Advanced Programming in the UNIX® Environment

Second Edition

W. Richard Stevens,
Stephen A. Rago

Addison-Wesley

HIGH TECH

UNIX

Профессиональное программирование

Второе издание

У. Ричард Стивенс,
Стивен А. Раго

Санкт-Петербург — Москва
2007

Серия «High tech»

У. Ричард Стивенс, Стивен А. Раго

UNIX. Профессиональное программирование, 2-е издание

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*М. Деркачев*
Редактор	*Р. Павлов*
Корректор	*Н. Ткачева*
Верстка	*Д. Орлова*

Стивенс Р., Раго С.
UNIX. Профессиональное программирование, 2-е издание. – СПб.: Символ-Плюс, 2007. – 1040 с., ил.
ISBN 5-93286-089-8

«UNIX. Профессиональное программирование» представляет собой подробнейшее справочное руководство для любого профессионального программиста, работающего с UNIX. Стивену Раго удалось обновить и дополнить текст фундаментального классического труда Стивенса, сохранив при этом точность и стиль оригинала. Содержание всех тем, примеров и прикладных программ обновлено в соответствии с последними версиями наиболее популярных реализаций UNIX. Среди важных дополнений главы, посвященные потокам и разработке многопоточных программ, использованию интерфейса сокетов для организации межпроцессного взаимодействия (IPC), а также широкий охват интерфейсов, добавленных в последней версии POSIX.1. Аспекты прикладного программного интерфейса разъясняются на простых и понятных примерах, протестированных на 4-х платформах: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9 и Mac OS X 10.3. Описывается множество ловушек, о которых следует помнить при написании программ для различных реализаций UNIX, и показывается, как их избежать, опираясь на стандарты POSIX.1 и Single UNIX Specification.

ISBN-13: 978-5-93286-089-2
ISBN-10: 5-93286-089-8
ISBN 0-201-43307-9 (англ)

© Издательство Символ-Плюс, 2007
Authorized translation of the English edition © 2005 Pearson Education, Inc. This translation is published and sold by permission of Pearson Education, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 18.04.2007. Формат 70x100 1/16 . Печать офсетная.
Объем 65 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Посвящается Джин

Оглавление

Отзывы ко второму и первому изданиям 15

Вступительное слово 17

Предисловия ко второму и первому изданиям 19

1. Обзор операционной системы UNIX 27

1.1. Введение 27
1.2. Архитектура UNIX 27
1.3. Вход в систему 28
1.4. Файлы и каталоги 30
1.5. Ввод и вывод 35
1.6. Программы и процессы 38
1.7. Обработка ошибок 41
1.8. Идентификация пользователя 44
1.9. Сигналы 46
1.10. Представление времени 48
1.11. Системные вызовы и библиотечные функции 49
1.12. Подведение итогов 52

2. Стандарты и реализации UNIX 53

2.1. Введение 53
2.2. Стандартизация UNIX 53
2.2.1. ISO C 53
2.2.2. IEEE POSIX 55
2.2.3. Single UNIX Specification 63
2.2.4. FIPS 64
2.3. Реализации UNIX 65
2.3.1. UNIX System V Release 4 65
2.3.2. 4.4BSD 66
2.3.3. FreeBSD 67
2.3.4. Linux 67
2.3.5. Mac OS X 67
2.3.6. Solaris 68
2.3.7. Прочие версии UNIX 68
2.4. Связь между стандартами и реализациями 68

2.5. Пределы 69
2.5.1. Пределы ISO C 70
2.5.2. Пределы POSIX 72
2.5.3. Пределы XSI 74
2.5.4. Функции sysconf, pathconf и fpathconf 75
2.5.5. Неопределенные пределы времени выполнения 84
2.6. Необязательные параметры 88
2.7. Макроопределения контроля функциональных особенностей . . . 92
2.8. Элементарные системные типы данных 93
2.9. Конфликты между стандартами 94
2.10. Подведение итогов 95

3. Файловый ввод-вывод 96

3.1. Введение 96
3.2. Дескрипторы файлов 96
3.3. Функция open 97
3.4. Функция creat 100
3.5. Функция close 101
3.6. Функция lseek 101
3.7. Функция read 105
3.8. Функция write 106
3.9. Эффективность операций ввода-вывода 107
3.10. Совместное использование файлов 109
3.11. Атомарные операции 113
3.12. Функции dup и dup2 115
3.13. Функции sync, fsync и fdatasync 117
3.14. Функция fcntl 118
3.15. Функция ioctl 124
3.16. /dev/fd 126
3.17. Подведение итогов 127

4. Файлы и каталоги 129

4.1. Введение 129
4.2. Функции stat, fstat и lstat 129
4.3. Типы файлов 130
4.4. set-user-ID и set-group-ID 134
4.5. Права доступа к файлу 135
4.6. Принадлежность новых файлов и каталогов 138
4.7. Функция access 139
4.8. Функция umask 140
4.9. Функции chmod и fchmod 143
4.10. Бит sticky 146
4.11. Функции chown, fchown и lchown 146

4.12. Размер файла 148
4.13. Усечение файлов 149
4.14. Файловые системы 150
4.15. Функции link, unlink, remove и rename 153
4.16. Символические ссылки 157
4.17. Функции symlink и readlink 160
4.18. Временные характеристики файлов 161
4.19. Функция utime 162
4.20. Функции mkdir и rmdir 165
4.21. Чтение каталогов 167
4.22. Функции chdir, fchdir и getcwd 172
4.23. Специальные файлы устройств 175
4.24. Коротко о битах прав доступа к файлам 177
4.25. Подведение итогов 179

5. Стандартная библиотека ввода–вывода 181

5.1. Введение 181
5.2. Потоки и объекты FILE 181
5.3. Стандартные потоки ввода, вывода и сообщений об ошибках . . . 183
5.4. Буферизация 183
5.5. Открытие потока 186
5.6. Чтение из потока и запись в поток 189
5.7. Построчный ввод-вывод 192
5.8. Эффективность стандартных функций ввода-вывода 193
5.9. Ввод-вывод двоичных данных 196
5.10. Позиционирование в потоке 198
5.11. Форматированный ввод-вывод 199
5.12. Подробности реализации 205
5.13. Временные файлы 207
5.14. Альтернативы стандартной библиотеке ввода-вывода 211
5.15. Подведение итогов 212

6. Информация о системе и файлы данных 213

6.1. Введение 213
6.2. Файл паролей 213
6.3. Теневые пароли 217
6.4. Файл групп 219
6.5. Идентификаторы дополнительных групп 220
6.6. Различия реализаций 222
6.7. Прочие файлы данных 223
6.8. Учет входов в систему 224
6.9. Информация о системе 225

6.10. Функции даты и времени 227
6.11. Подведение итогов 232

7. Среда окружения процесса 234

7.1. Введение 234
7.2. Функция main 234
7.3. Завершение работы процесса 235
7.4. Аргументы командной строки 240
7.5. Список переменных окружения 240
7.6. Раскладка памяти программы на языке C 241
7.7. Разделяемые библиотеки 243
7.8. Распределение памяти 244
7.9. Переменные окружения 248
7.10. Функции setjump и longjump 252
7.11. Функции getrlimit и setrlimit 259
7.12. Подведение итогов 264

8. Управление процессами 266

8.1. Введение 266
8.2. Идентификаторы процесса 266
8.3. Функция fork 268
8.4. Функция vfork 274
8.5. Функция exit 276
8.6. Функции wait и waitpid 279
8.7. Функция waitid 285
8.8. Функции wait3 и wait4 286
8.9. Гонка за ресурсами 287
8.10. Функция exec 291
8.11. Изменение идентификаторов пользователя и группы 298
8.12. Интерпретируемые файлы 304
8.13. Функция system 308
8.14. Учет использования ресурсов процессами 313
8.15. Идентификация пользователя 320
8.16. Временные характеристики процесса 320
8.17. Подведение итогов 323

9. Взаимоотношения между процессами 325

9.1. Введение 325
9.2. Вход с терминала 325
9.3. Вход в систему через сетевое соединение 331
9.4. Группы процессов 333
9.5. Сессии 335
9.6. Управляющий терминал 337

9.7. Функции tcgetpgrp, tcsetpgrp и tcgetsid 339
9.8. Управление заданиями 340
9.9. Выполнение программ командной оболочкой 343
9.10. Осиротевшие группы процессов 349
9.11. Реализация в FreeBSD 352
9.12. Подведение итогов 355

10. Сигналы 356

10.1. Введение 356
10.2. Концепция сигналов 356
10.3. Функция signal 367
10.4. Ненадежные сигналы 371
10.5. Прерванные системные вызовы 373
10.6. Реентерабельные функции 376
10.7. Семантика сигнала SIGCLD 379
10.8. Надежные сигналы. Терминология и семантика 382
10.9. Функции kill и raise 383
10.10. Функции alarm и pause 385
10.11. Наборы сигналов 391
10.12. Функция sigprocmask 393
10.13. Функция sigpending 394
10.14. Функция sigaction 397
10.15. Функции sigsetjmp и siglongjmp 403
10.16. Функция sigsuspend 407
10.17. Функция abort 414
10.18. Функция system 417
10.19. Функция sleep 422
10.20. Сигналы управления заданиями 424
10.21. Дополнительные возможности 427
10.22. Подведение итогов 429

11. Потоки 431

11.1. Введение 431
11.2. Концепция потоков 431
11.3. Идентификация потоков 433
11.4. Создание потока 434
11.5. Завершение потока 437
11.6. Синхронизация потоков 445
11.7. Подведение итогов 464

12. Управление потоками 465

12.1. Введение 465
12.2. Пределы для потоков 465

12.3. Атрибуты потока 466
12.4. Атрибуты синхронизации 472
12.5. Реентерабельность 480
12.6. Локальные данные потоков 485
12.7. Принудительное завершение потоков 490
12.8. Потоки и сигналы 494
12.9. Потоки и fork 498
12.10. Потоки и операции ввода-вывода 502
12.11. Подведение итогов 503

13. Процессы-демоны 504

13.1. Введение 504
13.2. Характеристики демонов 504
13.3. Правила программирования демонов 506
13.4. Журналирование ошибок 510
13.5. Демоны в единственном экземпляре 515
13.6. Соглашения для демонов 517
13.7. Модель клиент-сервер 522
13.8. Подведение итогов 522

14. Расширенные операции ввода-вывода 523

14.1. Введение 523
14.2. Неблокирующий ввод-вывод 523
14.3. Блокировка записей 527
14.4. STREAMS 544
14.5. Мультиплексирование ввода-вывода 558
14.5.1. Функции select и pselect 561
14.5.2. Функция poll 566
14.6. Асинхронный ввод-вывод 569
14.6.1. Асинхронный ввод-вывод в System V 570
14.6.2. Асинхронный ввод-вывод в BSD 571
14.7. Функции readv и writev 571
14.8. Функции readn и writen 574
14.9. Операции ввода-вывода с отображаемой памятью 576
14.10. Подведение итогов 583

15. Межпроцессное взаимодействие 585

15.1. Введение 585
15.2. Неименованные каналы 586
15.3. Функции popen и pclose 594
15.4. Сопроцессы 601
15.5. FIFO 605
15.6. XSI IPC 609

15.6.1. Идентификаторы и ключи 610
15.6.2. Структура прав доступа 611
15.6.3. Конфигурируемые пределы 612
15.6.4. Преимущества и недостатки 613
15.7. Очереди сообщений 615
15.8. Семафоры 621
15.9. Разделяемая память 628
15.10. Свойства взаимодействий типа клиент-сервер 636
15.11. Подведение итогов 639

16. Межпроцессное взаимодействие в сети: сокеты 642

16.1. Введение 642
16.2. Дескрипторы сокетов 643
16.3. Адресация 647
16.3.1. Порядок байтов 647
16.3.2. Форматы адресов 649
16.3.3. Определение адреса 651
16.3.4. Присвоение адресов сокетам 659
16.4. Установление соединения 660
16.5. Передача данных 664
16.6. Параметры сокетов 679
16.7. Экстренные данные 682
16.8. Неблокирующий и асинхронный ввод-вывод 683
16.9. Подведение итогов 684

17. Расширенные возможности IPC 686

17.1. Введение 686
17.2. Каналы на основе STREAMS 686
17.2.1. Именованные каналы STREAMS 690
17.2.2. Уникальные соединения 691
17.3. Сокеты домена UNIX 695
17.3.1. Именованные сокеты домена UNIX 696
17.3.2. Уникальные соединения 698
17.4. Передача дескрипторов файлов 703
17.4.1. Передача дескрипторов с помощью каналов STREAMS 705
17.4.2. Передача дескрипторов с помощью сокетов домена UNIX 708
17.5. Сервер открытия файлов, версия 1 717
17.6. Сервер открытия файлов, версия 2 723
17.7. Подведение итогов 731

18. Терминальный ввод-вывод 733
18.1. Введение 733
18.2. Обзор 733
18.3. Специальные символы ввода 742
18.4. Получение и изменение характеристик терминала 748
18.5. Флаги режимов терминала 749
18.6. Команда stty 757
18.7. Функции для работы со скоростью передачи 758
18.8. Функции управления линией связи 759
18.9. Идентификация терминала 760
18.10. Канонический режим 766
18.11. Неканонический режим 769
18.12. Размер окна терминала 776
18.13. termcap, terminfo и curses 778
18.14. Подведение итогов 779

19. Псевдотерминалы 781
19.1. Введение 781
19.2. Обзор 781
19.3. Открытие устройств псевдотерминалов 788
19.3.1. Псевдотерминалы на основе STREAMS 790
19.3.2. Псевдотерминалы в BSD 793
19.3.3. Псевдотерминалы в Linux 797
19.4. Функция pty_fork 799
19.5. Программа pty 801
19.6. Использование программы pty 806
19.7. Дополнительные возможности 814
19.8. Подведение итогов 815

20. Библиотека базы данных 818
20.1. Введение 818
20.2. Предыстория 818
20.3. Библиотека 820
20.4. Обзор реализации 822
20.5. Централизация или децентрализация? 826
20.6. Одновременный доступ 828
20.7. Сборка библиотеки 829
20.8. Исходный код 830
20.9. Производительность 858
20.10. Подведение итогов 864

21. Взаимодействие с сетевым принтером 866
21.1. Введение 866
21.2. Протокол печати через Интернет 866
21.3. Протокол передачи гипертекста 869
21.4. Очередь печати 870
21.5. Исходный код 872
21.6. Подведение итогов 919

A. Прототипы функций 921

B. Различные исходные тексты 956

C. Варианты решения некоторых упражнений 965

Список литературы 1000

Алфавитный указатель 1008

Отзывы ко второму изданию

«Обновление, выполненное Стивеном Раго (Stephen Rago), – это событие, которого давно и с нетерпением ждало все сообщество профессионалов, использующих в своей работе многоликое семейство UNIX и UNIX-подобных операционных систем. В этом издании исключены устаревшие и добавлены новейшие сведения. Содержание всех тем, примеров и прикладных программ обновлено в соответствии с последними версиями наиболее популярных реализаций UNIX и UNIX-подобных операционных систем. И кроме того, при этом полностью сохранен стиль изложения оригинала».

– Мукеш Кэкер (Mukesh Kacker),
соучредитель и бывший технический директор Pronto Networks, Inc.

«Один из фундаментальных классических трудов, посвященных программированию для UNIX».

– Эрик С. Рэймонд (Eric S. Raymond),
автор книги «The Art of UNIX Programming»

«Это издание представляет собой подробнейшее справочное руководство для любого профессионального программиста, работающего с UNIX. Стивену Раго удалось обновить и дополнить текст классического произведения Стивенса, сохранив при этом точность оригинала. Аспекты прикладного программного интерфейса разъясняются на простых и понятных примерах. В книге также описывается множество ловушек, о которых следует помнить при написании программ для различных реализаций UNIX, и показывается, как их избежать, опираясь на соответствующие стандарты, такие как POSIX 1003.1 (редакция от 2004 года) и Single UNIX Specification, Version 3».

– Эндрю Джози (Andrew Josey), директор по сертификации
The Open Group и председатель рабочей группы POSIX 1003.1

«Второе издание книги – жизненно необходимый справочник для любого, кто занимается разработкой программ для UNIX. Эту книгу я открываю первой, когда хочу изучить или вспомнить какие-либо из интерфейсов системы. Стивен Раго удачно переработал содержание книги и включил в нее сведения о новейших операционных системах, таких как GNU/Linux и Apple OS X, придерживаясь при этом стиля первого издания – как в смысле удобочитаемости, так и в смысле полноты изложения. Для нее всегда найдется место рядом с моим компьютером».

– Доктор Бенджамин Куперман (Dr. Benjamin Kuperman),
колледж г. Свортмора (Swarthmore)

Отзывы к первому изданию

«Книга «Advanced Programming in the UNIX® Environment» обязательно должна быть у любого серьезного программиста, который пишет для UNIX на языке C. По своей основательности, глубине и ясности подачи материала она не имеет себе равных».

– UniForum Monthly

«Многочисленные читатели рекомендовали мне книгу «Advanced Programming in the UNIX® Environment», написанную Ричардом Стивенсом (издательство Addison-Wesley), и я благодарен им за это. Раньше я даже не слышал об этой книге, хотя она вышла в свет в 1992 году. Получив экземпляр книги, я с первых же глав был очарован ею».

– Open Systems Today

«Очень понятное и подробное описание внутреннего устройства UNIX вы найдете в книге «Advanced Programming in the UNIX® Environment», написанной Ричардом Стивенсом (Addison-Wesley). Она включает в себя множество практических примеров, и я нахожу ее очень полезной при разработке системного программного обеспечения».

– RS/Magazine

Вступительное слово

Почти в каждом интервью или после лекций в какой-то момент мне задают один и тот же вопрос: «Ожидали ли вы, что UNIX продержится так долго?». Разумеется, в ответ я говорю одно и то же: «Нет, для нас это оказалось полной неожиданностью». Некоторые даже подсчитали, что система в том или ином виде существует уже более половины всей жизни компьютерной индустрии.

Процесс развития был бурным и сложным. С начала 70-х годов прошлого столетия компьютерные технологии сильно изменились, особенно за счет глобальных сетевых технологий, вездесущей графики и широкого распространения персональных компьютеров, тем не менее система сумела учесть и вобрать в себя все эти явления. Несмотря на то, что сегодня в области настольных систем доминируют Microsoft и Intel, рынок в определенной степени двигается в направлении от единого поставщика к нескольким, а в последние годы все более ориентируется на открытые стандарты и свободно распространяемые системы.

К счастью, система UNIX, которую следует рассматривать как явление, а не только как торговую марку, не просто двигалась вперед, но и сумела занять лидирующее положение. В 70-х и 80-х годах XX века корпорация AT&T была держателем авторских прав на исходные тексты UNIX, но она всячески поощряла усилия по стандартизации, основанные на системных интерфейсах и языках. Например, AT&T опубликовала SVID (System V Interface Definition, описание интерфейса System V), которое легло в основу стандарта POSIX и последующих его модификаций. Так случилось, что UNIX смогла достаточно изящным образом приспособиться к работе в сетевом окружении и, может быть, менее элегантно, но все-таки на достаточно приемлемом уровне к работе с графикой. Кроме того, основные интерфейсы ядра UNIX и инструментальные средства уровня пользователя стали технологической основой движения за программное обеспечение, распространяемое с открытым исходным кодом.

Существенно то, что статьи и книги, посвященные системе UNIX, всегда были востребованы, даже в то время, когда программное обеспечение самой системы было запатентовано. Примером может служить книга Мориса Баха (Maurice Bach) «The Design of the Unix Operating System». Честно говоря, я мог бы утверждать, что основная причина такой долговечности системы состоит в ее привлекательности для талантливых авторов, которые стремились объяснить ее красоты и тайны. Брайан Керниган (Brian Kernighan) – один из них, а другой – конечно же, Рич Стивенс (Rich Stevens). Первое издание этой книги, а также серия его книг, посвященных сетевым технологи-

ям, справедливо считаются одними из лучших работ и потому пользуются заслуженной популярностью.

Первое издание этой книги вышло в свет еще до того, как получили широкое распространение Linux и другие реализации UNIX с открытыми исходными текстами, берущие свое начало из Беркли, а также когда большинство людей имели лишь модемное подключение к сети. Стив Раго (Steve Rago) тщательно выполнил обновление этой книги, чтобы учесть изменения, произошедшие в компьютерных технологиях и в стандартах ISO и IEEE с момента выхода первой публикации. Поэтому все примеры в книге обновлены и вновь протестированы.

Это самое достойное второе издание классики.

Деннис Ритчи (Dennis Ritchie)
Мюррей Хилл, Нью Джерси
Март 2005

Предисловие ко второму изданию

Введение

Мой первый контакт с Ричем Стивенсом состоялся по электронной почте, когда я сообщил ему об опечатке в его книге «UNIX Network Programming». Позже он говорил, что я оказался первым, кто прислал ему сообщение о найденной ошибке. До самой его смерти в 1999 году мы время от времени обменивались электронными письмами. Обычно это происходило, когда один из нас задавался каким-либо вопросом и полагал, что другой мог бы на него ответить. Мы встречались с ним за обедом на конференциях USENIX и на лекциях, которые он читал.

Рич Стивенс был другом и настоящим джентльменом. Когда в 1993 году я написал книгу «UNIX System V Network Programming», я подразумевал, что она является версией книги Рича «UNIX Network Programming», ориентированной на System V. Благодаря своему характеру Рич охотно взялся за рецензирование моих глав, воспринимая меня не как конкурента, но как коллегу. Мы часто говорили о сотрудничестве над версией его книги «TCP/IP Illustrated», посвященной STREAMS. Если бы события сложились по-иному, вероятно, мы сделали бы это, но Ричарда больше нет с нами, поэтому обновление данной книги я рассматриваю как самую тесную совместную нашу работу.

Когда издательство Addison-Wesley сообщило мне, что оно заинтересовано в обновлении книги Рича, я думал, что дополнений будет не очень много. Даже по прошествии 13 лет его работа остается достаточно актуальной. Но современный мир UNIX значительно отличается от того, каким он был во время выхода первого издания книги.

- Версии System V постепенно вытесняются операционной системой Linux. Основные производители аппаратного обеспечения, поставляющие свою продукцию в комплекте с собственными версиями UNIX, либо выпустили версии своих продуктов для Linux, либо объявили о ее поддержке. ОС Solaris, вероятно, осталась последней наследницей System V Release 4, обладающей более или менее заметной долей рынка.
- После выхода 4.4BSD группа по проведению исследований в области информационных технологий (CSRG – Computing Science Research Group) из Калифорнийского университета в Беркли приняла решение о завершении разработки операционной системы UNIX, однако существует несколько групп добровольцев, которые продолжают осуществлять поддержку общедоступных версий.

- Появление ОС Linux, поддерживаемой тысячами добровольцев, позволило любому обладать компьютером, оборудованным самыми новейшими устройствами и работающим под управлением UNIX-подобной операционной системы, распространяемой с исходными текстами. Успех Linux выглядит не совсем обычно, учитывая, что существует несколько свободно распространяемых BSD-альтернатив.
- В очередной раз проявив себя в качестве инновационной компании, Apple Computer отказалась от устаревшей операционной системы Mac и заменила ее системой, созданной на основе Mach и FreeBSD.

В связи с этим я попытался дополнить сведения, содержащиеся в книге, чтобы охватить эти четыре платформы.

Когда в 1992 году Рич написал свою книгу «Advanced Programming in the UNIX Environment», я избавился от почти всех руководств по программированию в UNIX. С тех пор я держу на своем столе две книги: словарь и «Advanced Programming in the UNIX Environment». Надеюсь, что вы найдете это издание книги не менее полезным.

Изменения во втором издании

Работа Рича сохранила свою актуальность. Я старался не изменять оригинальное изложение материала, но слишком много всего произошло за последние 13 лет. Особенно это относится к стандартам, которые затрагивают программные интерфейсы UNIX.

Везде, где это было необходимо, я дополнил описания интерфейсов, которые изменились в результате деятельности по стандартизации. Особенно это заметно в главе 2, посвященной стандартам. В этом издании мы будем основываться на стандарте POSIX.1 от 2001 года как более универсальном по сравнению с версией 1990 года, на которой была основана первая редакция книги. Стандарт ISO C от 1990 года был изменен и дополнен в 1999 году, и некоторые изменения коснулись интерфейсов, описываемых стандартом POSIX.1.

Сегодня спецификация POSIX.1 охватывает намного большее количество интерфейсов. Основные спецификации Single UNIX Specification (опубликованные The Open Group, ранее X/Open) вошли в состав POSIX.1. Теперь POSIX.1 включает в себя ряд стандартов 1003.1 и некоторые из предварительных стандартов, опубликованных ранее.

В соответствии с этим я дополнил книгу новыми главами, охватывающими новые темы. Понятия потоков и многопоточных приложений очень важны, поскольку они предоставляют программистам более элегантный способ организации параллельных вычислений и асинхронной обработки.

Интерфейс сокетов теперь стал частью стандарта POSIX.1. Он обеспечивает единый интерфейс межпроцессного взаимодействия (IPC – Interprocess Communication), не зависящий от местонахождения процессов, и его обсуждение является естественным продолжением глав, посвященных IPC.

Я опустил рассмотрение большинства интерфейсов реального времени, которые появились в POSIX.1. Их лучше всего изучать по книгам, специально посвященным созданию приложений реального времени. Одну из таких книг вы найдете в библиографии.

Я изменил некоторые примеры в последних главах, чтобы приблизить их к задачам реальной жизни. Например, в наши дни не многие системы работают с PostScript-принтерами через последовательный или параллельный порт. Чаще встречается случай, когда доступ к таким принтерам осуществляется посредством сети, поэтому я изменил учебный пример взаимодействия с принтером так, чтобы учесть это обстоятельство.

Глава, посвященная взаимодействию с модемом, потеряла свою актуальность. Однако, чтобы оригинальный материал не был утерян окончательно, он выложен на веб-сайте книги в двух форматах: PostScript (*http://www.apuebook.com/lostchapter/modem.ps*) и PDF (*http://www.apuebook.com/lostchapter/modem.pdf*).

Все исходные тексты примеров из книги также доступны на сайте *www.apuebook.com*. Большая часть примеров была протестирована на четырех платформах:

1. FreeBSD 5.2.1 – системе, происходящей от 4.4BSD, работающей на процессоре Intel Pentium.
2. Linux 2.4.22 (дистрибутив Mandrake 9.2) – свободно распространяемой UNIX-подобной операционной системе, работающей на процессоре Intel Pentium.
3. Solaris 9 – происходящей от System V Release 4 системе от Sun Microsystems, работающей на 64-битном процессоре UltraSPARCIIi.
4. Darwin 7.4.0 – системе, основанной на FreeBSD и Mach, которая поддерживается Apple Mac OS X, версия 10.3, на процессоре PowerPC.

Благодарности

Первое издание этой книги, которое сразу же стало классикой, было полностью написано Ричем Стивенсом.

Вероятно, мне не удалось бы справиться с обновлением книги без поддержки моей семьи. Они стоически терпели груды разбросанных повсюду бумаг (определенно их было больше, чем обычно), монополизацию мною большинства компьютеров в доме и огромное количество времени, когда мое лицо было спрятано за терминалом. Моя супруга Джин (Jeanne) даже помогла мне, установив Linux на одну из тестовых машин.

Технические рецензенты предложили немало улучшений и исправлений и помогли удостовериться в правильности содержимого книги. Большое спасибо Дэвиду Бозуму (David Bausum), Дэвиду Борхему (David Boreham), Кейту Бостику (Keith Bostic), Марку Эллису (Mark Ellis), Филу Говарду (Phil Howard), Эндрю Джози (Andrew Josey), Мукешу Кэкеру (Mukesh Kacker), Брай-

ану Кернигану (Brian Kernighan), Бенгту Клебергу (Bengt Kleberg), Бену Куперману (Ben Kuperman), Эрику Рэймонду (Eric Raimond) и Энди Рудофу (Andy Rudoff).

Кроме того, я хотел бы поблагодарить Энди Рудофа за ответы на вопросы, касающиеся ОС Solaris, и Денниса Ритчи за то, что он, «зарывшись» в старые бумаги, отвечал на вопросы по истории UNIX. Еще раз хочу поблагодарить сотрудников издательства Addison-Wesley, с которыми было приятно работать. Спасибо Тиррелу Альбо (Tyrrell Albaugh), Мэри Франц (Mary Franz), Джону Фуллеру (John Fuller), Карен Геттман (Karen Gettman), Джессике Голдстейн (Jessica Goldstein), Норин Реджине (Noreen Regina) и Джону Уэйту (John Wait). Мои благодарности Эвелин Пайл (Evelyn Pyle) за отличную работу по техническому редактированию.

Я также буду благодарен всем читателям, кто пришлет по электронной почте свои комментарии, предложения и замечания об ошибках.

Уоррен, Нью Джерси
Апрель 2005

Стивен Раго
sar@apuebook.com

Предисловие к первому изданию

Введение

В этой книге описаны программные интерфейсы системы UNIX: интерфейс системных вызовов и многочисленные функции, предоставляемые стандартной библиотекой языка C. Она предназначена для всех, кто пишет программы, работающие под управлением UNIX.

Подобно большинству операционных систем, UNIX предоставляет работающим в ней программам разнообразные службы: открытие и чтение файлов, запуск новых программ, выделение областей памяти, получение текущего времени и т. д. Все это называется *интерфейсом системных вызовов* (*system call interface*). Дополнительно стандартная библиотека языка C предоставляет огромное количество функций, которые используются практически в любой программе, написанной на C (форматированный вывод значений переменных, сравнение строк и тому подобное).

Интерфейс системных вызовов и библиотечные функции традиционно описываются во втором и третьем разделах «Unix Programmer's Manual» (Руководства программиста UNIX). Эта книга не дублирует указанные разделы. В ней вы найдете примеры и пояснения, которые отсутствуют в упомянутом руководстве.

Стандарты UNIX

Быстрый рост количества версий UNIX, наблюдавшийся в 80-е годы, был урегулирован различными международными стандартами, которые стали появляться с конца 80-х. К ним относятся стандарт ANSI языка программирования C, семейство стандартов IEEE POSIX (которые продолжают развиваться и по сей день) и руководство по обеспечению переносимости X/Open.

Данная книга описывает эти стандарты. И не просто описывает, а рассматривает их применительно к популярным реализациям – System V Release 4 и грядущей 4.4BSD. Здесь представлены соответствующие действительности описания, которых зачастую недостает самим стандартам и книгам, которые только описывают стандарты.

Организация книги

Эта книга делится на шесть частей:

1. Обзор и введение в базовые концепции, связанные с программированием в UNIX, и в терминологию (глава 1). Обсуждение достижений в области стандартизации UNIX и различных реализаций UNIX (глава 2).
2. Ввод-вывод: небуферизованный ввод-вывод (глава 3), характеристики файлов и каталогов (глава 4), стандартная библиотека ввода-вывода (глава 5) и стандартные системные файлы (глава 6).
3. Процессы: окружение процессов в UNIX (глава 7), управление процессами (глава 8), взаимоотношения между различными процессами (глава 9) и сигналы (глава 10).
4. Дополнительно об операциях ввода-вывода: терминальный ввод-вывод (глава 11), расширенные операции ввода-вывода (глава 12) и процессы-демоны (глава 13).
5. IPC – взаимодействия между процессами (главы 14 и 15).
6. Примеры: библиотека базы данных (глава 16), взаимодействие с PostScript-принтером (глава 17), программа работы с модемом (глава 18) и использование псевдотерминалов (глава 19).

При чтении книги не лишним будет знание языка C, равно как и некоторый опыт использования UNIX. Изложение материала не предполагает наличия опыта разработки программ для UNIX. Книга предназначена для программистов, знакомых с UNIX или с другими операционными системами и желающих детально изучить возможности, предоставляемые большинством реализаций UNIX.

Примеры в книге

Данная книга содержит множество примеров – примерно 10 000 строк исходного кода. Все примеры написаны на языке C. Кроме того, примеры написаны в соответствии со стандартом ANSI C. Желательно, чтобы при чте-

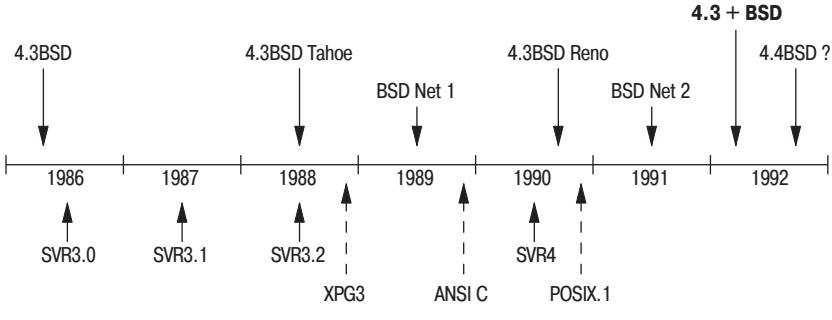
нии этой книги у вас под рукой была копия «Unix Programmer's Manual» (Руководства программиста UNIX) для вашей операционной системы, так как мы часто будем ссылаться на него при обсуждении малопонятных или зависящих от реализации особенностей.

Обсуждение практически каждой функции или системного вызова будет сопровождаться небольшой законченной программой. Это намного проще, чем рассматривать те же функции в больших программах, и позволит нам исследовать аргументы и возвращаемые значения. Так как некоторые из маленьких программ представляют собой достаточно искусственные примеры, мы включили в книгу несколько больших примеров (главы 16, 17, 18 и 19). Они демонстрируют решение задач, взятых из реальной жизни.

Все примеры программ были включены в текст книги прямо из файлов с исходными текстами. Копии всех примеров в виде файлов вы найдете на анонимном сервере FTP по адресу *ftp.uu.net*, в архиве *published/books/stevens.advprog.tar.Z*. Вы можете взять эти исходные тексты и экспериментировать с ними в вашей операционной системе.

Перечень систем, использовавшихся для тестирования примеров

К сожалению, все операционные системы находятся в постоянном движении. UNIX не является исключением. Следующая диаграмма показывает процесс эволюции различных версий System V и 4.xBSD.



Под обозначением 4.xBSD подразумеваются различные операционные системы от Computer Systems Research Group из Калифорнийского университета в Беркли. Эта группа также занимается распространением BSD Net 1 и BSD Net 2 – общедоступных исходные текстов операционных систем семейства 4.xBSD. Под обозначением SVRx подразумевается System V Release x от AT&T. XPG3 – это «X/Open Portability Guide, issue 3» (Руководство X/Open по обеспечению переносимости, выпуск 3). ANSI C – это стандарт ANSI языка программирования C. POSIX.1 – это стандарт ISO и IEEE на интерфейс UNIX-подобных операционных систем. Более подробно об этих стандартах и различных версиях UNIX мы поговорим в разделах 2.2 и 2.3.

В этой книге под обозначением *4.3+BSD* мы будем подразумевать версии UNIX, которые появились между выпусками BSD Net 2 и 4.4BSD.

К моменту написания книги версия 4.4BSD еще не была выпущена, поэтому было бы преждевременно говорить об операционной системе 4.4BSD. Однако эту серию операционных систем нужно было как-то обозначить, поэтому повсюду в книге мы будем использовать обозначение *4.3+BSD*.

Большинство примеров в книге были протестированы в четырех различных версиях UNIX:

1. UNIX System V/386 Release 4.0 Version 2.0 («чистая SVR4») от U.H. Corp. (UHC), работающая на процессоре Intel 80386.
2. 4.3+BSD от Computer Systems Research Group, отделение информатики, Калифорнийский университет в Беркли, работающая на рабочей станции от Hewlett Packard.
3. BSD/386 (производная от BSD Net 2) от Berkeley Software Design Inc., работающая на процессоре Intel 80386. Эта система практически полностью соответствует тому, что мы называем 4.3+BSD.
4. SunOS, версии 4.1.1 и 4.1.2 от Sun Microsystems (системы, в которых хорошо заметно наследие Беркли, но при этом много дополнений, пришедших из System V), работающие на SPARC-станциях SLC.

В книге представлены многочисленные тесты на производительность с указанием проверяемых операционных систем.

Благодарности

Я многим обязан моей семье за любовь, поддержку и множество выходных дней, потерянных за последние полтора года. Создание книги – во многом заслуга семьи. Спасибо вам, Салли, Билл, Эллен и Дэвид.

Я особенно благодарен Брайану Кернигану за его помощь при работе над книгой. Его многочисленные рецензии рукописи и ненавязчивые рекомендации по улучшению стиля изложения, надеюсь, будут заметны в окончательном варианте. Стив Раго также посвятил немало времени рецензированию рукописи и ответам на многие вопросы об устройстве и истории развития System V. Выражаю свою благодарность другим техническим рецензентам издательства Addison-Wesley, которые дали ценные комментарии по различным частям рукописи: Мори Баху (Maury Bach), Марку Эллису (Mark Ellis), Джефу Гитлину (Jeff Gitlin), Питеру Ханиману (Piter Honeyman), Джону Линдерману (John Linderman), Дугу Мак-Илрою (Doug McIlroy), Эви Немет (Evi Nemeth), Крэйгу Партриджу (Craig Partridge), Дейву Пресотто (Dave Presotto), Гэри Уилсону (Gary Wilson) и Гэри Райту (Gary Wright).

Кейт Бостик (Keith Bostic) и Кирк Мак-Кьюсик (Kirk McKusick) из U.C. Berkeley CSRG предоставили учетную запись, которая использовалась для проверки примеров на последней версии BSD (также большое спасибо Питеру Салусу (Peter Salus)). Сэм Нэйтарос (Sam Nataros) и Йоахим Саксен (Joachim Sacksen) из UHC предоставили копию операционной системы SVR4 для те-

стирования примеров. Трент Хейн (Trent Hein) помог получить альфа- и бета-версии BSD/386.

Другие мои друзья на протяжении последних лет часто оказывали мне немаловажную помощь: Пол Лукина (Paul Lucchina), Джо Годсил (Joe Godsil), Джим Хог (Jim Hogue), Эд Танкус (Ed Tankus) и Гэри Райт (Gary Wright). Редактор из издательства Addison-Wesley, Джон Уэйт (John Wait), был моим большим другом на протяжении всего этого времени. Он никогда не жаловался на срыв сроков и постоянное увеличение числа страниц. Отдельное спасибо Национальной оптической астрономической обсерватории (NOAO) и особенно Сиднею Уольфу (Sidney Wolff), Ричарду Уольфу (Richard Wolff) и Стиву Гранди (Steve Grandi) за предоставленное машинное время.

Настоящие книги о UNIX пишутся в формате troff, и данная книга также следует этой проверенной временем традиции. Копия книги, готовая к тиражированию, была подготовлена автором с помощью пакета groff, созданного Джеймсом Кларком (James Clark). Большое ему спасибо за такой замечательный пакет и за его быстрый отклик на замеченные ошибки. Быть может, мне когда-нибудь удастся разобраться с концевыми сносками в troff.

Я буду благодарен всем читателям, которые пришлют по электронной почте свои комментарии, предложения и замечания об ошибках.

Таксон, Аризона
Апрель 1992

Ричард Стивенс
rstevens@kohala.com
http://ww.kohala.com/~rstevens

1

Обзор ОС UNIX

1.1. Введение

Любая операционная система обслуживает работающие в ней программы. Обычно это обслуживание включает в себя запуск новых программ, открытие файлов, чтение из файлов, выделение областей памяти, получение текущего времени суток и многое другое. В этой книге рассказывается о сервисах, предоставляемых различными версиями операционной системы UNIX.

Строго линейное описание системы UNIX без опережающего использования терминов, которые фактически еще не были описаны, практически невозможно (и такое изложение, скорее всего, было бы скучным). Эта глава предлагает программисту обзорную экскурсию по системе UNIX. Мы дадим краткие описания и примеры некоторых терминов и понятий, которые будут встречаться на протяжении всей книги. В последующих главах мы рассмотрим их более подробно. Эта глава также содержит обзор сервисов, предоставляемых системой UNIX, для тех программистов, кто мало знаком с ней.

1.2. Архитектура UNIX

Строго говоря, операционная система определяется как программное обеспечение, которое управляет аппаратными ресурсами компьютера и предоставляет среду выполнения прикладных программ. Обычно это программное обеспечение называют *ядром* (*kernel*), так как оно имеет относительно небольшой объем и составляет основу системы. На рис. 1.1 изображена схема, отражающая архитектуру системы UNIX.

Интерфейс ядра – это слой программного обеспечения, называемый *системными вызовами* (заштрихованная область на рис. 1.1). Библиотеки функций общего пользования строятся на основе интерфейса системных вызовов, но прикладная программа может свободно пользоваться как теми, так и другими (более подробно о библиотечных функциях и системных вызовах мы по-

***Рис. 1.1.** Архитектура системы UNIX*

говорим в разделе 1.11). Командная оболочка – это особое приложение, которое предоставляет интерфейс для запуска других приложений.

В более широком смысле операционная система – это ядро и все остальное программное обеспечение, которое делает компьютер пригодным к использованию и обеспечивает его индивидуальность. В состав этого программного обеспечения входят системные утилиты, прикладные программы, командные оболочки, библиотеки функций общего пользования и тому подобное.

Например, Linux – это ядро операционной системы GNU. Некоторые так и называют эту операционную систему – GNU/Linux, но чаще всего ее именуют просто Linux. Хотя, строго говоря, такое наименование не является правильным, но оно вполне понятно, если учесть двоякий смысл выражения *операционная система*. (И кроме того, оно обладает таким преимуществом, как краткость.)

1.3. Вход в систему

Имя пользователя

При входе в систему UNIX мы вводим имя пользователя и пароль. После этого система отыскивает введенное имя в файле паролей; обычно это файл `/etc/passwd`. Файл паролей содержит записи, каждая из которых состоит из семи полей, разделенных двоеточиями: имя пользователя, зашифрованный пароль, числовой идентификатор пользователя (`205`), числовой идентификатор группы (`105`), поле комментария, домашний каталог (`/home/sar`) и командный интерпретатор (`/bin/ksh`).

```
sar:x:205:105:Stephen Rago:/home/sar:/bin/ksh
```

Все современные системы хранят пароли в отдельном файле. В главе 6 мы рассмотрим эти файлы и некоторые функции для доступа к ним.

Командные оболочки

Обычно после входа в систему на экран выводится некоторая системная информация, после чего мы можем вводить команды, предназначенные для командной оболочки. (В некоторых системах после ввода имени пользователя и пароля запускается графический интерфейс, но и в этом случае, как правило, можно получить доступ к командной оболочке, запустив командный интерпретатор в одном из окон.) *Командная оболочка* – это интерпретатор командной строки, который считывает ввод пользователя и выполняет команды. Ввод пользователя обычно осуществляется посредством терминала (интерактивная командная оболочка) или считывается из файла (который называется *сценарием командной оболочки*). Перечень наиболее распространенных командных оболочек приведен в табл. 1.1.

Таблица 1.1. Наиболее распространенные командные оболочки, используемые в UNIX

Название	Путь	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Bourne shell	/bin/sh	•	Ссылка на bash	Ссылка на bash	•
Bourne-again shell	/bin/bash	Необязательно	•	•	•
C shell	/bin/csh	Ссылка на tcsh	Ссылка на tcsh	Ссылка на tcsh	•
Korn shell	/bin/ksh				•
TENEX C shell	/bin/tcsh	•	•	•	•

Информацию о том, какой командный интерпретатор следует запустить, система извлекает из записи в файле паролей.

Командный интерпретатор Bourne shell был разработан в Bell Labs Стивом Борном (Steve Bourne). Он входит в состав практически всех существующих версий UNIX, начиная с Version 7. Управляющие конструкции в Bourne shell чем-то напоминают язык программирования Algol 68.

Командный интерпретатор C shell был разработан в Беркли Биллом Джоем (Bill Joy) и входит в состав всех версий BSD. Кроме того, C shell включен в состав System V/386 Release 3.2 от AT&T, а также в System V Release 4 (SVR4). (В следующей главе мы расскажем об этих версиях UNIX подробнее.) Командная оболочка C shell разработана на основе оболочки 6-й Редакции UNIX, а не Bourne shell. Управляющие конструкции этого интерпретатора напоминают язык программирования C, и кроме того, он поддерживает дополнительные особенности, которые отсутствуют в Bourne shell: управление заданиями, историю команд и возможность редактирования командной строки.

Командный интерпретатор Korn shell можно считать наследником Bourne shell. Он впервые появился в составе SVR4. Разработанный в Bell Labs Дэвидом Корном (David Korn), он может работать на большинстве версий UNIX, но до выхода SVR4 обычно распространялся как дополнение за отдельную плату и поэтому не получил такого широкого распространения, как преды-

дущие два интерпретатора. Он сохраняет обратную совместимость с Bourne shell и предоставляет те же возможности, благодаря которым C shell стал таким популярным: управление заданиями, возможность редактирования командной строки и прочее.

Bourne-again shell – это командный интерпретатор GNU, который входит в состав всех версий ОС Linux. Он был разработан в соответствии со стандартом POSIX и в то же время сохраняет совместимость с Bourne shell, а также поддерживает особенности, присущие C shell и Korn shell.

Командный интерпретатор TENEX C shell представляет собой расширенную версию C shell. Он заимствовал некоторые особенности, такие как автодополнение команд, из ОС TENEX, разработанной в 1972 году в компании Bolt Beranek and Newman. TENEX C shell расширяет возможности C shell и часто используется в качестве его замены.

> ОС Linux использует Bourne-again shell как командный интерпретатор по умолчанию. Фактически `/bin/sh` представляет собой ссылку на `/bin/bash`. В системах FreeBSD и Mac OS X командной оболочкой по умолчанию для пользователей является TENEX C shell, но в сценариях системного администрирования используется Bourne shell, поскольку язык командной оболочки C shell славится своей сложностью. Операционная система Solaris унаследовала от BSD и System V все командные интерпретаторы, перечисленные в табл. 1.1. Свободно распространяемые версии большинства командных интерпретаторов можно найти в Интернете.
>
> На протяжении всей книги в подобных примечаниях мы будем приводить исторические заметки и сравнения различных реализаций UNIX. Зачастую обоснования различных методов реализации становятся гораздо понятнее в историческом контексте.

В тексте книги приводятся многочисленные примеры взаимодействия с командным интерпретатором, демонстрирующие запуск разрабатываемых нами программ. В этих примерах используются возможности, общие для Bourne shell, Korn shell и Bourne-again shell.

1.4. Файлы и каталоги

Файловая система

Файловая система UNIX представляет собой иерархическую древовидную структуру, состоящую из каталогов и файлов. Начинается она с каталога, который называется *корнем* (*root*), а имя этого каталога представлено единственным символом – /.

Каталог представляет собой файл, в котором содержатся каталожные записи. Логически каждую такую запись можно представить в виде структуры, состоящей из имени файла и дополнительной информации, описывающей атрибуты файла. Атрибуты файла – это такие характеристики, как тип файла (обычный файл или каталог), размер файла, владелец файла, права доступа к файлу (есть ли у других пользователей доступ к файлу), время последней модификации файла. Функции `stat` и `fstat` возвращают структуру, в ко-

торой содержится информация о всех атрибутах файла. В главе 4 мы исследуем атрибуты файла более детально.

> Мы различаем логическое представление каталожной записи и способ, которым эта информация фактически хранится на диске. Большинство реализаций UNIX не хранят атрибуты в каталожных записях из-за сложностей, связанных с их синхронизацией в случае, когда имеется несколько жестких ссылок на файл. Это станет понятным, когда мы будем обсуждать жесткие ссылки в главе 4.

Имя файла

Имена элементов каталога называются *именами файлов*. Только два символа не могут встречаться в имени файла – это прямой слэш (/) и нулевой символ (\0). Символ слэша разделяет имена файлов, из которых состоит строка пути к файлу (описывается ниже), а нулевой символ обозначает конец этой строки. Однако на практике лучше ограничиться подмножеством обычных печатных символов. (Мы ограничиваем набор допустимых символов, потому что некоторые специальные символы имеют особое значение для командного интерпретатора, и при их использовании в именах файлов нам пришлось бы применять экранирование, а это может привести к определенным сложностям.)

Всякий раз, когда создается новый каталог, автоматически создаются два файла: . (называется *точка*) и .. (называется *точка-точка*). Под именем «точка» подразумевается текущий каталог, а под именем «точка-точка» – родительский. В корневом каталоге «точка-точка» представляет тот же самый каталог, что и «точка».

В некоторых устаревших версиях UNIX System V длина имени файла ограничена 14 символами. В версиях BSD этот предел был увеличен до 255 символов. Сегодня практически все файловые системы коммерческих версий UNIX поддерживают имена файлов длиной не менее 255 символов.

Путь к файлу

Последовательность одного или более имен файлов, разделенных слэшами, образует строку пути к файлу. Эта строка может также начинаться с символа слэша, и тогда она называется строкой *абсолютного пути*, в противном случае – строкой *относительного пути*. В случае относительного пути маршрут начинается от текущего каталога. Обозначение корня файловой системы (/) – особый случай строки абсолютного пути, которая не содержит ни одного имени файла.

Пример

Вывести список всех файлов в каталоге достаточно просто. Ниже приводится пример упрощенной реализации команды `ls`(1).

Листинг 1.1. Вывод списка всех файлов в каталоге

```
#include "apue.h"
#include <dirent.h>
```

```
int
main(int argc, char *argv[])
{
    DIR *dp;
    struct dirent *dirp;

    if (argc != 2)
        err_quit("Использование: ls имя_каталога");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("невозможно открыть %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Нотация `ls`**(1)** – это обычный способ указания страницы справочного руководства UNIX. В данном случае говорится, что страница с описанием команды `ls` находится в первом разделе. Разделы справочного руководства обычно нумеруются цифрами от 1 до 8, а страницы в каждом разделе отсортированы по алфавиту. Здесь и далее мы будем исходить из предположения, что у вас под рукой имеется копия справочного руководства по вашей версии UNIX.

> Исторически сложилось так, что все восемь разделов были объединены в документ, который называется «UNIX Programmer's Manual» (Руководство программиста UNIX). Но поскольку количество страниц в руководстве постоянно растет, появилась тенденция к распределению разделов по отдельным руководствам: одно для пользователей, одно для программистов и одно для системных администраторов.
>
> В некоторых версиях UNIX разделы справочного руководства делятся на подразделы, обозначенные заглавными буквами. Так, описания всех стандартных функций ввода-вывода в AT&T [1990e] находятся в разделе 3S, например `fopen`(3S). В некоторых системах в обозначениях разделов цифры заменены алфавитными символами, например «C» – для раздела с описаниями команд.

В наши дни большинство руководств распространяется в электронном виде. Если вы располагаете такой версией справочного руководства, просмотреть справку по команде `ls` можно следующим образом:

```
man 1 ls
```

или

```
man -sl ls
```

В предыдущем листинге представлена программа, которая просто выводит список файлов в указанном каталоге и ничего больше. Назовем файл с исходным текстом `myls.c` и скомпилируем его в исполняемый файл с именем по умолчанию `a.out`:

```
cc myls.c
```

> Исторически команда cc(1) запускает компилятор языка C. В системах, где используется компилятор GNU C, исполняемый файл компилятора носит имя gcc(1). В этих системах cc часто является ссылкой на gcc.

Примерный результат работы нашей программы:

```
$ ./a.out /dev
.
..
console
tty
mem
kmem
null
mouse
stdin
stdout
stderr
zero
                    и еще много строк, которые мы опустили
cdrom
$ ./a.out /var/spool/cron
невозможно открыть /var/spool/cron: Permission denied
$ ./a.out /dev/tty
невозможно открыть /dev/tty: Not a directory
```

Далее в книге мы будем демонстрировать запуск программ и результаты их работы именно таким образом: символы, вводимые с клавиатуры, напечатаны **жирным моноширинным шрифтом**, а результат выполнения команды – обычным моноширинным шрифтом. Если нам необходимо добавить комментарий среди строк, выводимых на терминал, мы будем печатать его *курсивом*. Вводимым с клавиатуры символам предшествует символ доллара; это приглашение командного интерпретатора. Мы всегда будем отображать приглашение в виде символа доллара.

Обратите внимание, что полученный нами список файлов не отсортирован по алфавиту. Команда ls сортирует имена файлов перед выводом.

Рассмотрим эту программу из 20 строк поближе.

- В первой строке подключается наш собственный заголовочный файл apue.h. Мы будем использовать его почти во всех программах в этой книге. Этот заголовочный файл в свою очередь подключает некоторые стандартные заголовочные файлы и определяет множество констант и прототипов функций, которые будут встречаться в наших примерах. Листинг этого файла вы найдете в приложении B.
- Функция main объявлена в соответствии со стандартом ISO C. (Более подробно об этом стандарте рассказывается в следующей главе.)
- Из командной строки мы принимаем аргумент, argv[1], который трактуется как имя каталога, список файлов которого нужно вывести. В главе 7 мы увидим, как вызывается функция main и каким образом программа

получает доступ к аргументам командной строки и переменным окружения.

- Поскольку на практике формат записей в каталоге различен в разных системах, для получения необходимой информации мы использовали стандартные функции `opendir`, `readdir` и `closedir`.
- Функция `opendir` возвращает указатель на структуру `DIR`, который затем передается функции `readdir`. (Нас пока не интересует содержимое структуры `DIR`.) Затем в цикле вызывается функция `readdir`, которая считывает очередную запись. Функция `readdir` возвращает указатель на структуру `dirent` или пустой указатель, если все записи были прочитаны. Все, что нам сейчас нужно в структуре `dirent`, – это имя файла (`d_name`). Используя это имя, мы можем вызвать функцию `stat` (раздел 4.2), чтобы определить все атрибуты файла.
- Для обработки ошибочных ситуаций вызываются наши собственные функции: `err_sys` и `err_quit`. В предыдущем примере мы видели, что функция `err_sys` выводит сообщение, описывающее возникшую ошибку («Permission denied» (Доступ запрещен) или «Not a directory» (Не является каталогом)). Исходный код этих функций и их описание приводятся в приложении B. Кроме того, мы еще будем говорить об обработке ошибок в разделе 1.7.
- По завершении программы вызывается функция `exit` с аргументом 0. Функция `exit` завершает выполнение программы. В соответствии с принятыми соглашениями значение 0 означает нормальное завершение программы, а значения в диапазоне от 1 до 255 свидетельствуют о наличии ошибки. В разделе 8.5 мы покажем, как любая программа, в том числе и наша, может получить код завершения другой программы.

Рабочий каталог

У каждого процесса имеется свой *рабочий каталог*, который иногда называют *текущим рабочим каталогом*. Это каталог, от которого отсчитываются все относительные пути, используемые в программе. Процесс может изменить свой рабочий каталог с помощью функции `chdir`.

Например, относительный путь к файлу `doc/memo/joe` означает, что файл или каталог `joe` находится в каталоге `memo`, который находится в каталоге `doc`, который в свою очередь должен размещаться в рабочем каталоге. Встретив такой путь, мы можем быть уверены, что `doc` и `memo` – это каталоги, но не можем утверждать, является ли `joe` каталогом или же файлом. Путь `/usr/lib/lint` – это абсолютный путь к файлу или каталогу `lint` в каталоге `lib`, расположенном в каталоге `usr`, который в свою очередь находится в корневом каталоге.

Домашний каталог

Когда пользователь входит в систему, рабочим каталогом становится его *домашний каталог*. Домашний каталог пользователя определяется в соответствии с записью в файле паролей (раздел 1.3).

1.5. Ввод и вывод

Дескрипторы файлов

Дескрипторы файлов – это, как правило, небольшие целые положительные числа, используемые ядром для идентификации файлов, к которым обращается конкретный процесс. Всякий раз, когда процесс открывает существующий или создает новый файл, ядро возвращает его дескриптор, который затем используется для выполнения над файлом операций чтения или записи.

Стандартный ввод, стандартный вывод, стандартный вывод сообщений об ошибках

По принятым соглашениям все командные оболочки при запуске новой программы открывают для нее три файловых дескриптора: файл стандартного ввода, файл стандартного вывода и файл стандартного вывода сообщений об ошибках. За исключением особых случаев, все три дескриптора по умолчанию связаны с терминалом, как в простой команде

```
ls
```

Большинство командных оболочек предоставляют возможность перенаправления любого из этих дескрипторов в любой файл. Например

```
ls > file.list
```

выполнит команду `ls` и перенаправит стандартный вывод в файл с именем `file.list`.

Небуферизованный ввод-вывод

Небуферизованный ввод-вывод осуществляется функциями `open`, `read`, `write`, `lseek` и `close`. Все эти функции работают с файловыми дескрипторами.

Пример

Ниже приводится пример программы, которая читает данные со стандартного ввода и копирует их на стандартный вывод. С ее помощью можно выполнять копирование любых обычных файлов.

Листинг 1.2. Копирование со стандартного ввода на стандартный вывод

```
#include "apue.h"

#define BUFFSIZE 4096

int
main(void)
{
    int n;
    char buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
```

```
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("ошибка записи ");

    if (n < 0)
        err_sys("ошибка чтения");

    exit(0);
}
```

Заголовочный файл <unistd.h>, подключаемый из файла apue.h, и константы STDIN_FILENO и STDOUT_FILENO являются частью стандарта POSIX (о котором мы будем много говорить в следующей главе). В этом файле объявлены прототипы функций, обеспечивающих множество сервисов, предоставляемых системой UNIX, в том числе функций read и write, используемых в этом примере.

Константы STDIN_FILENO и STDOUT_FILENO, определенные в файле <unistd.h>, устанавливают дескрипторы файлов стандартного ввода и стандартного вывода. Обычные значения этих констант – соответственно 0 и 1, однако для обеспечения переносимости мы будем обращаться к этим дескрипторам по именам констант.

Константу BUFFSIZE более детально мы исследуем в разделе 3.9, где увидим, как различные значения могут влиять на производительность программы. Однако независимо от значения этой константы наша программа будет в состоянии выполнять копирование файла.

Функция read возвращает количество прочитанных байт. Это число затем передается функции write с целью указать, сколько байт нужно записать. По достижении конца файла функция read вернет значение 0 и программа завершит свою работу. Если в процессе чтения возникнет ошибка, read вернет значение –1. Большинство системных функций в случае ошибки возвращают –1.

Если скомпилировать программу в исполняемый файл с именем по умолчанию (a.out) и запустить ее следующим образом:

```
./a.out > data
```

то стандартный вывод будет перенаправлен в файл data, а стандартным вводом и стандартным выводом сообщений об ошибках будет терминал. Если выходной файл не существует, командная оболочка создаст его. Программа будет копировать строки, вводимые с клавиатуры, на стандартный вывод до тех пор, пока мы не введем символ «конец-файла» (обычно Control-D).

Если мы запустим программу таким образом:

```
./a.out < infile > outfile
```

то файл с именем infile будет скопирован в файл с именем outfile.

В главе 3 мы опишем функции небуферизованного ввода-вывода более подробно.

Стандартные функции ввода-вывода

Стандартные функции ввода-вывода предоставляют буферизованный интерфейс к функциям небуферизованного ввода-вывода. Использование стандартных функций ввода-вывода избавляет нас от необходимости задумываться о выборе оптимального размера буфера, например о значении константы BUFFSIZE в предыдущем примере. Другое преимущество стандартных функций ввода-вывода в том, что они значительно упрощают обработку пользовательского ввода (что на каждом шагу встречается в прикладных программах UNIX). Например, функция fgets читает из файла строку целиком, в то время как функция read считывает указанное количество байт. Как мы увидим в разделе 5.4, библиотека стандартного ввода-вывода предоставляет функции, с помощью которых можно управлять типом буферизации.

Наиболее типичным примером стандартных функций ввода-вывода является функция printf. В программах, которые обращаются к этой функции, обязательно должен быть подключен заголовочный файл <stdio.h> (в нашем случае через подключение файла apue.h), так как он содержит прототипы всех стандартных функций ввода-вывода.

Пример

Программа, представленная в листинге 1.3, (ее мы будем более детально исследовать в разделе 5.8), подобна программе из предыдущего примера, использующей функции read и write. Она также копирует данные, полученные со стандартного ввода, на стандартный вывод и может выполнять копирование обычных файлов.

Листинг 1.3. Копирование со стандартного ввода на стандартный вывод с использованием стандартных функций ввода-вывода

```
#include "apue.h"

int
main(void)
{
    int c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("ошибка вывода");

    if (ferror(stdin))
        err_sys("ошибка ввода");

    exit(0);
}
```

Функция getc считывает один символ, который затем записывается с помощью функции putc. Прочитав последний байт, getc вернет признак конца файла – значение константы EOF (определена в <stdio.h>). Константы stdin и stdout также определены в файле <stdio.h> и обозначают стандартный ввод и стандартный вывод.

1.6. Программы и процессы

Программа

Программа – это исполняемый файл, размещенный на диске. Программа считывается в память и затем выполняется ядром через вызов одной из шести функций семейства exec. Мы будем рассматривать эти функции в разделе 8.10.

Процессы и идентификаторы процессов

Программа, находящаяся в процессе исполнения, называется *процессом*. Этот термин будет встречаться практически на каждой странице этой книги. В некоторых операционных системах для обозначения выполняемой в данный момент программы используется термин *задача*.

UNIX обеспечивает присвоение каждому процессу уникального числового идентификатора, который называется *идентификатором процесса*. Идентификатор процесса – всегда целое неотрицательное число.

Пример

В листинге 1.4 представлена программа, которая выводит собственный идентификатор процесса.

Листинг 1.4. Вывод идентификатора процесса

```
#include "apue.h"

int
main(void)
{
    printf("привет от процесса с идентификатором %d\n", getpid());
    exit(0);
}
```

Если скомпилировать эту программу в файл a.out и запустить ее, мы получим примерно следующее:

```
$ ./a.out
привет от процесса с идентификатором 851
$ ./a.out
привет от процесса с идентификатором 854
```

Эта программа получает идентификатор своего процесса с помощью функции getpid.

Управление процессами

Три основные функции отвечают за управление процессами: fork, exec и waitpid. (Функция exec имеет шесть разновидностей, но мы обычно будем ссылаться на них просто как на функцию exec.)

Пример

Особенности управления процессами в UNIX мы продемонстрируем на примере простой программы (листинг 1.5), которая читает команды со стандартного ввода и выполняет их. Это похоже на примитивную реализацию командной оболочки. В программе есть несколько моментов, которые мы хотели бы рассмотреть поближе.

- Для чтения строки со стандартного ввода используется функция `fgets`. Когда первым в строке вводится символ конца файла (обычно `Control-D`), `fgets` возвращает пустой указатель, цикл прерывается и процесс завершает работу. В главе 18 мы опишем все символы, имеющие специальное назначение – признак конца файла, забой, удаление строки и тому подобное, и покажем, как можно их изменять.
- Поскольку каждая строка, возвращаемая функцией `fgets`, завершается символом перевода строки, за которым следует нулевой символ, мы будем определять ее длину с помощью стандартной функции `strlen` и заменять перевод строки нулевым символом. Это необходимо, потому что строка, принимаемая функцией `execlp` в качестве аргумента, должна заканчиваться нулевым символом, а не символом перевода строки.

Листинг 1.5. Чтение команд со стандартного ввода и их выполнение

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char   buf[MAXLINE];  /* из apue.h */
    pid_t  pid;
    int    status;

    printf("%% ");      /* вывести приглашение (printf использует */
                        /* последовательность %%, чтобы вывести символ %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* заменить символ перевода строки */

        if ((pid = fork()) < 0) {
            err_sys("ошибка вызова fork");
        } else if (pid == 0) {        /* дочерний процесс */
            execlp(buf, buf, (char *)0);
            err_ret("невозможно выполнить: %s", buf);
            exit(127);
        }

        /* родительский процесс */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("ошибка вызова waitpid");

        printf("%% ");
```

```
    }
    exit(0);
}
```

- Для создания нового процесса вызывается функция `fork`, которая создает копию вызывающего процесса. Мы называем вызывающий процесс родительским процессом, а вновь созданный – дочерним. В родительском процессе функция `fork` возвращает идентификатор дочернего процесса, в дочернем процессе – 0. Поскольку `fork` создает новый процесс, мы можем сказать, что она вызывается один раз – родительским процессом, а возвращает управление дважды – как родительскому процессу, так и дочернему.
- Для запуска команды, прочитанной со стандартного ввода, в дочернем процессе вызывается функция `execlp`. Она замещает дочерний процесс новой программой из файла. Комбинация функций `fork` и `exec` – это своего рода двухступенчатый системный вызов, порождающий новый процесс. В UNIX эти два этапа выделены в самостоятельные функции. Более подробно мы поговорим о них в главе 8.
- Поскольку дочерний процесс запускает новую программу с помощью `execlp`, родительский процесс должен дождаться его завершения, прежде чем продолжить работу. Делается это с помощью вызова функции `waitpid`, которой передается идентификатор дочернего процесса – аргумент `pid`. Функция `waitpid` возвращает код завершения дочернего процесса (аргумент `status`), но в нашей программе мы не используем это значение. Мы могли бы проверить его, чтобы узнать точно, как завершился дочерний процесс.
- Одно из основных ограничений нашей программы заключается в том, что мы не можем передать аргументы выполняемой команде. Так, например, невозможно указать имя каталога, список файлов которого мы хотим получить. Мы можем выполнить команду `ls` только для рабочего каталога. Чтобы передать аргументы, нам необходимо проанализировать введенную строку, в соответствии с некоторыми признаками выделить аргументы (например, по символам пробела или табуляции) и затем передать их в виде отдельных аргументов функции `execlp`. Тем не менее наша программа достаточно наглядно демонстрирует, как работают функции управления процессами.

Если мы запустим программу, то получим примерно следующие результаты. Обратите внимание: наша программа выводит символ `%` в качестве приглашения, чтобы как-то отличить его от приглашения командной оболочки.

```
$ ./a.out
% date
Sun Aug 1 03:04:47 EDT 2004          программисты работают допоздна
% who
sar :0 Jul 26 22:54
sar pts/0 Jul 26 22:54 (:0)
sar pts/1 Jul 26 22:54 (:0)
sar pts/2 Jul 26 22:54 (:0)
```

```
% pwd
/home/sar/bk/apue/2e
% ls
Makefile
a.out
shell1.c
% ^D                                  ввод символа конца файла
$                                    приглашение командной оболочки
```

Нотация ^D указывает на управляющий символ. Управляющие символы – это специальные символы, которые формируются при нажатой и удерживаемой клавише Control или Ctrl (в зависимости от модели клавиатуры) и одновременном нажатии на другую клавишу. Символ Control-D, или ^D, представляет признак конца файла. Мы встретим еще много управляющих символов, когда будем обсуждать терминальный ввод-вывод в главе 18.

Потоки и идентификаторы потоков

Обычно процесс работает в одном потоке управления – только одна последовательность машинных инструкций выполняется в одно и то же время. С многими проблемами легче справиться, если решать различные части задачи одновременно в нескольких потоках. Кроме того, на многопроцессорных системах различные потоки одного и того же процесса могут выполняться параллельно.

Все потоки в процессе разделяют одно и то же адресное пространство, файловые дескрипторы, стеки и прочие атрибуты процесса. Поскольку потоки могут обращаться к одной и той же области памяти, они должны синхронизировать доступ к разделяемым данным, чтобы избежать несогласованности.

Как и в случае с процессами, каждый поток имеет свой числовой идентификатор. Однако идентификаторы потоков являются локальными для процесса. Они служат для того, чтобы ссылаться на конкретные потоки внутри данного процесса, и не имеют никакого значения для других процессов.

Функции управления потоками отличны от функций управления процессами. Однако, поскольку потоки были добавлены в UNIX намного позже появления модели процессов, эти две модели находятся в достаточно сложной взаимосвязи, как мы увидим в главе 12.

1.7. Обработка ошибок

Очень часто при возникновении ошибки в любой из функций системы UNIX эта функция возвращает отрицательное число, а в глобальную переменную errno записывается некоторое целое, которое несет дополнительную информацию о возникшей ошибке. Например, функция open возвращает либо файловый дескриптор – неотрицательное число, либо –1 в случае возникновения ошибки. Вообще через переменную errno функция open может возвращать 15 различных кодов ошибок, таких как отсутствие файла, недостаточность прав доступа и тому подобное. Некоторые функции следуют иному соглаше-

нию. Например, большинство функций, которые должны возвращать указатель на какой-либо объект, в случае ошибки возвращают пустой указатель.

Определения переменной errno и констант всех возможных кодов ошибок находятся в заголовочном файле <errno.h>. Имена констант начинаются с символа E. Кроме того, на первой странице второго раздела справочного руководства UNIX, которая называется intro(2), обычно перечислены все константы кодов ошибок. Например, если переменная errno содержит код, равный значению константы EACCES, это означает, что возникли проблемы с правами доступа, например при открытии файла.

> В ОС Linux коды ошибок и соответствующие им имена констант перечислены на странице errno(3).

Стандарты POSIX и ISO C определяют errno как символ, раскрывающийся в изменяемое выражение lvalue (то есть выражение, которое может стоять слева от оператора присваивания) целого типа. Это может быть целое число, соответствующее коду ошибки, или функция, возвращающая указатель на код ошибки. Изначально переменная errno определялась как

```
extern int errno;
```

Но в многопоточной среде адресное пространство процесса разделяется между несколькими потоками и каждый поток должен обладать своей локальной копией errno, чтобы исключить возможность пересечения. ОС Linux, например, поддерживает многопоточный доступ к переменной errno, определяя ее следующим образом:

```
extern int *__errno_location(void);
#define errno (*__errno_location())
```

Необходимо знать два правила, касающиеся errno. Во-первых, значение errno никогда не очищается процедурой, если ошибка не происходит. Следовательно, проверять это значение надо лишь в тех случаях, когда значение, возвращаемое функцией, указывает на то, что произошла ошибка. Во-вторых, ни одна функция никогда не устанавливает значение errno в 0, и ни одна из констант, определенных в <errno.h>, не имеет значение 0.

Для вывода сообщений об ошибках стандарт C предусматривает две функции.

```
#include <string.h>

char *strerror(int errnum);
```

Возвращает указатель на строку сообщения

Эта функция преобразует код ошибки *errnum*, обычно равный значению errno, в строку сообщения об ошибке и возвращает указатель на нее.

Функция perror, основываясь на значении errno, выводит сообщение об ошибке на стандартный вывод сообщений об ошибках и возвращает управление.

```
#include <stdio.h>

void perror(const char *msg);
```

Она выводит строку сообщения *msg*, двоеточие, пробел и текст сообщения об ошибке, соответствующий значению errno. Вывод заканчивается символом перевода строки.

Пример

В листинге 1.6 приводится пример использования этих функций.

Листинг 1.6. Демонстрация функций strerror и perror

```
#include "apue.h"
#include <errno.h>

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}
```

Если эту программу скомпилировать в исполняемый файл a.out, мы получим

```
$ ./a.out
EACCES: Permission denied
./a.out: No such file or directory
```

Обратите внимание: мы передали функции perror имя исполняемого файла программы – a.out, которое находится в argv[0]. Это стандартное соглашение, принятое в UNIX. Если программа выполняется в составе конвейера, как показано ниже,

```
prog1 < inputfile | prog2 | prog3 > outputfile
```

то, следуя этому соглашению, мы сможем точно определить, в какой из программ произошла ошибка.

Во всех примерах этой книги вместо функций strerror или perror мы будем использовать собственные функции вывода сообщений об ошибках, исходный код которых находится в приложении B. Они принимают переменное количество аргументов, что позволяет легко обрабатывать ошибочные ситуации единственным выражением на языке C.

Восстановление после ошибок

Ошибки, определенные в <errno.h>, могут быть разделены на две категории – фатальные и нефатальные. Восстановление нормальной работы после фатальных ошибок невозможно. Самое лучшее, что мы можем сделать, – это

вывести сообщение об ошибке на экран или записать его в файл журнала и завершить работу приложения. Нефатальные ошибки допускают нормальное продолжение работы. Большинство нефатальных ошибок по своей природе носят временный характер (например, нехватка ресурсов), и их можно избежать при меньшей загруженности системы.

К нефатальным ошибкам, связанным с нехваткой ресурсов, относятся EAGAIN, ENFILE, ENOBUFS, ENOLCK, ENOSPC, ENOSR, EWOULDBLOCK и иногда ENOMEM. Если ошибка EBUSY указывает на то, что разделяемый ресурс в настоящий момент времени занят, она также может рассматриваться как нефатальная. Иногда нефатальной может считаться ошибка EINTR, если она возникает в результате прерывания медленно работающего системного вызова (подробнее об этом мы поговорим в разделе 10.5).

Для восстановления после вышеперечисленных ошибок, как правило, достаточно приостановить работу на короткое время и повторить попытку. Эта методика может применяться и в других ситуациях. Например, если ошибка свидетельствует о разрыве сетевого соединения, можно подождать некоторое время и затем попытаться восстановить соединение. В некоторых приложениях используется алгоритм экспоненциального увеличения времени задержки, когда пауза между попытками увеличивается при каждой итерации.

В конечном счете сам разработчик приложения решает, после каких ошибок возможно продолжение работы. Применяя разумную стратегию восстановления после ошибок, мы можем существенно повысить отказоустойчивость приложения и избежать аварийного завершения его работы.

1.8. Идентификация пользователя

Идентификатор пользователя

Идентификатор пользователя из записи в файле паролей представляет собой числовое значение, которое однозначно идентифицирует пользователя в системе. Идентификатор пользователя назначается системным администратором при создании учетной записи и не может быть изменен пользователем. Как правило, каждому пользователю назначается уникальный идентификатор. Ниже мы узнаем, как ядро использует идентификатор пользователя для проверки прав на выполнение определенных операций.

Пользователь с идентификатором 0 называется *суперпользователем*, или *root*. В файле паролей этому пользователю обычно присвоено имя root. Обращаем ваше внимание на то, что этот пользователь обладает особыми привилегиями суперпользователя. Как мы увидим в главе 4, если процесс имеет привилегии суперпользователя, большинство проверок прав доступа к файлам просто не выполняется. Некоторые системные операции доступны только суперпользователю. Суперпользователь обладает неограниченной свободой действий в системе.

> В клиентских версиях Mac OS X учетная запись суперпользователя заблокирована, в серверных версиях – разблокирована. Инструкции по разблокированию учетной за-

писи суперпользователя вы найдете на веб-сайте компании Apple: *http://docs.info.apple.com/article.html?artnum=106290*.

Идентификатор группы

Кроме всего прочего, запись в файле паролей содержит числовой *идентификатор группы*. Он также назначается системным администратором при создании учетной записи. Как правило, в файле паролей имеется несколько записей с одинаковым идентификатором группы. Обычно группы используются для распределения пользователей по проектам или отделам. Это позволяет организовать совместное использование ресурсов, например файлов, членами определенной группы. В разделе 4.5 мы увидим, как назначить файлу такие права доступа, чтобы он был доступен всем членам группы и недоступен другим пользователям.

В системе существует файл групп, в котором указаны соответствия имен групп их числовым идентификаторам. Обычно этот файл называется `/etc/group`.

Представление идентификаторов пользователя и группы в числовом виде сложилось исторически. Для каждого файла на диске файловая система хранит идентификаторы пользователя и группы его владельца. Поскольку каждый идентификатор представлен двухбайтным целым числом, для хранения обоих идентификаторов требуется всего четыре байта. Если бы вместо идентификаторов использовались полные имена пользователей и групп, потребовалось бы хранить на диске значительно больший объем информации. Кроме того, сравнение строк вместо сравнения целых чисел при выполнении проверок прав доступа выполнялось бы гораздо медленнее.

Однако человеку удобнее работать с осмысленными именами, чем с числовыми идентификаторами, поэтому файл паролей хранит соответствия между именами и идентификаторами пользователей, а файл групп – между именами и идентификаторами групп. Например, команда `ls -l` выводит имена владельцев файлов, используя файл паролей для преобразования числовых идентификаторов в соответствующие им имена пользователей.

> В ранних версиях UNIX для представления идентификаторов использовались 16-битные числа, в современных версиях – 32-битные.

Пример

Программа, представленная листингом 1.7, выводит идентификаторы пользователя и группы.

Листинг 1.7. Вывод идентификаторов пользователя и группы

```
#include "apue.h"

int
main(void)
{
    printf("uid = %d, gid = %d\n", getuid(), getgid());
    exit(0);
}
```

Для получения идентификаторов пользователя и группы используются функции getuid и getgid. Запуск программы дает следующие результаты:

```
$ ./a.out
uid = 205, gid = 105
```

Идентификаторы дополнительных групп

В дополнение к группе, идентификатор которой указан в файле паролей, большинство версий UNIX позволяют пользователю быть членом других групп. Впервые такая возможность появилась в 4.2BSD, где можно было определить до 16 дополнительных групп, к которым мог принадлежать пользователь. Во время входа в систему из файла /etc/group извлекаются первые 16 групп, в которых присутствует имя данного пользователя, и их идентификаторы назначаются *идентификаторами дополнительных групп*. Как мы увидим в следующей главе, стандарт POSIX требует, чтобы операционная система поддерживала как минимум восемь дополнительных групп для одного процесса, однако большинство систем поддерживает не менее 16 таких групп.

1.9. Сигналы

Сигналы используются, чтобы известить процесс о наступлении некоторого состояния. Например, если процесс попытается выполнить деление на ноль, он получит уведомление в виде сигнала SIGFPE (floating-point exception – ошибка выполнения операции с плавающей точкой). Процесс может реагировать на сигнал тремя способами.

1. Игнорировать сигнал. Такая реакция не рекомендуется для сигналов, которые указывают на аппаратную ошибку (такую как деление на ноль или обращение к памяти, находящейся вне адресного пространства процесса), поскольку результат в этом случае непредсказуем.
2. Разрешить выполнение действия по умолчанию. В случае деления на ноль по умолчанию происходит аварийное завершение процесса.
3. Определить функцию, которая будет вызвана для обработки сигнала (такие функции называют *перехватчиками* сигналов). Определив свою собственную функцию, мы сможем отслеживать получение сигнала и реагировать на него по своему усмотрению.

Сигналы порождаются во многих ситуациях. Две клавиши терминала, известные как *клавиша прерывания* (Control-C или DELETE) и *клавиша выхода* (часто Control-\), используются для прерывания работы текущего процесса. Другой способ генерации сигнала – вызвать функцию kill. С помощью этой функции один процесс может послать сигнал другому процессу. Естественно, эта ситуация имеет свои ограничения: чтобы послать сигнал процессу, мы должны быть его владельцем (или суперпользователем).

Пример

Вспомните пример простейшей командной оболочки из листинга 1.5. Если запустить эту программу и нажать клавишу прерывания (Control-C), процесс завершит работу, поскольку реакция по умолчанию на этот сигнал, называемый SIGINT, заключается в завершении процесса. Процесс не сообщил ядру о том, что реакция на сигнал должна отличаться от действия по умолчанию, поэтому он завершается.

Чтобы перехватить этот сигнал, программа должна вызвать функцию signal, передав ей имя функции, которая должна быть вызвана при получении сигнала SIGINT. В следующем примере эта функция называется sig_int. Она просто выводит на экран сообщение и новое приглашение к вводу команды. Добавив 11 строк в программу из листинга 1.5, мы получим версию, представленную листингом 1.8 (добавленные строки обозначены символами «+»).

Листинг 1.8. Чтение команд со стандартного ввода и их выполнение

```
    #include "apue.h"
    #include <sys/wait.h>

+ static void sig_int(int); /* наша функция-перехватчик */
+
    int
    main(void)
    {
        char buf[MAXLINE];    /* из apue.h */
        pid_t pid;
        int status;

+       if (signal(SIGINT, sig_int) == SIG_ERR)
+           err_sys("ошибка вызова signal");
+
        printf("%% ");        /* вывести приглашение (printf использует */
                              /* последовательность %%, */
                              /* чтобы вывести символ %) */

        while (fgets(buf, MAXLINE, stdin) != NULL) {
            if (buf[strlen(buf) - 1] == '\n')
                buf[strlen(buf) - 1] = 0;  /* заменить символ перевода строки */

            if ((pid = fork()) < 0) {
                err_sys("ошибка вызова fork");
            } else if (pid == 0) {         /* дочерний процесс */
                execlp(buf, buf, (char *)0);
                err_ret("невозможно выполнить: %s", buf);
                exit(127);
            }

            /* родительский процесс */
            if ((pid = waitpid(pid, &status, 0)) < 0)
                err_sys("ошибка вызова waitpid");
                printf("%% ");
        }
```

```
        exit(0);
    }
+
+   void
+   sig_int(int signo)
+   {
+       printf("прервано\n%% ");
+   }
```

В главе 10 мы будем подробно рассказывать о сигналах, поскольку с ними работает большинство серьезных приложений.

1.10. Представление времени

Исторически в системе UNIX поддерживается два различных способа представления временных интервалов.

1. Календарное время. Значения в этом представлении хранят число секунд, прошедших с начала Эпохи: 00:00:00 1 января 1970 года по согласованному всемирному времени (Coordinated Universal Time – UTC). (Старые руководства описывают UTC как Greenwich Mean Time – время по Гринвичу.) Эти значения используются, например, для записи времени последней модификации файла.
2. Время работы процесса. Оно еще называется процессорным временем и измеряет ресурсы центрального процессора, использованные процессом. Значения в этом представлении измеряются в тактах (ticks). Исторически сложилось так, что в различных системах в одной секунде может быть 50, 60 или 100 тактов. Для хранения времени в этом представлении используется тип данных `clock_t`. (В разделе 2.5.4 мы покажем, как узнать количество тактов в секунде при помощи функции `sysconf`.)

В разделе 3.9 мы увидим, что при измерении времени выполнения процесса система UNIX хранит три значения для каждого процесса:

- Общее время (Clock time)
- Пользовательское время (User CPU time)
- Системное время (System CPU time)

Общее время, иногда его называют *временем настенных часов*, – это отрезок времени, затраченный процессом от момента запуска до завершения. Это значение зависит от общего количества процессов, выполняемых в системе. Всякий раз, когда нас интересует общее время, измерения должны делаться на незагруженной системе.

Пользовательское время – это время, затраченное на исполнение машинных инструкций самой программы. Системное время – это время, затраченное на выполнение ядром машинных инструкций от имени процесса. Например, всякий раз, когда процесс обращается к системному вызову, такому как `read` или `write`, время, затраченное ядром на выполнение запроса, приписывается процессу. Сумму пользовательского и системного времени часто называют *процессорным временем*.

Измерить общее, пользовательское и системное время весьма просто: запустите (выполните) команду time(1), передав ей в качестве аргумента команду, время работы которой мы хотим измерить. Например:

```
$ cd /usr/include
$ time -p grep _POSIX_SOURCE */*.h > /dev/null

real     0m0.81s
user     0m0.11s
sys      0m0.07s
```

Формат вывода результатов зависит от командной оболочки, поскольку некоторые из них вместо утилиты /usr/bin/time используют встроенную функцию, измеряющую время выполнения заданной команды.

В разделе 8.16 мы увидим, как можно получить все три значения из запущенного процесса. Собственно тема даты и времени будет рассматриваться в разделе 6.10.

1.11. Системные вызовы и библиотечные функции

Любая операционная система обеспечивает прикладным программам возможность обращения к системным службам. Во всех реализациях UNIX имеется строго определенное число точек входа в ядро, которые называются системными вызовами (вспомните рисунок 1.1). Седьмая версия Research UNIX System предоставляла около 50 системных вызовов, 4.4BSD – около 110, а SVR4 – примерно 120. В ОС Linux имеется от 240 до 260 системных вызовов в зависимости от версии. В ОС FreeBSD около 320 системных вызовов.

Интерфейс системных вызовов всегда документируется во втором разделе «Руководства программиста UNIX». Он определяется на языке C независимо от конкретных реализаций, использующих системные вызовы в той или иной системе. В этом отличие от многих более старых систем, которые традиционно определяли точки входа в ядро на языке ассемблера.

В системе UNIX для каждого системного вызова предусматривается одноименная функция в стандартной библиотеке языка C. Пользовательский процесс вызывает эту функцию стандартными средствами языка C. Затем эта функция вызывает соответствующую службу ядра, используя технику обращения, принятую в данной системе. Например, функция может разместить один или более своих аргументов в регистрах общего назначения и затем выполнить некоторую машинную инструкцию, которая генерирует программное прерывание. В нашем случае мы можем рассматривать системные вызовы как обычные функции языка C.

Раздел 3 «Руководства программиста UNIX» описывает функции общего назначения, доступные программисту. Эти функции не являются точками входа в ядро, хотя они могут обращаться к нему посредством системных вызовов. Например, функция printf может использовать системный вызов write для вывода строки, но функции strcpy (копирование строки) и atoi (преобразование ASCII-строки в число) не производят ни одного системного вызова.

С точки зрения разработчика системы между системным вызовом и библиотечной функцией имеются коренные различия. Но с точки зрения пользователя эти различия носят непринципиальный характер. В контексте нашей книги и системные вызовы, и библиотечные функции можно представлять как обычные функции языка С. И те, и другие предназначены для обслуживания прикладных программ. Однако при этом мы должны понимать, что можем заменить библиотечные функции, если в этом возникнет необходимость, а вот системные запросы – нет.

Рассмотрим в качестве примера функцию выделения памяти `malloc`. Существует масса способов распределения памяти и алгоритмов «сборки мусора» (метод наилучшего приближения, метод первого подходящего и т. д.). Но нет единой методики, оптимальной абсолютно для всех возможных ситуаций. Системный вызов `sbrk`(2), который занимается выделением памяти, не является менеджером памяти общего назначения. Он лишь увеличивает или уменьшает объем адресного пространства процесса на заданное количество байт, а управление этим пространством возлагается на сам процесс. Функция `malloc`(3) реализует одну конкретную модель распределения памяти. Если она нам не нравится по тем или иным причинам, мы можем написать собственную функцию `malloc`, которая, вероятно, будет обращаться к системному вызову `sbrk`. На самом деле многие программные пакеты реализуют свои собственные алгоритмы распределения памяти с использованием системного вызова `sbrk`. На рис. 1.2 показаны взаимоотношения между приложением, функцией `malloc` и системным вызовом `sbrk`.

Здесь мы видим четкое разделение обязанностей: системный вызов выделяет дополнительную область памяти от имени процесса, а библиотечная функция `malloc` распоряжается этой областью.

Еще один пример, иллюстрирующий различия между системным вызовом и библиотечной функцией, – интерфейс, предоставляемый системой UNIX для определения текущей даты и времени. В некоторых операционных сис-

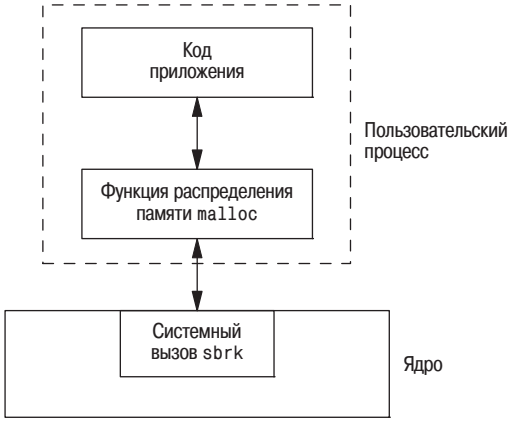


***Рис. 1.2.** Разделение обязанностей функции malloc и системного вызова sbrk*

темах имеется два системных вызова: один возвращает время, другой – дату. Любая специальная обработка, такая как переход на летнее время, выполняется ядром или требует вмешательства человека. UNIX предоставляет единственный системный вызов, который возвращает количество секунд, прошедших с начала Эпохи – 0 часов 00 минут 1 января 1970 года по согласованному всемирному времени (UTC). Любая интерпретация этого значения, например представление в удобном для человека виде с учетом поясного времени, полностью возлагается на пользовательский процесс. Стандартная библиотека языка C содержит функции практически для любых случаев. Они, например, реализуют различные алгоритмы, учитывающие переход на зимнее или летнее время.

Прикладная программа может обращаться как к системному вызову, так и к библиотечной функции. Кроме того, следует помнить, что библиотечные функции в свою очередь также могут обращаться к системным вызовам. Это наглядно продемонстрировано на рис. 1.3.

Другое отличие системных вызовов от библиотечных функций заключается в том, что системные вызовы обеспечивают лишь минимально необходимую функциональность, тогда как библиотечные функции часто предоставляют более широкие возможности. Мы уже видели это различие на примере сравнения системного вызова `sbrk` с библиотечной функцией `malloc`. Мы еще столкнемся с этим различием, когда будем сравнивать функции небуферизованного ввода-вывода (глава 3) и стандартные функции ввода-вывода (глава 5).

Системные вызовы управления процессами (`fork`, `exec` и `wait`) обычно вызываются пользовательским процессом напрямую. (Вспомните простую командную оболочку из листинга 1.5.) Но существуют и такие библиотечные функции, которые служат для упрощения самых распространенных случаев: например, функции `system` и `popen`. В разделе 8.3 мы продемонстрируем реализацию функции `system`, выполненную на основе системных вызовов управления процессами. В разделе 10.18 мы дополним этот пример обработкой сигналов.

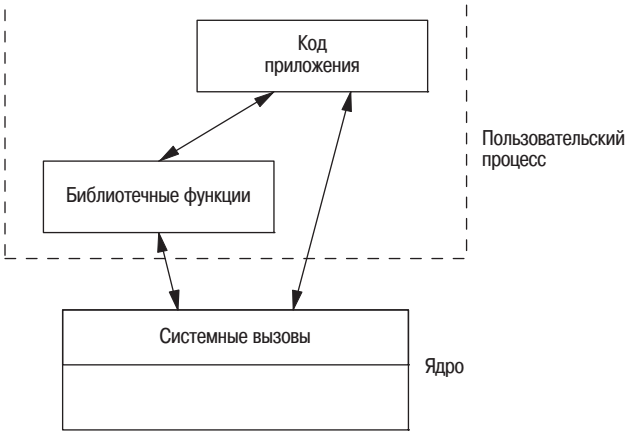


Рис. 1.3. *Различия между библиотечными функциями C и системными вызовами*

Чтобы охарактеризовать интерфейс системы UNIX, используемый большинством программистов, мы должны будем описать как системные вызовы, так и некоторые библиотечные функции. Описав, к примеру, только системный вызов `sbrk`, мы оставили бы без внимания более удобную для программиста функцию `malloc`, которая применяется во множестве приложений. В этой книге под термином *функция* мы будем подразумевать и системные вызовы, и библиотечные функции, за исключением тех случаев, когда необходимо будет подчеркнуть имеющиеся отличия.

1.12. Подведение итогов

Эта глава представляет собой обзорную экскурсию по системе UNIX. Мы дали определение некоторых фундаментальных понятий, с которыми столкнемся еще не раз, и привели многочисленные примеры небольших программ, чтобы вы могли представить себе, о чем пойдет речь в этой книге.

Следующая глава рассказывает о стандартизации UNIX и о влиянии деятельности в этой области на ее развитие. Стандарты, особенно ISO C и POSIX.1, будут постоянно встречаться нам на протяжении всей книги.

Упражнения

1.1. На своей системе проверьте и убедитесь, что каталоги «.» и «..» являются различными каталогами, за исключением корневой директории.

1.2. Просмотрите еще раз результат работы примера, представленного листингом 1.4, и объясните, куда пропали процессы с идентификаторами 852 и 853.

1.3. Входной аргумент функции `perror` в разделе 1.7 определен с атрибутом `const` (в соответствии со стандартом ISO C), в то время как целочисленный аргумент функции `strerror` определен без этого атрибута. Почему?

1.4. В функции обработки ошибок `err_sys` (приложение B) значение `errno` сохраняется в локальной переменной. Почему?

1.5. Если предположить, что календарное время хранится в виде 32-битного целого числа со знаком, то в каком году наступит переполнение? Какими способами можно отдалить дату переполнения? Будут ли найденные решения совместимы с существующими приложениями?

1.6. Предположим, что время работы процесса хранится в виде 32-битного целого числа со знаком и система отсчитывает 100 тактов в секунду. Через сколько дней наступит переполнение счетчика?

2

Стандарты и реализации UNIX

2.1. Введение

Немалая работа была проделана для стандартизации системы UNIX и языка программирования C. Хотя приложения всегда обладали высокой переносимостью между разными версиями UNIX, тем не менее появление многочисленных версий UNIX в течение 80-х годов привело к тому, что крупные пользователи, такие как правительство США, были вынуждены призвать разработчиков к выработке стандартов.

В этой главе мы сначала рассмотрим различные попытки стандартизации, которые предпринимались за последние два десятилетия, а затем обсудим их влияние на те реализации системы UNIX, которые обсуждаются в данной книге. Важной частью любых работ по стандартизации является спецификация различных ограничений, которые должны быть установлены для каждой реализации, поэтому мы рассмотрим эти ограничения и научимся определять их значения.

2.2. Стандартизация UNIX

2.2.1. ISO C

В конце 1989 года был одобрен стандарт ANSI для языка программирования C – X3.159-1989. Этот стандарт также был принят как международный стандарт ISO/IEC 9899:1990. Аббревиатура ANSI расшифровывается как American National Standards Institute (Американский национальный институт стандартов, представляющий США в Международной организации по стандартизации – International Organization for Standardization, ISO). Аббревиатура IEC означает International Electrotechnical Commission (Международная электротехническая комиссия).

Стандарт языка C теперь поддерживается и развивается международной рабочей группой ISO/IEC по стандартизации языка программирования C, из-

вестной как ISO/IEC JTC1/SC22/WG14, или сокращенно WG14. Назначение стандарта ISO C состоит в том, чтобы обеспечить переносимость программ, написанных на языке C, на самые различные операционные системы, не только UNIX. Этот стандарт определяет синтаксис и семантику языка, а также состав стандартной библиотеки [ISO 1999, глава 7; Plauger 1992; Kernighan and Ritchie 1988, приложение B]. Эта библиотека имеет большое значение, потому что все современные версии UNIX, в том числе описанные в этой книге, обязаны предоставлять библиотеки функций, определенные стандартом языка C.

В 1999 году стандарт ISO C обновлен и одобрен как ISO/IEC 9899:1999. Он в значительной степени улучшил поддержку приложений, выполняющих числовую обработку. Изменения не затронули стандарты POSIX, описываемые в этой книге, за исключением добавления ключевого слова `restrict` к некоторым прототипам функций. Это ключевое слово сообщает компилятору, какие ссылки по указателю можно оптимизировать, отмечая объекты, доступ к которым осуществляется из функций только посредством данного указателя.

В большинстве случаев между одобрением стандарта и модификацией программного обеспечения, учитывающей изменения в стандартах, проходит какое-то время. По мере своего развития все системы компиляции добавляют или совершенствуют поддержку последней версии стандарта ISO C.

> Информацию о текущем уровне соответствия gcc стандарту ISO C от 1999 года вы найдете по адресу: *http://www.gnu.org/software/gcc/c99status.html*.

Библиотеку ISO C можно разбить на 24 раздела, основываясь на именах заголовочных файлов, определяемых стандартом. В табл. 2.1 приводится перечень заголовочных файлов, определяемых стандартом языка C. Стандарт POSIX.1 включает эти файлы и, кроме того, определяет ряд дополнительных заголовочных файлов. Впоследствии мы перечислим те из них, которые поддерживаются четырьмя реализациями (FreeBSD 5.2.1, Linux 2.4.22, MAC X 10.3 и Solaris 9).

Таблица 2.1. Перечень заголовочных файлов, определяемых стандартом ISO C

Заголовочный файл	FreeBSD 5.2.1	Linux 2.4.22	MAC OS X 10.3	Solaris 9	Описание
<assert.h>	•	•	•	•	Проверка программных утверждений
<complex.h>	•	•	•		Поддержка арифметики комплексных чисел
<ctype.h>	•	•	•	•	Типы символов
<errno.h>	•	•	•	•	Коды ошибок (раздел 1.7)
<fenv.h>		•	•		Окружение операций с плавающей точкой
<float.h>	•	•	•	•	Арифметика с плавающей точкой
<inttypes.h>	•	•	•	•	Преобразования целочисленных типов

Заголовочный файл	FreeBSD 5.2.1	Linux 2.4.22	MAC OS X 10.3	Solaris 9	Описание
<iso646.h>	•	•	•	•	Альтернативные макросы операторов отношений
<limits.h>	•	•	•	•	Константы реализации (раздел 2.5)
<locale.h>	•	•	•	•	Классы региональных настроек (локалей)
<math.h>	•	•	•	•	Математические константы
<setjmp.h>	•	•	•	•	Нелокальные переходы (раздел 7.10)
<signal.h>	•	•	•	•	Сигналы (глава 10)
<stdarg.h>	•	•	•	•	Списки аргументов переменной длины
<stdbool.h>	•	•	•	•	Логический тип и значения
<stddef.h>	•	•	•	•	Стандартные определения
<stdint.h>	•	•	•		Целочисленные типы
<stdio.h>	•	•	•	•	Стандартная библиотека ввода-вывода (глава 5)
<stdlib.h>	•	•	•	•	Функции общего назначения
<string.h>	•	•	•	•	Операции над строками
<tgmath.h>		•			Макроопределения математических операций
<time.h>	•	•	•	•	Время и дата (раздел 6.10)
<wchar.h>	•	•	•	•	Расширенная поддержка многобайтных символов
<wctype.h>	•	•	•	•	Классификация и функции преобразования многобайтных символов

Перечень заголовочных файлов ISO C зависит от версии компилятора языка C, используемой в той или иной операционной системе. Изучая табл. 2.1, имейте в виду, что FreeBSD 5.2.1 распространяется с `gcc` версии 3.3.3, Solaris 9 – с двумя версиями `gcc`, 2.95.3 и 3.2, Mandrake 9.2 (Linux 2.4.22) – с `gcc` версии 3.3.1, а MAC OS X 10.3 – с `gcc` версии 3.3. MAC OS X также включает более старые версии `gcc`.

2.2.2. IEEE POSIX

POSIX – это семейство стандартов, разработанных организацией IEEE (Institute of Electrical and Electronics Engineers – Институт инженеров электроники и радиотехники). Аббревиатура POSIX расшифровывается как Portable Operating System Interface (Интерфейс переносимой операционной системы). Изначально это название относилось только к стандарту IEEE 1003.1-1988 (интерфейс операционной системы), но позднее оно стало объединять множество других стандартов и предварительных стандартов проекта под номером 1003, в том числе командную оболочку и утилиты (1003.2).

Основной интерес для нас будет представлять стандарт на интерфейс переносимой операционной системы 1003.1, цель которого состоит в повышении переносимости приложений между различными версиями UNIX. Этот стандарт определяет набор услуг, которые должна предоставлять операционная система, если она претендует на звание «POSIX-совместимой». Хотя стандарт 1003.1 и базируется на операционной системе UNIX, тем не менее он не ограничивается UNIX и UNIX-подобными операционными системами. Действительно, некоторые производители проприетарных операционных систем утверждают, что их системы являются POSIX-совместимыми, в то же время сохраняя все свои проприетарные особенности.

Поскольку стандарт 1003.1 определяет интерфейс, а не реализацию, между системными вызовами и библиотечными функциями не делается никаких различий. Стандарт именует все процедуры *функциями*.

Стандарты продолжают непрерывно развиваться, и 1003.1 не является исключением. Версия этого стандарта от 1988 года, IEEE Standard 1003.1-1988, была дополнена и представлена на рассмотрение Международной организации по стандартизации (ISO). Текст стандарта был полностью переработан, хотя при этом не было добавлено каких-либо новых интерфейсов или особенностей. Окончательный документ был опубликован как IEEE Std 1003.1-1990 [IEEE 1990]. Он также является международным стандартом ISO/IEC 9945-1:1990. Обычно этот стандарт называют *POSIX.1*, и в этой книге также используется это обозначение.

Рабочая группа IEEE 1003.1 продолжала вносить изменения в стандарт. В 1993 году была издана пересмотренная версия стандарта IEEE 1003.1. Она включала в себя стандарт 1003.1-1990 и стандарт на расширения реального времени 1003.1b-1993. В 1996 году стандарт снова был дополнен и опубликован как ISO/IEC 9945-1:1996. В нем появились интерфейсы многопоточного программирования, известные как *pthreads* (от «POSIX threads», потоки стандарта POSIX). В 1999 году с выходом стандарта IEEE Standard 1003.1d-1999 были добавлены улучшенные интерфейсы реального времени. Год спустя был опубликован стандарт IEEE Standard 1003.1j–2000», в котором появились дополнительные улучшенные интерфейсы реального времени. В этом же году вышел стандарт IEEE Standard 1003.1q–2000, добавивший расширения трассировки событий.

Версия стандарта 1003.1 от 2001 года отличалась от предшествующих версий тем, что она объединила в себе некоторые поправки из стандартов 1003.1, 1003.2 и часть Single UNIX Specification (SUS – единая спецификация UNIX) версии 2 (подробнее об этом стандарте см. в следующем разделе). В окончательный вариант IEEE Standard 1003.1–2001 вошли следующие стандарты:

- ISO/IEC 9945-1 (IEEE Standard 1003.1-1996), который включает в себя
  - IEEE Standard 1003.1-1990
  - IEEE Standard 1003.1b-1993 (расширения реального времени)
  - IEEE Standard 1003.1c-1995 (pthreads)
  - IEEE Standard 1003.1i-1995 (список технических опечаток)

- IEEE P1003.1a предварительный стандарт (пересмотр системных интерфейсов)
- IEEE Standard 1003.1d-1999 (улучшенные расширения реального времени)
- IEEE Standard 1003.1j–2000 (дополнительные улучшенные расширения реального времени)
- IEEE Standard 1003.1q–2000 (трассировка)
- IEEE Standard 1003.2d-1994 (пакетные расширения)
- IEEE P1003.2b предварительный стандарт (дополнительные утилиты)
- Части стандарта IEEE Standard 1003.1g–2000 (независимые от протокола интерфейсы)
- ISO/IEC 9945-2 (IEEE Standard 1003.2-1993)
- Основные спецификации Single UNIX Specification версии 2, которые включают
  - System Interface Definitions, Issue 5 (определения системных интерфейсов, выпуск 5)
  - Commands and Utilities, Issue 5 (команды и утилиты, выпуск 5)
  - System Interfaces and Headers, Issue 5 (системные интерфейсы и заголовочные файлы, выпуск 5)
- Open Group Technical Standard, Networking Services, Issue 5.2 (технический стандарт на сетевые службы, выпуск 5.2)
- ISO/IEC 9899:1999, Programming Languages – C (языки программирования – C)

В табл. 2.2, 2.3 и 2.4 приводятся списки обязательных и дополнительных заголовочных файлов, предусматриваемых стандартом POSIX.1. Поскольку POSIX.1 включает стандартные библиотечные функции ISO C, то он также требует наличия заголовочных файлов, перечисленных в табл. 2.1. Все четыре таблицы представляют собой перечень заголовочных файлов, которые включены в обсуждаемые здесь реализации операционных систем.

Таблица 2.2. Перечень обязательных заголовочных файлов, определяемых стандартом POSIX

Заголовочный файл	FreeBSD 5.2.1	Linux 2.4.22	MAC OS X 10.3	Solaris 9	Описание
<dirent.h>	•	•	•	•	Работа с каталогами (раздел 4.21)
<fcntl.h>	•	•	•	•	Управление файлами (раздел 3.14)
<fnmatch.h>	•	•	•	•	Шаблоны имен файлов
<glob.h>	•	•	•	•	Шаблоны путей файловой системы
<grp.h>	•	•	•		Файл групп (раздел 6.4)
<netdb.h>	•	•	•	•	Операции с распределенной базой системных данных
<pwd.h>	•	•	•	•	Файл паролей (раздел 6.2)

Таблица 2.2 (продолжение)

Заголовочный файл	FreeBSD 5.2.1	Linux 2.4.22	MAC OS X 10.3	Solaris 9	Описание
<regex.h>	•	•	•	•	Регулярные выражения
<tar.h>	•	•	•	•	Архиватор tar
<termios.h>	•	•	•	•	Терминальный ввод-вывод (глава 18)
<unistd.h>	•	•	•	•	Символьные константы
<utime.h>	•	•	•	•	Время последнего доступа и последней модификации файла (раздел 4.19)
<wordexp.h>	•	•		•	Дополнение слов по шаблону
<arpa/inet.h>	•	•	•	•	Сеть Интернет (глава 16)
<net/if.h>	•	•	•	•	Локальные сетевые интерфейсы (глава 16)
<netinet/in.h>	•	•	•	•	Семейство адресов Интернет (раздел 16.3)
<netinet/tcp.h>	•	•	•	•	Определения протокола TCP
<sys/mman.h>	•	•	•	•	Управление памятью
<sys/select.h>	•	•	•	•	Функция select (раздел 14.5.1)
<sys/socket.h>	•	•	•	•	Интерфейс сокетов (глава 16)
<sys/stat.h>	•	•	•	•	Получение сведений о файлах (глава 4)
<sys/times.h>	•	•	•	•	Время работы процесса (раздел 8.16)
<sys/types.h>	•	•	•	•	Примитивы системных типов данных (раздел 2.8)
<sys/un.h>	•	•	•	•	Определения сокетов домена UNIX (раздел 17.3)
<sys/utsname.h>	•	•	•	•	Название системы (раздел 6.9)
<sys/wait.h>	•	•	•	•	Управление процессами (раздел 8.6)

Таблица 2.3. Заголовочные файлы расширений XSI, определяемые стандартом POSIX

Заголовочный файл	FreeBSD 5.2.1	Linux 2.4.22	MAC OS X 10.3	Solaris 9	Описание
<cpio.h>	•	•		•	Архиватор cpio
<dlfcn.h>	•	•	•	•	Динамическое связывание
<fmtmsg.h>	•	•		•	Вывод сообщений в форматированном виде
<ftw.h>		•		•	Обход дерева файлов (раздел 4.21)

Заголовочный файл	FreeBSD 5.2.1	Linux 2.4.22	MAC OS X 10.3	Solaris 9	Описание
<iconv.h>		•	•	•	Преобразование кодировок
<langinfo.h>	•	•	•	•	Константы сведений о языках
<libgen.h>	•	•	•	•	Определения для функций поиска по шаблону
<monetary.h>	•	•	•	•	Денежные типы
<ndbm.h>	•		•	•	Операции с базой данных (стандарта dbm)
<nl_types.h>	•	•	•	•	Каталоги с сообщениями
<poll.h>	•	•	•	•	Функция poll (раздел 14.5.2)
<search.h>	•	•	•	•	Поиск по таблицам
<strings.h>	•	•	•	•	Операции над строками
<syslog.h>	•	•	•	•	Системное журналирование (раздел 13.4)
<ucontext.h>	•	•	•	•	Пользовательский контекст
<ulimit.h>	•	•	•	•	Ограничения пользователей
<utmpx.h>		•		•	Работа с учетными записями пользователей
<sys/ipc.h>	•	•	•	•	IPC (раздел 15.6)
<sys/msg.h>	•	•		•	Очереди сообщений (раздел 15.7)
<sys/resource.h>	•	•	•	•	Операции над ресурсами (раздел 7.11)
<sys/sem.h>	•	•	•	•	Семафоры (раздел 15.8)
<sys/shm.h>	•	•	•	•	Разделяемая память (раздел 15.9)
<sys/statvfs.h>	•	•		•	Получение сведений о файловой системе
<sys/time.h>	•	•	•	•	Типы данных для представления времени
<sys/timeb.h>	•	•	•	•	Дополнительные определения даты и времени
<sys/uio.h>	•	•	•	•	Векторные операции ввода-вывода (раздел 14.7)

В этой книге мы описываем версию стандарта POSIX.1 от 2001 года, которая включает в себя функции, определенные стандартом ISO C. Его интерфейсы подразделяются на обязательные для реализации и дополнительные. Кроме того, дополнительные интерфейсы по своей функциональности подразделяются на 50 категорий. Категории еще не устаревших интерфейсов программирования приведены в табл. 2.5 с соответствующими им кодами. Коды –

это двух- или трехсимвольные сокращения, которые помогают идентифицировать функциональную область интерфейса. С помощью этих кодов в тексте справочного руководства отмечаются места, где описываемые интерфейсы зависят от поддержки соответствующего дополнения. Многие из дополнительных интерфейсов относятся к расширениям реального времени.

Таблица 2.4. Необязательные заголовочные файлы, определяемые стандартом POSIX

Заголовочный файл	FreeBSD 5.2.1	Linux 2.4.22	MAC OS X 10.3	Solaris 9	Описание
<aio.h>	•	•	•	•	Асинхронный ввод-вывод
<mqueue.h>	•			•	Очереди сообщений
<pthread.h>	•	•	•	•	Потоки (главы 11 и 12)
<sched.h>	•	•	•	•	Планировщик
<semaphore.h>	•	•	•	•	Семафоры
<spawn.h>		•			Интерфейс запуска программ в системах реального времени
<stropts.h>		•		•	Интерфейс XSI STREAMS (раздел 14.4)
<trace.h>					Трассировка событий

Таблица 2.5. Необязательные группы интерфейсов POSIX.1 и их коды

Код	Обязательные для SUS	Символическая константа	Описание
ADV		_POSIX_ADVISORY_INFO	Консультативная информация (расширение реального времени)
AIO		_POSIX_ASYNCHRONOUS_IO	Асинхронный ввод-вывод (расширение реального времени)
BAR		_POSIX_BARRIERS	Барьеры (расширение реального времени)
CPT		_POSIX_CPUTIME	Измерение времени работы процесса (расширение реального времени)
CS		_POSIX_CLOCK_SELECTION	Выбор часов (расширение реального времени)
CX	•		Расширение стандарта ISO C
FSC	•	_POSIX_FSYNC	Синхронизация файлов
IP6		_POSIX_IPV6	Интерфейсы IPv6
MF	•	_POSIX_MAPPED_FILES	Файлы, отображаемые в память
ML		_POSIX_MEMLOCK	Блокировка памяти процесса (расширение реального времени)

Код	Обязательные для SUS	Символическая константа	Описание
MLR		_POSIX_MEMLOCK_RANGE	Блокировка области памяти (расширение реального времени)
MON		_POSIX_MONOTONIC_CLOCK	Монотонные часы (расширение реального времени)
MPR	•	_POSIX_MEMORY_PROTECTION	Защита памяти
MSG		_POSIX_MESSAGE_PASSING	Передача сообщений (расширение реального времени)
MX			Дополнение с плавающей точкой, соответствующее IEC 60599
PIO		_POSIX_PRIORITIZED_IO	Приоритетный вывод-вывод
PS		_POSIX_PRIORITIZED_SCHEDULING	Планирование процессов (расширение реального времени)
RS		_POSIX_RAW_SOCKETS	Низкоуровневые сокеты
RTS		_POSIX_REALTIME_SIGNALS	Расширение сигналов реального времени
SEM		_POSIX_SEMAPHORES	Семафоры (расширение реального времени)
SHM		_POSIX_SHARED_MEMORY_OBJECTS	Объекты разделяемой памяти (расширение реального времени)
SIO		_POSIX_SYNCHRONIZED_IO	Синхронизированный ввод-вывод (расширение реального времени)
SPI		_POSIX_SPIN_LOCKS	Взаимоблокировки (расширение реального времени)
SPN		_POSIX_SPAWN	Запуск процессов (расширение реального времени)
SS		_POSIX_SPORADIC_SERVER	Сервер непериодических (спорадических) процессов (расширение реального времени)
TCT		_POSIX_THREAD_CPUTIME	Измерение процессорного времени для потоков (расширение реального времени)
TEF		_POSIX_TRACE_EVENT_FILTER	Фильтр трассировки событий
THR	•	_POSIX_THREADS	Потоки
TMO		_POSIX_TIMEOUTS	Тайм-ауты (расширение реального времени)
TMR		_POSIX_TIMERS	Таймеры (расширение реального времени)
TPI		_POSIX_THREAD_PRIO_INHERIT	Наследование приоритета потока (расширение реального времени)

Таблица 2.5 (продолжение)

Код	Обязательные для SUS	Символическая константа	Описание
TPP		_POSIX_THREAD_PRIO_PROTECT	Защита приоритета потока (расширение реального времени)
TPS		_POSIX_THREAD_PRIORITY_SCHEDULING	Планирование выполнения потоков (расширение реального времени)
TRC		_POSIX_TRACE	Трассировка (расширение реального времени)
TRI		_POSIX_TRACE_INHERIT	Наследование трассировки
TRL		_POSIX_TRACE_LOG	Журналирование трассировки
TSA	•	_POSIX_THREAD_ATTR_STACKADDR	Адрес стека потока
TSF	•	_POSIX_THREAD_SAFE_FUNCTIONS	Функции, безопасные в контексте потока
TSH	•	_POSIX_THREAD_PROCESS_SHARED	Синхронизация потоков, разделяемая процессом
TSP		_POSIX_THREAD_SPORADIC_SERVER	Сервер непериодических (спорадических) потоков (расширение реального времени)
TSS	•	_POSIX_THREAD_ATTR_STACKSIZE	Размер стека потока
TYM		_POSIX_TYPED_MEMORY_OBJECTS	Типизированная память (расширение реального времени)
XSI	•	_XOPEN_UNIX	Интерфейсы расширений X/Open
XSR		_XOPEN_STREAMS	XSI STREAMS

POSIX.1 не включает в себя понятие суперпользователя. Вместо этого говорится, что некоторые действия требуют «соответствующих привилегий», но определение этого термина POSIX.1 оставляет на усмотрение конкретной реализации. Версии UNIX, разработанные в соответствии с принципами безопасности Министерства обороны США, имеют многоуровневую систему безопасности. Однако в этой книге мы будем пользоваться традиционной терминологией и называть такие действия требующими привилегий суперпользователя.

По прошествии почти двадцати лет работы сформировались стандарты, которые можно считать достаточно зрелыми и устоявшимися. Стандарт POSIX.1 поддерживается открытой рабочей группой, известной как Austin Group (*http://www.opengroup.org/austin*). Чтобы стандарты оставались актуальными, время от времени они должны подтверждаться или обновляться.

2.2.3. Single UNIX Specification

Single Unix Specification (Единая спецификация UNIX) представляет собой надмножество стандарта POSIX.1 и определяет дополнительные интерфейсы для расширения функциональных возможностей, предоставляемых базовой спецификацией POSIX.1. Полный набор системных интерфейсов называется *X/Open System Interface* (XSI). Интерфейсы POSIX.1, которые являются частью расширений XSI, идентифицируются символьной константой `_XOPEN_UNIX`.

XSI также определяет дополнительные интерфейсы POSIX.1, которые должны поддерживаться реализацией, чтобы она получила право именоваться «XSI-совместимой». Это синхронизация файлов, отображение файлов в память, защита памяти, интерфейсы потоков – все они отмечены в табл. 2.5 как «Обязательные для SUS». Только XSI-совместимые реализации могут называться операционными системами UNIX.

> Торговая марка UNIX принадлежит The Open Group, которая использует единую спецификацию UNIX для определения интерфейсов, обязательных для реализации в системе, чтобы она получила право называться системой UNIX. Чтобы получить лицензию на право использования торговой марки UNIX, реализация должна пройти серию тестов на соответствие.

Некоторые из дополнительных интерфейсов, определяемых XSI, являются обязательными для реализации, тогда как другие необязательны. По своим функциональным возможностям интерфейсы разделяются на группы следующим образом:

- Шифрование: обозначаются символьной константой `_XOPEN_CRYPT`
- Расширения реального времени: обозначаются символьной константой `_XOPEN_REALTIME`
- Дополнения реального времени
- Потоки реального времени: обозначаются символьной константой `_XOPEN_REALTIME_THREADS`
- Дополнения к потокам реального времени
- Трассировка
- XSI STREAMS: обозначаются символьной константой `_XOPEN_STREAMS`
- Совместимость с предыдущими версиями: обозначаются символьной константой `_XOPEN_LEGACY`

Единая спецификация UNIX (SUS) публикуется The Open Group, сформированной в 1996 году в результате слияния X/Open и Open Software Foundation (OSF). X/Open принадлежит издание «X/Open Portability Guide» (Руководство X/Open по переносимости), которое заимствовало определенные стандарты и заполнило пробелы, связанные с отсутствующими функциональными возможностями. Целью этих руководств было повышение переносимости прикладных программ, которое стало возможным благодаря простому следованию опубликованным стандартам.

Первая версия Single UNIX Specification была издана X/Open в 1994 году. Она известна также под названием «Spec 1170», поскольку содержала примерно 1170 интерфейсов. Своими корнями она уходит в инициативу Common Open Software Environment (COSE – Общая открытая программная среда), цель которой состояла в том, чтобы еще больше повысить переносимость приложений между различными реализациями UNIX. Группа COSE – Sun, IBM, HP, Novell/USL и OSF – шагнула значительно дальше простого одобрения стандартов. Дополнительно она исследовала интерфейсы, обычно используемые коммерческими приложениями. В результате были отобраны 1170 интерфейсов и, кроме того, добавлены X/Open Common Application Environment, Issue 4 (CAE – Общая среда приложений, известная также как XPG4, поскольку исторически ее предшественником было руководство X/Open Portability Guide), System V Interface Definition, Issue 3 (SVID – Определение интерфейса System V) и OSF Application Environment Specification (AES – Спецификация среды приложений).

Вторая версия Single UNIX Specification была издана The Open Group в 1997 году. В новую версию была добавлена поддержка потоков, интерфейсов реального времени, 64-битной арифметики, файлов большого размера и многобайтных символов.

Третья версия Single UNIX Specification (сокращенно – SUSv3) была опубликована The Open Group в 2001 году. Базовые спецификации SUSv3 те же, что и в стандарте IEEE Standard 1003.1-2001, и разделяются на четыре категории: «Основные определения», «Системные интерфейсы», «Командная оболочка и утилиты» и «Обоснование». SUSv3 также включает в себя X/Open Curses Issue 4, Version 2, но эта спецификация не является частью POSIX.1.

В 2002 Международная Организация по Стандартизации одобрила эту версию как международный стандарт ISO/IEC 9945:2002. В 2003 году The Open Group снова обновила стандарт 1003.1, добавив в него исправления технического характера, после чего ISO одобрила его как ISO/IEC 9945:2003. В апреле 2004 года The Open Group опубликовала Single UNIX Specification, Version 3, 2004 Edition. В нее были включены дополнительные технические исправления основного текста стандарта.

2.2.4. FIPS

Аббревиатура *FIPS* означает Federal Information Processing Standard (Федеральный стандарт обработки информации). Этот стандарт был опубликован правительством США, которое использовало его при покупке компьютерных систем. Стандарт FIPS 151-1 (апрель 1989 года) был основан на IEEE Std. 1003.1-1988 и на проекте стандарта ANSI C. За ним последовал FIPS 151-2 (май 1993 года) на основе IEEE Standard 1003.1-1990. FIPS 151-2 требовал наличия некоторых возможностей, которые стандартом POSIX.1 были объявлены необязательными. Все они стали обязательными в стандарте POSIX.1-2001.

В результате любой разработчик, желавший продавать POSIX.1-совместимые компьютерные системы американскому правительству, должен был

поддерживать некоторые из дополнительных особенностей POSIX.1. Позднее стандарт POSIX.1 FIPS был отменен, поэтому мы больше не будем возвращаться к нему в этой книге.

2.3. Реализации UNIX

В предыдущем разделе были описаны ISO C, IEEE POSIX и Single UNIX Specification – три стандарта, разработанные независимыми организациями. Однако стандарты – это лишь спецификации интерфейса. А как они связаны с реальностью? Производители берут эти стандарты и воплощают в конкретные реализации. Для нас интерес представляют как сами стандарты, так и их воплощение.

В разделе 1.1 [McKusick et al. 1996] приводится подробная (и отлично иллюстрированная) история генеалогического дерева UNIX. Все началось с 6-й (1976) и 7-й (1979) редакций UNIX Time-Sharing System для PDP-11 (обычно они именуются Version 6 и Version 7). Они стали первыми версиями, получившими широкое распространение за пределами Bell Laboratories. Начали самостоятельно развиваться три ветви UNIX:

1. Одна в AT&T; она привела к появлению System III и System V (так называемые коммерческие версии UNIX).
2. Другая – в Калифорнийском университете города Беркли; она привела к появлению 4.xBSD.
3. Третья – исследовательская версия UNIX, которая продолжала разрабатываться в исследовательском центре вычислительной техники (Computing Science Research Center) AT&T Bell Laboratories и привела к появлению UNIX Time-Shared System 8-й и 9-й редакций и завершилась выходом 10-й редакции в 1990 году.

2.3.1. UNIX System V Release 4

Версия UNIX System V Release 4 (SVR4) была выпущена подразделением AT&T – UNIX System Laboratories (USL, ранее – UNIX Software Operation). Версия SVR4 объединила функциональность AT&T UNIX System Release 3.2 (SVR3.2), SunOS – операционной системы от Sun Microsystems, 4.3BSD, выпущенной Калифорнийским университетом, и Xenix – операционной системы от корпорации Microsoft – в единую операционную систему. (Изначально Xenix разрабатывалась на основе 7-й редакции и позднее вобрала в себя многие особенности, присущие System V.) Исходные тексты SVR4 появились в конце 1989 года, а первые копии стали доступны конечным пользователям в 1990 году. Реализация SVR4 соответствовала как стандарту POSIX 1003.1, так и X/Open Portability Guide, Issue 3 (XPG3).

Корпорация AT&T также опубликовала «System V Interface Definition» (SVID, Определение интерфейса System V) [AT&T 1989]. Выпуск 3 SVID определил функциональные возможности, которые должны поддерживаться операционной системой, чтобы она могла быть квалифицирована как реализа-

ция, соответствующая System V Release 4. Как и в случае с POSIX.1, SVID определяет интерфейс, но не реализацию. В SVID не проводится каких-либо различий между системными вызовами и библиотечными функциями. Чтобы обнаружить эти различия, необходимо обращаться к справочному руководству по фактической реализации SVR4 [AT&T 1990e].

2.3.2. 4.4BSD

Версии Berkeley Software Distribution (BSD) разрабатывались и распространялись Computer Systems Research Group (CSRG) – Группой исследования компьютерных систем Калифорнийского университета в городе Беркли. Версия 4.2BSD была выпущена в 1983, а 4.3BSD – в 1986 году. Обе эти версии работали на миникомпьютерах VAX. Следующая версия, 4.3BSD Tahoe, была выпущена в 1988 году и также работала на специфическом миникомпьютере под названием Tahoe. (Книга Леффлера (Leffler) и др. [1989] описывает версию 4.3BSD Tahoe.) Затем в 1990 году последовала версия 4.3BSD Reno, которая поддерживала большую часть функциональных возможностей, определяемых стандартом POSIX.1.

Изначально BSD-системы содержали исходный код, запатентованный AT&T, и подпадали под действие лицензий AT&T. Чтобы получить исходный код BSD-системы, требовалась лицензия AT&T на UNIX. С тех пор положение вещей изменилось, так как на протяжении нескольких лет все большая часть исходного кода AT&T замещалась кодом сторонних разработчиков; кроме того, в системе появилось много новых функциональных возможностей, исходный код которых был получен из сторонних источников.

В 1989 году в версии 4.3BSD Tahoe большая часть кода, не принадлежащего AT&T, была идентифицирована и выложена в публичный доступ под названием BSD Networking Software, Release 1.0. Затем, в 1991 году, последовал второй выпуск BSD Networking Software (Release 2.0), который был развитием версии 4.3BSD Reno. Основная цель состояла в том, чтобы освободить большую часть или даже всю систему 4.4BSD от любых лицензионных ограничений AT&T, сделав исходные тексты общедоступными.

Версия 4.4BSD-Lite должна была стать заключительным релизом CSRG. Однако ее официальный выпуск был отложен из-за юридических споров с USL. Как только в 1994 году юридические разногласия были устранены, вышла 4.4BSD-Lite, полностью свободно распространяемая, так что теперь, чтобы получить ее, уже не требовалось приобретать какие-либо лицензии на исходные тексты UNIX. Вслед за этим, в 1995 году, CSRG выпустила вторую, исправленную версию. Второй выпуск 4.4BSD-Lite стал заключительной версией BSD от CSRG. (Эта версия BSD описана в книге Мак-Кьюсика [1996].)

Разработка операционной системы UNIX в Беркли началась с PDP-11, переместилась на миникомпьютеры VAX и затем на так называемые рабочие станции. В первой половине 90-х годов была добавлена поддержка популярных персональных компьютеров, собранных на базе микропроцессора 80386, что привело к появлению версии 386BSD. Реализация была выполнена Биллом Джолитцом (Bill Jolitz) и описана в серии ежемесячных статей

в журнале «Dr. Dobb's Journal» за 1991 год. Значительная часть исходного кода была заимствована из BSD Networking Software, Release 2.0.

2.3.3. FreeBSD

Операционная система FreeBSD базируется на 4.4BSD-Lite. Проект FreeBSD был образован с целью дальнейшего развития линейки BSD-систем после того, как в Беркли было принято решение о прекращении работ над BSD-версиями операционной системы UNIX, и проект 386BSD оказался заброшенным.

Все программное обеспечение, разработанное в рамках проекта FreeBSD, является свободно распространяемым, как в исходных текстах, так и в виде бинарных дистрибутивов. ОС FreeBSD 5.2.1 стала одной из четырех платформ, на которых тестировались примеры для данной книги.

> Существует еще несколько свободных операционных систем, основанных на BSD. Проект NetBSD (*http://www.netbsd.org*) аналогичен проекту FreeBSD, основной акцент в нем сделан на переносимости между различными аппаратными платформами. Проект OpenBSD (*http://www.openbsd.org*) также аналогичен FreeBSD, но с акцентом на безопасности.

2.3.4. Linux

Linux – это операционная система, которая предоставляет все богатства программного окружения UNIX и свободно распространяется в соответствии с Общественной Лицензией GNU (GNU Public License). Популярность Linux – это нечто феноменальное в компьютерной индустрии. Linux часто отличается тем, что первой из операционных систем начинает поддерживать новейшие аппаратные решения.

ОС Linux была создана Линусом Торвальдсом (Linus Torvalds) в 1991 году в качестве замены ОС MINIX. Семена дали быстрые всходы, потому что множество разработчиков по всему миру добровольно взялись за работу по ее улучшению.

Дистрибутив Linux Mandrake 9.2 стал одной из систем, на которых тестировались примеры из этой книги. В этом дистрибутиве используется ядро Linux версии 2.4.22.

2.3.5. Mac OS X

Mac OS X отличается от предыдущих версий этой системы тем, что она основана на совершенно иных технологиях. Ядро этой операционной системы называется Darwin и представляет собой комбинацию ядра Mach ([Accetta et al. 1986]) и ОС FreeBSD. Ядро Darwin является проектом с открытым исходным кодом, подобно FreeBSD и Linux.

Mac OS X 10.3 (Darwin 7.4.0) использовалась как одна из тестовых платформ при написании этой книги.

2.3.6. Solaris

Solaris – это разновидность ОС UNIX, разработанная в Sun Microsystems. Основанная на System V Release 4, она совершенствовалась инженерами из Sun Microsystems в течение более 10 лет. Это единственный коммерчески успешный потомок SVR4, формально сертифицированный как UNIX-система. (Дополнительную информацию о сертификации UNIX-систем вы найдете по адресу *http://www.opengroup.org/certification/idx/unix.html*.)

Версия Solaris 9 использовалась при написании этой книги в качестве одной из тестовых платформ.

2.3.7. Прочие версии UNIX

Среди прочих операционных систем, которые были сертифицированы как UNIX-системы, можно назвать:

- AIX, версия UNIX от IBM
- HP-UX, версия UNIX от Hewlett-Packard
- IRIX, UNIX-система, распространяемая компанией Silicon Graphics
- UnixWare, версия UNIX, которая происходит от SVR4 и ныне принадлежит корпорации SCO

2.4. Связь между стандартами и реализациями

Упомянутые выше стандарты определяют подмножество любой фактически существующей системы. Основное внимание в этой книге будет уделяться четырем операционным системам: FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3 и Solaris 9. Несмотря на то, что только Solaris может называться UNIX-системой, тем не менее, все четыре предоставляют программную среду UNIX. Поскольку все четыре системы в различной степени являются POSIX-совместимыми, мы сосредоточимся на тех функциональных возможностях, которые считаются обязательными в соответствии со стандартом POSIX.1, указывая любые различия между POSIX и фактической реализацией в этих системах. Особенности и технические приемы, характерные только для конкретной реализации, будут отмечены особо. Так как SUSv3 является надмножеством POSIX.1, мы также обратим внимание на любые особенности, которые являются частью SUSv3, но не являются частью POSIX.1.

Следует знать, что реализации обеспечивают обратную совместимость по функциональным особенностям с более ранними версиями, такими как SVR3.2 и 4.3BSD. Например, Solaris поддерживает как спецификацию неблокирующих операций ввода-вывода (`O_NONBLOCK`) стандарта POSIX.1, так и традиционный для System V метод (`O_NDELAY`). В этой книге мы будем говорить о тех характеристиках, которые предписываются стандартом POSIX.1, и при этом будем лишь иногда упоминать нестандартные особенности, сохраняемые для обратной совместимости. Например, и SVR3.2, и 4.3BSD реализуют механизм сигналов способом, который отличается от стандарта POSIX.1. В главе 10 мы опишем только сигналы POSIX.1.

2.5. Пределы

Реализации определяют множество системных кодов и констант. Многие из них жестко зашиты в тексты программ, для получения значений других используются специальные методы. Благодаря описанной выше деятельности по стандартизации, сейчас преобладают более универсальные методы определения значений констант и предусматриваемых реализациями пределов, что очень помогает в разработке переносимого программного обеспечения.

Существует два типа пределов:

1. Пределы времени компиляции (например, наибольшее значение, которое может принимать переменная типа `short int`).
2. Пределы времени выполнения (например, максимальная длина имени файла).

Пределы времени компиляции могут определяться в заголовочных файлах, которые подключаются программой на этапе компиляции. Пределы времени выполнения требуют, чтобы процесс получил их значения, вызвав соответствующие функции.

Кроме того, некоторые пределы для одной реализации имеют фиксированные значения и потому могут определяться статически в заголовочных файлах. Для других реализаций они могут варьироваться, вследствие чего для получения их значений требуется обращаться к соответствующим функциям во время исполнения. Примером предела такого типа может служить максимальная длина имени файла. System V до появления SVR4 ограничивала длину имени файла 14 символами, тогда как BSD-системы увеличили это значение до 255 символов. Сегодня большинство реализаций UNIX поддерживают множество различных типов файловых систем, и каждая из них имеет свои собственные пределы – это случай предела времени выполнения, который зависит от того, в какой файловой системе находится рассматриваемый файл. Например, корневая файловая система может ограничивать длину имени файла 14 символами, тогда как в другой файловой системе это ограничение может составлять 255 символов.

Для решения этих проблем существует три типа пределов:

1. Пределы времени компиляции (заголовочные файлы).
2. Пределы времени выполнения, не связанные с файлами или каталогами (функция `sysconf`).
3. Пределы времени выполнения, связанные с файлами или каталогами (функции `pathconf` и `fpathconf`).

Еще больше путаницы добавляет то, что если конкретный предел времени выполнения не изменяется для данной системы, он может быть определен статически в заголовочном файле. Однако если он не определен в заголовочном файле, тогда приложение должно вызвать одну из трех функций `conf` (которые вскоре будут описаны), чтобы определить его значение во время исполнения.

2.5.1. Пределы ISO C

Все пределы, которые определены стандартом ISO C, являются пределами времени компиляции. В табл. 2.6 приведены пределы, задаваемые стандартом языка C и определенные в файле <limits.h>. Эти константы всегда определяются заголовочным файлом и не изменяются. В третьей колонке указаны минимально допустимые значения, определяемые стандартом ISO C. Они были выбраны с учетом 16-битной целочисленной арифметики с поразрядным дополнением до единицы (one's-complement). В четвертой колонке приводятся значения для системы Linux, использующей 32-битную целочисленную арифметику с поразрядным дополнением до двойки (two's-complement). Обратите внимание на то, что для целочисленных типов без знака не приводится минимальное значение, так как оно всегда будет равно 0. В 64-битных системах максимальное значение для типа long соответствует максимальному значению для типа long long.

Таблица 2.6. Пределы значений целочисленных типов из файла <limits.h>

Имя	Описание	Минимально допустимое значение	Типовое значение
CHAR_BIT	Количество бит на символ	8	8
CHAR_MAX	Максимальное значение типа char	(см. ниже)	127
CHAR_MIN	Минимальное значение типа char	(см. ниже)	−128
SCHAR_MAX	Максимальное значение типа signed char	127	127
SCHAR_MIN	Минимальное значение типа signed char	−127	−128
UCHAR_MAX	Максимальное значение типа unsigned char	255	255
INT_MAX	Максимальное значение типа int	32 767	2 147 483 647
INT_MIN	Минимальное значение типа int	−32 767	−2 147 483 648
UINT_MAX	Максимальное значение типа unsigned int	65 535	4 294 967 295
SHRT_MAX	Максимальное значение типа short	32 767	32 767
SHRT_MIN	Минимальное значение типа short	−32 767	−32 768

Имя	Описание	Минимально допустимое значение	Типовое значение
USHRT_MAX	Максимальное значение типа unsigned short	65 535	65 535
LONG_MAX	Максимальное значение типа long	2 147 483 647	2 147 483 647
LONG_MIN	Минимальное значение типа long	−2 147 483 647	−2 147 483 648
ULONG_MAX	Максимальное значение типа unsigned long	4 294 967 295	4 294 967 295
LLONG_MAX	Максимальное значение типа long long	9 223 372 036 854 775 807	9 223 372 036 854 775 807
LLONG_MIN	Минимальное значение типа long long	−9 223 372 036 854 775 807	−9 223 372 036 854 775 808
ULLONG_MAX	Максимальное значение типа unsigned long long	18 446 744 073 709 551 615	18 446 744 073 709 551 615
MB_LEN_MAX	Максимальное количество байт в многобайтных символах	1	16

Одно из различий между системами, с которым мы столкнемся, состоит в том, как система представляет тип char – со знаком или без него. В четвертой колонке табл. 2.6 мы видим, что в данной системе тип char представлен как целое со знаком. Значение константы CHAR_MIN эквивалентно SCHAR_MIN, а CHAR_MAX эквивалентно SCHAR_MAX. Если тип char в системе представляется как целое без знака, следовательно, значение CHAR_MIN будет равно 0, а CHAR_MAX равно UCHAR_MAX.

Предельные значения для типов чисел с плавающей точкой определяются в заголовочном файле <float.h> подобным же образом. Каждый, кто всерьез занимается вычислениями с плавающей точкой, должен ознакомиться с содержимым этого файла.

Еще одна константа стандарта ISO C, с которой мы встретимся, – это FOPEN_MAX. Она определяет гарантированное системой минимальное количество стандартных потоков ввода-вывода, которые могут быть открыты одновременно. Это значение хранится в заголовочном файле <stdio.h> и не может быть меньше 8. Согласно стандарту POSIX.1 константа STREAM_MAX, если таковая определена, должна иметь то же самое значение.

В файле <stdio.h> стандарт ISO C определяет также константу TMP_MAX. Это максимальное количество уникальных имен файла, которые могут быть сгенерированы функцией tmpnam. Более подробно мы поговорим об этом в разделе 5.13.

В табл. 2.7 приводятся значения FOPEN_MAX и TMP_MAX для всех четырех платформ, обсуждаемых в данной книге.

Таблица 2.7. Пределы, определяемые стандартом ISO для различных платформ

Предел	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
FOPEN_MAX	20	16	20	20
TMP_MAX	308 915 776	238 328	308 915 776	17 576

Стандарт ISO C определяет также константу FILENAME_MAX, но мы избегаем ее использования, поскольку в некоторых операционных системах ее значение слишком мало для применения на практике.

2.5.2. Пределы POSIX

Стандарт POSIX.1 определяет многочисленные константы, связанные с предельными значениями. К сожалению, это один из самых запутанных аспектов POSIX.1. Хотя POSIX.1 и определяет огромное количество констант и предельных значений, тем не менее мы сосредоточим свое внимание лишь на тех из них, которые затрагивают базовые интерфейсы POSIX.1. Эти пределы и константы подразделяются на следующие категории:

1. Неизменяемые минимальные значения: 19 констант, список которых приводится в табл. 2.8.
2. Неизменяемое значение SSIZE_MAX.
3. Значения, которые могут быть увеличены во время выполнения: CHARCLASS_NAME_MAX, COLL_WEIGHTS_MAX, LINE_MAX, NGROUPS_MAX и RE_DUP_MAX.
4. Значения, не изменяемые во время выполнения, возможно неопределенные: ARG_MAX, CHILD_MAX, HOST_NAME_MAX, LOGIN_NAME_MAX, OPEN_MAX, PAGESIZE, RE_DUP_MAX, STREAM_MAX, SYMLOOP_MAX, TTY_NAME_MAX и TZNAME_MAX.
5. Изменяемые значения, связанные с размером строки пути, возможно неопределенные: FILESIZEBITS, LINK_MAX, MAX_CANON, MAX_INPUT, NAME_MAX, PATH_MAX, PIPE_BUF, SYMLINK_MAX.

Из этих 44 пределов и констант некоторые могут быть определены в файле <limits.h>, а другие могут быть и не определены – в зависимости от некоторых условий. Пределы и константы, которые не обязательно должны быть определены, мы рассмотрим в разделе 2.5.4, когда будем говорить о функциях sysconf, pathconf и fpathconf. Девятнадцать неизменяемых минимальных значений приведены в табл. 2.8.

Эти значения являются неизменяемыми и не зависят от конкретной реализации операционной системы. Они задают большинство ограничений, накладываемых на функциональные возможности. Реализации, претендующие на звание POSIX-совместимых, должны обеспечивать значения не ниже указанных. Именно поэтому они называются минимально допустимыми, хотя в их именах присутствует постфикс MAX. Кроме того, чтобы обеспечить максимальную переносимость, приложения, строго следующие стандарту,

не должны требовать более высоких значений. Описания каждой из этих констант мы будем приводить по мере того, как они будут нам встречаться.

Таблица 2.8. Неизменяемые минимальные значения из файла <limits.h>, определяемые стандартом POSIX.1

Имя	Описание: минимально допустимое значение для	Значение
_POSIX_ARG_MAX	Длины аргументов функции exec	4 096
_POSIX_CHILD_MAX	Количества дочерних процессов на реальный идентификатор пользователя	25
_POSIX_HOST_NAME_MAX	Максимальной длины имени сетевого узла, возвращаемого функцией gethostname	255
_POSIX_LINK_MAX	Количества ссылок на один файл	8
_POSIX_LOGIN_NAME_MAX	Максимальной длины имени пользователя	9
_POSIX_MAX_CANON	Количества байт в канонической входной очереди терминала	255
_POSIX_MAX_INPUT	Количества байт, доступного во входной очереди терминала	255
_POSIX_NAME_MAX	Количества байт в имени файла, не считая завершающего нулевого символа	14
_POSIX_NGROUPS_MAX	Количества идентификаторов дополнительных групп на процесс	8
_POSIX_OPEN_MAX	Количества открытых файлов на процесс	20
_POSIX_PATH_MAX	Максимальной длины строки пути к файлу, включая завершающий нулевой символ	256
_POSIX_PIPE_BUF	Количества байт, которые могут быть записаны в канал атомарно	512
_POSIX_RE_DUP_MAX	Количества повторяющихся вхождений для основного регулярного выражения, принимаемого функциями regexec и regcomp, при использовании интервальной нотации \{m,n\}	255
_POSIX_SSIZE_MAX	Значения, которое может быть сохранено в переменной типа ssize_t	32 767
_POSIX_STREAM_MAX	Количества одновременно открытых стандартных потоков ввода-вывода на процесс	8
_POSIX_SYMLINK_MAX	Количества байт в символической ссылке	255
_POSIX_SYMLOOP_MAX	Количества переходов по символическим ссылкам допустимого в строке пути	8
_POSIX_TTY_NAME_MAX	Длины имени терминального устройства, включая завершающий нулевой символ	9
_POSIX_TZNAME_MAX	Количества байт в имени временной зоны	6

> Приложения, строго следующие стандарту POSIX, отличаются от просто POSIX-совместимых приложений. Последние используют только интерфейсы, определяемые стандартом IEEE Standard 1003.1-2001. Приложение, строго следующее стандарту, – это POSIX-совместимое приложение, которое не полагается ни на какое не определенное в стандарте поведение, не использует никаких устаревающих интерфейсов и не требует значений констант больших, чем минимумы, приведенные в табл. 2.8.

К сожалению, некоторые из этих неизменяемых минимальных значений слишком малы, чтобы найти практическое применение. Например, большинство современных UNIX-систем предоставляют возможность открывать намного больше 20 файлов на процесс. Минимальный предел 255 для `_POSIX_PATH_MAX` также слишком мал. Длина строки пути может превысить это значение, что говорит о том, что мы не можем использовать константы `_POSIX_OPEN_MAX` и `_POSIX_PATH_MAX` в качестве размеров массивов на этапе компиляции.

Каждому из 19 неизменяемых минимальных значений, приведенных в табл. 2.8, соответствует значение, зависящее от реализации, имя которого отличается отсутствием приставки `_POSIX_`. Константы без приставки `_POSIX_` предназначены для хранения фактических значений, поддерживаемых конкретной реализацией. (Эти 19 констант, значения которых определяются реализацией, перечислены в пунктах 2–5 списка, приведенного выше: неизменяемое значение, значения, которые могут быть увеличены во время выполнения, значения, не изменяемые во время выполнения и изменяемые значения, связанные с размером строки пути.) Основная проблема состоит в том, что не все 19 значений, зависящих от реализации, обязательно будут определены в заголовочном файле `<limits.h>`.

Например, определение конкретного значения может не быть включено в заголовочный файл, если его фактическая величина для данного процесса зависит от количества памяти в системе. Если значения не определены в заголовочном файле, мы не сможем использовать их для задания границ массивов на этапе компиляции. Поэтому стандарт POSIX.1 определяет функции `sysconf`, `pathconf` и `fpathconf`, с помощью которых можно определить фактические значения пределов во время выполнения. Однако существует еще одна проблема: некоторые из значений определены стандартом POSIX.1 как «возможно неопределенные» (следовательно, бесконечные). Это означает, что на практике значение не имеет верхней границы. Для ОС Linux, например, количество структур `iovec`, которые можно передать функциям `readv` или `writev`, ограничено только объемом доступной памяти. Поэтому предел `IOV_MAX` для Linux считается неопределенным. Мы еще вернемся к этой проблеме в разделе 2.5.5.

2.5.3. Пределы XSI

Стандарт XSI также определяет ряд констант, значения которых зависят от реализации. Они включают:

1. Неизменяемые минимальные значения: десять констант, перечисленных в табл. 2.9.
2. Числовые пределы: `LONG_BIT` и `WORD_BIT`.

3. Значения, не изменяемые во время выполнения, возможно неопределенные: ATEXIT_MAX, IOV_MAX и PAGE_SIZE.

Неизменяемые минимальные значения перечислены в табл. 2.9. Большая часть их имеет отношение к каталогам сообщений. Две последние константы наглядно показывают ситуацию, когда минимумы, декларируемые стандартом POSIX.1, слишком малы (вероятно, для того, чтобы сделать возможной реализацию POSIX-совместимых операционных систем для встраиваемых устройств), поэтому стандартом Single UNIX Specification были добавлены увеличенные минимальные значения для использования в XSI-совместимых системах.

Таблица 2.9. Неизменяемые минимальные значения из файла <limits.h>, определяемые стандартом XSI

Имя	Описание	Минимально допустимое значение	Типовое значение
NL_ARGMAX	Максимальное количество позиционированных аргументов функций printf и scanf	9	9
NL_LANGMAX	Максимальный размер переменной окружения LANG в байтах	14	14
NL_MSGMAX	Максимальный номер сообщения	32 767	32 767
NL_NMAX	Максимальное число байт при преобразовании символов «несколько-к-одному»	Не определено	1
NL_SETMAX	Максимальный номер набора сообщений	255	255
NL_TEXTMAX	Максимальный размер строки сообщения	_POSIX2_LINE_MAX	2 048
NZERO	Приоритет процесса по умолчанию	20	20
_XOPEN_IOV_MAX	Максимальное количество структур iovec, которое может быть передано функциям readv или writev	16	16
_XOPEN_NAME_MAX	Максимальная длина имени файла в байтах	255	255
_XOPEN_PATH_MAX	Максимальная длина строки пути к файлу в байтах	1 024	1 024

2.5.4. Функции sysconf, pathconf и fpathconf

Мы перечислили различные минимальные значения, которые должны поддерживаться реализацией, но как мы узнаем фактические пределы, которые поддерживает конкретная система? Как мы уже упоминали ранее, некоторые из этих пределов могут быть определены на этапе компиляции, другие – во время исполнения. Мы также говорили, что некоторые из них являются не изменяемыми в данной системе, тогда как другие, связанные с файлами или каталогами, могут изменяться. На этапе выполнения значения пределов можно получить с помощью одной из следующих функций.

```
#include <unistd.h>

long sysconf(int name);

long pathconf(const char *pathname, int name);

long fpathconf(int filedes, int name);
```

Все три возвращают значение соответствующего предела в случае успеха, −1 в случае ошибки (см. ниже)

Различие между двумя последними функциями состоит в том, что первая из них получает в качестве аргумента строку пути к файлу, а вторая – файловый дескриптор.

В табл. 2.10 перечисляются значения аргумента *name*, которые можно передать функции `sysconf` для идентификации пределов времени выполнения. Эта функция использует константы, имена которых начинаются с префикса `_SC_`. В табл. 2.11 перечисляются значения аргумента *name* для функций `pathconf` и `fpathconf`. Эти функции используют для идентификации пределов времени выполнения константы, имена которых начинаются с префикса `_PC_`.

Таблица 2.10. Пределы и идентификаторы для аргумента name функции sysconf

Имя предела	Описание	Аргумент *name*
ARG_MAX	Максимальная длина аргументов функций семейства exec (в байтах)	_SC_ARG_MAX
ATEXIT_MAX	Максимальное количество функций, которые могут быть зарегистрированы с помощью функции atexit	_SC_ATEXIT_MAX
CHILD_MAX	Максимальное количество процессов на один реальный идентификатор пользователя	_SC_CHILD_MAX
Количество тактов системных часов в секунду	Количество тактов системных часов в секунду	_SC_CLK_TCK
COLL_WEIGHTS_MAX	Максимальное количество весовых коэффициентов для одного элемента категории LC_COLLATE в файле региональных настроек	_SC_COLL_WEIGHTS_MAX
HOST_NAME_MAX	Максимальная длина имени сетевого узла, возвращаемого функцией gethostname	_SC_HOST_NAME_MAX
IOV_MAX	Максимальное количество структур iovec, которое можно передать функциям readv и writev	_SC_IOV_MAX
LINE_MAX	Максимальная длина строки ввода, принимаемой утилитами	_SC_LINE_MAX
LOGIN_NAME_MAX	Максимальная длина имени пользователя	_SC_LOGIN_NAME_MAX
NGROUPS_MAX	Максимальное количество идентификаторов дополнительных групп на процесс	_SC_NGROUPS_MAX

Имя предела	Описание	Аргумент *name*
OPEN_MAX	Максимальное количество открытых файлов на процесс	_SC_OPEN_MAX
PAGESIZE	Системный размер страницы памяти в байтах	_SC_PAGESIZE
PAGE_SIZE	Системный размер страницы памяти в байтах	_SC_PAGE_SIZE
RE_DUP_MAX	Максимальное количество повторяющихся вхождений для основного регулярного выражения, принимаемого функциями regexec и regcomp, при использовании интервальной нотации \{m, n\}	_SC_RE_DUP_MAX
STREAM_MAX	Максимальное количество стандартных потоков ввода-вывода на процесс в любой конкретный момент времени; значение, если определено, должно быть равно FOPEN_MAX	_SC_STREAM_MAX
SYMLOOP_MAX	Максимальное количество переходов по символическим ссылкам, допустимое в строке пути	_SC_SYMLOOP_MAX
TTY_NAME_MAX	Максимальная длина имени терминального устройства, включая завершающий нулевой символ	_SC_TTY_NAME_MAX
TZNAME_MAX	Количество байт в имени временной зоны	_SC_TZNAME_MAX

Таблица 2.11. Пределы и идентификаторы для аргумента name функций pathconf и fpathconf

Имя предела	Описание	Аргумент *name*
FILESIZEBITS	Минимальное количество бит, необходимое для представления максимального размера обычного файла, допустимого для заданного каталога, в виде целого значения со знаком	_PC_FILESIZEBITS
LINK_MAX	Максимальное значение счетчика ссылок на один файл	_PC_LINK_MAX
MAX_CANON	Максимальное количество байт в канонической входной очереди терминала	_PC_MAX_CANON
MAX_INPUT	Количество байт, доступное во входной очереди терминала	_PC_MAX_INPUT
NAME_MAX	Максимальная длина имени файла в байтах (за исключением завершающего нулевого символа)	_PC_NAME_MAX
PATH_MAX	Максимальная длина строки пути к файлу, включая завершающий нулевой символ	_PC_PATH_MAX
PIPE_BUF	Максимальное количество байт, которые могут быть записаны в канал атомарно	_PC_PIPE_BUF
SYMLINK_MAX	Количество байт в символической ссылке	_PC_SYMLINK_MAX

Мы должны поближе рассмотреть значения, возвращаемые этими тремя функциями.

1. Все три функции возвращают значение -1 и код ошибки EINVAL в переменной errno, если аргумент name содержит имя неподдерживаемого предела. В третьей колонке табл. 2.10 и табл. 2.11 даны имена пределов, которые будут использоваться на протяжении всей книги.
2. Для некоторых пределов могут возвращаться либо определенные числовые значения (≥0), либо признак неопределенности – возвращаемое значение равно –1, но при этом значение errno не изменяется.
3. Значение предела _SC_CLK_TCK представляет собой количество тактов системных часов в секунду; эта величина используется при работе со значениями, возвращаемыми функцией times (раздел 8.16).

Ниже перечислены ограничения, накладываемые на аргумент *pathname* функции pathconf и аргумент *fileldes* функции fpathconf. Если какое-либо из этих ограничений не будет соблюдено, это может привести к непредсказуемым результатам.

1. Файл, к которому относятся параметры _PC_MAX_CANON и _PC_MAX_INPUT, должен быть файлом терминального устройства.
2. Файл, к которому относится параметр _PC_LINK_MAX, должен быть либо файлом, либо каталогом. Если файл является каталогом, то возвращаемое значение применимо только к самому каталогу, но не к файлам, находящимся в нем.
3. Файл, к которому относятся параметры _PC_FILESIZEBITS и _PC_NAME_MAX, должен быть каталогом. Возвращаемое значение относится к именам файлов этого каталога.
4. Файл, к которому относится параметр _PC_PATH_MAX, должен быть каталогом. Возвращаемое значение представляет собой максимальную длину относительного пути, когда заданный каталог является рабочим каталогом. (К сожалению, эта величина не отражает фактическую максимальную длину абсолютного пути, которую мы в действительности хотим узнать. Мы еще вернемся к этой проблеме в разделе 2.5.5.)
5. Файл, к которому относится параметр _PC_PIPE_BUF, должен быть неименованным каналом, именованным каналом или каталогом. В первых двух случаях возвращаемое значение относится к самим каналам. В случае каталога ограничение будет относиться к любым именованным каналам, созданным в этом каталоге.
6. Файл, к которому относится параметр _PC_SYMLINK_MAX, должен быть каталогом. Возвращаемое значение – максимальная длина строки, которую может хранить символическая ссылка в этом каталоге.

Пример

Программа на языке `awk(1)`**, представленная в листинге 2.1, генерирует программу на языке C, которая в свою очередь выводит значения всех идентификаторов функций** `pathconf` **и** `sysconf`**.**

Листинг 2.1. Генерация программы на языке C, которая выводит значения всех конфигурационных ограничений

```
BEGIN {
    printf("#include \"apue.h\"\n")
    printf("#include <errno.h>\n")
    printf("#include <limits.h>\n")
    printf("\n")
    printf("static void pr_sysconf(char *, int);\n")
    printf("static void pr_pathconf(char *, char *, int);\n")
    printf("\n")
    printf("int\n")
    printf("main(int argc, char *argv[])\n")
    printf("{\n")
    printf("\tif (argc != 2)\n")
    printf("\t\terr_quit(\"Использование: a.out <каталог>\");\n\n")
    FS="\t+"
    while (getline <"sysconf.sym" > 0) {
        printf("#ifdef %s\n", $1)
        printf("\tprintf(\"%s определен как %%d\\n\", %s+0);\n", $1, $1)
        printf("#else\n")
        printf("\tprintf(\"идентификатор %s не найден\\n\");\n", $1)
        printf("#endif\n")
        printf("#ifdef %s\n", $2)
        printf("\tpr_sysconf(\"%s =\", %s);\n", $1, $2)
        printf("#else\n")
        printf("\tprintf(\"идентификатор %s не найден\\n\");\n", $2)
        printf("#endif\n")
    }
    close("sysconf.sym")
    while (getline <"pathconf.sym" > 0) {
        printf("#ifdef %s\n", $1)
        printf("\tprintf(\"%s определен как %%d\\n\", %s+0);\n", $1, $1)
        printf("#else\n")
        printf("\tprintf(\"идентификатор %s не найден\\n\");\n", $1)
        printf("#endif\n")
        printf("#ifdef %s\n", $2)
        printf("\tpr_pathconf(\"%s =\", argv[1], %s);\n", $1, $2)
        printf("#else\n")
        printf("\tprintf(\"идентификатор %s не найден\\n\");\n", $2)
        printf("#endif\n")
    }
    close("pathconf.sym")
    exit
}
END {
```

```
        printf("\texit(0);\n")
        printf("}\n\n")
        printf("static void\n")
        printf("pr_sysconf(char *mesg, int name)\n")
        printf("{\n")
        printf("\tlong val;\n\n")
        printf("\tfputs(mesg, stdout);\n")
        printf("\terrno = 0;\n")
        printf("\tif ((val = sysconf(name)) < 0) {\n")
        printf("\t\tif (errno != 0) {\n")
        printf("\t\t\tif (errno == EINVAL)\n")
        printf("\t\t\t\tfputs(\" (не поддерживается)\\n\", stdout);\n")
        printf("\t\t\telse\n")
        printf("\t\t\t\terr_sys(\"ошибка вызова sysconf\");\n")
        printf("\t\t} else {\n")
        printf("\t\t\tfputs(\" (нет ограничений)\\n\", stdout);\n")
        printf("\t\t}\n")
        printf("\t} else {\n")
        printf("\t\tprintf(\" %%ld\\n\", val);\n")
        printf("\t}\n")
        printf("}\n\n")
        printf("static void\n")
        printf("pr_pathconf(char *mesg, char *path, int name)\n")
        printf("{\n")
        printf("\tlong val;\n")
        printf("\n")
        printf("\tfputs(mesg, stdout);\n")
        printf("\terrno = 0;\n")
        printf("\tif ((val = pathconf(path, name)) < 0) {\n")
        printf("\t\tif (errno != 0) {\n")
        printf("\t\t\tif (errno == EINVAL)\n")
        printf("\t\t\t\tfputs(\" (не поддерживается)\\n\", stdout);\n")
        printf("\t\t\telse\n")
        printf("\t\t\t\terr_sys(\"ошибка вызова pathconf, path = %%s\", path);\n")
        printf("\t\t} else {\n")
        printf("\t\t\tfputs(\" (нет ограничений)\\n\", stdout);\n")
        printf("\t\t}\n")
        printf("\t} else {\n")
        printf("\t\tprintf(\" %%ld\\n\", val);\n")
        printf("\t}\n")
        printf("}\n")
}
```

Программа на языке `awk` **считывает два входных файла –** `pathconf.sym` **и** `sysconf.sym`**, которые содержат перечень пределов и идентификаторов, разделенных символами табуляции. Не на каждой платформе определены все идентификаторы, поэтому программа на языке** `awk` **окружает каждый вызов** `pathconf` **и** `sysconf` **директивами условной компиляции** `#ifdef`**.**

Например, программа на языке `awk` **трансформирует строку входного файла, которая выглядит следующим образом:**

```
NAME_MAX    _PC_NAME_MAX
```

в следующий код на языке C:

```
#ifdef NAME_MAX
    printf("NAME_MAX определен как %d\n", NAME_MAX+0);
#else
    printf("идентификатор NAME_MAX не найден\n");
#endif
#ifdef _PC_NAME_MAX
    pr_pathconf("NAME_MAX =", argv[1], _PC_NAME_MAX);
#else
    printf("идентификатор _PC_NAME_MAX не найден\n");
#endif
```

Программа, представленная в листинге 2.2, сгенерирована предыдущей программой. Она выводит значения всех пределов, корректно обрабатывая случаи, когда идентификатор не определен.

Листинг 2.2. Вывод всех возможных значений sysconf и pathconf

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

static void pr_sysconf(char *, int);
static void pr_pathconf(char *, char *, int);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("Использование: a.out <каталог>");
#ifdef ARG_MAX
    printf("ARG_MAX определен как %d\n", ARG_MAX+0);
#else
    printf("идентификатор ARG_MAX не найден\n");
#endif
#ifdef _SC_ARG_MAX
    pr_sysconf("ARG_MAX =", _SC_ARG_MAX);
#else
    printf("идентификатор _SC_ARG_MAX не найден\n");
#endif
/* аналогичным образом производится обработка всех остальных */
/* идентификаторов sysconf... */
#ifdef MAX_CANON
    printf("MAX_CANON определен как %d\n", MAX_CANON+0);
#else
    printf("идентификатор MAX_CANON не найден\n");
#endif
#ifdef _PC_MAX_CANON
    pr_pathconf("MAX_CANON =", argv[1], _PC_MAX_CANON);
#else
    printf("идентификатор _PC_MAX_CANON не найден\n");
#endif
/* аналогичным образом производится обработка всех остальных */
```

```
/* идентификаторов pathconf... */
    exit(0);
}

static void
pr_sysconf(char *mesg, int name)
{
    long val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = sysconf(name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs(" (не поддерживается)\n", stdout);
            else
                err_sys("ошибка вызова sysconf");
        } else {
            fputs(" (нет ограничений)\n", stdout);
        }
    } else {
        printf(" %ld\n", val);
    }
}

static void
pr_pathconf(char *mesg, char *path, int name)
{
    long val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = pathconf(path, name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs(" (не поддерживается)\n", stdout);
            else
                err_sys("ошибка вызова pathconf, path = %s", path);
        } else {
            fputs(" (нет ограничений)\n", stdout);
        }
    } else {
        printf(" %ld\n", val);
    }
}
```

В табл. 2.12 приводятся результаты работы программы, представленной листингом 2.2, на каждой из четырех систем, обсуждаемых в данной книге. «Нет идентификатора» означает, что данная платформа не имеет соответствующего идентификатора _SC или _PC, с помощью которого можно было бы узнать значение константы. В этом случае предел считается неопределенным. В противоположность этому обозначение «не поддерживается» говорит о том, что идентификатор определен, но он не распознается функциями

pathconf и sysconf. «Нет ограничений» означает, что система не задает этот предел, но это вовсе не означает, что предела нет вообще.

Таблица 2.12. Примеры конфигурационных пределов

Предел	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	
				файловая система UFS	файловая система PCFS
ARG_MAX	65 536	131 072	262 144	1 048,320	1 048 320
ATEXIT_MAX	32	2 147 483 647	Нет идентификатора	Нет ограничений	Нет ограничений
CHARCLASS_NAME_MAX	Нет идентификатора	2 048	Нет идентификатора	14	14
CHILD_MAX	867	999	100	7 877	7 877
Количество тактов системных часов в секунду	128	100	100	100	100
COLL_WEIGHTS_MAX	0	255	2	10	10
FILESIZEBITS	Не поддерживается	64	Нет идентификатора	41	Не поддерживается
HOST_NAME_MAX	255	Не поддерживается	Нет идентификатора	Нет идентификатора	Нет идентификатора
IOV_MAX	1 024	Нет ограничений	Нет идентификатора	16	16
LINE_MAX	2 048	2 048	2 048	2 048	2 048
LINK_MAX	32 768	32 000	32 768	32 768	1
LOGIN_NAME_MAX	17	256	Нет идентификатора	9	9
MAX_CANON	255	255	255	256	256
MAX_INPUT	255	255	255	512	512
NAME_MAX	255	255	765	255	8
NGROUPS_MAX	16	32	16	16	16
OPEN_MAX	1 735	1 024	256	256	256
PAGESIZE	4 096	4 096	4 096	8 192	8 192
PAGE_SIZE	4 096	4 096	Нет идентификатора	8 192	8 192
PATH_MAX	1 024	4 096	1 024	1 024	1 024
PIPE_BUF	512	4 096	512	5 120	5 120
RE_DUP_MAX	255	32 767	255	255	255

Таблица 2.12 (продолжение)

Предел	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	
				файловая система UFS	файловая система PCFS
STREAM_MAX	1 735	16	20	256	256
SYMLINK_MAX	Не поддерживается	Нет ограничений	Нет идентификатора	Нет идентификатора	Нет идентификатора
SYMLOOP_MAX	32	Нет ограничений	Нет идентификатора	Нет идентификатора	Нет идентификатора
TTY_NAME_MAX	255	32	Нет идентификатора	128	128
TZNAME_MAX	255	6	255	Нет ограничений	Нет ограничений

В разделе 4.14 мы увидим, что UFS – это реализация Berkeley fast file system для SVR4, а PCFS – это реализация файловой системы MS-DOS FAT для Solaris.

2.5.5. Неопределенные пределы времени выполнения

Мы уже упоминали, что некоторые пределы могут быть не определены. Проблема состоит в том, что, если они не определены в заголовочном файле <limits.h>, мы не сможем использовать их на этапе компиляции. Но, кроме того, они могут оставаться неопределенными даже во время выполнения! Давайте рассмотрим два конкретных случая: размещение в памяти строки пути и определение количества файловых дескрипторов.

Строка пути

Многим программам приходится выделять память для хранения строки пути. Обычно память выделяется на этапе компиляции; в этом случае в качестве размеров массивов выбираются некоторые «магические» числа (немногие из которых корректны), например 256, 512, 1024 или стандартная константа BUFSIZ. В операционной системе 4.3BSD константа MAXPATHLEN, определяемая в заголовочном файле <sys/param.h>, представляет собой правильное значение, но большинство приложений, написанных под 4.3BSD, ее не используют.

Для таких случаев стандартом POSIX.1 предусматривается константа PATH_MAX, но ее значение вполне может оказаться неопределенным. В листинге 2.3 приводится функция, которая будет использоваться в этой книге для определения объема памяти, необходимого для размещения строки пути.

Если константа PATH_MAX определена в файле <limits.h>, то используется ее значение. Если нет, мы должны вызвать функцию pathconf. Значение, возвращаемое этой функцией, представляет собой максимальный размер строки относительного пути для случая, когда первый аргумент является рабочим каталогом. Таким образом, в качестве первого аргумента мы указываем

корневой каталог и прибавляем к полученному результату единицу. Если pathconf сообщает, что константа PATH_MAX не определена, нам остается лишь надеяться на удачу и выбрать достаточно большое число самостоятельно.

Стандарты, предшествовавшие SUSv3, не уточняли, должна ли константа PATH_MAX учитывать завершающий нулевой символ в конце строки пути. Если реализация операционной системы соответствует одному из этих ранних стандартов, следует на всякий случай добавить единицу к полученному объему памяти.

Выбор того или иного алгоритма в случае неопределенного результата зависит от того, как используется выделяемая память. Если память выделяется для вызова функции getcwd, например для того, чтобы получить абсолютное имя рабочего каталога (раздел 4.22), то в случае, когда выделенный объем памяти окажется слишком мал, мы получим признак ошибки и в errno будет записан код ошибки ERANGE. В такой ситуации можно увеличить объем памяти, выделенной под строку, вызвав функцию realloc (раздел 7.8 и упражнение 4.16), и повторить попытку. При необходимости можно продолжать увеличение размера строки до тех пор, пока вызов функции getcwd не завершится успехом.

Листинг 2.3. Динамическое выделение памяти для строки пути

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

#ifdef PATH_MAX
static int pathmax = PATH_MAX;
#else
static int pathmax = 0;
#endif

#define SUSV3 200112L

static long posix_version = 0;

/* Если константа PATH_MAX не определена, то нельзя гарантировать, */
/* что следующее число будет достаточно адекватным */
#define PATH_MAX_GUESS 1024

char *
path_alloc(int *sizep) /* если результат не пустой указатель, */
                       /* то также возвращается размер выделенной памяти */
{
    char *ptr;
    int size;

    if (posix_version == 0)
        posix_version = sysconf(_SC_VERSION);

    if (pathmax == 0) { /* первый вызов функции */
        errno = 0;

        if ((pathmax = pathconf("/", _PC_PATH_MAX)) < 0) {
            if (errno == 0)
```

```
                pathmax = PATH_MAX_GUESS; /* если константа не определена */
            else
                err_sys("ошибка вызова pathconf с параметром _PC_PATH_MAX");
        } else {
            pathmax++;          /* добавить 1, т.к. путь относительно корня */
        }
    }

    if (posix_version < SUSV3)
        size = pathmax + 1;
    else
        size = pathmax;

    if ((ptr = malloc(size)) == NULL)
        err_sys("ошибка функции malloc");

    if (sizep != NULL)
        *sizep = size;
    return(ptr);
}
```

Максимальное количество открытых файлов

Как правило, процесс-демон, то есть процесс, который выполняется в фоновом режиме и не связан с терминальным устройством, закрывает все открытые файлы. Некоторые программы предусматривают следующую последовательность действий, исходя из предположения, что в заголовочном файле `<sys/param.h>` определена константа `NOFILE`:

```
#include <sys/param.h>

for (i = 0; i < NOFILE; i++)
    close(i);
```

В других программах используется константа `_NFILE`, которая определена в некоторых версиях `<stdio.h>` как верхний предел. В третьих в качестве верхнего предела жестко зашито число 20.

Мы могли бы понадеяться на константу `OPEN_MAX`, определяемую стандартом POSIX.1, чтобы переносимым образом узнать значение этого предела, но если она не определена, то проблема останется нерешенной. Ниже приведен код, в котором в случае неопределенного значения `OPEN_MAX` цикл не будет выполнен ни разу, так как `sysconf` вернет –1:

```
#include <unistd.h>

for (i = 0; i < sysconf(_SC_OPEN_MAX); i++)
    close(i);
```

Лучшее, что можно предпринять в такой ситуации, – закрыть все дескрипторы до некоторого произвольного предела, например 256. Как и в случае со строкой пути, такой подход не гарантирует желаемого результата во всех возможных случаях, но это лучшее, что можно сделать. Мы продемонстрируем данный подход в листинге 2.4.

Листинг 2.4. Определение количества файловых дескрипторов

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

#ifdef OPEN_MAX
static long openmax = OPEN_MAX;
#else
static long openmax = 0;
#endif

/*
 * Если константа OPEN_MAX не определена, мы не можем
 * гарантировать адекватность следующего значения.
 */
#define OPEN_MAX_GUESS 256

long
open_max(void)
{
    if (openmax == 0) { /* первый вызов функции */
        errno = 0;

        if ((openmax = sysconf(_SC_OPEN_MAX)) < 0) {
            if (errno == 0)
                openmax = OPEN_MAX_GUESS; /* неопределенный предел */
            else
                err_sys("ошибка вызова sysconf с параметром _SC_OPEN_MAX");
        }
    }
    return(openmax);
}
```

Легко поддаться искушению просто вызывать функцию close до тех пор, пока она не вернет признак ошибки, но дело в том, что по коду ошибки EBADF, которую возвращает close, нельзя сказать, была ли это попытка закрыть неправильный дескриптор или дескриптор просто не был открыт. Если бы мы реализовали такой алгоритм, то в случае, когда дескриптор 10 был бы открытым, а дескриптор 9 – нет, выполнение цикла остановилось бы на дескрипторе 9 и дескриптор 10 остался бы незакрытым. С другой стороны, функция dup (раздел 3.12) возвращает признак ошибки, если будет превышен предел OPEN_MAX, но создание сотен копий дескриптора – слишком экстремальный способ выяснения значения искомого предела.

Некоторые реализации возвращают LONG_MAX в качестве значения пределов, которые в действительности не ограничены. Так обстоит дело с пределом ATEXIT_MAX в операционной системе Linux (табл. 2.12). Вообще такой подход нельзя назвать приемлемым, потому что он может привести к непредсказуемой работе программ.

Например, с помощью команды ulimit, встроенной в командный интерпретатор Bourne-again shell, можно изменить максимальное количество файлов,

которые процессы могут держать одновременно открытыми. Вообще, если предел должен быть фактически неограниченным, выполнение этой операции требует привилегий суперпользователя. Но если мы сделаем верхний предел практически неограниченным, функция sysconf будет возвращать число LONG_MAX в качестве предела OPEN_MAX. Тогда программа, которая ориентируется на значение верхнего предела, как в листинге 2.4, будет затрачивать огромное количество времени на попытки закрыть 2 147 483 647 дескрипторов, большинство из которых даже не открывались.

Системы, которые поддерживают расширения XSI стандарта Single UNIX Specification, предоставляют функцию getrlimit(2) (раздел 7.11). Она может использоваться для получения максимального количества открытых дескрипторов на процесс. Таким способом можно узнать, определено ли ограничение на количество открытых файлов, и избежать дальнейших проблем.

> Значение OPEN_MAX согласно определению стандарта POSIX относится к разряду не изменяемых во время выполнения. Это означает, что данный предел не изменяется в течение всей жизни процесса. Однако в системах, которые поддерживают расширения XSI, мы можем изменить его посредством вызова функции setrlimit(2) (раздел 7.11). (Значение этого предела также может быть изменено командой limit командной оболочки C shell или командой ulimit командных оболочек Bourne, Bourne-again и Korn shell.) Если ваша система поддерживает данную возможность, то можно изменить функцию, представленную листингом 2.4, таким образом, чтобы она вызывала sysconf при каждом обращении к ней, а не только на первом вызове.

2.6. Необязательные параметры

Мы уже видели список необязательных параметров, определяемых стандартом POSIX.1, в табл. 2.5 и обсуждали необязательные категории XSI в разделе 2.2.3. Для написания переносимых приложений, зависящих от любой из этих необязательных особенностей, нам необходим переносимый способ определять, поддерживает ли система заданный необязательный параметр.

Как и в случае с пределами (см. раздел 2.5), в Single UNIX Specification есть три разновидности параметров:

1. Параметры времени компиляции, описанные в файле <unistd.h>.
2. Параметры времени выполнения, не связанные с файлами или каталогами и идентифицируемые функцией sysconf.
3. Параметры времени выполнения, связанные с файлами или каталогами, значения которых можно получить с помощью функций pathconf или fpathconf.

В перечень необязательных параметров входят символьные константы из третьей колонки табл. 2.5, а также символьные константы, представленные в табл. 2.13 и 2.14. Если символьная константа не определена, мы должны использовать функции sysconf, pathconf или fpathconf, чтобы узнать, поддерживается ли заданный необязательный параметр. В этом случае через аргумент *name* функции передается имя, сформированное заменой префикса _POSIX на _SC или _PC. Для констант, которые начинаются с префикса _XOPEN,

через аргумент *name* передается идентификатор, сформированный путем добавления префикса _SC или _PC. Предположим, что константа _POSIX_THREADS не определена. В таком случае, чтобы узнать, поддерживает ли система потоки POSIX, мы можем вызвать функцию sysconf, передав ей в качестве аргумента name идентификатор _SC_THREADS. Далее, если константа _XOPEN_UNIX также не определена, то чтобы узнать, поддерживает ли система расширения XSI, можно вызвать функцию sysconf, передав ей в качестве аргумента *name* идентификатор _SC_XOPEN_UNIX.

Если символьная константа определена в системе, есть три возможных варианта:

1. Если константа определена со значением −1, это означает, что данная функциональная возможность не поддерживается системой.
2. Если константа определена и имеет значение больше нуля, это означает, что данная функциональная возможность поддерживается.
3. Если константа определена и имеет значение равное нулю, это означает, что мы должны использовать функции sysconf, pathconf или fpathconf, чтобы узнать, поддерживается ли заданная функциональная возможность.

В табл. 2.13 перечисляются необязательные параметры и соответствующие им символьные константы, используемые при обращении к функции sysconf, в дополнение к перечисленным в табл. 2.5.

Таблица 2.13. Необязательные параметры и их идентификаторы для функции sysconf

Имя параметра	Описание	Аргумент *name*
_POSIX_JOB_CONTROL	Указывает, поддерживается ли системой управление заданиями	_SC_JOB_CONTROL
_POSIX_READER_WRITER_LOCKS	Указывает, поддерживаются ли системой блокировки чтения-записи	_SC_READER_WRITER_LOCKS
_POSIX_SAVED_IDS	Указывает, поддерживает ли система сохраненные идентификаторы пользователя и группы	_SC_SAVED_IDS
_POSIX_SHELL	Указывает, поддерживает ли система стандартную командную оболочку POSIX	_SC_SHELL
_POSIX_VERSION	Указывает версию POSIX.1	_SC_VERSION
_XOPEN_CRYPT	Указывает, поддерживается ли системой группа интерфейсов шифрования XSI	_SC_XOPEN_CRYPT
_XOPEN_LEGACY	Указывает, поддерживается ли системой группа интерфейсов XSI, обеспечивающих совместимость с предыдущими версиями	_SC_XOPEN_LEGACY
_XOPEN_REALTIME	Указывает, поддерживается ли системой группа интерфейсов реального времени XSI	_SC_XOPEN_REALTIME
_XOPEN_REALTIME_THREADS	Указывает, поддерживается ли системой группа интерфейсов потоков реального времени XSI	_SC_XOPEN_REALTIME_THREADS
_XOPEN_VERSION	Указывает версию XSI	_SC_XOPEN_VERSION

В табл. 2.14 перечисляются символьные константы, которые могут использоваться при обращении к функциям pathconf и fpathconf. Как и в случае с системными пределами, есть некоторые обстоятельства, на которые мы хотим обратить ваше внимание.

Таблица 2.14. Необязательные параметры и идентификаторы для функций pathconf и fpathconf

Имя параметра	Описание	Аргумент *name*
_POSIX_CHOWN_RESTRICTED	Указывает, ограничено ли действие функции chown	_PC_CHOWN_RESTRICTED
_POSIX_NO_TRUNC	Указывает, приводит ли к ошибке использование путей длиннее чем NAME_MAX	_PC_NO_TRUNC
_POSIX_VDISABLE	Если определен, действие специальных терминальных символов может быть запрещено этим значением	_PC_VDISABLE
_POSIX_ASYNC_IO	Указывает, поддерживаются ли операции асинхронного ввода-вывода для заданного файла	_PC_ASYNC_IO
_POSIX_PRIO_IO	Указывает, поддерживаются ли приоритетные операции ввода-вывода для заданного файла	_PC_PRIO_IO
_POSIX_SYNC_IO	Указывает, поддерживаются ли операции синхронизированного ввода-вывода для заданного файла	_PC_SYNC_IO

1. Параметр _SC_VERSION указывает год (первые четыре цифры) и месяц (последние две цифры) публикации стандарта. Его значение может быть 198808L, 199009L, 199506L или иным для более поздних версий. Так, версии 3 Single UNIX Specification соответствует значение 200112L.
2. Значение параметра _SC_XOPEN_VERSION указывает версию XSI, которой соответствует система. Третьей версии Single UNIX Specification соответствует значение 600.
3. Параметры _SC_JOB_CONTROL, _SC_SAVED_IDS и _PC_VDISABLE сейчас не относятся к дополнительным функциональным возможностям, так как начиная с версии 3 стандарта Single UNIX Specification они перешли в разряд обязательных, хотя сами идентификаторы сохранены для обратной совместимости.
4. Для параметров _PC_CHOWN_RESTRICTED и _PC_NO_TRUNC возвращается значение –1 без изменения errno, если функциональная возможность не поддерживается для указанного значения аргумента *pathname* или *filedes*.
5. Файл, к которому относится параметр _PC_CHOWN_RESTRICTED, должен быть либо файлом, либо каталогом. Если это каталог, то данная функциональная возможность будет применяться к файлам в этом каталоге.
6. Файл, к которому относится параметр _PC_NO_TRUNC, должен быть каталогом. Возвращаемое значение применяется к именам файлов в этом каталоге.

7. Файл, к которому относится параметр _PC_VDISABLE, должен быть файлом терминального устройства.

В табл. 2.15 приведены некоторые конфигурационные параметры и соответствующие им значения для четырех платформ, обсуждаемых в данной книге. Обратите внимание на то, что некоторые системы еще не соответствуют последней версии Single UNIX Specification. Так, например, Mac OS X 10.3 поддерживает потоки POSIX, но определяет соответствующую константу как

```
#define _POSIX_THREADS
```

без указания определенного значения. Однако в соответствии с Single UNIX Specification версии 3 эта константа должна быть определена со значением –1, 0 или 200112.

Таблица 2.15. Примеры значений конфигурационных параметров

Предел	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	
				файловая система UFS	файловая система PCFS
_POSIX_CHOWN_RESTRICTED	1	1	1	1	1
_POSIX_JOB_CONTROL	1	1	1	1	1
_POSIX_NO_TRUNC	1	1	1	1	Не поддерживается
_POSIX_SAVED_IDS	Не поддерживается	1	Не поддерживается	1	1
_POSIX_THREADS	200112	200112	Определено	1	1
_POSIX_VDISABLE	255	0	255	0	0
_POSIX_VERSION	200112	200112	198808	199506	199506
_XOPEN_UNIX	Не поддерживается	1	Не определено	1	1
_XOPEN_VERSION	Не поддерживается	500	Не определено	3	3

Если в ячейке таблицы указано «не определено», это означает, что данная функциональная особенность не определена, то есть система не определяет данную символьную константу или соответствующую ей константу с префиксом _PC или _SC. Напротив, «определено» означает, что символьная константа определена, но ей не соответствует какое-либо числовое значение, как в предыдущем примере с _POSIX_THREADS. «Не поддерживается» означает, что система определяет символьную константу и ей присвоено значение –1 или 0, но функция sysconf или pathconf возвратила значение –1.

Обратите внимание, что функция pathconf в операционной системе Solaris 9 возвращает значение –1 для параметра _PC_NO_TRUNC, если вызывается для файла, находящегося в файловой системе PCFS. Эта файловая система под-

держивает формат DOS (для дискет) и без предупреждения усекает имена файлов до формата 8.3, как того требует файловая система DOS.

2.7. Макроопределения контроля функциональных особенностей

Огромное количество идентификаторов стандарта POSIX.1 и XSI определяются в заголовочных файлах. Но большинство реализаций добавляют в эти файлы и свои собственные определения в дополнение к тем, что описываются стандартами POSIX.1 и XSI. Если возникает необходимость скомпилировать программу так, чтобы она зависела только от определений POSIX и не использовала определения, зависящие от реализации, необходимо определить константу `_POSIX_C_SOURCE`. Эта константа используется во всех заголовочных файлах стандарта POSIX.1 для исключения любых зависящих от реализации определений.

> Ранние версии стандарта POSIX.1 определяли константу `_POSIX_SOURCE`. Она была заменена константой `_POSIX_C_SOURCE` в версии POSIX.1 от 2001 года.

Константы `_POSIX_C_SOURCE` и `_POSIX_SOURCE` называются *макроопределениями контроля функциональных особенностей*. Все подобные макроопределения начинаются с символа подчеркивания. Обычно они используются в командной строке компилятора `cc`, например

```
cc -D_POSIX_C_SOURCE=200112 file.c
```

В таком случае контролирующий макрос будет определен прежде, чем программа на языке C подключит какой-либо заголовочный файл. Чтобы использовать только определения стандарта POSIX.1, мы также можем в первой строке файла с исходным текстом программы указать следующее определение:

```
#define _POSIX_C_SOURCE 200112
```

Чтобы приложениям стали доступны функциональные особенности, определяемые версией 3 Single UNIX Specification, мы должны определить константу `_XOPEN_SOURCE` со значением 600. Это даст тот же эффект, что и определение константы `_POSIX_C_SOURCE` со значением 200112L, когда речь идет о функциональности, определяемой стандартом POSIX.1.

Стандарт Single UNIX Specification определяет утилиту `c99` в качестве интерфейса к среде компиляции языка C. С ее помощью можно скомпилировать файл следующим образом:

```
c99 -D_XOPEN_SOURCE=600 file.c -o file
```

Чтобы разрешить компилятору `gcc` использовать расширения 1999 ISO C, можно добавить в командную строку параметр `-std=c99`, как показано ниже:

```
gcc -D_XOPEN_SOURCE=600 -std=c99 file.c -o file
```

Другой макрос контроля функциональности, `__STDC__`, автоматически определяется компилятором языка C, если он соответствует стандарту ISO C. Это

позволяет писать программы, которые будут успешно компилироваться не только компиляторами ISO C, но и компиляторами, не соответствующими этому стандарту. Например, чтобы воспользоваться преимуществами стандарта ISO C, если таковой поддерживается, при определении прототипов функций можно включить в заголовочный файл следующий код:

```
#ifdef _ _STDC_ _
void *myfunc(const char *, int);
#else
void *myfunc();
#endif
```

Хотя большинство современных компиляторов языка C поддерживают стандарт ISO C, тем не менее во многих заголовочных файлах можно обнаружить подобное использование макроса `__STDC__`.

2.8. Элементарные системные типы данных

Исторически некоторые типы данных языка C были связаны с некоторыми переменными системы UNIX. Например, старшие и младшие номера устройств исторически хранились в виде 16-битного целого числа, где 8 бит отводилось для старшего номера устройства и 8 бит – для младшего номера. Но большинство крупных систем требуют возможности определения более чем 256 различных номеров устройств, поэтому потребовалось предусмотреть иной подход к нумерации. (Так, Solaris использует 32 бита для хранения номеров устройств – 14 бит для старшего и 18 бит для младшего номера устройства.)

Заголовочный файл `<sys/types.h>` определяет ряд зависящих от реализации типов данных, которые называются *элементарными системными типами данных*. Кроме того, некоторые из этих типов данных объявляются и в других заголовочных файлах. Все они объявлены посредством директивы `typedef`. Названия их обычно завершаются последовательностью `_t`. В табл. 2.16 приводится большинство элементарных типов, с которыми мы будем сталкиваться в этой книге.

Таблица 2.16. Некоторые наиболее распространенные элементарные системные типы данных

Тип	Описание
`caddr_t`	Адрес в памяти (раздел 14.9)
`clock_t`	Счетчик тактов системных часов (время работы процесса) (раздел 1.10)
`comp_t`	Счетчик тактов в упакованном виде (раздел 8.14)
`dev_t`	Номер устройства (старший и младший) (раздел 4.23)
`fd_set`	Набор файловых дескрипторов (раздел 14.5.1)
`fpos_t`	Позиция в файле (раздел 5.10)
`gid_t`	Числовой идентификатор группы

Таблица 2.16 (продолжение)

Тип	Описание
ino_t	Номер индексного узла (i-node) (раздел 4.14)
mode_t	Тип файла, режим создания файла (раздел 4.5)
nlink_t	Счетчик ссылок для записей в файле каталога (раздел 4.14)
off_t	Размер файла и смещение в файле (со знаком) (lseek, раздел 3.6)
pid_t	Идентификатор процесса и идентификатор группы процессов (со знаком) (разделы 8.2 и 9.4)
ptrdiff_t	Разность двух указателей (со знаком)
rlim_t	Предельное значение для ресурса (раздел 7.11)
sig_atomic_t	Тип данных, доступ к которым может выполняться атомарно (раздел 10.15)
sigset_t	Набор сигналов (раздел 10.11)
size_t	Размер объекта (например, строки) (без знака) (раздел 3.7)
ssize_t	Возвращаемый функциями результат, который представляет собой счетчик байт (со знаком) (read, write, раздел 3.7)
time_t	Счетчик секунд календарного времени (раздел 1.10)
uid_t	Числовой идентификатор пользователя
wchar_t	Может представлять символы любой кодировки

Определение этих типов данных выполнено таким образом, что при написании программ нет необходимости погружаться в детали конкретной реализации, которые могут меняться от системы к системе. Мы будем описывать, как используется каждый из этих типов, по мере необходимости.

2.9. Конфликты между стандартами

В общем и целом, различные стандарты прекрасно уживаются друг с другом. В основном мы будем обращать внимание на различия между стандартами ISO C и POSIX.1, поскольку стандарт SUSv3 является надмножеством стандарта POSIX.1. Ниже приводятся некоторые отличия.

Стандарт ISO C определяет функцию clock, которая возвращает количество процессорного времени, использованного процессом. Возвращаемое значение имеет тип clock_t. Чтобы преобразовать это значение в секунды, мы должны разделить его на константу CLOCKS_PER_SEC, определенную в заголовочном файле <time.h>. Стандарт POSIX.1 определяет функцию times, которая возвращает как процессорное время (для вызывающего процесса и для всех его дочерних процессов, завершивших свою работу), так и общее время. Все эти значения имеют тип clock_t. С помощью функции sysconf необходимо получить количество тактов в секунду и затем использовать его для перевода значений типа clock_t в секунды. Получается, что одна и та же характе-

ристика – количество тактов в секунду – определяется стандартами ISO C и POSIX.1 различными способами. Кроме того, оба стандарта используют один и тот же тип данных (clock_t) для хранения различных значений. Разницу можно наблюдать в ОС Solaris, где функция clock возвращает время в микросекундах (следовательно, константа CLOCKS_PER_SEC имеет значение 1 000 000), тогда как функция sysconf возвращает значение 100 (количество тактов в секунду).

Конфликт возможен также в тех случаях, когда стандарт ISO C определяет некоторую функцию, но не так строго, как это делает стандарт POSIX.1. Так, например, обстоит дело с функциями, которые требуют иной реализации в среде POSIX (многозадачной), чем в среде ISO C (где очень немногое можно предположить о целевой операционной системе). Тем не менее большинство POSIX-совместимых систем реализуют функции в соответствии со стандартом ISO C для сохранения совместимости. Примером может служить функция signal. Если мы по незнанию будем использовать функцию signal из ОС Solaris (надеясь написать переносимый код, который будет работать в среде ISO C и на устаревших версиях UNIX), то получим семантику, отличную от той, которую имеет функция sigaction, определяемая стандартом POSIX.1. Более подробно о функции signal мы поговорим в главе 10.

2.10. Подведение итогов

Очень многое произошло в сфере стандартизации программной среды UNIX за прошедшие два десятилетия. Мы описали наиболее важные стандарты – ISO C, POSIX и Single UNIX Specification – и их влияние на четыре реализации UNIX, обсуждаемые в данной книге: FreeBSD, Linux, Mac OS X и Solaris. Эти стандарты пытаются определить некоторые конфигурационные параметры, которые могут варьироваться от системы к системе, и мы видели, что они далеки от совершенства. В этой книге мы еще не раз столкнемся со многими из них.

Упражнения

2.1. Мы упоминали в разделе 2.8, что некоторые из элементарных системных типов данных определены более чем в одном заголовочном файле. Так, например, в ОС FreeBSD 5.2.1 тип size_t определен в 26 различных файлах. Поскольку программа может подключить все 26 файлов, а стандарт ISO C не допускает множественного определения одного и того же типа, подумайте, как должны быть написаны эти заголовочные файлы.

2.2. Просмотрите заголовочные файлы в своей системе и перечислите фактические типы данных, используемые для реализации элементарных системных типов.

2.3. Измените программу из листинга 2.4 так, чтобы избежать лишней работы, когда функция sysconf возвращает значение LONG_MAX в качестве предела OPEN_MAX.

3

Файловый ввод–вывод

3.1. Введение

Обсуждение системы UNIX мы начнем с операций файлового ввода-вывода, таких как открытие файла, чтение из файла, запись в файл и т. д. Большинство операций файлового ввода-вывода в UNIX можно выполнить с помощью всего пяти функций: `open`, `read`, `write`, `lseek` и `close`. Далее мы рассмотрим, как изменение размера буфера влияет на производительность функций `read` и `write`.

Функции, описываемые в этой главе, очень часто называют функциями *небуферизованного ввода-вывода* в противоположность стандартным функциям ввода-вывода, о которых пойдет речь в главе 5. Термин *небуферизованный* означает, что каждая операция чтения или записи обращается к системному вызову ядра. Функции небуферизованного ввода-вывода не являются частью стандарта ISO C, но они определены стандартами POSIX.1 и Single UNIX Specification.

Всякий раз, когда речь заходит о совместном использовании ресурсов несколькими процессами, особую важность приобретает понятие атомарного выполнения операций. Мы рассмотрим это понятие применительно к операциям файлового ввода-вывода и аргументам функции `open`. Далее мы увидим, как осуществляется одновременный доступ к файлам из нескольких процессов и какие структуры данных ядра с этим связаны. Затем мы перейдем к функциям `dup`, `fcntl`, `sync`, `fsync` и `ioctl`.

3.2. Дескрипторы файлов

Все открытые файлы представлены в ядре файловыми дескрипторами. Файловый дескриптор – это неотрицательное целое число. Когда процесс открывает существующий файл или создает новый, ядро возвращает ему файловый дескриптор. Чтобы выполнить запись в файл или чтение из него, нужно

передать функции read или write его файловый дескриптор, полученный в результате вызова функции open или creat.

В соответствии с принятыми соглашениями командные оболочки UNIX ассоциируют файловый дескриптор 0 со стандартным устройством ввода процесса, 1 – со стандартным устройством вывода и 2 – со стандартным устройством вывода сообщений об ошибках. Это соглашение используется командными оболочками и большинством приложений, но не является особенностью ядра UNIX. Тем не менее многие приложения не смогли бы работать, если это соглашение было бы нарушено.

В POSIX-совместимых приложениях вместо фактических значений 0, 1 и 2 следует использовать константы STDIN_FILENO, STDOUT_FILENO и STDERR_FILENO. Определения этих констант находятся в заголовочном файле <unistd.h>.

Под файловые дескрипторы отводится диапазон чисел от 0 до OPEN_MAX. (Вспомните табл. 2.10.) В ранних реализациях UNIX максимальным значением файлового дескриптора было число 19, что позволяло каждому процессу держать открытыми до 20 файлов, но многие системы увеличили это число до 63.

> В операционных системах FreeBSD 5.2.1, Mac OS X 10.3 и Solaris 9 этот предел практически бесконечен и ограничен лишь объемом памяти в системе, представлением целых чисел и прочими жесткими и мягкими ограничениями, задаваемыми администратором системы. Операционная система Linux 2.4.22 жестко ограничивает количество файловых дескрипторов на процесс числом 1 048 576.

3.3. Функция open

Создание или открытие файла производится функцией open.

```
#include <fcntl.h>

int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

Возвращает дескриптор файла в случае успеха, −1 в случае ошибки

Третий аргумент обозначен многоточием (...), таким способом стандарт ISO C указывает, что количество остальных аргументов и их типы могут варьироваться. В этой функции третий аргумент используется только при создании нового файла, о чем мы поговорим немного позже. Этот аргумент мы привели в прототипе функции как комментарий.

Аргумент *pathname* представляет имя файла, который будет открыт или создан. Эта функция может принимать большое количество параметров, которые определяются аргументом *oflag*. Значение этого аргумента формируется объединением по ИЛИ (OR) одной или более констант, определяемых в заголовочном файле <fcntl.h> и перечисленных ниже:

O_RDONLY Файл открывается только на чтение.

`O_WRONLY` Файл открывается только на запись.

`O_RDWR` Файл открывается как для чтения, так и для записи.

> В большинстве реализаций для сохранения совместимости с устаревшим программным обеспечением константа `O_RDONLY` определяется значением 0, `O_WRONLY` – 1 и `O_RDWR` – 2.

Должна быть указана одна и только одна из этих трех констант. Далее приводится список констант, присутствие которых в аргументе *oflag* необязательно:

`O_APPEND` Запись производится в конец файла. Более подробное описание этого флага мы дадим чуть позже, в разделе 3.11.

`O_CREAT` Если файл не существует, он будет создан. Этот флаг требует наличия третьего аргумента функции `open` (*mode*), который определяет значения битов прав доступа к создаваемому файлу. (В разделе 4.5, где рассказывается о правах доступа к файлу, мы увидим, как определяется значение аргумента *mode* и какое влияние на него оказывает значение `umask` процесса.)

`O_EXCL` Приводит к появлению ошибки, если файл уже существует и задан флаг `O_CREAT`. При такой комбинации флагов атомарно выполняется проверка существования файла и его создание, если файл не существует. Более подробно мы опишем атомарные операции в разделе 3.11.

`O_TRUNC` Если файл существует и успешно открывается на запись либо на чтение и запись, то его размер усекается до нуля.

`O_NOCTTY` Если аргумент *pathname* ссылается на файл терминального устройства, то это устройство не назначается управляющим терминалом вызывающего процесса. Об управляющих терминалах мы подробнее поговорим в разделе 9.6.

`O_NONBLOCK` Если аргумент *pathname* ссылается на именованный канал (FIFO), специальный блочный файл или специальный символьный файл, этот флаг задает неблокирующий режим открытия файла и последующих операций ввода-вывода. Мы опишем этот режим в разделе 14.2.

> В ранних выпусках System V появился флаг `O_NDELAY`. Он подобен флагу `O_NONBLOCK`, однако вносит двусмысленность в трактовку значения, возвращаемого функцией `read`. Использование флага `O_NODELAY` приводит к тому, что функция `read` возвращает значение 0 в случае отсутствия данных в именованном или неименованном канале или в файле устройства, но тогда возникает конфликт со значением 0, которое возвращается по достижении конца файла. В системах, основанных на SVR4, сохранилась поддержка флага `O_NODELAY` с устаревшей семантикой, однако все новые приложения должны использовать флаг `O_NONBLOCK`.

Следующие три флага также относятся к разряду необязательных. Они предназначены для поддержки синхронизированных операций ввода-вывода, определяемых стандартом Single UNIX Specification (а также POSIX.1):

`O_DSYNC` Каждый вызов функции `write` ожидает завершения физической операции ввода-вывода, но не ожидает, пока будут обновлены атрибуты файла, если они не влияют на возможность чтения только что записанных данных.

O_RSYNC Каждый вызов функции read приостанавливается до тех пор, пока не будут закончены ожидающие завершения операции записи в ту же самую часть файла.

O_SYNC Каждый вызов функции write ожидает завершения физической операции ввода-вывода, включая операцию обновления атрибутов файла. Мы будем использовать этот флаг в разделе 3.14.

Флаги O_DSYNC и O_SYNC очень похожи друг на друга, но все-таки чуть-чуть отличаются. Флаг O_DSYNC влияет на атрибуты файла, только если их необходимо обновить, чтобы отразить изменения в данных (например, обновить размер файла, если в файл были записаны дополнительные данные). При использовании флага O_SYNC данные и атрибуты всегда обновляются синхронно. При перезаписи существующей части файла, открытого с флагом O_DSYNC, атрибуты времени файла не будут обновляться синхронно с данными. Напротив, если файл открывается с флагом O_SYNC, каждое обращение к функции write будет приводить к изменению атрибутов времени файла независимо от того, были ли перезаписаны существующие данные или в конец файла добавлены новые.

ОС Solaris 9 поддерживает все три флага, в системах FreeBSD 5.2.1 и MacOS X 10.3 есть отдельный флаг (O_FSYNC), который имеет то же значение, что и O_SYNC. Поскольку оба эти флага полностью эквивалентны, FreeBSD 5.2.1 объявляет их с одним и тем же значением (однако Mac OS X 10.3 не определяет флаг O_SYNC). Ни FreeBSD 5.2.1, ни Mac OS X 10.3 не поддерживают флаги O_DSYNC и O_RSYNC. ОС Linux 2.4.22 трактует оба флага так же, как O_SYNC.

Функция open гарантирует, что возвращаемый ею дескриптор файла будет представлять собой наименьшее не используемое в качестве дескриптора положительное число. Это обстоятельство используется в некоторых приложениях для открытия нового файла вместо стандартного ввода, стандартного вывода или стандартного вывода сообщений об ошибках. Например, приложение может закрыть файл стандартного вывода (обычно это дескриптор 1) и затем открыть другой файл, зная, что он будет открыт с дескриптором 1. В разделе 3.12 мы продемонстрируем более надежный способ открытия файла на конкретном дескрипторе при помощи функции dup2.

Усечение имени файла и строки пути

Что произойдет, если конфигурационный параметр NAME_MAX определен со значением 14 и при этом мы попытаемся создать новый файл, имя которого состоит из 15 символов? Традиционно ранние версии System V, такие как SVR2, допускали это, просто усекая длину имени файла до 14 символов. BSD-системы возвращали признак ошибки с кодом ENAMETOOLONG в переменной errno. Простое усечение имени файла создает проблему, которая проявляет себя не только при создании нового файла. Так, если параметр NAME_MAX определен со значением 14 и существует файл с именем ровно из 14 символов, ни одна из функций, принимающих аргумент *pathname*, например open или stat, не имеет никакой возможности определить первоначальное имя файла, которое, возможно, было обрезано.

Конфигурационный параметр _POSIX_NO_TRUNC, предусматриваемый стандартом POSIX.1, определяет, усекаются ли слишком длинные имена файлов

и строки пути или возвращается признак ошибки. Как мы уже говорили в главе 2, значение этого параметра может варьироваться в зависимости от типа файловой системы.

> Возвращается признак ошибки или нет, во многом обусловлено историческими причинами. Так, например, операционные системы, базирующиеся на SVR4, не генерируют ошибку для традиционной файловой системы S5. Для файловой системы UFS те же самые системы возвращают признак ошибки.
>
> Другой пример (см. табл. 2.15): ОС Solaris сгенерирует ошибку для файловой системы UFS, но не для PCFS, которая совместима с файловой системой DOS, поскольку она «молча» усекает имена файлов, не соответствующие формату 8.3.
>
> BSD-системы и Linux всегда возвращают признак ошибки.

Когда параметр `_POSIX_NO_TRUNC` определен и полный путь к файлу превышает значение `PATH_MAX` или какой-либо компонент имени файла или строки пути превышает значение `NAME_MAX`, возвращается признак ошибки и в переменную `errno` записывается код ошибки `ENAMETOOLONG`.

3.4. Функция creat

Новый файл можно также создать с помощью функции `creat`.

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

В случае успеха возвращает файловый дескриптор, доступный только для записи, −1 в случае ошибки

Обратите внимание: эта функция эквивалентна

```
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

> В ранних версиях UNIX второй аргумент функции `open` мог принимать только три значения: 0, 1 или 2. Открыть несуществующий файл не было никакой возможности. Таким образом, для создания нового файла был необходим отдельный системный вызов. В настоящее время флаги `O_CREAT` и `O_TRUNC` обеспечивают функцию `open` необходимыми средствами для создания файлов, и потребность в функции `creat` отпала.

Порядок определения аргумента *mode* мы покажем в разделе 4.5, когда во всех подробностях будем описывать права доступа к файлам.

У функции `creat` есть один недостаток: файл открывается только на запись. До появления обновленной версии функции `open`, чтобы создать временный файл, записать в него некоторые данные и потом прочитать их, требовалось вызывать `creat`, `close` и затем `open`. Гораздо удобнее использовать в таких случаях функцию `open` следующим образом:

```
open(pathname, O_RDWR | O_CREAT | O_TRUNC, mode);
```

3.5. Функция close

Закрытие открытого файла производится обращением к функции close.

```
#include <unistd.h>

int close(int filedes);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Закрытие файла также приводит к снятию любых блокировок, которые могли быть наложены процессом. Мы обсудим этот вопрос в разделе 14.3.

При завершении процесса все открытые им файлы автоматически закрываются ядром. Многие приложения используют это обстоятельство и не закрывают файлы явным образом. Примером тому служит программа, представленная листингом 1.2.

3.6. Функция lseek

С любым открытым файлом связано такое понятие, как *текущая позиция файла*. Как правило, это неотрицательное целое число, которым выражается количество байт от начала файла. (Некоторые исключения, касающиеся слова «неотрицательное», будут упомянуты чуть позже.) Обычно операции чтения и записи начинают выполняться с текущей позиции файла и увеличивают ее значение на количество байт, которое было прочитано или записано. По умолчанию при открытии файла текущая позиция инициализируется числом 0, если не был установлен флаг O_APPEND.

Явное изменение текущей позиции файла выполняется с помощью функции lseek.

```
#include <unistd.h>

off_t lseek(int filedes, off_t offset, int whence);
```

Возвращает новую текущую позицию файла в случае успеха, −1 в случае ошибки

Интерпретация аргумента *offset* зависит от значения аргумента *whence*.

- Если аргумент *whence* имеет значение SEEK_SET, то *offset* интерпретируется как смещение от начала файла.
- Если аргумент *whence* имеет значение SEEK_CUR, то *offset* интерпретируется как смещение от текущей позиции файла. В этом случае *offset* может принимать как положительные, так и отрицательные значения.
- Если аргумент *whence* имеет значение SEEK_END, то *offset* интерпретируется как смещение от конца файла. В этом случае *offset* может принимать как положительные, так и отрицательные значения.

Поскольку в случае успеха функция lseek возвращает новую текущую позицию файла, мы можем задать в аргументе *offset* значение 0, чтобы узнать текущую позицию:

```
off_t  currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

Можно воспользоваться этим приемом, чтобы определить, имеется ли возможность свободного перемещения текущей позиции файла. Если файловый дескриптор относится к именованному или неименованному каналу или к сокету, функция lseek вернет значение –1 и запишет в переменную errno код ошибки ESPIPE.

> Символьные константы SEEK_SET, SEEK_CUR и SEEK_END изначально появились в System V. До этого аргумент *whence* мог принимать значения 0 (смещение от начала файла), 1 (смещение относительно текущей позиции) или 2 (смещение от конца файла). Весьма многие программы до сих пор используют эти предопределенные числовые значения.
>
> Буква l в названии функции lseek означает «long integer» (длинное целое). До введения типа данных off_t аргумент *offset* и возвращаемое значение имели тип long. Сама функция lseek впервые появилась в Version 7, когда в язык C был добавлен тип длинных целых чисел. (В Version 6 была похожая функциональность, которая обеспечивалась функциями seek и tell.)

Пример

Программа, представленная в листинге 3.1, проверяет возможность свободного перемещения текущей позиции в файле стандартного ввода.

Листинг 3.1. Проверка возможности свободного перемещения текущей позиции в файле стандартного ввода

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("перемещение невозможно\n");
    else
        printf("перемещение выполнено\n");
exit(0);
}
```

Запустив эту программу, мы получим следующее

```
$ ./a.out < /etc/motd
перемещение выполнено
$ cat < /etc/motd | ./a.out
перемещение невозможно
$ ./a.out < /var/spool/cron/FIFO
перемещение невозможно
```

Обычно смещение относительно текущей позиции должно быть неотрицательным целым числом. Однако некоторые устройства допускают использо-

вание отрицательных смещений. Но для обычных файлов смещение должно быть неотрицательным. Поскольку отрицательные смещения все-таки возможны, возвращаемое функцией `lseek` значение следует сравнивать именно с числом –1, а не проверять, не является ли оно отрицательным.

> В ОС FreeBSD на платформе Intel x86 устройство `/dev/kmem` поддерживает отрицательные смещения.
>
> Поскольку тип `off_t` является целым числом со знаком (табл. 2.16), теряется половина возможного максимального размера файла. Так, если `off_t` представляет собой 32-битное целое со знаком, то максимальный размер файла будет равен $2^{31}-1$ байт.

Функция `lseek` изменяет значение текущей позиции файла лишь в области данных ядра – фактически она не выполняет никаких операций ввода-вывода. Это значение текущей позиции будет использовано ближайшей операцией чтения или записи.

Текущая позиция файла может превышать его текущий размер. В этом случае следующая операция записи увеличит размер файла. Это вполне допустимо и может рассматриваться как создание «дырки» в файле. Байты, которые фактически не были записаны, считываются как нули. «Дырка» в файле не обязательно должна занимать место на диске. В некоторых файловых системах в случае переноса текущей позиции за пределы файла на диске могут быть выделены новые блоки для данных, но это совершенно необязательно.

Пример

Программа из листинга 3.2 создает файл с «дыркой».

Листинг 3.2. Создание файла с «дыркой»

```
#include "apue.h"
#include <fcntl.h>

char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int  fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("ошибка вызова creat");

    if (write(fd, buf1, 10) != 10)
        err_sys("ошибка записи buf1");
    /* теперь текущая позиция = 10 */

    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("ошибка вызова lseek");
    /* теперь текущая позиция = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("ошибка записи buf2");
```

```
    /* теперь текущая позиция = 16394 */

    exit(0);
}
```

Запустив эту программу, мы получим следующее

```
$ ./a.out
$ ls -l file.hole                проверим размер файла
-rw-r--r-- 1 sar      16394 Nov 25 01:01 file.hole
$ od -c file.hole                посмотрим его фактическое содержимое
0000000 a b c d e f g h i j \0 \0 \0 \0 \0 \0
0000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0040000 A B C D E F G H I J
0040012
```

Чтобы просмотреть содержимое файла, мы воспользовались командой `od`(1). Флаг `-c` сообщает ей о том, что содержимое следует выводить в виде символов. Мы видим, что байты, которые не были фактически записаны, читаются как нули. Семизначные числа в начале каждой строки – это смещение от начала файла в восьмеричном виде.

Чтобы убедиться в том, что в файле действительно имеется «дырка», сравним только что созданный файл с файлом того же размера, но без «дырки»:

```
$ ls -ls file.hole file.nohole          сравним размеры
  8 -rw-r--r--  1 sar    16394 Nov 25 01:01 file.hole
 20 -rw-r--r--  1 sar    16394 Nov 25 01:03 file.nohole
```

Несмотря на то, что файлы имеют одинаковый размер, файл без «дырки» занимает 20 дисковых блоков, в то время как файл с «дыркой» – всего 8.

В этом примере мы вызывали функцию `write` (раздел 3.8). О файлах с «дырками» мы еще поговорим в разделе 4.12.

Поскольку для представления смещения функция `lseek` использует тип `off_t`, реализации поддерживают тот размер, который определен для конкретной платформы. Большинство современных платформ предоставляют два набора интерфейсов для работы со смещением в файле: 32-битный и 64-битный.

Стандарт Single UNIX Specification предоставляет приложениям возможность с помощью функции `sysconf` (раздел 2.5.4) определить интерфейсы, которые поддерживаются системой. В табл. 3.1 приводится список констант, передаваемых функции `sysconf`.

Компилятор `c99` требует, чтобы желаемая модель размерностей была отображена во флаги компиляции и связывания с помощью команды `getconf`(1). В зависимости от конкретной реализации могут потребоваться различные флаги и библиотеки.

> К сожалению, это одна из тех областей, в которых реализации отстают от стандартов. Хуже того, имена констант не совпадают в версиях 2 и 3 Single UNIX Specification.

Чтобы обойти эти препятствия, приложение может присвоить константе _FILE_OFFSET_BITS значение 64. Тогда тип off_t будет определен как 64-битное целое со знаком. Установив константу _FILE_OFFSET_BITS равной 32, мы сможем работать с 32-битными смещениями. Как 32-битные, так и 64-битные смещения поддерживаются на всех четырех платформах, обсуждаемых в данной книге, однако следует помнить, что определение константы _FILE_OFFSET_BITS не гарантирует сохранения переносимости приложений.

Таблица 3.1. Размеры типов данных и имена констант, передаваемые функции sysconf

Имя конфигурационного параметра	Описание	Значение аргумента *name*
_POSIX_V6_ILP32_OFF32	Типы int, long, указатели и off_t представлены 32 битами	_SC_V6_ILP32_OFF32
_POSIX_V6_ILP32_OFFBIG	Типы int, long и указатели представлены 32 битами, тип off_t имеет размер не менее 64 бит	_SC_V6_ILP32_OFFBIG
_POSIX_V6_LP64_OFF64	Тип int представлен 32 битами, типы long, указатели и off_t имеют размер 64 бита	_SC_V6_LP64_OFF64
_POSIX_V6_LP64_OFFBIG	Тип int представлен 32 битами, типы long, указатели и off_t имеют размер не менее 64 бит	_SC_V6_LP64_OFFBIG

Обратите внимание: даже если у вас установлены 64-битные смещения, возможность создания файлов размером более 2Тб ($2^{31}-1$ байт) зависит от реализации файловой системы.

3.7. Функция read

Чтение данных из открытого файла производится функцией read.

```
#include <unistd.h>

ssize_t read(int filedes, void *buf, size_t nbytes);
```

Возвращает количество прочитанных байт,
0 – если достигнут конец файла, –1 в случае ошибки

В случае успеха функция read возвращает количество прочитанных байт. Если был достигнут конец файла, возвращается 0.

Существует несколько ситуаций, когда количество фактически прочитанных байт меньше, чем было запрошено:

- При чтении из обычного файла, когда конец файла встретился до того, как было прочитано требуемое количество байт. Например, если до конца файла осталось 30 байт, а запрошено было 100 байт, то функция read вернет число 30. При следующем вызове она вернет 0 (конец файла).

- При чтении с терминального устройства. Обычно за одно обращение читается одна строка. (В главе 18 мы увидим, как это можно изменить.)
- При чтении данных из сети. Промежуточная буферизация в сети может стать причиной того, что будет получено меньшее количество байт, чем было запрошено.
- При чтении из именованных или неименованных каналов. Если в канале содержится меньше байт, чем было запрошено, функция `read` вернет только то, что ей будет доступно.
- При чтении с устройства, ориентированного на доступ к отдельным записям. Примером такого устройства является накопитель на магнитной ленте, который может вернуть только одну запись за одно обращение.
- При прерывании операции чтения сигналом в тот момент, когда часть данных уже была прочитана. Эту ситуацию мы обсудим подробнее в разделе 10.5.

Операция чтения начинается с текущей позиции файла. В случае успеха текущая позиция будет увеличена на число фактически прочитанных байт.

Стандарт POSIX.1 изменил прототип функции `read`. Классическое определение этой функции выглядит следующим образом:

```
int read(int filedes, char *buf, unsigned nbytes);
```

- Во-первых, тип второго аргумента (`char*`) был изменен на `void*` для совместимости со стандартом ISO C: тип `void*` используется для определения нетипизированных указателей.
- Далее, возвращаемое значение должно быть целым числом со знаком (`ssize_t`), чтобы была возможность возвращать положительное число (количество прочитанных байт), 0 (признак конца файла) или –1 (признак ошибки).
- И наконец, третий аргумент исторически был целым числом без знака, что позволяло в 16-битных реализациях читать и записывать до 65534 байт за одно обращение. С появлением стандарта POSIX.1 от 1990 года были введены новые типы данных: `ssize_t` для представления возвращаемого значения как целого со знаком и `size_t`, целое без знака, для представления третьего аргумента. (Вспомните константу `SSIZE_MAX` из раздела 2.5.2.)

3.8. Функция write

Запись данных в открытый файл производится функцией `write`.

```
#include <unistd.h>
ssize_t write(int filedes, const void *buf, size_t nbytes);
```

Возвращает количество записанных байт
в случае успеха, –1 в случае ошибки

Возвращаемое значение обычно совпадает со значением аргумента *nbytes*, в противном случае возвращается признак ошибки. Наиболее распространенные случаи, когда возникает ошибка записи, – это переполнение диска или превышение ограничения на размер файла для заданного процесса (раздел 7.11 и упражнение 10.11).

Для обычных файлов запись начинается с текущей позиции файла. Если при открытии файла был указан флаг `O_APPEND`, текущая позиция устанавливается в конец файла перед началом каждой операции записи. По окончании записи значение текущей позиции увеличивается на количество фактически записанных байт.

3.9. Эффективность операций ввода-вывода

Программа из листинга 3.3 выполняет копирование файлов, используя функции `read` и `write`. К этой программе необходимо сделать несколько пояснений.

- Если чтение производится из файла стандартного ввода, а запись – в файл стандартного вывода, предполагается, что они были должным образом открыты командной оболочкой до запуска программы. В действительности все командные оболочки UNIX, как правило, предоставляют возможность открытия файла для чтения на стандартном устройстве ввода и создания (или перезаписи) файла на стандартном устройстве вывода. Это освобождает программы от необходимости открывать входной и выходной файлы.

Листинг 3.3. Копирование со стандартного ввода на стандартный вывод

```
#include "apue.h"

#define BUFFSIZE 4096

int
main(void)
{
    int n;
    char buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("ошибка записи");

    if (n < 0)
        err_sys("ошибка чтения");

    exit(0);
}
```

- Большинство приложений предполагают, что с файлом стандартного ввода связан дескриптор 0, а с файлом стандартного вывода – дескриптор 1. В этой программе используются константы `STDIN_FILENO` и `STDOUT_FILENO`, определения которых находятся в файле `<unistd.h>`.

- Программа не закрывает входной и выходной файлы. Все открытые дескрипторы закрываются ядром UNIX по завершении процесса, и она пользуется этим обстоятельством.
- Этот пример одинаково хорошо работает как с текстовыми, так и с двоичными файлами, поскольку ядро не делает никаких различий между этими двумя форматами.

Еще один вопрос, на который нам предстоит ответить: как было выбрано значение константы `BUFFSIZE`. Прежде чем дать на него ответ, давайте попробуем запустить программу с различными значениями `BUFFSIZE`. В табл. 3.2 приведены результаты чтения файла размером 103 316 352 байта с использованием 20 различных размеров буфера.

Таблица 3.2. Производительность операции чтения с различными размерами буфера в ОС Linux

BUFFSIZE	Пользовательское время (секунды)	Системное время (секунды)	Общее время (секунды)	Количество циклов
1	124,89	161,65	288,64	103 316 352
2	63,10	80,96	145,81	51 658 176
4	31,84	40,00	72,75	25 829 088
8	15,17	21,01	36,85	12 914 544
16	7,86	10,27	18,76	6 457 272
32	4,13	5,01	9,76	3 228 636
64	2,11	2,48	6,76	1 614 318
128	1,01	1,27	6,82	807 159
256	0,56	0,62	6,80	403 579
512	0,27	0,41	7,03	201 789
1 024	0,17	0,23	7,84	100 894
2 048	0,05	0,19	6,82	50 447
4 096	0,03	0,16	6,86	25 223
8 192	0,01	0,18	6,67	12 611
16 384	0,02	0,18	6,87	6 305
32 768	0,00	0,16	6,70	3 152
65 536	0,02	0,19	6,92	1 576
131 072	0,00	0,16	6,84	788
262 144	0,01	0,25	7,30	394
524 288	0,00	0,22	7,35	198

Файл читался программой из листинга 3.3 с перенаправлением стандартного вывода на устройство /dev/null. В эксперименте участвовала файловая система Linux ext2 с размером дискового блока 4096 байт. (Значение st_blksize, которое мы рассмотрим в разделе 4.12, составляет 4096.) Это объясняет, почему наименьшее системное время приходится именно на этот размер BUFFSIZE. Дальнейшее увеличение буфера дает лишь незначительный положительный эффект.

Большинство файловых систем для повышения производительности поддерживают возможность опережающего чтения. Обнаружив ряд последовательных операций чтения, система пытается прочитать больший объем данных, чем было запрошено приложением, предполагая, что программа вскоре продолжит чтение. Из последних строк табл. 3.2 видно, что опережающее чтение в ext2 перестает играть какую-либо роль при размерах буфера более 128 Кб.

Позднее мы еще вернемся к этой таблице. В разделе 3.14 мы покажем результат выполнения операции синхронной записи, в разделе 5.8 сравним время выполнения операций небуферизованного ввода-вывода и функций стандартной библиотеки ввода-вывода.

> Будьте внимательны, проводя эксперименты по определению производительности программ, работающих с файлами. Операционная система попытается кэшировать файл в оперативной памяти (*incore*), поэтому при проведении серии экспериментов с одним и тем же файлом каждый последующий результат, скорее всего, будет лучше самого первого. Это происходит потому, что первая операция ввода-вывода поместит файл в системный кэш и каждый последующий прогон программы будет получать данные из кэша, а не с диска. (Термин *incore* означает *оперативную память*. Много лет назад оперативная память компьютеров строилась на магнитных ферритовых сердечниках (по-английски *core*). Отсюда же взялся и термин *core dump* (дамп памяти) – образ оперативной памяти программы, сохраненный в файле на диске для последующего анализа.)
>
> В эксперименте, результаты которого показаны в табл. 3.2, участвовали различные копии файла, и поэтому использование системного кэша было сведено к минимуму. Размер этих файлов достаточно велик, так что они не могут одновременно находиться в кэше (тестовая система имела в своем распоряжении 512 Мб ОЗУ).

3.10. Совместное использование файлов

ОС UNIX поддерживает совместное использование открытых файлов несколькими процессами. Нам необходимо разобраться с этой возможностью, прежде чем мы перейдем к описанию функции dup. Для этого мы рассмотрим структуры данных, которые используются ядром при выполнении всех операций ввода-вывода.

> Далее следует лишь концептуальное описание, которое может совпадать, а может и не совпадать с конкретной реализацией. За описанием структур в System V вам следует обращаться к [Bach 1986]. В [McKusick et al. 1996] описаны те же структуры применительно к 4.4BSD. В [McKusick and Neville-Neil 2005] рассматривается FreeBSD 5.2. Аналогичное описание для Solaris вы найдете в [Mauro and McDougall 2001].

Ядро использует три структуры данных для представления открытого файла, а отношения между ними определяют взаимовлияние процессов при совместном использовании файлов.

1. Каждому процессу соответствует запись в таблице процессов. С каждой записью в таблице процессов связана таблица открытых файловых дескрипторов, которую можно представить как таблицу, в которой каждая строка соответствует одному файловому дескриптору. Для каждого дескриптора хранится следующая информация:
   a. Флаги дескриптора (флаг close-on-exec («закрыть-при-вызове-exec»), см. рис. 3.1 и раздел 3.14).
   b. Указатель на запись в таблице файлов.
2. Все открытые файлы представлены в ядре таблицей файлов. Каждая запись в таблице содержит:
   a. Флаги состояния файла, такие как чтение, запись, добавление в конец файла, синхронный режим операций ввода-вывода, неблокирующий режим (подробнее эти флаги будут описаны в разделе 3.14).
   b. Текущая позиция файла.
   c. Указатель на запись в таблице виртуальных узлов (v-node).
3. Каждому открытому файлу соответствует структура виртуального узла (v-node), в которой хранится информация о типе файла и указатели для функций, работающих с файлом. Для большинства файлов структура v-node также содержит индексный узел (i-node) файла. Эта информация считывается с диска при открытии файла, так что вся информация о файле сразу же становится доступной. Индексный узел (i-node) содержит, например, сведения о владельце файла, размере файла, указатели на блоки данных файла на диске и тому подобное. (Более подробно об индексных узлах мы поговорим в разделе 4.14 при описании типичной файловой системы UNIX.)

   В ОС Linux отсутствует понятие виртуальных узлов (v-node). Вместо него используются структуры индексных узлов (i-node). Хотя реализация их различна, концептуально они представляют собой одно и то же. В обоих случаях индексный узел хранит информацию, специфичную для конкретной файловой системы.

Мы опустим некоторые особенности отдельных реализаций, не имеющие для нас большого значения. Например, таблица открытых дескрипторов может храниться не в таблице процессов, а в пространстве пользователя. Сами таблицы могут быть реализованы различными способами: они не обязательно должны быть массивами, вместо этого они могут быть оформлены в виде связанных списков структур. Все эти подробности несущественны для нашего обсуждения совместного доступа к файлам.

На рис. 3.1 показаны все три таблицы для одного процесса, которым открыты два файла, а именно файл стандартного ввода (дескриптор 0) и файл стандартного вывода (дескриптор 1). Взаимоотношения между таблицами определились начиная еще с ранних версий UNIX [Thompson 1978], и они оказы-

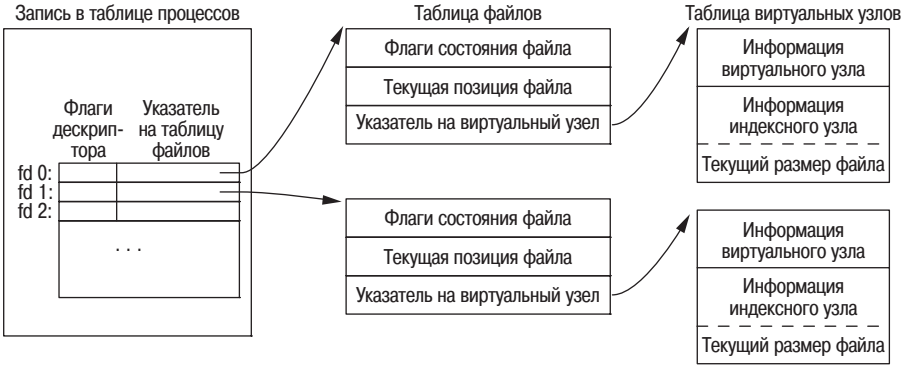


Рис. 3.1. *Структуры данных ядра для открытых файлов*

вают весьма существенное влияние на способ совместного использования одного файла несколькими процессами. Мы еще вернемся к этому рисунку в последующих главах, когда будем обсуждать другие способы совместного использования файлов.

> Концепция виртуального узла (v-node) была придумана для того, чтобы обеспечить поддержку нескольких типов файловых систем в рамках одной операционной системы. Эта работа была проделана независимо Питером Вейнбергером (Peter Weinberger) из Bell Labaratories и Биллом Джоем (Bill Joy) из Sun Microsystems. В Sun эта концепция получила название Virtual File System (виртуальная файловая система), а часть индексного узла (i-node), не зависящая от типа файловой системы, была названа виртуальным узлом (v-node) [Kleiman 1986]. Концепция виртуальных узлов распространилась на различные реализации UNIX вместе с поддержкой Network File System (NFS – сетевая файловая система) компании Sun. Первой версией из Беркли, поддерживающей виртуальные узлы, стала 4.3BSD Reno, в которую была добавлена поддержка NFS.
>
> В SVR4 виртуальные узлы заменили индексные узлы версии SVR3. ОС Solaris, как наследник SVR4, также использует концепцию виртуальных узлов.
>
> Вместо разделения структур данных на виртуальные и индексные узлы, в Linux используются понятия индексных узлов, не зависимых от типа файловой системы, и индексных узлов, зависящих от типа файловой системы.

Ситуация, когда два независимых процесса открывают один и тот же файл, показана на рис. 3.2. Здесь мы предполагаем, что первый процесс открывает этот файл с дескриптором 3, а второй процесс открывает тот же самый файл с дескриптором 4. Каждый процесс, открывающий файл, создает собственную запись в таблице файлов, но двум этим записям соответствует единственная запись в таблице виртуальных узлов. Одна из причин для создания отдельной записи в таблице файлов для каждого процесса состоит в том, что у каждого процесса должна быть собственная текущая позиция файла.

Теперь, разобравшись с этими структурами данных, рассмотрим более подробно, что происходит в процессе описанных выше операций ввода-вывода.

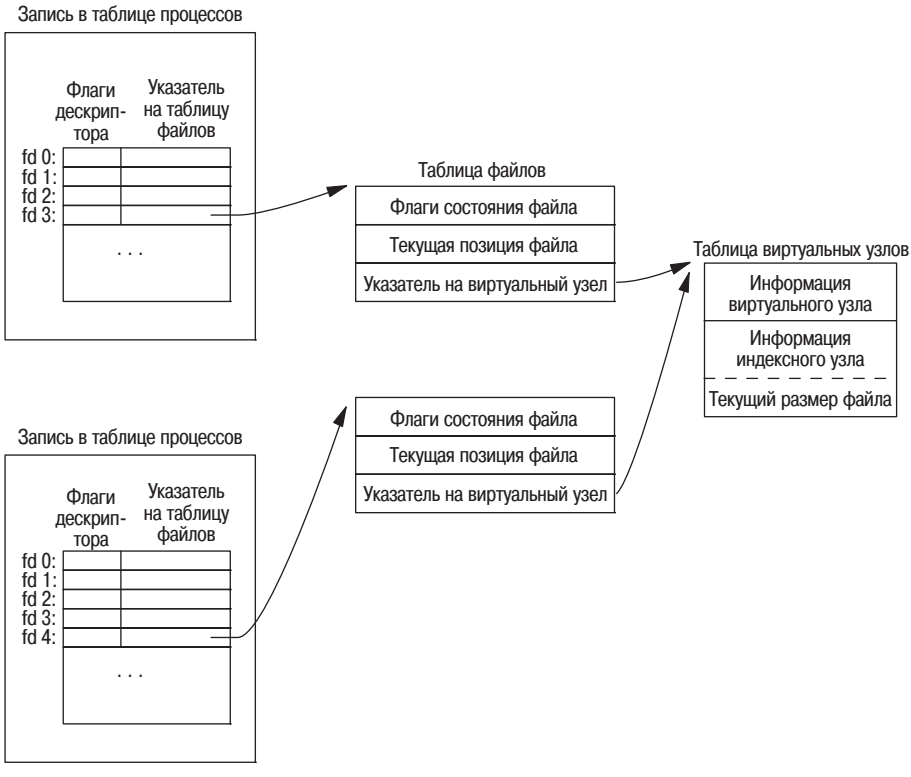


Рис. 3.2. *Два независимых процесса открыли один и тот же файл*

- В таблице файлов после завершения каждой операции записи текущая позиция файла увеличивается на количество записанных байт. В том случае, если текущая позиция файла оказывается больше текущего размера файла, в таблице индексных узлов изменяется размер файла в соответствии с текущей позицией (это происходит, например, при добавлении новых данных в конец файла).
- Если файл был открыт с флагом `O_APPEND`, соответствующий флаг устанавливается в таблице файлов. Каждый раз при выполнении операции записи в качестве текущей позиции принимается значение размера файла из таблицы индексных узлов. В результате запись всегда производится в конец файла.
- Если текущая позиция переносится в конец файла с помощью функции `lseek`, выполняется только перезапись значения текущего размера файла из таблицы индексных узлов в поле текущей позиции в таблице файлов. (Обратите внимание: это не то же самое, что открытие файла с флагом `O_APPEND`, о чем мы будем говорить в разделе 3.11.)
- Функция `lseek` изменяет только значение текущей позиции файла в таблице файлов. Никаких операций ввода-вывода при этом не производится.

Существует возможность открыть несколько дескрипторов, которые будут ссылаться на одну и ту же запись в таблице файлов; мы увидим это в разделе 3.12 при обсуждении функции dup. То же самое происходит в результате вызова функции fork, когда родительский и дочерний процессы совместно используют одни и те же записи в таблице файлов для каждого из открытых дескрипторов (раздел 8.3).

Обратите внимание на различия, которые существуют между флагами дескриптора и флагами состояния файла. Флаги дескриптора уникальны для каждого отдельно взятого дескриптора, открытого процессом, тогда как флаги состояния файла имеют отношение ко всем дескрипторам в любом процессе, которые ссылаются на одну и ту же запись в таблице файлов. Рассматривая функцию fcntl в разделе 3.14, мы узнаем, как можно получить и изменить значения флагов дескриптора и флагов состояния файла.

Все описанное ранее в этом разделе прекрасно работает в том случае, когда несколько процессов читают данные из одного и того же файла. Каждый процесс имеет собственную запись в таблице файлов со своим собственным значением текущей позиции файла. Однако можно получить совершенно неожиданные результаты, если несколько процессов попытаются выполнить запись данных в один и тот же файл. Чтобы избежать в будущем неприятных сюрпризов, мы должны разобраться с понятием атомарности операций.

3.11. Атомарные операции

Добавление данных в конец файла

Рассмотрим процесс, который дописывает данные в конец файла. Старые версии UNIX не поддерживали флаг O_APPEND для функции open, в результате приходилось писать нечто вроде:

```
if (lseek(fd, 0L, 2) < 0)         /* переместить текущую позицию в конец файла */
   err_sys("ошибка вызова функции lseek");
if (write(fd, buf, 100) != 100)              /* и выполнить запись */
   err_sys("ошибка вызова функции write");
```

Такой код будет прекрасно работать в случае единственного процесса, но могут возникнуть определенные проблемы, если добавление данных в конец файла производится сразу несколькими процессами. (Подобная ситуация возможна, например, когда несколько процессов добавляют сообщения в файл журнала.)

Допустим, существуют два независимых процесса A и B, которые выполняют запись данных в конец одного и того же файла. Каждый из процессов открыл файл, но без флага O_APPEND. Эта ситуация изображена на рис. 3.2. Каждый процесс имеет собственную запись в таблице файлов, но при этом они ссылаются на одну и ту же запись в таблице виртуальных узлов. Предположим, что процесс A вызывает функцию lseek и устанавливает текущую позицию файла в значение 1500 (текущий размер файла). Затем ядро приостанавливает работу процесса A и передает управление процессу B, который

в свою очередь также вызывает функцию lseek и также устанавливает текущую позицию файла в значение 1500 (текущий размер файла). После этого процесс B вызывает функцию write, которая увеличивает текущую позицию файла до 1600. Поскольку размер файла был увеличен, ядро записывает новое значение размера файла (1600) в таблицу виртуальных узлов. После этого ядро опять переключает процессы и передает управление процессу A. Когда процесс A вызывает функцию write, запись будет выполняться с места, на которое указывает значение текущей позиции файла для процесса A, то есть 1500. В результате данные окажутся записанными поверх тех, что записаны процессом B.

Проблема в том, что операция «перейти в конец файла и записать данные» требует обращения к двум отдельным функциям (как мы только что показали). Решением проблемы было бы атомарное[1] выполнение операции позиционирования и записи. Любая операция, которая требует обращения более чем к одной функции, не может быть атомарной, поскольку всегда существует вероятность того, что ядро временно приостановит процесс между двумя последовательными вызовами функций (как это было показано выше).

ОС UNIX предоставляет возможность атомарного выполнения этой операции, если мы укажем флаг O_APPEND при открытии файла. Как мы уже говорили в предыдущем разделе, этот флаг заставит ядро выполнять перенос текущей позиции в конец файла непосредственно перед операцией записи. А кроме того, отпадает необходимость вызывать функцию lseek перед каждым вызовом функции write.

Функции pread и pwrite

Стандарт Single UNIX Specification включает в себя расширения XSI, которые позволяют процессам атомарно выполнять операции перемещения текущей позиции и ввода-вывода. Эти расширения представлены функциями pread и pwrite.

```
#include <unistd.h>

ssize_t pread(int filedes, void *buf, size_t nbytes, off_t offset);
```

Возвращает количество прочитанных байт, 0 по достижении конца файла и −1 в случае ошибки

```
ssize_t pwrite(int filedes, const void *buf, size_t nbytes, off_t offset);
```

Возвращает количество записанных байт или −1 в случае ошибки

Вызов функции pread эквивалентен двум последовательным вызовам функций lseek и read со следующими отличиями:

- При использовании pread нет возможности прервать выполнение этих двух операций

[1] То есть неделимое. – *Примеч. перев.*

- Значение текущей позиции файла не изменяется

Вызов функции pwrite эквивалентен двум последовательным вызовам функций lseek и write с аналогичными отличиями.

Создание файла

Пример еще одной атомарной операции мы видели при описании флагов O_CREAT и O_EXCL функции open. При одновременном указании обоих флагов функция open будет завершаться ошибкой, если файл уже существует. Мы также говорили, что проверка существования файла и создание файла будут выполняться атомарно. Если бы не было такой атомарной операции, мы могли бы попробовать написать нечто вроде:

```
if ((fd = open(pathname, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(pathname, mode)) < 0)
            err_sys("ошибка вызова функции creat");
    } else {
        err_sys("ошибка вызова функции open");
    }
}
```

Эта ситуация чревата проблемами, если файл с тем же именем будет создан другим процессом между обращениями к функциям open и creat. Если другой процесс создаст файл между вызовами этих функций и успеет туда что-либо записать, то эти данные будут утеряны, когда первый процесс вызовет функцию creat. Объединение проверки существования файла и его создания в единую атомарную операцию решает эту проблему.

Вообще говоря, термин *атомарная операция* относится к таким операциям, которые могут состоять из нескольких действий. Если операция атомарна, то либо все необходимые действия будут выполнены до конца, либо не будет выполнено ни одно из них. Атомарность не допускает выполнения лишь некоторой части действий. К теме атомарных операций мы еще вернемся, когда будем рассматривать функцию link (раздел 4.15) и блокировку отдельных записей в файле (раздел 14.3).

3.12. Функции dup и dup2

Дубликат дескриптора существующего файла можно создать с помощью одной из следующих функций:

```
#include <unistd.h>

int dup(int filedes);

int dup2(int filedes, int filedes2);
```

Возвращают новый дескриптор файла или −1 в случае ошибки

Функция dup гарантирует, что возвращаемый ею новый файловый дескриптор будет иметь наименьшее возможное значение. При вызове функции dup2 мы указываем значение нового дескриптора в аргументе *filedes2*. Если дескриптор *filedes2* перед вызовом функции уже был открыт, то он предварительно закрывается. Если значения аргументов *filedes* и *filedes2* эквивалентны, то функция dup2 вернет дескриптор filedes2, не закрывая его.

Новый файловый дескриптор, возвращаемый функциями, будет ссылаться на ту же самую запись в таблице файлов, что и дескриптор *filedes*. Продемонстрируем это на рис. 3.3.

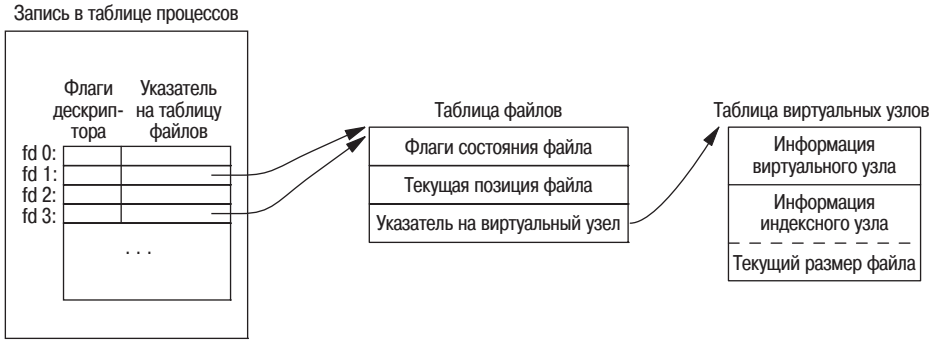


***Рис. 3.3.** Структуры в ядре после вызова функции dup(1)*

На рисунке предполагается, что процесс при запуске выполняет код

```
newfd = dup(1);
```

Предполагается, что следующий доступный дескриптор – это число 3 (что наиболее вероятно, потому что командная оболочка уже открыла для процесса дескрипторы 0, 1 и 2). Поскольку оба дескриптора указывают на одну и ту же запись в таблице файлов, они совместно будут использовать флаги состояния файла – чтение, запись, добавление в конец файла и прочие, и текущая позиция файла будет для них также одинаковой.

Каждый из дескрипторов будет иметь свой собственный набор флагов дескриптора. Как мы увидим в следующем разделе, функция dup всегда сбрасывает флаг close-on-exec («закрыть-при-вызове-exec») в новом дескрипторе.

Дубликат дескриптора можно создать также с помощью функции fcntl, которая будет описана в разделе 3.14. На самом деле вызов

```
dup(filedes);
```

эквивалентен вызову

```
fcntl(filedes, F_DUPFD, 0);
```

Аналогично вызов

```
dup2(filedes, filedes2);
```

эквивалентен вызову

```
close(filedes2);
fcntl(filedes, F_DUPFD, filedes2);
```

В последнем случае для функции dup2 приведен не совсем точный эквивалент из двух последовательных вызовов функций close и fcntl. Имеются следующие различия:

1. Функция dup2 представляет собой атомарную операцию, тогда как альтернативная форма состоит из обращений к двум функциям. В результате возникает вероятность того, что между обращениями к функциям close и fcntl будет вызван обработчик сигнала, который изменит дескриптор файла. (Сигналы будут обсуждаться в главе 10.)
2. Существуют некоторые отличия в кодах ошибок, возвращаемых через errno функциями dup2 и fcntl.

Системный вызов dup2 впервые появился в Version 7 и затем перекочевал в BSD. Возможность создания дубликатов дескрипторов с помощью fcntl появилась в System III и перешла в System V. В SVR3.2 была включена функция dup2, а в 4.2BSD – функция fcntl и функциональность F_DUPFD. Стандарт POSIX.1 требует как наличия функции dup2, так и поддержки функцией fcntl параметра F_DUPFD.

3.13. Функции sync, fsync и fdatasync

Традиционные реализации UNIX имеют в своем распоряжении буферный кэш или кэш страниц, через который выполняется большинство дисковых операций ввода-вывода. Когда мы записываем данные в файл, они, как правило, сначала помещаются ядром в один из буферов, а затем ставятся в очередь для записи на диск в более позднее время. Этот прием называется *отложенной записью*. (В главе 3 [Bach 1986] детально рассматривается работа буферного кэша.)

Ядро обычно записывает отложенные данные на диск, когда возникает необходимость в повторном использовании буфера. Для синхронизации файловой системы на диске и содержимого буферного кэша существуют функции sync, fsync и fdatasync.

```
#include <unistd.h>

int fsync(int filedes);

int fdatasync(int filedes);
```

Возвращают значение 0 в случае успеха, −1 в случае ошибки

```
void sync(void);
```

Функция sync просто ставит все измененные блоки буферов в очередь для записи и возвращает управление – она не ждет, пока физически будет выполнена запись на диск.

Функция sync, как правило, вызывается периодически (обычно каждые 30 секунд) из системного демона, часто называемого update. Это обеспечивает регулярную очистку буферного кэша ядра. Команда sync(1) также обращается к функции sync.

Функция fsync применяется только к одному файлу, который определяется файловым дескриптором *filedes*, кроме того, она ожидает завершения физической записи данных на диск, прежде чем вернуть управление. В основном функция fsync предназначена для таких приложений, как базы данных, чтобы гарантировать запись измененных блоков с данными на диск.

Функция fdatasync похожа на функцию fsync, но воздействует только на содержимое файла. (При использовании fsync также синхронно обновляются атрибуты файла.)

> Все четыре платформы, обсуждаемые в книге, поддерживают функции sync и fsync. Однако функция fdatasync не поддерживается в FreeBSD 5.2.1 и Mac OS X 10.3.

3.14. Функция fcntl

С помощью функции fcntl можно изменять свойства уже открытого файла.

```
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* int arg */);
```

Возвращаемое значение зависит от аргумента *cmd* (см. ниже) в случае успеха, −1 в случае ошибки

В примерах этого раздела третий аргумент всегда будет представлен целым числом – в соответствии с комментарием в приведенном прототипе функции. Однако при обсуждении блокировки записей в разделе 14.3 третий аргумент будет представлять собой указатель на структуру.

Функция fcntl используется в пяти различных случаях.

1. Создание дубликата существующего дескриптора (*cmd* = F_DUPFD).
2. Получение/установка флагов дескриптора (*cmd* = F_GETFD или F_SETFD).
3. Получение/установка флагов состояния файла (*cmd* = F_GETFL или F_SETFL).
4. Проверка/установка владельца для асинхронных операций ввода-вывода (*cmd* = F_GETOWN или F_SETOWN).
5. Получение/установка блокировки на отдельную запись в файле (*cmd* = F_GETLK, F_SETLK или F_SETLKW).

Теперь мы рассмотрим первые семь значений аргумента *cmd* из десяти возможных. (Описание остальных трех, связанных с блокировкой записей, отложим до раздела 14.3.) Взгляните еще раз на рис. 3.1, так как мы будем ссылаться как на флаги дескрипторов файлов, связанные с каждым дескриптором в таблице дескрипторов процесса, так и на флаги состояния файла, связанные с каждым файлом в таблице файлов.

F_DUPFD Создает дубликат дескриптора *filedes*. Новый файловый дескриптор передается в вызывающую программу в виде возвращаемого значения. Это будет наименьший неиспользованный дескриптор, значение которого больше или равно третьему аргументу (заданному в виде целого числа). Новый дескриптор будет ссылаться на ту же запись в таблице файлов, что и *filedes* (рис. 3.3). Но при этом новый дескриптор будет иметь свой собственный набор флагов, а флаг FD_CLOEXEC будет сброшен. (Это означает, что дескриптор останется открытым после вызова функции exec, которая обсуждается в главе 8.)

F_GETFD Передает в вызывающую программу флаги дескриптора *filedes* в виде возвращаемого значения. В настоящее время определен только один флаг – FD_CLOEXEC.

F_SETFD Устанавливает флаги дескриптора *filedes*. Новые значения флагов берутся из третьего аргумента (заданного в виде целого числа).

Вы должны знать, что существуют программы, которые работают с флагами дескрипторов, но не используют константу FD_CLOEXEC. Вместо этого они используют значение 0 (сбросить флаг FD_CLOEXEC) или 1 (установить флаг FD_CLOEXEC).

F_GETFL Передает в вызывающую программу флаги состояния файла *filedes* в виде возвращаемого значения. Мы уже описывали флаги состояния файла, когда обсуждали функцию open. Они перечислены в табл. 3.3.

Таблица 3.3. Флаги состояния файла, используемые функцией fcntl

Флаг состояния файла	Описание
O_RDONLY	Файл открыт только для чтения
O_WRONLY	Файл открыт только для записи
O_RDWR	Файл открыт для чтения и записи
O_APPEND	Файл открыт для добавления в конец
O_NONBLOCK	Неблокирующий режим
O_SYNC	Ожидать завершения операции записи (данных и атрибутов)
O_DSYNC	Ожидать завершения операции записи (только данных)
O_RSYNC	Синхронизировать операции чтения и записи
O_FSYNC	Ожидать завершения операции записи (только FreeBSD и Mac OS X)
O_ASYNC	Асинхронный режим ввода-вывода (только FreeBSD и Mac OS X)

К сожалению, три флага, O_RDONLY, O_WRONLY и O_RDWR, представлены числовыми значениями, а не отдельными битами, которые можно было бы проверить. (Как уже говорилось ранее, в силу исторических причин они обычно имеют значения 0, 1 и 2 соответственно. Кроме того, эти значения являются взаимоисключающими – для файла может быть установлен только один из этих трех флагов.) Поэтому следует сначала применить маску O_ACCMODE, чтобы выделить режимы доступа, и лишь потом сравнивать полученный результат с любым из трех значений.

`F_SETFL` Устанавливает флаги состояния файла. Новые значения флагов берутся из третьего аргумента (заданного в виде целого числа). Изменить можно только флаги `O_APPEND`, `O_NONBLOCK`, `O_SYNC`, `O_DSYNC`, `O_RSYNC`, `O_FSYNC` и `O_ASYNC`.

`F_GETOWN` Возвращает идентификатор процесса или группы процессов, которые в настоящее время получают сигналы `SIGURG` и `SIGIO`. Эти сигналы асинхронного ввода-вывода рассматриваются в разделе 14.6.2.

`F_SETOWN` Назначает идентификатор процесса или группы процессов, которые будут получать сигналы `SIGIO` и `SIGURG`. Положительное значение аргумента *arg* интерпретируется как идентификатор процесса, отрицательное – как идентификатор группы процессов, эквивалентный абсолютному значению аргумента *arg*.

Возвращаемое значение функции `fcntl` зависит от конкретной команды. Все команды возвращают значение -1 в случае ошибки и другие значения – в случае успешного завершения. Команды `_F_DUPFD`, `F_GETFD`, `F_GETFL` и `F_GETOWN` возвращают специальные значения. Первая возвращает дескриптор файла, следующие две – соответствующие флаги и последняя – либо идентификатор процесса (положительное значение), либо идентификатор группы процессов (отрицательное значение).

Пример

Программа, представленная листингом 3.4, принимает из командной строки один аргумент, который определяет дескриптор файла, и выводит значения флагов состояния файла для этого дескриптора.

Листинг 3.4. Вывод флагов состояния файла для заданного дескриптора

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int  val;

    if (argc != 2)
        err_quit("Использование: a.out <номер_дескриптора>");

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("ошибка fcntl для дескриптора %d", atoi(argv[1]));

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("только для чтения");
        break;
    case O_WRONLY:
        printf("только для записи");
        break;
    case O_RDWR:
        printf("для чтения и для записи");
        break;
```

```
    default:
        err_dump("неизвестный режим доступа");
    }

    if (val & O_APPEND)
        printf(", добавление в конец");
    if (val & O_NONBLOCK)
        printf(", неблокирующий режим");
#if defined(O_SYNC)
    if (val & O_SYNC)
        printf(", синхронный режим записи");
#endif
#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC)
    if (val & O_FSYNC)
        printf(", синхронный режим записи");
#endif
    putchar('\n');
    exit(0);
}
```

Обратите внимание: мы использовали макроопределение контроля функциональных особенностей `_POSIX_C_SOURCE` и условную компиляцию для тех флагов, которые не являются частью стандарта POSIX.1. Следующий сценарий демонстрирует работу программы при запуске ее из bash (Bourne-again shell). В зависимости от используемой командной оболочки полученные результаты могут несколько отличаться от приведенных здесь.

```
$ ./a.out 0 < /dev/tty
Только для чтения
$ ./a.out 1 > temp.foo
$ cat temp.foo
Только для записи
$ ./a.out 2 2>>temp.foo
Только для записи, добавление в конец
$ ./a.out 5 5<>temp.foo
Для чтения и для записи
```

Выражение `5<>temp.foo` открывает файл `temp.foo` для чтения и для записи на дескрипторе 5.

Пример

Изменяя флаги дескриптора или флаги состояния файла, необходимо сначала получить все имеющиеся значения флагов, изменить желаемые и затем записать полученное значение обратно. Мы не можем просто изменить отдельные флаги с помощью команд `F_SETFD` или `F_SETFL`, поскольку так можно сбросить другие флаги, которые были установлены.

В листинге 3.5 приводится текст функции, которая устанавливает один или более флагов состояния файла.

Листинг 3.5. Включает один или более флагов состояния файла

```
#include "apue.h"
#include <fcntl.h>

void
set_fl(int fd, int flags) /* flags - флаги, которые нужно включить */
{
    int  val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("ошибка выполнения команды F_GETFL функции fcntl");

    val |= flags;         /* включить флаги */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("ошибка выполнения команды F_SETFL функции fcntl");
}
```

Если изменить код в середине на

```
val &=  flags;            /* выключить флаги */
```

то мы получим функцию `clr_fl`, которая будет использоваться в ряде последующих примеров. В этой строке производится объединение по И (AND) текущего значения переменной `val` с логическим дополнением до единицы значения аргумента `flags`.

Если в начало программы, приведенной в листинге 3.3, добавить строку

```
set_fl(STDOUT_FILENO, O_SYNC);
```

то будет включен режим синхронной записи. Это приведет к тому, что каждый раз функция `write` будет ожидать завершения физической записи данных на диск, прежде чем вернуть управление. Обычно в UNIX функция `write` лишь ставит записываемые данные в очередь, а собственно запись на диск производится несколько позднее. Флаг `O_SYNC` часто используется в системах управления базами данных, так как он дает дополнительные гарантии того, что данные будут своевременно записаны на диск и не пропадут в случае отказа системы.

Предполагается, что использование флага `O_SYNC` увеличивает общее время работы программы. Чтобы проверить это предположение, можно воспользоваться программой из листинга 3.3. Скопируем с ее помощью 98,5 Мб данных из одного файла на диске в другой и сравним результаты с версией программы, которая устанавливает флаг `O_SYNC`. Результаты, полученные нами в ОС Linux с файловой системой `ext2`, приводятся в табл. 3.4.

Во всех шести случаях замеры производились со значением `BUFFSIZE`, равным 4096. Результаты, приведенные в табл. 3.2, были получены при чтении файла с диска и записи в устройство `/dev/null`, то есть запись на диск не производилась. Вторая строка табл. 3.4 соответствует чтению файла с диска и записи в другой файл на диске. По этой причине значения времени в первой и во второй строках табл. 3.4 отличаются. При записи в файл на диске

системное время увеличивается, потому что в этом случае ядро должно скопировать данные, полученные от процесса, и поставить их в очередь на запись, которая будет выполняться драйвером диска. Мы ожидали, что общее время также увеличится при записи файла на диск, но в этом случае прирост оказался незначительным. Это указывает на то, что записываемые данные попадают в системный кэш и мы не можем измерить фактическое время записи данных на диск.

Таблица 3.4. Результаты проверки производительности различных режимов синхронизации в ОС Linux с файловой системой ext2

Операция	Пользовательское время (секунды)	Системное время (секунды)	Общее время (секунды)
Время чтения из табл. 3.2 для `BUFFSIZE`=4096	0,03	0,16	6,86
Нормальный режим записи файла на диск	0,02	0,30	6,87
Запись на диск с установленным флагом `O_SYNC`	0,03	0,30	6,83
Запись на диск с последующим вызовом `fdatasync`	0,03	0,42	18,28
Запись на диск с последующим вызовом `fsync`	0,03	0,37	17,95
Запись на диск с установленным флагом `O_SYNC` и последующим вызовом `fsync`	0,05	0,44	17,95

Когда мы разрешаем синхронную запись, системное время и общее время должны значительно увеличиться. Как видно из третьей строки, время синхронной записи практически то же самое, что и время отложенной записи. Это означает, что файловая система `ext2` в Linux не обслуживает флаг `O_SYNC`. Данное предположение подтверждается шестой строкой: оказывается, что время, затраченное на выполнение синхронной записи с последующим вызовом функции `fsync`, практически равно времени, затраченному на обычную операцию записи с последующим вызовом функции `fsync` (пятая строка). После выполнения синхронной записи ожидается, что вызов `fsync` не будет иметь никакого эффекта.

В табл. 3.5 приводятся результаты тех же самых экспериментов для ОС Mac OS X. Обратите внимание на то, что полученные значения времени полностью соответствуют нашим ожиданиям: синхронная запись оказывается более дорогостоящей по сравнению с отложенной записью, а вызов функции `fsync` не сказывается на времени при использовании синхронного режима записи. Обратите также внимание на то, что добавление вызова функции `fsync` после выполнения обычной отложенной записи не дает существенного прироста времени. Это, скорее всего, говорит о том, что данные из кэша переписывались

операционной системой на диск по мере поступления новых данных, так что к моменту вызова функции fsync в кэше их оставалось не так уж и много.

Таблица 3.5. Результаты проверки производительности различных режимов синхронизации в Mac OS X

Операция	Пользовательское время (секунды)	Системное время (секунды)	Общее время (секунды)
Запись в устройство /dev/null	0,06	0,79	4,33
Нормальный режим записи файла на диск	0,05	3,56	14,40
Запись на диск с установленным флагом O_FSYNC	0,13	9,53	22,48
Запись на диск с последующим вызовом fsync	0,11	3,31	14,12
Запись на диск с установленным флагом O_FSYNC и последующим вызовом fsync	0,17	9,14	22,12

Сравните эффект от использования функций fsync и fdatasync, которые обновляют содержимое файла при обращении к ним, и действие флага O_SYNC, который обновляет содержимое файла при каждой операции записи.

В этом примере наглядно демонстрируется, для чего нужна функция fcntl. Наша программа работает с дескриптором (стандартный вывод), не зная названия файла, открытого командной оболочкой на этом дескрипторе. Мы лишены возможности установить флаг O_SYNC при открытии файла, так как его открывает командная оболочка. С помощью функции fcntl можно изменить свойства дескриптора, зная только дескриптор открытого файла. Позднее мы рассмотрим еще одну область применения функции fcntl, когда будем рассказывать о неблокирующих операциях ввода-вывода для неименованных каналов (раздел 15.2), поскольку при работе с ними нам доступен только дескриптор.

3.15. Функция ioctl

Функция ioctl всегда была универсальным инструментом ввода-вывода. Все, что невозможно выразить с помощью функций, описанных в этой главе, как правило, делается с помощью ioctl. Возможности этой функции чаще всего использовались в операциях терминального ввода-вывода. (Когда мы доберемся до главы 18, то увидим, что стандарт POSIX.1 заменил операции терминального ввода-вывода отдельными функциями.)

```
#include <unistd.h>    /* System V */
#include <sys/ioctl.h> /* BSD и Linux */
#include <stropts.h>   /* XSI STREAMS */

int ioctl(int filedes, int request, ...);
```

Возвращает –1 в случае ошибки, другие значения – в случае успеха

Функция `ioctl` была включена в стандарт Single UNIX Specification только как расширение для работы с устройствами STREAMS [Rago 1993]. Однако различные версии UNIX используют ее для выполнения самых разнообразных операций с устройствами. Некоторые реализации даже расширили ее функциональность для использования с обычными файлами.

Приведенный выше прототип функции определяется стандартом POSIX.1. В операционных системах FreeBSD 5.2.1 и Mac OS X 10.3 второй аргумент определен как `unsigned long`. Это не имеет большого значения, так как в качестве второго аргумента всегда передается имя константы, определяемой в заголовочном файле.

Согласно стандарту ISO C необязательные аргументы обозначены многоточием. Однако в большинстве случаев передается только один дополнительный аргумент, который представляет собой указатель на переменную или структуру.

В этом прототипе мы указали только те заголовочные файлы, которые требуются для самой функции `ioctl`. Но, как правило, при работе с ней необходимо подключать дополнительные заголовочные файлы для конкретных устройств. Например, все команды `ioctl` для операций терминального ввода-вывода, определяемые стандартом POSIX.1, требуют подключения заголовочного файла `<termios.h>`.

Каждый драйвер устройства может определять свой собственный набор команд `ioctl`. Тем не менее операционная система предоставляет набор универсальных команд `ioctl` для различных классов устройств. Примеры некоторых категорий универсальных команд `ioctl`, поддерживаемых FreeBSD, приводятся в табл. 3.6.

Таблица 3.6. Команды ioctl в ОС FreeBSD

Категория	Имена констант	Заголовочный файл	Количество команд `ioctl`
Метки диска	`DIOxxx`	`<sys/disklabel.h>`	6
Файловый ввод-вывод	`FIOxxx`	`<sys/filio.h>`	9
Ввод-вывод для накопителей на магнитной ленте	`MTIOxxx`	`<sys/mtio.h>`	11
Ввод-вывод для сокетов	`SIOxxx`	`<sys/sockio.h>`	60
Терминальный ввод-вывод	`TIOxxx`	`<sys/ttycom.h>`	44

Операции с накопителями на магнитной ленте позволяют записывать на ленту признак конца файла, перематывать ленту в начало, перемещаться вперед через заданное число файлов или записей и тому подобное. Ни одну из этих операций нельзя достаточно просто выразить в терминах других функций, описанных в данной главе (`read`, `write`, `lseek` и т. д.). Таким образом, простейший способ взаимодействия с такими устройствами всегда заключался в управлении ими через функцию `ioctl`.

Мы еще вернемся к функции `ioctl` при описании системы STREAMS в разделе 14.4, а также в разделе 18.12, где с ее помощью будем получать и изменять размер окна терминала, и в разделе 19.7, когда будем исследовать расширенные возможности псевдотерминалов.

3.16. /dev/fd

В современных операционных системах имеется каталог `/dev/fd`, в котором находятся файлы с именами 0, 1, 2 и т. д. Открытие файла `/dev/fd/`*n* эквивалентно созданию дубликата дескриптора с номером *n*, где *n* – это номер открытого дескриптора.

> Поддержка каталога `/dev/fd` была реализована Томом Даффом (Tom Duff) и впервые появилась в 8-й Редакции Research UNIX System. Эта функциональная особенность поддерживается всеми четырьмя операционными системами, о которых идет речь в данной книге: FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3 и Solaris 9. Она не является частью стандарта POSIX.1.

В вызове функции

```
fd = open("/dev/fd/0", mode);
```

большинство систем игнорируют аргумент `mode`, но есть и такие, которые требуют, чтобы он представлял собой подмножество флагов, которые использовались при открытии оригинального файла (в данном случае файл стандартного ввода). Поскольку данный вызов эквивалентен вызову

```
fd = dup(0);
```

дескрипторы 0 и `fd` будут совместно использовать одну и ту же запись в таблице файлов (рис. 3.3). Например, если дескриптор 0 был открыт только для чтения, то и дескриптор `fd` будет доступен только для чтения. Даже если система игнорирует режим открытия дескриптора и вызов

```
fd = open("/dev/fd/0", O_RDWR);
```

не завершается ошибкой, мы все равно не сможем ничего записать в файл с дескриптором `fd`.

Кроме того, можно использовать имя каталога `/dev/fd` в аргументе *pathname* функции `creat`, равно как и в функции `open` с флагом `O_CREAT`. Это позволяет программам, обращающимся к `creat`, продолжать работу, даже если аргумент *pathname*, например, содержит строку `/dev/fd/1`.

В некоторых системах имеются файлы /dev/stdin, /dev/stdout и /dev/stderr, которые эквивалентны /dev/fd/0, /dev/fd/1 и /dev/fd/2 соответственно.

В основном файлы из каталога /dev/fd используются командными оболочками. Это позволяет программам, которые требуют указания имени файла, работать со стандартными устройствами ввода и вывода таким же образом, как с обычными файлами. Например, в следующем примере программа cat(1) использует в качестве входного файла стандартный ввод, обозначаемый символом "-":

```
filter file2 | cat file1 - file3 | lpr
```

Сначала утилита cat читает содержимое файла file1, затем файл стандартного ввода (результат работы утилиты filter, обрабатывающей файл file2) и наконец файл file3. Если система поддерживает /dev/fd, можно опустить символ «-» и переписать команду следующим образом:

```
filter file2 | cat file1 /dev/fd/0 file3 | lpr
```

Символ «-» в качестве аргумента командной строки для обозначения файла стандартного ввода или стандартного вывода – своего рода ляп, который присутствует во многих программах. Например, если он будет стоять на месте первого файла, то будет очень похож на начало другого аргумента командной строки. Использование /dev/fd – это шаг к единообразию и к порядку.

3.17. Подведение итогов

В этой главе были описаны базовые функции ввода-вывода, предоставляемые системой UNIX. Их часто называют функциями небуферизованного ввода-вывода, потому что каждый вызов функции read или write обращается к системному вызову ядра. Мы увидели, как воздействует изменение размера буфера ввода-вывода на время, необходимое для чтения файла. Мы также рассмотрели несколько способов сбрасывания записываемых данных на диск и их влияние на производительность приложения.

Были рассмотрены атомарные операции, которые используются для доступа к одному и тому же файлу из нескольких процессов. Мы также увидели структуры данных, используемые ядром для организации совместного доступа к информации об открытых файлах. В дальнейшем мы еще вернемся к этим структурам.

Также были описаны функции fcntl и ioctl. Мы еще поговорим о них в главе 14, где ioctl будет использоваться с системой ввода-вывода STREAMS, а fcntl – для организации блокировки отдельных записей в файле.

Упражнения

3.1. Действительно ли функции чтения и записи файлов, описанные в данной главе, являются небуферизованными? Объясните почему.

3.2. Напишите свою версию функции dup2, которая реализует ту же функциональность, что и функция dup2, описанная в разделе 3.12, без ис-

пользования функции fcntl. Предусмотрите корректную обработку ошибок.

3.3. Предположим, что некоторый процесс вызывает следующие функции:

```
fd1 = open(pathname, oflags);
fd2 = dup(fd1);
fd3 = open(pathname, oflags);
```

Нарисуйте диаграмму, подобную той, что приведена на рис. 3.3. На какой дескриптор окажет влияние функция fcntl, если ей передать в качестве аргументов fd1 и F_SETFD? На какой дескриптор окажет влияние функция fcntl, если ей передать в качестве аргументов fd1 и F_SETFL?

3.4. Следующую последовательность операций можно наблюдать в различных программах.

```
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
if (fd > 2)
    close(fd);
```

Чтобы понять, для чего понадобился условный оператор if, предположите, что fd изначально имеет значение 1, и нарисуйте картинку, отображающую, что происходит со всеми тремя дескрипторами и соответствующими записями в таблице файлов после каждого вызова функции dup2. Затем нарисуйте аналогичную картинку, исходя из предположения, что изначально fd имел значение 3.

3.5. Командные оболочки Bourne shell, Bourne-again shell и Korn shell предусматривают такую нотацию:

```
digit1>&digit2
```

Она говорит о том, что дескриптор *digit1* должен быть перенаправлен в тот же файл, что и дескриптор *digit2*. Чем отличаются следующие две команды:

```
./a.out > outfile 2>&1
./a.out 2>&1 > outfile
```

(Подсказка: командные оболочки обрабатывают командную строку слева направо.)

3.6. Если файл открыт для чтения и записи с флагом O_APPEND, можно ли читать данные из произвольного места в файле с помощью функции lseek? Можно ли воспользоваться функцией lseek для изменения данных в произвольном месте в файле? Напишите программу, чтобы получить ответы на эти вопросы.

4

Файлы и каталоги

4.1. Введение

В предыдущей главе мы рассказали о базовых функциях, выполняющих операции ввода-вывода. Основным предметом обсуждения были операции ввода-вывода для обычных файлов: открытие, чтение файла или запись в файл. Теперь мы рассмотрим дополнительные характеристики файловой системы и свойства файла. Познакомившись с функцией `stat`, мы затем пройдемся по каждому элементу структуры `stat`, исследуя все существующие атрибуты файлов: владельца файла, права доступа к файлу и прочие. По ходу изложения мы также опишем каждую из функций, которые изменяют эти атрибуты. Кроме того, мы более подробно рассмотрим структуру файловой системы UNIX и символические ссылки. Закончим мы эту главу функциями для работы с каталогами и напишем функцию, которая обходит дерево каталогов.

4.2. Функции stat, fstat и lstat

В этой главе основное внимание будет сосредоточено вокруг трех функций `stat` и информации, которую они возвращают.

```
#include <sys/stat.h>

int stat(const char *restrict pathname, struct stat *restrict buf);

int fstat(int filedes, struct stat *buf);

int lstat(const char *restrict pathname, struct stat restrict buf);
```

Все три возвращают 0 в случае успеха, −1 в случае ошибки

Функция `stat` возвращает структуру с информацией о файле, указанном в аргументе *pathname*. Функция `fstat` возвращает информацию об открытом файле, который определяется дескриптором *filedes*. Функция `lstat` похожа

на функцию stat, но когда ей передается имя символической ссылки, она возвращает сведения о самой символической ссылке, а не о файле, на который она ссылается. (В разделе 4.21 нам потребуется эта функция, когда мы будем спускаться вниз по дереву каталогов. Более подробно символические ссылки будут описаны в разделе 4.16.)

Второй аргумент, *buf*, является указателем на структуру, которую функция будет заполнять информацией. Определение структуры может отличаться для разных реализаций, но основная ее часть выглядит следующим образом:

```
struct stat {
    mode_t st_mode;   /* тип файла и режим (права доступа) */
    ino_t st_ino;     /* номер индексного узла */
    dev_t st_dev;     /* номер устройства (файловой системы) */
    dev_t st_rdev;    /* номер устройства для специальных файлов */
    nlink_t st_nlink; /* количество ссылок */
    uid_t st_uid;     /* идентификатор пользователя владельца */
    gid_t st_gid;     /* идентификатор группы владельца */
    off_t st_size;    /* размер в байтах, для обычных файлов */
    time_t st_atime;  /* время последнего обращения к файлу */
    time_t st_mtime;  /* время последнего изменения файла */
    time_t st_ctime;  /* время последнего изменения флагов состояния файла */
    blksize_t st_blksize; /* оптимальный размер блока ввода-вывода */
    blkcnt_t st_blocks;   /* количество занятых дисковых блоков */
};
```

> Стандарт POSIX.1 не требует наличия полей st_rdev, st_blksize и st_blocks. Они определены как расширения XSI в стандарте Single UNIX Specification.

Обратите внимание, что каждый член структуры имеет элементарный системный тип данных (раздел 2.8). Чтобы исследовать атрибуты файла, мы рассмотрим каждый из членов структуры.

Вероятно, наиболее часто функцию stat использует команда ls -l, которая выводит полную информацию о файле.

4.3. Типы файлов

Мы уже упоминали файлы двух типов: обычные файлы и каталоги. Большинство файлов в UNIX являются либо обычными файлами, либо каталогами, но есть и другие типы файлов. Перечислим возможные типы:

1. Обычный файл – наиболее распространенный тип файлов, который хранит данные в том или ином виде. Ядро UNIX не делает различий между текстовыми и двоичными файлами. Любая интерпретация содержимого файла полностью возлагается на прикладную программу, обрабатывающую файл.

   > Одно из наиболее известных исключений из этого правила – исполняемые файлы. Чтобы запустить программу, ядро должно понять ее формат. Все двоичные исполняемые файлы следуют конкретному формату, который позволяет ядру определить, куда следует загрузить исполняемый код и данные программы.

2. Файл каталога. Файлы этого типа содержат имена других файлов и ссылки на информацию о них. Любой процесс, обладающий правом на чтение каталога, может проверить его содержимое, но только ядро обладает правом на запись в файл каталога. Чтобы внести изменения в каталог, процессы должны пользоваться функциями, обсуждаемыми в данной главе.
3. Специальный файл блочного устройства. Этот тип файлов обеспечивает буферизованный ввод-вывод для таких устройств, как дисковые устройства с фиксированным размером блока.
4. Специальный файл символьного устройства. Этот тип файлов обеспечивает небуферизованный ввод-вывод для устройств с переменным размером блока. Все устройства в системе являются либо специальными файлами блочных устройств, либо специальными файлами символьных устройств.
5. FIFO, или именованный канал. Этот тип файлов используется для организации обмена информацией между процессами. Именованные каналы будут описаны в разделе 15.5.
6. Сокет. Этот тип файлов используется для организации обмена информацией между процессами через сетевые соединения. Сокеты можно применять и для обмена информацией между процессами на одной и той же машине. Мы будем использовать сокеты для организации взаимодействий между процессами в главе 16.
7. Символическая ссылка. Файлы этого типа представляют собой ссылки на другие файлы. Более подробно о символических ссылках мы поговорим в разделе 4.16.

Тип файла хранится в поле `st_mode` структуры `stat`. Определить тип файла можно с помощью макроопределений, приведенных в табл. 4.1. В качестве аргумента для каждого из них используется значение поля `st_mode` структуры `stat`.

Таблица 4.1. Макросы для определения типа файла из <sys/stat.h>

Макроопределение	Тип файла
`S_ISREG()`	Обычный файл
`S_ISDIR()`	Каталог
`S_ISCHR()`	Специальный файл символьного устройства
`S_ISBLK()`	Специальный файл блочного устройства
`S_ISFIFO()`	Канал (именованный или неименованный)
`S_ISLNK()`	Символическая ссылка
`S_ISSOCK`	Сокет

Стандарт POSIX.1 допускает реализацию и представление объектов межпроцессного взаимодействия (IPC), таких как очереди сообщений и семафоры, в виде файлов. Макроопределения из табл. 4.2 позволяют нам определить тип объекта IPC из структуры `stat`. Главное их отличие от макросов, пере-

численных в табл. 4.1, заключается в том, что аргументом для них является указатель на структуру `stat`, а не значение поля `st_mode`.

Таблица 4.2. Макросы для определения типа объекта IPC из <sys/stat.h>

Макроопределение	Тип файла
`S_TYPEISMQ()`	Очередь сообщений
`S_TYPEISSEM()`	Семафор
`S_TYPEISSHM()`	Объект разделяемой памяти

Очереди сообщений, семафоры и объекты разделяемой памяти будут рассматриваться в главе 15. Однако ни одна из реализаций, обсуждаемых в данной книге, не представляет эти объекты в виде файлов.

Пример

Программа, представленная листингом 4.1, выводит тип файла для каждого аргумента командной строки.

Листинг 4.1. Вывод типа файла для каждого аргумента командной строки

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int i;
    struct stat buf;
    char *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("ошибка вызова функции lstat");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "обычный файл";
        else if (S_ISDIR(buf.st_mode))
            ptr = "каталог";
        else if (S_ISCHR(buf.st_mode))
            ptr = "файл символьного устройства";
        else if (S_ISBLK(buf.st_mode))
            ptr = "файл блочного устройства";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "символическая ссылка";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "сокет";
        else
```

```
                ptr = "** неизвестный тип файла **";
            printf("%s\n", ptr);
        }
        exit(0);
    }
```

Пример вывода программы из листинга 4.1:

```
$ ./a.out /etc/passwd /etc /dev/initctl /dev/log /dev/tty \
> /dev/scsi/host0/bus0/target0/lun0/cd /dev/cdrom
/etc/passwd: обычный файл
/etc: каталог
/dev/initctl: fifo
/dev/log: сокет
/dev/tty: файл символьного устройства
/dev/scsi/host0/bus0/target0/lun0/cd: файл блочного устройства
/dev/cdrom: символическая ссылка
```

(Символ обратного слэша в конце первой строки сообщает командной оболочке о том, что ввод команды не закончен. В таких случаях командная оболочка выводит на следующей строке вторичное приглашение к вводу – символ >.) Мы нарочно использовали функцию `lstat` вместо `stat`, чтобы обнаружить символические ссылки. Используя функцию `stat`, мы никогда не увидели бы их.

Для компиляции этой программы в операционной системе Linux необходимо определить символ `_GNU_SOURCE`, чтобы включить определение макроса `S_ISSOCK`.

Ранние версии UNIX не имели макроопределений `S_ISxxx`. Вместо этого необходимо было выполнять объединение по И (AND) значения `st_mode` с маской `S_IFMT` и затем сравнивать результат с константами `S_IFxxx`. Определение этой маски и связанных с нею констант в большинстве систем находится в файле `<sys/stat.h>`. Заглянув в этот файл, мы обнаружим, что макрокоманда `S_ISDIR` определена примерно таким образом

```
#define S_ISDIR(mode) (((mode) & S_IFMT) == S_IFDIR)
```

Мы уже говорили, что обычные файлы являются самыми распространенными, но было бы интересно узнать, какой процент от всех файлов на данной системе занимают файлы каждого типа. В табл. 4.3 приводится количество файлов каждого типа и его процентное выражение для ОС Linux, используемой в качестве однопользовательской рабочей станции. Эти данные были получены с помощью программы, которую мы продемонстрируем в разделе 4.21.

Таблица 4.3. Количество файлов различных типов и его процентное выражение

Тип файла	Количество	Процент от общего числа
Обычные файлы	226 856	82,22%
Каталоги	23 017	8,95
Символические ссылки	6 442	2,51

Таблица 4.3 (продолжение)

Тип файла	Количество	Процент от общего числа
Файлы символьных устройств	447	0,17
Файлы блочных устройств	312	0,12
Сокеты	69	0,03
FIFO	1	0,00

4.4. set-user-ID и set-group-ID

С каждым процессом связаны шесть или более идентификаторов. Все они перечислены в табл. 4.4.

Таблица 4.4. Идентификаторы пользователя и группы, связанные с каждым процессом

Реальный идентификатор пользователя Реальный идентификатор группы	Определяет, кто мы на самом деле
Эффективный идентификатор пользователя Эффективный идентификатор группы Идентификаторы дополнительных групп	Используются при проверке прав доступа к файлам
Сохраненный идентификатор пользователя Сохраненный идентификатор группы	Идентификаторы, сохраняемые функциями `exec`

- Реальные идентификаторы пользователя и группы определяют, кто мы на самом деле. Эти идентификаторы извлекаются из файла паролей во время входа в систему. Обычно в течение сессии значения этих идентификаторов не меняются, хотя процессы, обладающие правами суперпользователя, имеют возможность изменять их, о чем мы поговорим в разделе 8.11.
- Эффективные идентификаторы пользователя и группы и идентификаторы дополнительных групп определяют права доступа к файлам, о чем мы поговорим в следующем разделе. (Определение дополнительных групп было дано в разделе 1.8.)
- Сохраненные идентификаторы пользователя и группы представляют собой копии эффективных идентификаторов, которые создаются в момент запуска программы. Мы расскажем о назначении этих двух идентификаторов, когда будем описывать функцию `setuid` в разделе 8.11.

> Сохраненные идентификаторы перешли в разряд обязательных для реализации в соответствии с версией POSIX.1 от 2001 года. В более ранних версиях POSIX они находились в категории необязательных. Приложение может проверить наличие константы `_POSIX_SAVED_IDS` на этапе компиляции или вызвать функцию `sysconf` с аргументом `_SC_SAVED_IDS` на этапе выполнения, чтобы определить, поддерживает ли реализация эту функциональную возможность.

Обычно эффективный идентификатор пользователя совпадает с реальным идентификатором пользователя, а эффективный идентификатор группы – с реальным идентификатором группы.

У каждого файла в системе есть идентификатор владельца и идентификатор группы владельца. Идентификатор владельца файла хранится в поле `st_uid` структуры `stat`, а идентификатор группы владельца – в поле `st_gid`.

Когда мы запускаем файл программы, эффективным идентификатором процесса обычно становится реальный идентификатор пользователя, а эффективным идентификатором группы – реальный идентификатор группы. Но существует возможность установить специальный флаг в поле `st_mode`, который как бы говорит: «при запуске этого файла взять в качестве эффективного идентификатора процесса идентификатор пользователя владельца файла (`st_uid`)». Точно так же в поле `st_mode` может быть установлен другой флаг, который назначит в качестве эффективного идентификатора группы идентификатор группы владельца файла (`st_gid`). Эти два флага в поле `st_mode` называются битами *set-user-ID* и *set-group-ID* соответственно.

Например, если владельцем файла является суперпользователь и у файла установлен бит set-user-ID, то во время работы программы соответствующий процесс будет обладать правами суперпользователя. Это происходит независимо от того, каков реальный идентификатор пользователя процесса, запустившего файл. Так, системная утилита UNIX, позволяющая любому пользователю изменять свой пароль, `passwd`(1), является программой с установленным битом set-user-ID. Это требуется для того, чтобы утилита могла записать новый пароль в файл паролей (обычно это файл `/etc/passwd` или `/etc/shadow`), который должен быть доступен на запись только суперпользователю. Поскольку в подобных случаях программы, запускаемые рядовыми пользователями, обычно расширяют их привилегии, при их написании следует проявлять осторожность. Такие программы мы обсудим более подробно в главе 8.

Биты set-user-ID и set-group-ID хранятся в поле `st_mode` структуры `stat`, ассоциированной с файлом. Проверить их можно с помощью констант `S_ISUID` и `S_ISGID`.

4.5. Права доступа к файлу

Поле `st_mode` кроме всего прочего содержит в себе биты прав доступа к файлу. Под *файлом* мы подразумеваем файл любого типа из описанных выше. Любые файлы – каталоги, специальные файлы устройств и прочие – обладают правами доступа. Многие полагают, что понятие прав доступа присуще только обычным файлам.

Права доступа к файлу определяются девятью битами, которые подразделяются на три категории. Все они перечислены в табл. 4.5.

Таблица 4.5. Биты прав доступа из файла <sys/stat.h>

Маска для поля st_mode	Назначение
S_IRUSR	user-read – доступно пользователю для чтения
S_IWUSR	user-write – доступно пользователю для записи
S_IXUSR	user-execute – доступно пользователю для исполнения
S_IRGRP	group-read – доступно группе для чтения
S_IWGRP	group-write – доступно группе для записи
S_IXGRP	group-execute – доступно группе для исполнения
S_IROTH	other-read – доступно остальным для чтения
S_IWOTH	other-write – доступно остальным для записи
S_IXOTH	other-execute – доступно остальным для исполнения

Команда chmod(1), которая обычно используется для изменения прав доступа к файлам, позволяет определять имя категории посредством символов: u – user (пользователь, или владелец), g – group (группа) и o – other (остальные). В некоторых книгах эти три категории обозначаются как owner (владелец), group (группа) и world (весь остальной мир), что может привести к путанице, так как команда chmod использует символ o не в смысле owner (владелец), а в смысле other (остальные). Мы будем использовать термины *user* (пользователь), *group* (группа) и *other* (остальные), чтобы сохранить совместимость с командой chmod.

Три категории из табл. 4.5 – чтение, запись и исполнение – используются различными функциями самыми разными способами. Сейчас мы коротко опишем их, а затем еще будем к ним возвращаться при обсуждении конкретных функций.

- Первое правило: чтобы открыть файл любого типа по его полному имени, необходимо иметь право на исполнение для всех каталогов, указанных в имени файла, включая текущий. По этой причине бит права на исполнение для каталогов часто называют битом права на поиск.

  Например, чтобы открыть файл /usr/include/stdio.h, мы должны иметь право на исполнение для каталогов /, /usr и /usr/include. Далее мы должны обладать соответствующими правами на доступ к открываемому файлу в зависимости от того, в каком режиме мы собираемся его открыть – только для чтения, для чтения и записи и т. д.

  Если текущим каталогом является каталог /usr/include, то для того чтобы открыть файл stdio.h, мы должны обладать правом на исполнение для текущего каталога. В этом примере текущий каталог не указан явным образом, но подразумевается. С тем же успехом можно было бы обозначить имя файла как ./stdio.h.

  Обратите внимание, что право на чтение и право на исполнение для каталогов имеют разный смысл. Право на чтение дает возможность прочитать файл каталога, получив полный список файлов, находящихся в нем. Право

на исполнение дает возможность войти в каталог, когда он является одним из компонентов пути к файлу, к которому требуется получить доступ. (Чтобы отыскать нужный файл, необходимо выполнить поиск по каталогу).

Еще один пример неявной ссылки на каталог – переменная окружения `PATH` (обсуждается в разделе 8.10). Если она определяет какой-либо каталог, для которого у нас нет права на исполнение, то командная оболочка никогда не будет просматривать его при поиске исполняемых файлов.

- Право на чтение для файла определяет, можем ли мы открыть существующий файл для чтения (флаги `O_RDONLY` и `O_RDWR`, функции `open`).
- Право на запись для файла определяет, можем ли мы открыть существующий файл для записи (флаги `O_WRONLY` и `O_RDWR`, функции `open`).
- Чтобы указать флаг `O_TRUNC` в функции `open`, нужно обладать правом на запись.
- Нельзя создать новый файл в каталоге при отсутствии права на запись и права на исполнение для этого каталога.
- Чтобы удалить существующий файл, необходимо обладать правом на запись и правом на исполнение для каталога, который содержит этот файл. Не нужно обладать правом на чтение или на запись для самого файла.
- Чтобы запустить файл на исполнение с помощью одной из шести функций семейства `exec` (раздел 8.10), нужно обладать правом на исполнение. Кроме того, файл должен быть обычным файлом.

Решение о выдаче полномочий на доступ к файлу, которое принимается ядром всякий раз, когда процесс открывает, создает или удаляет файл, зависит от того, кому принадлежит файл (`st_uid` и `st_gid`), от значений эффективных идентификаторов процесса (эффективный идентификатор пользователя и эффективный идентификатор группы) и от идентификаторов дополнительных групп процесса, если таковые поддерживаются. Оба идентификатора владельца являются свойствами самого файла, тогда как эффективные идентификаторы и идентификаторы дополнительных групп – это свойства процесса. Решение принимается ядром по следующему алгоритму.

1. Если процесс имеет эффективный идентификатор пользователя, равный 0 (суперпользователь), то доступ разрешается. Это дает суперпользователю абсолютную свободу действий во всей файловой системе.
2. Если процесс имеет эффективный идентификатор пользователя, совпадающий с идентификатором владельца файла (то есть процесс является владельцем файла), то доступ разрешается, если установлен соответствующий бит права доступа для владельца. В противном случае доступ к файлу запрещается. Под выражением *соответствующий бит права доступа* понимается следующее: если процесс открывает файл для чтения, то должен быть установлен бит user-read, если файл открывается для записи, то должен быть установлен бит user-write, если процесс собирается запустить файл на исполнение, то должен быть установлен бит user-execute.
3. Если эффективный идентификатор группы или один из идентификаторов дополнительных групп процесса совпадает с идентификатором группы

файла, то доступ разрешается, если установлен соответствующий бит права доступа. В противном случае доступ к файлу запрещается.

4. Если установлен соответствующий бит права доступа для остальных, то доступ разрешается, в противном случае доступ запрещается.

Эти четыре шага выполняются в указанной последовательности. Обратите внимание: если процесс является владельцем файла (шаг 2), то решение о предоставлении доступа или отказе в доступе к файлу принимается только на основании прав доступа владельца, права группы уже не проверяются. Аналогичным образом, если процесс не является владельцем файла, но принадлежит к соответствующей группе, то решение принимается на основе анализа прав доступа группы – права остальных не принимаются во внимание.

4.6. Принадлежность новых файлов и каталогов

Рассматривая в главе 3 процедуру создания новых файлов с помощью функций `open` или `creat`, мы не упоминали о том, какие значения принимаются в качестве идентификатора пользователя и группы для нового файла. Как создаются каталоги, мы покажем в разделе 4.20 при описании функции `mkdir`. Правила выбора владельца для нового каталога аналогичны приводимым здесь правилам выбора владельца для нового файла.

В качестве идентификатора пользователя (владельца) для нового файла принимается значение эффективного идентификатора пользователя процесса. При определении идентификатора группы для нового файла стандарт POSIX.1 допускает выбор одного из двух вариантов.

1. В качестве идентификатора группы для нового файла может быть принят эффективный идентификатор группы процесса.
2. В качестве идентификатора группы для нового файла может быть принят идентификатор группы каталога, в котором создается файл.

> Операционные системы FreeBSD 5.2.1 и Mac OS X 10.3 всегда используют идентификатор группы каталога в качестве идентификатора группы для создаваемого файла.
>
> В файловых системах `ext2` и `ext3` в ОС Linux допускается возможность выбора любого из этих двух вариантов с помощью специального флага команды `mount`(1). В операционных системах Linux 2.4.22 (если установлен соответствующий флаг команды `mount`) и Solaris 9 выбор идентификатора группы для нового файла зависит от того, установлен ли бит set-group-ID у каталога, в котором создается файл. Если этот бит установлен, идентификатором группы для нового файла назначается идентификатор группы каталога, в противном случае – эффективный идентификатор группы процесса.

Второй вариант – наследование идентификатора группы от каталога – дает гарантию, что все файлы и подкаталоги, создаваемые в заданном каталоге, будут принадлежать той же группе, что и родительский каталог. Порядок назначения группы владельца для файлов и каталогов будет распространяться вниз по всем вложенным каталогам. Например, таким образом организована структура каталога `/var/spool/mail` в ОС Linux.

Как уже упоминалось ранее, этот вариант назначения идентификатора группы принят по умолчанию в операционных системах FreeBSD 5.2.1 и Mac OS X 10.3, но в Linux и Solaris он является одним из возможных. Чтобы описанная схема работала в Linux 2.4.22 и Solaris 9, для каталога необходимо установить бит set-group-ID, а функция `mkdir` должна устанавливать его автоматически для всех вложенных каталогов. (Это будет описано в разделе 4.20.)

4.7. Функция access

Как уже говорилось ранее, при открытии файла ядро выполняет серию проверок прав доступа, основываясь на эффективных идентификаторах пользователя и группы процесса. Однако в некоторых случаях процессу необходимо проверить права доступа на основе реальных идентификаторов пользователя и группы. Это бывает удобно, когда процесс запущен с правами другого пользователя с помощью set-user-ID или set-group-ID. Даже когда установка бита set-user-ID предоставляет процессу права суперпользователя, все еще может потребоваться необходимость проверить права реального пользователя на доступ к тому или иному файлу. Функция `access` выполняет проверку прав доступа, основываясь на реальных идентификаторах пользователя и группы процесса. (Замените слово *эффективный* на слово *реальный* в алгоритме принятия решения, приведенном в конце раздела 4.5.)

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Аргумент *mode* представляет собой набор констант из табл. 4.6, объединяемых по ИЛИ (OR).

Таблица 4.6. Константы, используемые в аргументе mode функции access

mode	Описание	mode	Описание
R_OK	Проверка права на чтение	X_OK	Проверка права на исполнение
W_OK	Проверка права на запись	F_OK	Проверка существования файла

Пример

В программе, представленной листингом 4.2, показан пример использования функции `access`.

Листинг 4.2. Пример использования функции access

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
```

```
    if (argc != 2)
        err_quit("Использование: a.out <имя_файла>");
    if (access(argv[1], R_OK) < 0)
        err_ret("ошибка вызова функции access для файла %s", argv[1]);
    else
        printf("доступ для чтения разрешен\n");
    if (open(argv[1], O_RDONLY) < 0)
        err_ret("ошибка вызова функции open для файла %s", argv[1]);
    else
        printf("файл благополучно открыт для чтения\n");
    exit(0);
}
```

Ниже приводится пример работы с этой программой:

```
$ ls -l a.out
-rwxrwxr-x   1 sar               15945 Nov 30 12:10 a.out
$ ./a.out a.out
Доступ для чтения разрешен
Файл благополучно открыт для чтения
$ ls -l /etc/shadow
-r--------   1 root                1315 Jul 17 2002 /etc/shadow
$ ./a.out /etc/shadow
ошибка вызова функции access для файла /etc/shadow: Permission denied
ошибка вызова функции open для файла /etc/shadow: Permission denied
$ su                          получим права суперпользователя
Password:                     вводим пароль суперпользователя
# chown root a.out            делаем суперпользователя владельцем файла
# chmod u+s a.out             и устанавливаем бит set-user-ID
# ls -l a.out                 проверяем владельца файла и состояние бита SUID
-rwsrwxr-x    1 root              15945 Nov 30 12:10 a.out
# exit                        возвращаемся к правам обычного пользователя
$ ./a.out /etc/shadow
ошибка вызова функции access для файла /etc/shadow: Permission denied
файл благополучно открыт для чтения
```

В этом примере программа, у которой установлен бит set-user-ID, смогла определить, что реальный пользователь не сможет читать указанный файл, хотя функция `open` отрабатывает успешно.

> В предыдущем примере и в главе 8 мы иногда переходим в режим суперпользователя, чтобы продемонстрировать некоторые приемы. Если вы работаете в многопользовательской системе и не обладаете правами суперпользователя, вы не сможете полностью протестировать такие примеры.

4.8. Функция umask

Теперь, когда мы рассмотрели биты прав доступа, свойственные всем файлам, перейдем к маске режима создания файла, которой обладает каждый процесс.

Функция `umask` устанавливает маску режима создания файлов для процесса и возвращает предыдущее значение маски. (Это одна из немногих функций, которые не возвращают признак ошибки.)

```
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

Возвращает предыдущее значение маски

Аргумент *cmask* представляет собой набор констант из табл. 4.5 (S_IRUSR, S_IWUSR и т. д.), объединяемых по ИЛИ (OR).

Маска режима создания файлов используется при создании процессом новых файлов или новых каталогов. (Загляните в разделы 3.3 и 3.4, где были описаны функции open и creat. Обе функции принимают аргумент *mode*, в котором указываются биты прав доступа к создаваемому файлу.) Процедуру создания новых каталогов мы рассмотрим в разделе 4.20. Любые биты, которые *включены* в маске, *выключают* соответствующие биты прав доступа к файлу.

Пример

Программа, представленная листингом 4.3, создает два файла: один со значением маски, равным нулю, и второй – с маской, которая выключает все биты прав доступа для группы и остальных.

Листинг 4.3. Пример использования функции umask

```
#include "apue.h"
#include <fcntl.h>

#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("ошибка вызова функции creat для файла foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
        err_sys("ошибка вызова функции creat для файла bar");
    exit(0);
}
```

Запустив эту программу, мы сможем увидеть, как устанавливаются биты прав доступа.

```
$ umask                 прежде всего выведем текущее значение маски
002
$ ./a.out
$ ls -l foo bar
-rw-------    1 sar        0 Dec 7 21:20 bar
-rw-rw-rw-    1 sar        0 Dec 7 21:20 foo
$ umask                 проверим, изменилось ли значение маски
002
```

Большинство пользователей UNIX никогда не имеют дела с этой маской. Она обычно устанавливается командной оболочкой единожды, в момент входа

в систему, и никогда не изменяется. Тем не менее при разработке программ, которые создают новые файлы, необходимо модифицировать значение маски на время работы процесса, чтобы обеспечить установку конкретных битов прав доступа. Например, чтобы предоставить любому пользователю право на чтение создаваемого файла, мы должны установить значение маски в 0. В противном случае вследствие применения действующей маски может получиться так, что необходимые биты прав доступа окажутся сброшенными.

В предыдущем примере мы использовали команду `umask` для вывода значения маски режимов создания файлов до и после запуска программы. Тем самым мы показали, что изменение маски в процессе не влияет на маску родительского процесса (которым часто является командная оболочка). Все командные оболочки имеют встроенную команду `umask`, которая используется для вывода и изменения значения маски режима создания новых файлов.

Пользователи могут установить значение `umask` для управления правами доступа к создаваемым файлам по умолчанию. Значение маски задается в восьмеричной системе счисления, где каждый бит маски соответствует биту прав доступа, который он отключает, как это показано в табл. 4.7. Права доступа отключаются установкой соответствующих битов. Наиболее распространенные значения маски – 002 (запрещает запись в файл всем пользователям, кроме владельца), 022 (запрещает запись в файл членам группы и остальным пользователям) и 027 (запрещает членам группы запись в файл, а всем остальным – чтение, запись и исполнение).

Таблица 4.7 Биты прав доступа для маски

Бит маски	Значение	Бит маски	Значение
0400	user-read	0010	group-execute
0200	user-write	0004	other-read
0100	user-execute	0002	other-write
0040	group-read	0001	other-execute
0020	group-write		

Стандарт Single UNIX Specification требует, чтобы командная оболочка поддерживала возможность определения маски в символической форме. В отличие от восьмеричного формата, символический формат определяет набор прав, которые разрешаются (то есть сброшены в маске), а не прав, которые запрещаются (то есть установлены в маске). Сравните два варианта вызова команды `umask`:

```
$ umask              прежде всего выведем текущее значение маски
                     режима создания новых файлов
002
$ umask -S           выведем значение маски в символическом представлении
u=rwx,g=rwx,o=rx
$ umask 027          изменим значение маски режима создания файлов
$ umask -S           выведем значение маски в символическом представлении
u=rwx,g=rx,o=
```

4.9. Функции chmod и fchmod

Эти функции позволяют изменять права доступа к существующим файлам.

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);

int fchmod(int filedes, mode_t mode);
```

Возвращают 0 в случае успеха, –1 в случае ошибки

Функция `chmod` работает с файлом, заданным его именем, а функция `fchmod` – с уже открытым файлом, заданным дескриптором.

Чтобы можно было изменить права доступа к файлу, эффективный идентификатор процесса должен совпадать с идентификатором владельца файла либо процесс должен обладать привилегиями суперпользователя.

Аргумент `mode` представляет собой набор констант из табл. 4.8, объединяемых по ИЛИ (OR).

Таблица 4.8. Константы режимов для функции chmod, определенные в файле <sys/stat.h>

mode	Описание
S_ISUID	set-user-ID при запуске на исполнение
S_ISGID	set-group-ID при запуске на исполнение
S_ISVTX	saved-text (бит sticky)
S_IRWXU	Право на чтение, запись и исполнение для пользователя (владельца)
S_IRUSR	Право на чтение для пользователя (владельца)
S_IWUSR	Право на запись для пользователя (владельца)
S_IXUSR	Право на исполнение для пользователя (владельца)
S_IRWXG	Право на чтение, запись и исполнение для группы
S_IRGRP	Право на чтение для группы
S_IWGRP	Право на запись для группы
S_IXGRP	Право на исполнение для группы
S_IRWXO	Право на чтение, запись и исполнение для остальных
S_IROTH	Право на чтение для остальных
S_IWOTH	Право на запись для остальных
S_IXOTH	Право на исполнение для остальных

Обратите внимание: имена девяти констант из табл. 4.8 совпадают с именами констант из табл. 4.5. Здесь добавились две константы set-ID (`S_ISUID` и `S_ISGID`), константа saved-text (`S_ISVTX`) и три комбинированных константы (`S_IRWXU`, `S_IRWXG` и `S_IRWXO`).

Бит saved-text (S_ISVTX) не является частью стандарта POSIX.1. Он определен как расширение XSI в стандарте Single UNIX Specification. Его назначение будет описано в следующем разделе.

Пример

Вспомните состояние файлов foo и bar, которые были созданы программой из листинга 4.3, демонстрирующей работу функции umask:

```
$ ls -l foo bar
-rw-------  1 sar      0 Dec 7 21:20 bar
-rw-rw-rw-  1 sar      0 Dec 7 21:20 foo
```

Программа, представленная листингом 4.4, изменяет режимы доступа к этим файлам.

Листинг 4.4. Пример использования функции chmod

```
#include "apue.h"

int
main(void)
{
    struct stat   statbuf;

    /* включить бит set-group-ID и выключить group-execute */

    if (stat("foo", &statbuf) < 0)
        err_sys("ошибка вызова функции stat для файла foo");
    if (chmod("foo", (statbuf.st_mode &  S_IXGRP) | S_ISGID) < 0)
        err_sys("ошибка вызова функции chmod для файла foo");

    /* установить режим в значение "rw-r--r--" */

    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys("ошибка вызова функции chmod для файла bar");
    exit(0);
}
```

После запуска программы из листинга 4.4 мы увидим, что режимы доступа к файлам изменились следующим образом:

```
$ ls -l foo bar
-rw-r--r--    1 sar         0 Dec 7 21:20 bar
-rw-rwSrw-    1 sar         0 Dec 7 21:20 foo
```

В этом примере мы использовали абсолютное значение прав доступа для файла bar, совершенно не задумываясь о текущих правах доступа. Для файла foo мы, наоборот, установили права доступа относительно их текущего состояния. Для этого мы сначала получили набор битов прав доступа с помощью функции stat и затем изменили их. Мы явным образом включили бит set-group-ID и выключили бит group-execute. Обратите внимание: команда ls вывела значение бита group-execute в виде символа S, подчеркивая тем самым, что бит set-group-ID установлен при сброшенном бите group-execute.

В ОС Solaris вместо символа `S` команда `ls` выводит символ `l`, чтобы подчеркнуть, что для файла включен режим обязательных блокировок файла и отдельных записей. Это справедливо только для обычных файлов, и мы еще будем обсуждать эту тему в разделе 14.3.

И наконец, обратите внимание на то, что дата и время, отображаемые командой `ls`, не изменились после запуска программы из листинга 4.4. Немного позднее, в разделе 4.18, мы увидим, что функция `chmod` обновляет только время последнего изменения индексного узла (i-node). Команда `ls` по умолчанию выводит время последнего изменения содержимого файла.

Функция `chmod` автоматически сбрасывает два бита прав доступа при следующих условиях:

- В некоторых системах, таких как Solaris, бит sticky имеет особое значение для обычных файлов. Если мы попытаемся установить бит sticky (`S_ISVTX`) для обычного файла, не обладая при этом привилегиями суперпользователя, то этот бит в аргументе *mode* будет автоматически сброшен. (Бит sticky рассматривается в следующем разделе.) Отсюда следует, что он может быть установлен только суперпользователем. Сделано это для того, чтобы предотвратить установку бита `S_ISVTX` злоумышленником и тем самым избежать нанесения ущерба производительности системы в целом.

  В операционных системах FreeBSD 5.2.1, Mac OS X 10.3 и Solaris 9 только суперпользователь может устанавливать бит sticky на обычные файлы. В Linux 2.4.22 это ограничение отсутствует, поскольку для обычных файлов в ОС Linux этот бит не имеет никакого смысла. Несмотря на то, что в FreeBSD 5.2.1 и Mac OS X 10.3 этот бит также не имеет значения для обычных файлов, тем не менее данные системы все-таки разрешают его установку для обычных файлов только суперпользователю.

- При создании файла ему может быть назначен идентификатор группы, отличный от идентификатора группы процесса, создающего файл. В разделе 4.6 мы уже говорили, что возможна ситуация, когда файл наследует идентификатор группы от каталога, в котором он размещается. Таким образом, если идентификатор группы создаваемого файла не является эффективным идентификатором группы процесса или одним из идентификаторов дополнительных групп и при этом процесс не обладает привилегиями суперпользователя, то бит set-group-ID автоматически сбрасывается. Это предотвращает возможность создания файла с идентификатором группы, с которой пользователь никак не связан.

  Операционные системы FreeBSD 5.2.1, Mac OS X 10.3, Linux 2.4.22 и Solaris 9 имеют еще одну особенность, предотвращающую злонамеренное использование некоторых битов. Так, если процесс, не обладающий привилегиями суперпользователя, производит запись в файл, то биты set-user-ID и set-group-ID автоматически сбрасываются. Даже если злоумышленнику удастся отыскать доступные на запись файлы с установленными битами set-user-ID или set-group-ID, то в момент модификации эти файлы утратят особые привилегии.

4.10. Бит sticky

Интересна история появления бита S_ISVTX. В версиях UNIX, которые поддерживали предварительную подкачку страниц, этот бит был известен под названием *sticky bit* («липкий»). Если этот бит был установлен на исполняемый файл, то при первом запуске программы ее сегмент кода записывался в файл подкачки и сохранялся там после ее завершения. (Сегмент кода программы состоит из машинных инструкций.) Это приводило к тому, что при следующем вызове программы она запускалась намного быстрее, поскольку файл подкачки представлял собой непрерывную область на диске, в отличие от обычных файлов, которые могут размещаться в разрозненных дисковых блоках. Бит sticky обычно устанавливался на наиболее часто используемые программы, такие как текстовые редакторы или компилятор языка C. Естественно, существовало ограничение на количество таких «липких» файлов, которые могли одновременно разместиться в файле подкачки, тем не менее этот прием был очень удобен. Название sticky (липкий) объясняется тем, что сегмент кода программы как бы вклеивался в пространство файла подкачки и оставался там до перезагрузки системы. В более поздних версиях UNIX этот бит стал называться *saved-text bit* (закрепляемый сегмент кода), отсюда и название константы – S_ISVTX. Сегодня большинство версий UNIX обладают системой виртуальной памяти и более быстродействующими файловыми системами, поэтому надобность в подобном «закреплении» отпала.

В современных системах назначение бита sticky было расширено. Стандарт Single UNIX Specification допускает установку бита sticky на каталоги. Если бит sticky установлен на каталог, то удалять или изменять файлы в таком каталоге сможет только тот пользователь, который обладает правом на запись в каталог и является владельцем файла, владельцем каталога или суперпользователем.

Типичные примеры каталогов, на которые, как правило, устанавливается бит sticky – /tmp и /var/spool/uucppublic. Обычно любой пользователь может создавать файлы в этих каталогах. Часто эти каталоги доступны для записи, чтения и исполнения всем пользователям, но удалить или изменить файлы смогут только их владельцы.

> Бит saved-text не является частью стандарта POSIX.1. Он входит в состав определяемых стандартом Single UNIX Specification расширений XSI и поддерживается операционными системами FreeBSD 5.2.1, Mac OS X 10.3, Solaris 9 и Linux 2.4.22.
>
> В ОС Solaris 9 бит sticky для обычных файлов имеет специальное значение. Если для файла установлен бит sticky и сброшены биты прав на исполнение, то операционная система не будет кэшировать содержимое этого файла.

4.11. Функции chown, fchown и lchown

Функции семейства chown позволяют изменять идентификаторы пользователя и группы файла.

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int filedes, uid_t owner, gid_t group);

int lchown(const char *pathname, uid_t owner, gid_t group);
```

Все три функции возвращают 0 в случае успеха, –1 в случае ошибки

Эти три функции практически идентичны, за исключением случая, когда они применяются к символическим ссылкам. В случае символической ссылки функция `lchown` изменяет владельца самой символической ссылки, а не файла, на который она указывает.

> Функция `lchown` является расширением XSI, определяемым стандартом Single UNIX Specification. Практически все версии UNIX реализуют эту функцию.

Если один из аргументов, *owner* или *group*, имеет значение –1, то соответствующий идентификатор остается без изменений.

Исторически сложилось так, что в BSD-системах существует ограничение на эту операцию – только суперпользователь может изменить владельца файла. Это предотвращает возможность несанкционированной передачи права на владение файлами другим пользователям и тем самым возможность превышения установленных дисковых квот.

> Стандарт POSIX.1 допускает любой из двух режимов в зависимости от значения константы `_POSIX_CHOWN_RESTRICTED`.
>
> В ОС Solaris 9 режим работы зависит от значения параметра конфигурации, значение по умолчанию – ограниченный режим изменения владельца файла. В операционных системах FreeBSD 5.2.1, Linux 2.4.22 и Mac OS X 10.3 ограниченный режим изменения владельца файла действует всегда.

В разделе 2.6 мы уже упоминали, что константа `_POSIX_CHOWN_RESTRICTED` не всегда определяется в заголовочном файле `<unistd.h>`, но ее значение всегда можно получить с помощью функции `pathconf` или `fpathconf`. Значение этого параметра может зависеть от конкретного файла; кроме того, это ограничение может поддерживаться или не поддерживаться самой файловой системой. Мы будем употреблять выражение «если действует ограничение `_POSIX_CHOWN_RESTRICTED`» в отношении конкретных файлов, о которых идет речь, вне зависимости от того, определено значение константы в заголовочном файле `<unistd.h>` или нет.

Если ограничение `_POSIX_CHOWN_RESTRICTED` действует, то

1. Только процесс, обладающий правами суперпользователя, сможет изменить идентификатор пользователя файла.
2. Процесс, не обладающий правами суперпользователя, сможет изменить идентификатор группы файла, если процесс является владельцем этого файла (эффективный идентификатор пользователя процесса совпадает с идентификатором пользователя файла), аргумент *owner* имеет значение –1 или совпадает с идентификатором пользователя файла и аргумент

group совпадает с эффективным идентификатором группы или с одним из идентификаторов дополнительных групп процесса.

Это означает, что если ограничение _POSIX_CHOWN_RESTRICTED действует, то вы не сможете изменить идентификатор пользователя (владельца) файла. Изменить идентификатор группы файла может только владелец этого файла и только при условии, что он присваивает ему идентификатор одной из групп, к которой принадлежит сам.

Если эти функции вызываются из процесса, не обладающего привилегиями суперпользователя, в случае успешного завершения они сбрасывают биты set-user-ID и set-group-ID.

4.12. Размер файла

Поле st_size структуры stat содержит размер файла в байтах. Это поле имеет смысл только для обычных файлов, каталогов и символических ссылок.

> ОС Solaris поддерживает размер файла также для каналов, он обозначает доступное для чтения количество байт в канале. О каналах речь пойдёт в разделе 15.2.

Обычные файлы могут иметь размер, равный нулю. В этом случае будет получен признак конца файла при первой же операции чтения.

Для каталогов размер файла обычно кратен некоторому числу, такому как 16 или 512. О чтении каталогов мы поговорим в разделе 4.21.

Для символических ссылок размер файла обозначает длину имени файла в байтах. Например, в следующем случае число 7 обозначает длину пути к каталогу usr/lib:

```
lrwxrwxrwx 1 root              7 Sep 25 07:14 lib -> usr/lib
```

(Обратите внимание: символические ссылки не имеют типичного для строк языка С завершающего нулевого символа в конце имени, таким образом, поле st_size всегда определяет длину строки имени файла.)

В большинстве современных версий UNIX есть поля st_blksize и st_blocks. Первое из них определяет оптимальный размер блока для операций ввода-вывода, а второй – фактическое количество 512-байтных блоков, занимаемых файлом. В разделе 3.9 мы определили, что наименьшее время на операции чтения затрачивается, если используется буфер с размером st_blksize. Стандартная библиотека ввода-вывода, которую мы рассмотрим в главе 5, также старается производить операции ввода-вывода блоками по st_blksize байт, что повышает производительность.

> Существуют такие версии UNIX, которые измеряют величину st_blocks не в 512-байтных блоках. Использование этого значения снижает переносимость программ.

Дырки в файлах

В разделе 3.6 мы уже говорили, что обычные файлы могут содержать «дырки». Мы продемонстрировали это на примере программы из листинга 3.2.

Дырки создаются в результате переноса текущей позиции за пределы файла и последующей записи некоторых данных. Рассмотрим следующий пример:

```
$ ls -l core
-rw-r--r-- 1 sar 8483248 Nov 18 12:18 core
$ du -s core
272 core
```

Размер файла `core` превышает 8 Мбайт, хотя команда `du` сообщает о том, что он занимает всего 272 блока по 512 байт (139 264 байта). (Команда `du` в большинстве систем, происходящих от BSD, выводит количество 1024-байтных блоков, тогда как в Solaris – количество 512-байтных блоков.) Очевидно, этот файл содержит много дырок.

Как уже упоминалось в разделе 3.6, функция `read` возвращает значение 0 для байтов, которые фактически не были записаны. Запустив следующую команду, мы увидим, что обычные операции ввода-вывода считывают полное количество байт, соответствующее размеру файла:

```
$ wc -c core
 8483248 core
```

Команда `wc`(1) с ключом `-c` подсчитывает количество символов (байт) в файле.

Если мы скопируем этот файл, например, с помощью утилиты `cat`(1), то все эти дырки будут скопированы как обычные байты данных со значением 0:

```
$ cat core > core.copy
$ ls -l core*
-rw-r--r-- 1 sar 8483248 Nov 18 12:18 core
-rw-rw-r-- 1 sar 8483248 Nov 18 12:27 core.copy
$ du -s core*
272 core
16592 core.copy
```

Здесь фактический размер нового файла составил 8 495 104 (т. е. 512×16 592) байт. Различие между этим числом и размером, выведенным командой `ls`, обусловлено тем, что файловая система использует некоторое количество блоков для хранения указателей на блоки с фактическими данными.

Тем, кому интересны вопросы, связанные с физическим размещением файлов, рекомендуем обратиться к разделу 4.2 [Bach 1986], разделам 7.2 и 7.3 [McKusick 1996] (или к разделам 8.2 и 8.3 в [McKusick and Neville-Neil 2005]) и к разделу 14.2 [Mauro and McDougall 2001].

4.13. Усечение файлов

Иногда возникает необходимость отсечь некоторые данные, расположенные в конце файла. Усечение размера файла до нуля, которое осуществляется при использовании флага `O_TRUNC` функции `open`, есть частный случай усечения файла.

```
#include <unistd.h>
int truncate(const char *pathname, off_t length);
int ftruncate(int filedes, off_t length);
```

Возвращают 0 в случае успеха, –1 в случае ошибки

Эти функции выполняют усечение существующего файла до размера, определяемого аргументом *length*. Если первоначальный размер файла превышал значение *length*, то данные, которые оказались за этим пределом, будут недоступны. Если первоначальный размер файла меньше значения *length*, результат операции зависит от системы, однако XSI-совместимые системы должны увеличить размер файла. Если это действительно происходит, то данные, расположенные между старым и новым концом файла, будут читаться как нули (то есть, вероятно, в файле будет создана «дырка»).

> Функция `ftruncate` является частью стандарта POSIX.1. Функция `truncate` – XSI-расширение базовой функциональности POSIX.1, предусмотренное стандартом Single UNIX Specification.
>
> В версиях BSD, предшествовавших 4.4BSD, функция truncate может только уменьшать размер файла.
>
> ОС Solaris включает в себя расширение `fcntl` (`F_FREESP`), которое позволяет вырезать любую часть файла, а не только ту, что находится в конце.

Мы будем использовать функцию `ftruncate` в программе из листинга 13.2, где необходимо очистить содержимое файла после получения блокировки.

4.14. Файловые системы

Чтобы понять концепцию ссылок на файлы, нам прежде всего необходимо в общих чертах разобраться в устройстве файловой системы UNIX. Также полезно будет понимание разницы между индексным узлом (i-node) и записью в файле каталога, которая указывает на индексный узел.

Сегодня используются самые разные реализации файловых систем UNIX. Например, Solaris поддерживает несколько типов дисковых файловых систем: традиционную для BSD-систем UNIX File System (UFS), DOS-совместимую файловую систему под названием PCFS и файловую систему, предназначенную для компакт-дисков – HSFS. Мы уже видели одно из различий между разными типами файловых систем в табл. 2.15. UFS основана на системе Berkeley fast file system, которая рассматривается в этом разделе.

Представим себе диск, поделенный на несколько разделов. Каждый из разделов может содержать файловую систему, как показано на рис. 4.1.

Индексные узлы – это записи фиксированной длины, которые содержат большую часть сведений о файлах.

Присмотревшись поближе к той части группы цилиндров, где находятся индексные узлы и блоки данных, мы увидим картину, изображенную на рис. 4.2.

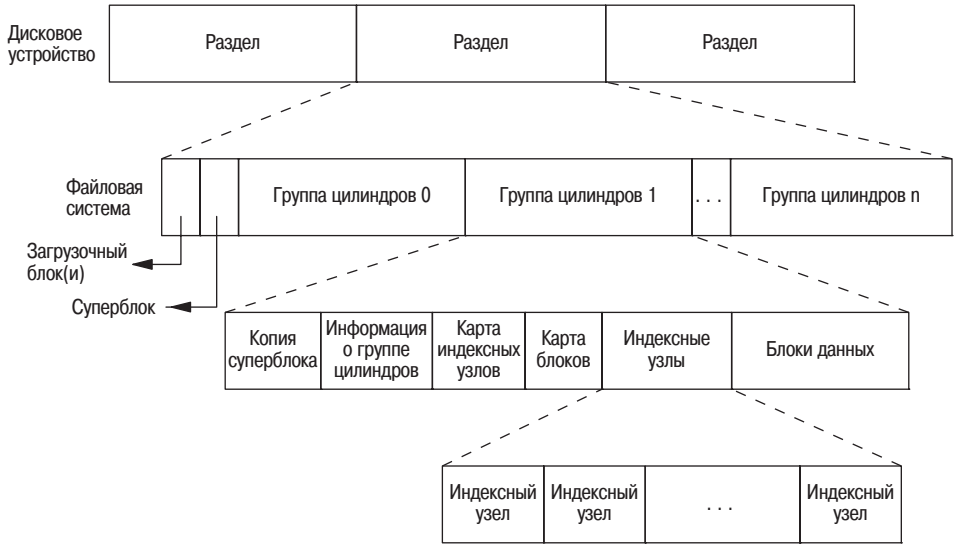


Рис. 4.1. *Дисковое устройство, разделы и файловая система*

Отметим на рис. 4.2 следующие моменты:

- Здесь мы видим две записи в файле каталога, которые ссылаются на один и тот же индексный узел. Каждый индексный узел имеет счетчик ссылок; в этом счетчике хранится число записей в файле каталога, которые ссылаются на данный индексный узел. Только в том случае, если этот счетчик достигнет значения 0, файл будет удален (то есть блоки данных, связанные с файлом, перейдут в список свободных блоков). Поэтому операция «отсоединения файла» (unlink) не всегда приводит к «удалению блоков, ассоциированных с файлом» (delete). Именно по этой причине

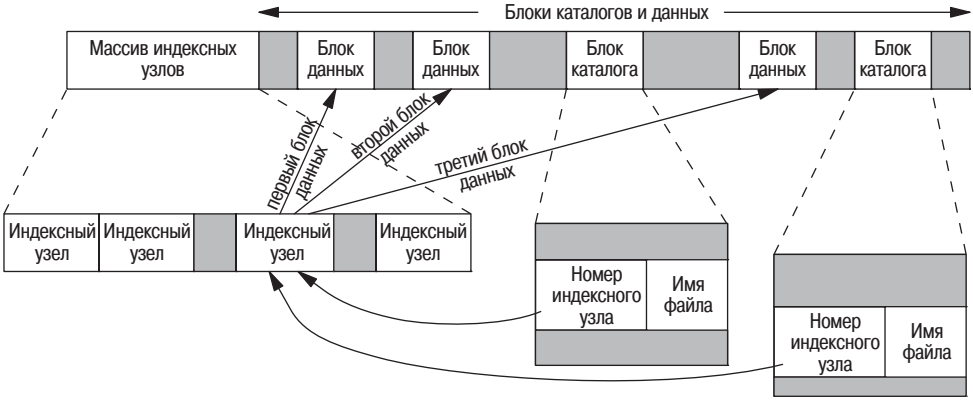


Рис. 4.2. *Часть группы цилиндров с индексными узлами и блоками данных более детально*

функция удаления записей из каталога носит имя unlink (отцепить), а не delete (удалить). В структуре stat счетчик ссылок находится в поле st_nlink. Оно имеет элементарный системный тип nlink_t. Этот тип ссылок называется *жесткими ссылками*. Напоминаем, что в разделе 2.5.2 мы говорили о константе LINK_MAX, которая задает максимальное значение для счетчика ссылок.

- Другой тип ссылок называется *символическими ссылками*. В этом случае фактическое содержимое файла, хранимое в блоках данных, представляет собой имя файла, на который указывает символическая ссылка. В следующем примере имя файла в каталожной записи представлено трехсимвольной строкой lib, сам же файл содержит 7 символов – usr/lib. Тип файла в индексном узле должен быть определен как S_IFLNK, благодаря чему файловая система будет корректно распознавать символические ссылки.

```
lrwxrwxrwx 1 root 7 Sep 25 07:14 lib -> usr/lib
```

- Индексный узел содержит полную информацию о файле: тип файла, биты прав доступа, размер файла, указатели на блоки данных файла и тому подобное. Большая часть информации для структуры stat берется из индексного узла. Только два элемента, которые могут представлять для нас интерес, берутся из записи в файле каталога: имя файла и номер индексного узла. Другие элементы, такие как длина имени файла и размер записи в файле каталога, пока для нас особого интереса не представляют. Тип данных, который хранит номер индексного узла, – ino_t.
- Поскольку номер индексного узла в каталожной записи ссылается на индексный узел, находящийся в той же самой файловой системе, нельзя создать запись, которая указывала бы на индексный узел в другой файловой системе. По этой причине команда ln(1) (создающая новую запись в каталоге, которая указывает на индексный узел существующего файла) не в состоянии создавать ссылки на файлы, расположенные в других файловых системах. Функция link будет рассматриваться в следующем разделе.
- При переименовании/перемещении файла в пределах одной и той же файловой системы фактическое содержимое файла никуда не перемещается. Все, что нужно сделать, это добавить в каталог новую запись, которая будет указывать на существующий индексный узел, а затем отцепить старую запись. При этом значение счетчика ссылок не изменится. Например, чтобы переименовать файл /usr/lib/foo в /usr/foo, нет необходимости перемещать содержимое файла foo, если каталоги /usr и /usr/lib расположены в одной файловой системе. Обычно именно так работает команда mv(1).

Мы только что обсудили смысл счетчика ссылок для обычных файлов, а что означает понятие счетчика ссылок для каталога? Предположим, что мы создаем новый каталог в текущем каталоге:

```
$ mkdir testdir
```

На рис. 4.3 показан результат выполнения этой команды. Обратите внимание: на рисунке мы явно показали наличие записей о каталогах «точка» (текущий каталог) и «точка-точка» (родительский каталог).

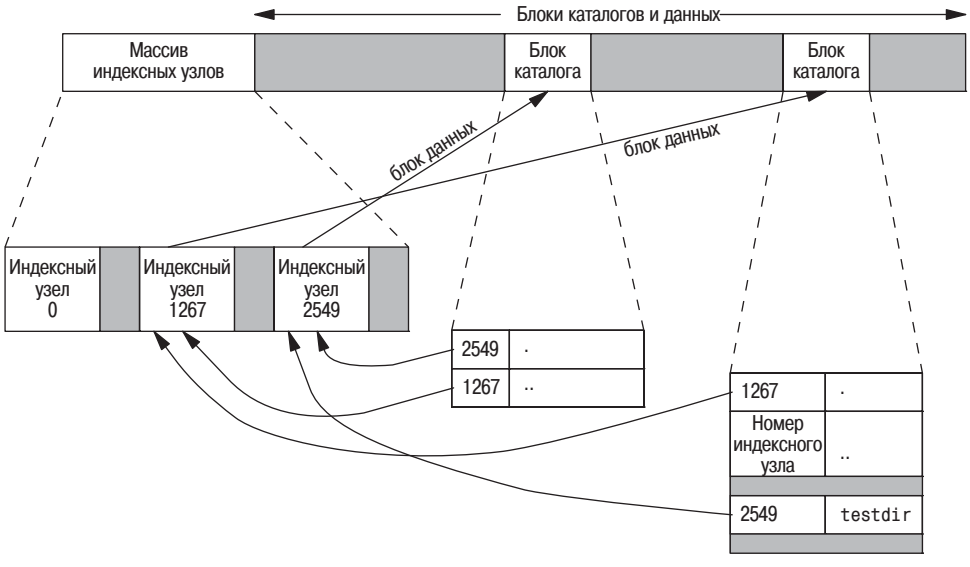


Рис. 4.3. *Группа цилиндров после создания каталога testdir*

Индексный узел с номером 2549 в поле «тип» хранит значение «каталог», а в счетчике ссылок – значение 2. Любой оконечный каталог (который не содержит других каталогов) в счетчике ссылок всегда хранит число 2, потому что на индексный узел ссылаются две каталожные записи: запись, которая указывает на каталог `testdir`, и запись в этом же каталоге, которая указывает на каталог «точка». Индексный узел с номером 1267 в поле «тип» хранит значение «каталог», а в счетчике ссылок – значение 3 или выше. Причина, по которой значение счетчика ссылок больше или равно 3, теперь должна быть нам понятна. Число 3 – это минимальное значение, которое учитывает записи в каталоге верхнего уровня (который на рисунке не показан), в самом каталоге («точка») и в каталоге `testdir` («точка-точка»). Обратите внимание, что появление каждого нового подкаталога увеличивает счетчик ссылок в родительском каталоге на единицу.

Этот формат похож на классический формат файловой системы UNIX, описанный в главе 4 [Bach 1986]. За дополнительной информацией по изменениям, которые появились в Berkeley fast file system, обращайтесь к главе 7 [McKusik 1996] или к главе 8 [McKusik and Neville-Neil 2005]. Подробную информацию о файловой системе UFS вы найдете в главе 14 [Mauro and McDougall 2001].

4.15. Функции link, unlink, remove и rename

Как мы уже говорили в предыдущем разделе, на индексный узел любого файла могут указывать несколько каталожных записей. Такие ссылки создаются с помощью функции `link`.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Эта функция создает в каталоге новую запись с именем *newpath*, которая будет указывать на существующий файл *existingpath*. Если запись с именем *newpath* уже существует, функция вернет признак ошибки. Создается только последний компонент полного пути *newpath*, все промежуточные компоненты должны существовать к моменту вызова функции.

Операции создания новой записи в каталоге и увеличения счетчика ссылок должны выполняться атомарно. (Вспомните обсуждение атомарных операций в разделе 3.11.)

Большинство реализаций требуют, чтобы оба пути находились в пределах одной файловой системы, хотя стандарт POSIX.1 допускает возможность создания ссылок на файлы, расположенные в других файловых системах. Если поддерживается создание жестких ссылок на каталоги, то эта операция может выполняться только суперпользователем. Причина такого ограничения состоит в том, что создание жесткой ссылки на каталог может привести к появлению замкнутых «петель» в файловой системе, и большинство обслуивающих ее утилит не смогут обработать их надлежащим образом. (В разделе 4.16 мы покажем пример замкнутой петли, образованной с помощью символической ссылки.) По этой же причине многие реализации файловых систем вообще не допускают создания жестких ссылок на каталоги.

Удаление записей из каталога производится с помощью функции `unlink`.

```
#include <unistd.h>

int unlink(const char *pathname);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Эта функция удаляет запись из файла каталога и уменьшает значение счетчика ссылок на файл *pathname*. Если на файл указывает несколько ссылок, то его содержимое будет через них по-прежнему доступно. В случае ошибки файл не изменяется.

Как мы уже говорили, чтобы удалить жесткую ссылку на файл, необходимо обладать правом на запись и на исполнение для каталога, в котором находится удаляемая запись. Кроме того, в разделе 4.10 говорилось, что если для каталога установлен бит sticky, то мы должны обладать правом на запись в каталог и являться либо владельцем файла, либо владельцем каталога, либо суперпользователем.

Содержимое файла может быть удалено, только если счетчик ссылок достиг значения 0. Кроме того, содержимое файла нельзя удалить, если он открыт каким-либо процессом. Во время закрытия файла ядро в первую очередь проверяет счетчик процессов, которые открыли этот файл. Если значение

этого счетчика достигло нуля, то ядро проверяет счетчик ссылок и только в том случае, если значение и этого счетчика достигло нуля, содержимое файла будет удалено.

Пример

Программа, приведенная в листинге 4.5, открывает файл, а затем отцепляет его (то есть удаляет с помощью функции `unlink`). После этого программа приостанавливается на 15 секунд и завершает свою работу.

Листинг 4.5. Открывает файл, а затем удаляет его

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("ошибка вызова функции open");
    if (unlink("tempfile") < 0)
        err_sys("ошибка вызова функции unlink");
    printf("файл удален\n");
    sleep(15);
    printf("конец\n");
    exit(0);
}
```

Запуск программы дает следующие результаты:

```
$ ls -l tempfile       посмотрим размер файла
-rw-r----- 1 sar       413265408 Jan 21 07:14 tempfile
$ df /home             проверим объем доступного свободного пространства
Filesystem  1K-blocks     Used   Available  Use%   Mounted on
/dev/hda4    11021440  1956332     9065108   18%   /home
$ ./a.out &            запустим программу из листинга 4.5 как фоновый процесс
1364                   командная оболочка выводит идентификатор процесса
$ файл удален          файл отцеплен
ls -l tempfile         проверим, остался ли файл на месте
ls: tempfile: No such file or directory       запись из каталога была удалена
$ df /home             проверим, освободилось ли дисковое пространство
Filesystem  1K-blocks     Used   Available  Use%   Mounted on
/dev/hda4    11021440  1956332     9065108   18%   /home
$ конец                программа завершила работу, все файлы были закрыты
df /home               теперь должно освободиться пространство на диске
Filesystem  1K-blocks     Used   Available  Use%   Mounted on
/dev/hda4    11021440  1552352     9469088   15%   /home
                       на диске освободилось 394.1 Мб
```

Эта характерная особенность функции `unlink` очень часто используется программами, чтобы обеспечить удаление временных файлов в случае аварийного завершения. Процесс создает файл с помощью функции `open` или `creat`, а затем сразу же вызывает функцию `unlink`. Однако файл не будет удален,

поскольку он остается открытым. Только когда процесс закроет файл или завершит свою работу, что в свою очередь заставит ядро закрыть все файлы, открытые процессом, файл будет удален.

Если аргумент *pathname* является символической ссылкой, то будет удалена сама символическая ссылка, а не файл, на который она ссылается – не существует функции, которая удаляла бы файл по символической ссылке.

Процесс, обладающий привилегиями суперпользователя, может вызвать функцию unlink для удаления каталога, но вообще в таких случаях следует использовать функцию rmdir, которую мы рассмотрим в разделе 4.20.

Удалить жесткую ссылку на файл или каталог можно также с помощью функции remove. Для файлов функция remove абсолютно идентична функции unlink, для каталогов – функции rmdir.

```
#include <stdio.h>

int remove(const char *pathname);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Стандарт ISO C определяет remove как функцию для удаления файлов. Исторически сложившееся в UNIX название unlink было заменено, потому что в то время в большинстве других операционных систем, которые следовали стандарту ISO C, понятие ссылок на файлы не поддерживалось.

Для переименования файла или каталога используется функция rename.

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Стандарт ISO C определяет rename как функцию для переименования файлов. (ISO C вообще не касается каталогов.) Стандарт POSIX.1 расширил это определение, включив в него каталоги и символические ссылки.

Следует отдельно рассмотреть случаи, когда аргумент *oldname* представляет файл, символическую ссылку или каталог. Также необходимо упомянуть о том, что произойдет, если *newname* уже существует.

1. Если аргумент *oldname* указывает на файл, который не является каталогом, то происходит переименование файла или символической ссылки. Если файл *newname* уже существует, он не должен быть каталогом. Если файл *newname* существует и не является каталогом, то он будет удален, а файл *oldname* будет переименован в *newname*. Процесс должен обладать правом записи в каталоги, где находятся файлы *newname* и *oldname*, поскольку предполагается внесение изменений в оба каталога.
2. Если аргумент *oldname* указывает на каталог, то выполняется переименование каталога. Если *newname* существует, то он также должен быть ката-

логом, и этот каталог должен быть пустым. (Под «пустым каталогом» мы имеем в виду каталог, который содержит только две записи: «точка» и «точка-точка».) Если *newname* существует и является пустым каталогом, он будет удален, а каталог *oldname* будет переименован в *newname*. Кроме того, при переименовании каталога аргумент *newname* не должен начинаться с имени каталога *oldname*. Так, например, мы не сможем переименовать каталог `/usr/foo` в `/usr/foo/testdir`, поскольку прежнее имя каталога (`/usr/foo`) содержится в начале нового имени и он не может быть удален.

3. Если аргументы *oldname* или *newname* содержат имя символической ссылки, то будет переименована сама символическая ссылка, а не файл, на который она ссылается.
4. В особом случае, когда аргументы *oldname* и *newname* указывают на один и тот же файл, функция завершается без признака ошибки, но и не производит никаких изменений.

Если файл *newname* уже существует, то мы должны обладать теми же правами, что и для удаления файла. Кроме того, поскольку мы удаляем запись из каталога, который содержит файл *oldname* и создаем новую запись в файле каталога, в котором будет находиться *newname*, мы должны обладать правом на запись и исполнение для обоих каталогов.

4.16. Символические ссылки

Символическая ссылка представляет собой косвенную ссылку на файл, в отличие от жесткой ссылки, которая является прямым указателем на индексный узел файла. Символические ссылки были придуманы с целью обойти ограничения, присущие жестким ссылкам.

- Жесткие ссылки обычно требуют, чтобы ссылка и файл размещались в пределах одной файловой системы
- Только суперпользователь имеет право создавать жесткие ссылки на каталоги

Символические ссылки не имеют ограничений, связанных с файловой системой, и любой пользователь сможет создать символическую ссылку на каталог. Символические ссылки обычно используются для перемещения файлов или даже целой иерархии каталогов в другое местоположение в системе.

> Впервые символические ссылки появились в 4.2BSD и впоследствии стали поддерживаться в SVR4.

Работая с функциями, которые обращаются к файлам по именам, всегда нужно знать, как функция обрабатывает символические ссылки. Если функция следует по символической ссылке, то она будет воздействовать на файл, на который указывает символическая ссылка. В противном случае операция будет производиться над самой символической ссылкой, а не над файлом, на который она указывает. В табл. 4.9 приводится перечень описываемых в этой главе функций, которые следуют по символическим ссылкам. В этом списке отсутствуют функции `mkdir`, `mkfifo`, `mknod` и `rmdir` – они возвращают

признак ошибки, если им в качестве аргумента передается символическая ссылка. Кроме того, функции, которые принимают в качестве аргумента дескриптор файла, такие как fstat и fchmod, также не были включены в список, поскольку в этом случае обработка символических ссылок производится функциями, возвращающими файловые дескрипторы (как правило, open). Следует ли функция chown по символическим ссылкам, зависит от конкретной реализации.

> В старых версиях Linux (до 2.1.81) функция chown не следовала по символическим ссылкам. Начиная с версии 2.1.81 функция chown следует по символическим ссылкам. В операционных системах FreeBSD 5.2.1 и Mac OS X 10.3 функция chown также следует по символическим ссылкам. (В версиях, предшествовавших 4.4BSD, функция chown не следовала по символическим ссылкам, ее поведение было изменено в 4.4.BSD.) В ОС Solaris 9 функция chown также следует по символическим ссылкам. Все четыре платформы предоставляют функцию lchown для изменения владельца самих символических ссылок.

Существует одно исключение, не отмеченное в табл. 4.9, – когда функция open вызывается с установленными одновременно флагами O_CREAT и O_EXCL. Если в этом случае аргумент *pathname* содержит имя символической ссылки, то функция будет завершаться ошибкой с кодом EEXIST. Сделано это с целью закрыть брешь в системе безопасности и предотвратить возможность «обмана» привилегированных процессов путем подмены файлов символическими ссылками.

Таблица 4.9. Интерпретация символических ссылок различными функциями

Функция	Не следует по символической ссылке	Следует по символической ссылке
access		•
chdir		•
chmod		•
chown	•	•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

Пример

С помощью символической ссылки можно создать замкнутую петлю в файловой системе. Большинство функций, анализирующих путь к файлу, обнаружив такую петлю, возвращают в errno код ошибки ELOOP. Рассмотрим следующую последовательность команд:

```
$ mkdir foo                    создать новый каталог
$ touch foo/a                  создать пустой файл
$ ln -s ../foo foo/testdir     создать символическую ссылку
$ ls -l foo
total 0
-rw-r----- 1 sar        0 Jan 22 00:16 a
lrwxrwxrwx 1 sar        6 Jan 22 00:16 testdir -> ../foo
```

Эта последовательность команд создает каталог foo, который содержит файл a и символическую ссылку на каталог foo. На рис. 4.4 приводится схема, на которой каталоги представлены в виде окружностей, а файл – в виде квадрата. Написав простую программу, которая использует стандартную функцию ftw(3) для обхода дерева каталогов и вывода имен всех встретившихся файлов, и запустив ее в ОС Solaris 9, мы получим:

```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
    (и еще много строк, пока не произойдет ошибка с кодом ELOOP)
```

В разделе 4.21 будет представлена версия функции ftw, которая использует функцию lstat вместо stat, чтобы предотвратить следование по символическим ссылкам.

> Обратите внимание: в ОС Linux функция ftw использует функцию lstat, поэтому вы не сможете наблюдать подобный эффект.

Разорвать такую замкнутую петлю не составляет труда. Для этого можно воспользоваться функцией unlink, чтобы удалить файл foo/testdir, так как unlink не следует по символическим ссылкам. Однако, если аналогичная

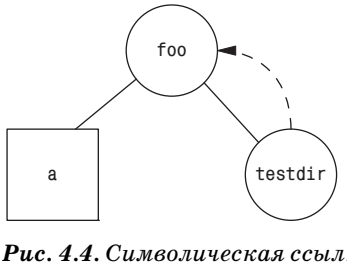


Рис. 4.4. Символическая ссылка testdir создает замкнутую петлю

петля создана с помощью жесткой ссылки, то разорвать ее будет намного сложнее. По этой причине функция link создает жесткие ссылки на каталоги только при наличии у процесса привилегий суперпользователя.

> При написании оригинального текста к этому разделу Ричард Стивенс ради эксперимента действительно создал такую петлю на собственной системе. В результате файловая система была повреждена, и не помогла даже утилита fsck(1). Для восстановления файловой системы пришлось прибегнуть к помощи утилит clri(8) и dcheck(8).
>
> Потребность в жестких ссылках на каталоги давно прошла. Пользователи больше не нуждаются в них – благодаря функции mkdir и символическим ссылкам.

Когда мы открываем файл и передаем функции open имя символической ссылки, функция следует по ссылке и открывает файл, на который она указывает. Если файл, на который указывает ссылка, отсутствует, функция open возвращает признак ошибки, сообщая о невозможности открытия файла. Это может ввести в заблуждение пользователей, которые не знакомы с символическими ссылками, например:

```
$ ln -s /no/such/file myfile        создать символическую ссылку
$ ls myfile
myfile                              команда ls говорит, что файл существует
$ cat myfile                        попробуем заглянуть внутрь файла
cat: myfile: No such file or directory
$ ls -l myfile                      попробуем с ключом -l
lrwxrwxrwx  1 sar        13 Jan 22 00:26 myfile -> /no/such/file
```

Файл myfile существует, однако утилита cat утверждает обратное, потому что myfile – это символическая ссылка, а сам файл, на который она указывает, отсутствует. Запуск команды ls с ключом -l дает нам две подсказки: во-первых, строка вывода ls начинается с символа l, который обозначает символическую ссылку (link), а во-вторых, последовательность -> говорит о том же. У команды ls есть еще один ключ (-F), который добавляет к именам символических ссылок символ «@», благодаря чему можно без труда распознать их даже без ключа -l.

4.17. Функции symlink и readlink

Символические ссылки создаются с помощью функции symlink.

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

В файле каталога создается новая запись *sympath*, которая указывает на файл *actualpath*. Функция не требует существования файла *actualpath* на момент создания символической ссылки. (Мы продемонстрировали эту возможность на примере в предыдущем разделе.) Кроме того, не требуется, чтобы файлы *actualpath* и *sympath* находились в одной и той же файловой системе.

Поскольку функция open следует по символическим ссылкам, нам необходим инструмент, с помощью которого можно было бы открыть саму символическую ссылку, чтобы прочитать имя файла, на который она ссылается. Эти действия выполняет функция readlink.

```
#include <unistd.h>

ssize_t readlink(const char *restrict pathname, char *restrict buf,
                 size_t bufsize);
```

Возвращает количество прочитанных байт в случае успех, −1 в случае ошибки

Эта функция совмещает в себе функции open, read и close. В случае успеха она возвращает количество прочитанных байт, размещенных в *buf*. Строка, помещаемая в буфер *buf*, не завершается нулевым символом.

4.18. Временные характеристики файлов

Каждый файл характеризуется тремя атрибутами времени. Их назначение приводится в табл. 4.10.

Таблица 4.10. Три атрибута времени, связанные с каждым файлом

Поле	Описание	Пример	Ключи команды ls(1)
st_atime	Время последнего доступа к содержимому файла	read	-u
st_mtime	Время последнего изменения содержимого файла	write	По умолчанию
st_ctime	Время последнего изменения статуса индексного узла	chmod, chown	-c

Обратите внимание на различие между временем последнего изменения содержимого файла (st_mtime) и временем последнего изменения статуса индексного узла (st_ctime). Время последнего изменения содержимого файла показывает, когда в последний раз вносились изменения в файл. Время последнего изменения статуса индексного узла определяет время последней модификации индексного узла файла. В этой главе мы упоминали множество операций, которые изменяют индексный узел, не затрагивая при этом содержимое файла: изменение прав доступа, идентификатора пользователя (владельца), количества ссылок на файл и другие. Поскольку информация индексного узла хранится отдельно от содержимого файла, то кроме времени последнего изменения содержимого файла существует и такая характеристика, как время последнего изменения его статуса.

Заметьте, что система не отслеживает время последнего доступа к индексному узлу. По этой причине функции access и stat, например, не изменяют ни одну из трех величин.

Время последнего доступа к файлу часто используется системными администраторами для удаления ненужных файлов, к которым давно никто не обращался. Классический пример – удаление файлов с именами `a.out` и `core`, к которым не обращались более одной недели. Для выполнения подобного рода действий очень часто применяется утилита `find`(1).

Время последнего изменения содержимого файла и время последнего изменения статуса индексного узла могут использоваться для того, чтобы отобрать для архивирования только те файлы, содержимое которых претерпело изменения или у которых был изменен статус индексного узла.

Команда `ls` может отображать или сортировать только по одному из трех значений. По умолчанию при запуске с ключом `-l` или `-t` она использует время последнего изменения содержимого файла. Ключ `-u` заставляет ее использовать время последнего доступа, а ключ `-c` – время последнего изменения статуса индексного узла.

В табл. 4.11 приводится перечень функций и их воздействие на эти три величины. В разделе 4.14 мы уже говорили, что каталог – это на самом деле файл, состоящий из серии записей, каждая из которых содержит имя файла и номер индексного узла файла. Добавление, удаление или изменение этих записей приводит к изменению временных характеристик, связанных с каталогом. По этой причине табл. 4.11 содержит одну колонку для атрибутов времени, связанных с самим файлом или каталогом, и отдельную колонку для атрибутов времени родительского каталога. Создание нового файла, например, воздействует на временные характеристики не только самого файла, но и каталога, в котором этот файл размещается. Однако операции чтения и записи оказывают влияние только на индексный узел файла и никак не сказываются на содержащем этот файл каталоге. (Функции `mkdir` и `rmdir` будут рассматриваться в разделе 4.20. Функция `utime` будет описана в следующем разделе. Шесть функций семейства `exec` будут рассматриваться в разделе 8.10. Функции `mkfifo` и `pipe` мы рассмотрим в главе 15.)

4.19. Функция utime

Функция `utime` изменяет время последнего доступа и время последней модификации файла.

```
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *times);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Эта функция использует следующую структуру:

```
struct utimbuf {
    time_t actime;    /* время последнего доступа */
    time_t modtime;   /* время последнего изменения */
}
```

Таблица 4.11. Воздействие различных функций на время последнего доступа к файлу, последнего изменения содержимого файла и последнего изменения статуса индексного узла

Функция	Файл или каталог			Родительский каталог			Раздел	Примечание
	a	m	c	a	m	c		
chmod, fchmod			•				4.9	
chown, fchown			•				4.11	
creat	•	•	•		•	•	3.4	Создание нового файла (O_CREAT)
creat		•	•				3.4	Усечение существующего файла (O_TRUNC)
exec	•						8.10	
lchown			•				4.11	
link			•		•	•	4.15	Родительский каталог для второго аргумента
mkdir	•	•	•		•	•	4.20	
mkfifo	•	•	•		•	•	15.5	
open	•	•	•		•	•	3.3	Создание нового файла (O_CREAT)
open		•	•				3.3	Усечение существующего файла (O_TRUNC)
pipe	•	•	•				15.2	
read	•						3.7	
remove			•		•	•	4.15	Когда remove = unlink
remove					•	•	4.15	Когда remove = rmdir
rename			•		•	•	4.15	Для обоих аргументов
rmdir					•	•	4.20	
truncate, ftruncate		•	•				4.13	
unlink			•		•	•	4.15	
utime	•	•	•				4.19	
write		•	•				3.8	

Оба поля структуры содержат календарное время, которое выражается количеством секунд, прошедших с начала Эпохи, как это описано в разделе 1.10.

Действие этой функции и привилегии, необходимые для ее выполнения, зависят от того, является ли аргумент *times* пустым указателем.

- Если в аргументе *times* передается пустой указатель (NULL), время последнего доступа к файлу и время последнего изменения файла устанавливаются равными текущему времени. Для этого процесс должен обладать правом записи в файл или иметь эффективный идентификатор пользователя, совпадающий с идентификатором владельца файла.
- Если в аргументе *times* передается не пустой указатель, значения временных характеристик берутся из структуры, на которую указывает аргумент *times*. В этом случае процесс должен иметь эффективный идентификатор пользователя, совпадающий с идентификатором владельца файла, либо обладать привилегиями суперпользователя.

Обратите внимание: невозможно задать время последней модификации статуса индексного узла, так как оно автоматически изменяется в результате вызова функции utime.

В некоторых версиях UNIX команда touch(1) использует эту функцию. Кроме того, стандартные архиваторы tar(1) и cpio(1) могут вызывать функцию utime для установки сохраненных при архивации временных характеристик распакованных файлов.

Пример

Программа, приведенная в листинге 4.6, производит усечение длины файла до нуля, используя функцию open с флагом O_TRUNC, но не изменяет при этом ни время последнего доступа к файлу, ни время последнего изменения файла. Чтобы добиться такого эффекта, программа сначала получает значения временных характеристик файла с помощью функции stat, затем усекает размер файла до нуля и в заключение переустанавливает значения времени с помощью функции utime.

Листинг 4.6. Пример использования функции utime

```
#include "apue.h"
#include <fcntl.h>
#include <utime.h>

int
main(int argc, char *argv[])
{
    int i, fd;
    struct stat statbuf;
    struct utimbuf timebuf;

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) { /* получить значения времени */
            err_ret("%s: ошибка вызова функции stat", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* усечение */
            err_ret("%s: ошибка вызова функции open", argv[i]);
            continue;
```

```
            }
            close(fd);
            timebuf.actime = statbuf.st_atime;
            timebuf.modtime = statbuf.st_mtime;
            if (utime(argv[i], &timebuf) < 0) { /* установить значения времени */
                err_ret("%s: ошибка вызова функции utime", argv[i]);
                continue;
            }
        }
        exit(0);
    }
```

Мы можем продемонстрировать работу программы из листинга 4.6 следующим примером.

```
$ ls -l changemod times          определим размер и время последнего
                                 изменения файлов
-rwxrwxr-x  1 sar   15019 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar   16172 Nov 19 20:05 times
$ ls -lu changemod times         определим время последнего доступа
-rwxrwxr-x  1 sar   15019 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar   16172 Nov 19 20:05 times
$ date                           выведем текущее время и дату
Thu Jan 22 06:55:17 EST 2004
$ ./a.out changemod times        запустим программу из листинга 4.6
$ ls -l changemod times          и проверим результаты
-rwxrwxr-x  1 sar       0 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar       0 Nov 19 20:05 times
$ ls -lu changemod times         проверим так же время последнего доступа
-rwxrwxr-x  1 sar       0 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar       0 Nov 19 20:05 times
$ ls -lc changemod times         и время последнего изменения статуса
                                 индексного узла
-rwxrwxr-x  1 sar       0 Jan 22 06:55 changemod
-rwxrwxr-x  1 sar       0 Jan 22 06:55 times
```

Как мы и ожидали, время последнего доступа к файлу и время последней модификации его содержимого не изменились. Однако время последнего изменения статуса индексного узла было установлено равным времени запуска программы.

4.20. Функции mkdir и rmdir

Создание каталогов производится с помощью функции `mkdir`, а удаление – с помощью функции `rmdir`.

```
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Эта функция создает новый пустой каталог. Записи «точка» и «точка-точка» создаются автоматически. Права доступа к каталогу, задаваемые аргументом *mode*, модифицируются маской режима создания файлов процесса.

Очень часто встречается ошибка, когда аргумент *mode* назначается по аналогии с файлами: выдаются только права на запись и на чтение. Но для каталогов, как правило, необходимо устанавливать хотя бы один бит, дающий право на исполнение, чтобы разрешить доступ к файлам по их именам, находящимся в каталоге (упражнение 4.16).

Идентификаторы пользователя и группы устанавливаются в соответствии с правилами, приведенными в разделе 4.6.

> В операционных системах Solaris 9 и Linux 2.4.22 новый каталог наследует бит set-group-ID от родительского каталога. Файлы, созданные в новом каталоге, наследуют от каталога идентификатор группы. В Linux это поведение определяется реализацией файловой системы. Например, файловые системы `ext2` и `ext3` предоставляют такую возможность при использовании определенных ключей команды `mount`(1). Однако реализация файловой системы UFS для Linux не предполагает возможности выбора: бит set-group-ID наследуется всегда, чтобы имитировать исторически сложившуюся реализацию BSD, где идентификатор группы каталога наследуется от родительского каталога.
>
> Реализации, основанные на BSD, не передают бит set-group-ID по наследству – в них просто наследуется идентификатор группы. Поскольку операционные системы FreeBSD 5.2.1 и Mac OS X 10.3 основаны на 4.4BSD, они не требуют наследования бита set-group-ID. На этих платформах вновь создаваемые файлы и каталоги всегда наследуют идентификатор группы родительского каталога независимо от состояния бита set-group-ID.
>
> В ранних версиях UNIX не было функции `mkdir`. Она впервые появилась в 4.2BSD и SVR3. В более ранних версиях, чтобы создать новый каталог, процесс должен был вызывать функцию `mknod`. Однако использовать эту функцию мог только процесс, обладающий привилегиями суперпользователя. Чтобы как-то обойти это ограничение, обычная команда создания каталога `mkdir`(1) должна была иметь установленный бит set-user-ID и принадлежать пользователю `root`. Чтобы создать каталог из процесса, необходимо было вызывать команду `mkdir`(1) с помощью функции `system`(3).

Удаление пустого каталога производится с помощью функции `rmdir`. Напоминаем, что пустым называется каталог, который содержит только две записи: «точка» и «точка-точка».

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Если в результате вызова этой функции счетчик ссылок на каталог становится равным нулю и при этом никакой процесс не держит каталог открытым, то пространство, занимаемое каталогом, освобождается. Если один или более процессов держат каталог открытым в момент, когда счетчик ссылок достигает значения 0, то функция удаляет последнюю ссылку и перед возвратом управления удаляет записи «точка» и «точка-точка». Кроме того, в таком

каталоге не могут быть созданы новые файлы. Однако файл каталога не удаляется, пока последний процесс не закроет его. (Даже если другой процесс держит каталог открытым, вряд ли он там делает что-то особенное, так как для успешного завершения функции `rmdir` каталог должен был быть пуст.)

4.21. Чтение каталогов

Прочитать информацию из файла каталога может любой, кто имеет право на чтение этого каталога. Но только ядро может выполнять запись в каталоги, благодаря чему обеспечивается сохранность файловой системы. В разделе 4.5 мы утверждали, что возможность создания и удаления файлов в каталоге определяется битами прав на запись и на исполнение, но это не относится к непосредственной записи в файл каталога.

Фактический формат файлов каталогов зависит от реализации UNIX и архитектуры файловой системы. В ранних версиях UNIX, таких как Version 7, структура каталогов была очень простой: каждая запись имела фиксированную длину 16 байт – 14 байт отводилось для имени файла и 2 байта для номера индексного узла. Когда в 4.2BSD была добавлена поддержка более длинных имен файлов, записи стали иметь переменную длину. Это означало, что любая программа, выполняющая прямое чтение данных из файла каталога, попадала в зависимость от конкретной реализации. Чтобы упростить положение дел, был разработан набор функций для работы с каталогами, который стал частью стандарта POSIX.1. Многие реализации не допускают чтения содержимого файлов каталогов с помощью функции `read`, тем самым препятствуя зависимости приложений от особенностей, присущих конкретной реализации.

```
#include <dirent.h>

DIR *opendir(const char *pathname);
```

Возвращает указатель в случае успеха или `NULL` в случае ошибки

```
struct dirent *readdir(DIR *dp);
```

Возвращает указатель в случае успеха, `NULL` в случае достижения конца каталога или ошибки

```
void rewinddir(DIR *dp);

int closedir(DIR *dp);
```

Возвращает 0 в случае успеха или –1 в случае ошибки

```
long telldir(DIR *dp);
```

Возвращает значение текущей позиции в каталоге, ассоциированном с *dp*

```
void seekdir(DIR *dp, long loc);
```

Функции telldir и seekdir не являются частью стандарта POSIX.1. Это расширения XSI стандарта Single UNIX Specification – таким образом, предполагается, что они должны быть реализованы во всех версиях UNIX, следующих этой спецификации.

Как вы помните, некоторые из этих функций использовались в программе из листинга 1.1, которая воспроизводила ограниченную функциональность команды ls.

Структура dirent определена в файле <dirent.h> и зависит от конкретной реализации. Однако в любой версии UNIX эта структура содержит как минимум следующие два поля:

```
struct dirent {
    ino_t d_ino;               /* номер индексного узла */
    char d_name[NAME_MAX + 1]; /* строка имени файла, завершающаяся */
                               /* нулевым символом */
}
```

> Поле d_ino не определено в стандарте POSIX.1, поскольку эта характеристика зависит от конкретной реализации, но оно определяется в расширении XSI базового стандарта POSIX.1. Сам же стандарт POSIX.1 определяет только поле d_name в этой структуре.

Обратите внимание: параметр NAME_MAX не определен как константа в ОС Solaris 9 – его значение зависит от файловой системы, в которой размещается каталог, и определяется, как правило, с помощью функции fpathconf. Наиболее часто встречается значение NAME_MAX, равное 255 (вспомните табл. 2.12). Так как строка имени файла заканчивается нулевым символом, то не имеет значения, как определен массив d_name в заголовочном файле, поскольку размер массива не соответствует длине имени файла.

DIR является внутренней структурой, которая используется этими шестью функциями для хранения информации о каталоге. По своему назначению структура DIR похожа на структуру FILE, используемую функциями стандартной библиотеки ввода-вывода, которая будет описана в главе 5.

Указатель на структуру DIR, возвращаемый функцией opendir, используется в качестве аргумента остальных пяти функций. Функция opendir выполняет первичную инициализацию таким образом, чтобы последующий вызов readdir прочитал первую запись из файла каталога. Порядок следования записей в каталоге, как правило, зависит от реализации и обычно не совпадает с алфавитным.

Пример

Мы воспользуемся этими функциями работы с каталогами при написании программы, которая обходит дерево каталогов. Цель программы состоит в том, чтобы подсчитать количество файлов каждого типа из перечисленных в табл. 4.3. Программа, приведенная в листинге 4.7, принимает единственный параметр – имя начального каталога – и рекурсивно спускается от этой точки вниз по иерархии каталогов. В ОС Solaris имеется функция ftw(3), ко-

торая обходит дерево каталогов, вызывая определяемую пользователем функцию для каждого встреченного файла. Но с ней связана одна проблема: она вызывает функцию stat для каждого файла, в результате чего программа следует по символическим ссылкам. Например, если мы начнём просмотр каталогов от корня файловой системы, в котором имеется символическая ссылка с именем /lib, указывающая на каталог /usr/lib, то все файлы в каталоге /usr/lib будут сосчитаны дважды. Чтобы устранить эту проблему, ОС Solaris предоставляет дополнительную функцию nftw(3), для которой можно отключить следование по символическим ссылкам. Хотя можно было бы бы использовать функцию nftw, давайте все-таки напишем собственную версию функции для обхода дерева каталогов, чтобы показать принципы работы с каталогами.

> Обе функции, ftw и nftw, включены в стандарт Single UNIX Specification как расширения XSI базового стандарта POSIX.1. Реализации этих функций имеются в операционных системах Solaris 9 и Linux 2.4.22. Системы, основанные на BSD, предоставляют функцию fts(3) с аналогичной функциональностью. Она реализована в операционных системах FreeBSD 5.2.1, Mac OS X 10.3 и Linux 2.4.22.

Листинг 4.7. Рекурсивный обход дерева каталогов с подсчетом количества файлов по типам

```
#include "apue.h"
#include <dirent.h>
#include <limits.h>

/* тип функции, которая будет вызываться для каждого встреченного файла */
typedef int Myfunc(const char *, const struct stat *, int);

static Myfunc  myfunc;
static int     myftw(char *, Myfunc *);
static int     dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

int
main(int argc, char *argv[])
{
    int ret;

    if (argc != 2)
        err_quit("Использование: ftw <начальный_каталог>");

    ret = myftw(argv[1], myfunc); /* выполняет всю работу */

    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
    if (ntot == 0)
        ntot = 1;/* во избежание деления на 0; вывести 0 для всех счетчиков */
    printf("обычные файлы = %7ld, %5.2f %%\n", nreg,
            nreg*100.0/ntot);
    printf("каталоги = %7ld, %5.2f %%\n", ndir,
            ndir*100.0/ntot);
    printf("специальные файлы блочных устройств = %7ld, %5.2f %%\n", nblk,
```

```
            nblk*100.0/ntot);
    printf("специальные файлы символьных устройств = %7ld, %5.2f %%\n", nchr,
            nchr*100.0/ntot);
    printf("FIFO = %7ld, %5.2f %%\n", nfifo,
            nfifo*100.0/ntot);
    printf("символические ссылки = %7ld, %5.2f %%\n", nslink,
            nslink*100.0/ntot);
    printf("сокеты = %7ld, %5.2f %%\n", nsock,
        nsock*100.0/ntot);
    exit(ret);
}

/*
 * Обойти дерево каталогов, начиная с каталога "pathname".
 * Пользовательская функция func() вызывается для каждого встреченного файла.
 */

#define FTW_F   1    /* файл, не являющийся каталогом */
#define FTW_D   2    /* каталог */
#define FTW_DNR 3    /* каталог, который не доступен для чтения */
#define FTW_NS  4    /* файл, информацию о котором */
                     /* невозможно получить с помощью stat */

static char *fullpath; /* полный путь к каждому из файлов */

static int             /* возвращаем то, что вернула функция func() */
myftw(char *pathname, Myfunc *func)
{
    int len;

    fullpath = path_alloc(&len); /* выделить память для PATH_MAX+1 байт */
                                 /* (листинг 2.3) */
    strncpy(fullpath, pathname, len); /* защита от */
    fullpath[len-1] = 0;              /* переполнения буфера */

    return(dopath(func));
}

/*
 * Обход дерева каталогов, начиная с "fullpath". Если "fullpath" не является
 * каталогом, для него вызывается lstat(), func() и затем выполняется возврат.
 * Для каталогов производится рекурсивный вызов функции.
 */
static int         /* возвращаем то, что вернула функция func() */
dopath(Myfunc* func)
{
    struct stat    statbuf;
    struct dirent *dirp;
    DIR           *dp;
    int            ret;
    char          *ptr;

    if (lstat(fullpath, &statbuf) < 0) /* ошибка вызова функции stat */
        return(func(fullpath, &statbuf, FTW_NS));
```

```
    if (S_ISDIR(statbuf.st_mode) == 0) /* не каталог */
        return(func(fullpath, &statbuf, FTW_F));

/*
 * Это каталог. Сначала вызовем функцию func(),
 * а затем обработаем все файлы в этом каталоге.
 */
    if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);

    ptr = fullpath + strlen(fullpath); /* установить указатель */
                                       /* в конец fullpath */
    *ptr++ = '/';
    *ptr = 0;

    if ((dp = opendir(fullpath)) == NULL) /* каталог недоступен */
        return(func(fullpath, &statbuf, FTW_DNR));

    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0 ||
            strcmp(dirp->d_name, "..") == 0)
                continue;              /* пропустить каталоги "." и ".." */

        strcpy(ptr, dirp->d_name);     /* добавить имя после слэша */

        if ((ret = dopath(func)) != 0) /* рекурсия */
            break;                     /* выход по ошибке */
    }

    ptr[-1] = 0; /* стереть часть строки от слэша и до конца */

    if (closedir(dp) < 0)
        err_ret("невозможно закрыть каталог %s", fullpath);

    return(ret);
}

static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG: nreg++; break;
        case S_IFBLK: nblk++; break;
        case S_IFCHR: nchr++; break;
        case S_IFIFO: nfifo++; break;
        case S_IFLNK: nslink++; break;
        case S_IFSOCK: nsock++; break;
        case S_IFDIR:
            err_dump("признак S_IFDIR для %s", pathname);
                    /* каталоги должны иметь тип = FTW_D */
    }
    break;

    case FTW_D:
```

```
            ndir++;
            break;

        case FTW_DNR:
            err_ret("закрыт доступ к каталогу %s", pathname);
            break;

        case FTW_NS:
            err_ret("ошибка вызова функции stat для %s", pathname);
            break;

        default:
            err_dump("неизвестный тип %d для файла %s", type, pathname);
        }

        return(0);
    }
```

Эта программа получилась даже более универсальной, чем было необходимо. Таким образом мы хотели пояснить работу функции `ftw`. Например, функция `myfunc` всегда возвращает значение 0, хотя функция, которая ее вызывает, готова обработать и ненулевое значение.

За дополнительной информацией о технике обхода дерева каталогов и использовании ее в стандартных командах UNIX – `find`, `ls`, `tar` и других – обращайтесь к [Fowler, Korn and Vo 1989].

4.22. Функции chdir, fchdir и getcwd

Для каждого процесса определен текущий рабочий каталог. Относительно этого каталога вычисляются все относительные пути (то есть пути, которые не начинаются с символа слэша). Когда пользователь входит в систему, текущим рабочим каталогом обычно становится каталог, указанный в шестом поле записи из файла `/etc/passwd`, – домашний каталог пользователя. Текущий рабочий каталог – это атрибут процесса, домашний каталог – атрибут пользователя.

Процесс может изменить текущий рабочий каталог с помощью функции `chdir` или `fchdir`.

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int filedes);
```

Возвращают 0 в случае успеха, –1 в случае ошибки

Новый рабочий каталог может быть представлен как в виде строки *pathname*, так и файловым дескриптором.

> Функция `fchdir` не является частью базовой спецификации стандарта POSIX.1. Это расширение XSI из стандарта Single UNIX Specification. Все четыре платформы, обсуждаемые в книге, поддерживают функцию `fchdir`.

Пример

Поскольку текущий рабочий каталог является атрибутом процесса, то вызов функции chdir в дочернем процессе никак не влияет на текущий рабочий каталог родительского процесса. (Отношения между процессами более подробно мы рассмотрим в главе 8.) Это означает, что программа, приведенная в листинге 4.8, работает не так, как мы ожидаем.

Листинг 4.8. Пример использования функции chdir

```
#include "apue.h"

int
main(void)
{
    if (chdir("/tmp") < 0)
        err_sys("ошибка вызова функции chdir");
    printf("каталог /tmp стал текущим рабочим каталогом\n");
    exit(0);
}
```

После компиляции и запуска этой программы мы получим следующие результаты (mycd – исполняемый файл программы):

```
$ pwd
/usr/lib
$ mycd
каталог /tmp стал текущим рабочим каталогом
$ pwd
/usr/lib
```

Текущий рабочий каталог командной оболочки, которая запустила программу mycd, не изменился. Это побочный эффект способа запуска программ командной оболочкой. Каждая программа выполняется как отдельный процесс, благодаря чему текущий рабочий каталог самой командной оболочки нельзя изменить вызовом функции chdir из программы. По этой причине функция chdir должна вызываться самой командной оболочкой, для чего командные оболочки предоставляют встроенную команду cd.

Поскольку ядро хранит сведения о текущем рабочем каталоге, должен быть способ получить его текущее значение. К сожалению, ядро хранит не полный путь к каталогу, а некоторую иную информацию, такую как указатель на виртуальный узел (v-node) каталога.

Чтобы определить абсолютный путь к текущему рабочему каталогу, нужна функция, которая будет перемещаться вверх по дереву каталогов, начиная с текущего («точка») и далее через специальные каталоги «точка-точка», пока не достигнет корневого каталога. В каждом из промежуточных каталогов функция будет читать записи из файла каталога, пока не найдет название, которое соответствует индексному узлу предыдущего каталога. Повторяя эту процедуру до тех пор, пока не будет достигнут корневой каталог, мы в результате получим абсолютный путь к текущему рабочему каталогу. К счастью, такая функция уже существует.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Возвращает указатель на *buf* в случае успеха, NULL в случае ошибки

В эту функцию мы должны передать адрес буфера *buf* и его размер в байтах. Буфер должен быть достаточно большим, чтобы вместить строку абсолютного пути к каталогу плюс завершающий нулевой символ. (Проблему выделения памяти для строки абсолютного пути к файлу мы уже обсуждали в разделе 2.5.5.)

> Некоторые старые версии UNIX допускают в качестве указателя на буфер передавать значение NULL. В этом случае функция сама выделяет для буфера память размером *size* байт с помощью функции malloc. Такое поведение не предусматривается стандартами POSIX.1 или Single UNIX Specification, и его не следует использовать в программах.

Пример

Программа, представленная в листинге 4.9, переходит в определенный каталог, после чего вызывает getcwd и выводит строку пути к текущему рабочему каталогу. После запуска программы мы получили следующее:

```
$ ./a.out
cwd = /var/spool/uucppublic
$ ls -l /usr/spool
lrwxrwxrwx 1 root 12 Jan 31 07:57 /usr/spool -> ../var/spool
```

Листинг 4.9. Пример использования функции getcwd

```
#include "apue.h"

int
main(void)
{
    char  *ptr;
    int   size;

    if (chdir("/usr/spool/uucppublic") < 0)
        err_sys("ошибка вызова функции chdir");

    ptr = path_alloc(&size);         /* наша собственная функция */
    if (getcwd(ptr, size) == NULL)
        err_sys("ошибка вызова функции");

    printf("cwd = %s\n", ptr);
    exit(0);
}
```

Обратите внимание: функция getcwd следует по символическим ссылкам, как это и должно быть в соответствии с табл. 4.9, но она понятия не имеет, что попала в каталог /var/spool по символической ссылке /usr/spool. Это одна из особенностей символических ссылок.

Функция getcwd очень удобна для приложений, в которых возникает необходимость возврата к первоначальному текущему каталогу. Для этого перед сменой текущего рабочего каталога нужно вызвать getcwd и сохранить полученное значение. По окончании работы мы можем передать сохраненную строку функции chdir и вернуться в первоначальный рабочий каталог.

Функция fchdir предоставляет еще более простой способ решения этой задачи. Вместо вызова функции getcwd можно открыть текущий каталог, сохранить файловый дескриптор и после этого сменить текущий каталог. Когда возникнет необходимость вернуться к первоначальному местоположению, остается просто передать дескриптор функции fchdir.

4.23. Специальные файлы устройств

Очень часто возникает путаница с полями st_dev и st_rdev. Нам они потребуются в разделе 18.9 при написании функции ttyname. Правила для их различения очень просты.

- Каждая файловая система характеризуется старшим и младшим номерами устройства, которые представлены элементарным системным типом dev_t. Старший номер устройства идентифицирует драйвер устройства и иногда указывает, с какой платой периферийного устройства следует взаимодействовать. Младший номер идентифицирует конкретное подустройство. На рис. 4.1 было показано, что на одном и том же дисковом устройстве может размещаться несколько файловых систем. Все файловые системы на одном и том же дисковом устройстве, как правило, имеют одинаковые старшие номера, но различные младшие номера устройства.
- Обычно старший и младший номера устройства можно получить с помощью макросов, определенных в большинстве реализаций: major и minor. Это означает, что нам не нужно задумываться о том, как хранятся два номера в одной переменной типа dev_t.

  В ранних версиях UNIX старший и младший номера устройств хранились в виде 16-битного целого числа, в котором 8 бит отводилось для старшего номера и 8 бит – для младшего номера устройства. FreeBSD 5.2.1 и Mac OS X 10.3 используют для этих целей 32-битные целые числа, в которых для хранения старшего номера устройства отводится 8 бит, а для младшего номера устройства – 24 бита. На 32-битных платформах Solaris 9 использует 32-битные целые числа, в которых для старшего номера отводится 14 бит, а для младшего номера – 18 бит. На 64-битных платформах Solaris 9 использует 64-битные целые числа, в которых под каждый номер отводится по 32 бита. В Linux 2.4.22, несмотря на то, что тип dev_t определен как 64-битное целое число, под старший и младший номера в настоящее время отводится по 8 бит.

  Стандарт POSIX.1 оговаривает существование типа dev_t, но не определяет формат хранения и способ интерпретации его содержимого. В большинстве систем для этих целей существуют макросы major и minor, но имя заголовочного файла, в котором они определены, зависит от конкретной системы. В BSD-системах их определения находятся в файле <sys/types.h>. В ОС Solaris они определены в файле <sys/mkdev.h>, в Linux – в файле <sys/sysmacros.h>, который подключается в файле <sys/types.h>.

- В поле st_dev для каждого файла хранится номер устройства файловой системы, в которой располагается файл и соответствующий ему индексный узел.
- Поле st_rdev имеет определенное значение только для специальных файлов символьных или блочных устройств. В этом поле хранится номер фактического устройства, представленного файлом.

Пример

Программа, представленная листингом 4.10, выводит номера устройств для каждого из аргументов командной строки. Кроме того, если аргумент представляет специальный файл блочного или символьного устройства, то дополнительно выводится содержимое поля st_rdev.

Листинг 4.10. Вывод содержимого полей st_dev и st_rdev

```
#include "apue.h"
#ifdef SOLARIS
#include <sys/mkdev.h>
#endif

int
main(int argc, char *argv[])
{
    int   i;
    struct stat buf;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (stat(argv[i], &buf) < 0) {
            err_ret("ошибка вызова функции stat");
            continue;
        }
        printf("dev = %d/%d", major(buf.st_dev), minor(buf.st_dev));
        if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
            printf(" (%s) rdev = %d/%d",
                    (S_ISCHR(buf.st_mode)) ? "симв. устр." : "блочное устр.",
                    major(buf.st_rdev), minor(buf.st_rdev));
        }
        printf("\n");
    }

    exit(0);
}
```

Запуск этой программы дает следующие результаты:

```
$ ./a.out / /home/sar /dev/tty[01]
/: dev = 3/3
/home/sar: dev = 3/4
/dev/tty0: dev = 0/7 (симв. устр.) rdev = 4/0
/dev/tty1: dev = 0/7 (симв. устр.) rdev = 4/1
$ mount                          какие устройства в какие каталоги смонтированы?
```

```
/dev/hda3 on / type ext2 (rw,noatime)
/dev/hda4 on /home type ext2 (rw,noatime)
$ ls -lL /dev/tty[01] /dev/hda[34]
brw------- 1 root        3,   3 Dec 31  1969 /dev/hda3
brw------- 1 root        3,   4 Dec 31  1969 /dev/hda4
crw------- 1 root        4,   0 Dec 31  1969 /dev/tty0
crw------- 1 root        4,   1 Jan 18 15:36 /dev/tty1
```

Первые два аргумента программы – это каталоги (/ и /home/sar), другие два – специальные файлы устройств /dev/tty[01]. (Мы воспользовались регулярными выражениями языка командной оболочки, чтобы сократить объем вводимого с клавиатуры текста. Командная оболочка преобразует строку /dev/tty[01] в /dev/tty0 /dev/tty1.)

Мы предполагаем, что специальные файлы представляют символьные устройства. Наша программа показала, что номера устройств для каталогов / и /home/sar различны, следовательно, они находятся в разных файловых системах. Это подтверждается командой mount(1).

Затем мы воспользовались командой ls, чтобы отыскать дисковые устройства, о которых нам сообщила команда mount, и терминальные устройства. Два дисковых устройства представлены специальными файлами блочных устройств, терминальные устройства – специальными файлами символьных устройств. (Обычно файлы блочных устройств представляют устройства, которые могут содержать файловые системы с произвольным доступом к данным – жесткие диски, накопители на гибких магнитных дисках, CD-ROM. Некоторые старые версии UNIX поддерживали накопители на магнитных лентах, но они не получили широкого распространения.)

Обратите внимание: имена файлов и индексные узлы терминальных устройств (st_dev) находятся на устройстве 0/7, в псевдофайловой системе devfs, которая реализована в виде каталога /dev, но их фактические номера устройств – 4/0 и 4/1.

4.24. Коротко о битах прав доступа к файлам

Мы рассмотрели все биты прав доступа к файлам, некоторые из которых могут иметь множество интерпретаций. В табл. 4.12 приводится полный перечень битов прав доступа и их интерпретация для обычных файлов и для каталогов.

И наконец девять констант, которые могут быть сгруппированы по три:

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

Таблица 4.12. Перечень битов прав доступа к файлам

Константа	Описание	Значение для обычных файлов	Значение для каталогов
S_ISUID	set-user-ID	Устанавливает эффективный идентификатор пользователя при исполнении	(Не используется)
S_ISGID	set-group-ID	Если установлен бит group-execute, то устанавливает эффективный идентификатор группы при исполнении, в противном случае включает режим обязательной блокировки файла или отдельных записей (если поддерживается)	Устанавливает идентификатор группы для файлов, создаваемых в этом каталоге, в соответствии с идентификатором группы самого каталога
S_ISVTX	бит sticky	Управляет кэшированием содержимого файлов (если поддерживается)	Ограничивает возможность удаления и переименования файлов в каталоге
S_IRUSR	user-read	Разрешает пользователю читать файл	Разрешает пользователю читать записи в файле каталога
S_IWUSR	user-write	Разрешает пользователю писать в файл	Разрешает пользователю удалять и создавать файлы в каталоге
S_IXUSR	user-execute	Разрешает пользователю запускать файл на исполнение	Разрешает пользователю производить поиск по каталогу
S_IRGRP	group-read	Разрешает группе читать файл	Разрешает группе читать записи в файле каталога
S_IWGRP	group-write	Разрешает группе писать в файл	Разрешает группе удалять и создавать файлы в каталоге
S_IXGRP	group-execute	Разрешает группе запускать файл на исполнение	Разрешает группе производить поиск по каталогу
S_IROTH	other-read	Рразрешает всем остальным читать файл	Разрешает всем остальным читать записи в файле каталога
S_IWOTH	other-write	Разрешает всем остальным писать в файл	Разрешает всем остальным удалять и создавать файлы в каталоге
S_IXOTH	other-execute	Разрешает всем остальным запускать файл на исполнение	Разрешает всем остальным производить поиск по каталогу

4.25. Подведение итогов

Основным предметом обсуждения в этой главе была функция stat. Мы детально рассмотрели каждое поле структуры stat. Это, в свою очередь, заставило нас исследовать все существующие в UNIX атрибуты файлов. Уверенное знание всех свойств файла и всех функций, которые работают с файлами, составляет основу программирования в системе UNIX.

Упражнения

4.1. Измените программу, представленную листингом 4.1, таким образом, чтобы вместо функции lstat она использовала функцию stat. Что изменится, если в качестве аргумента командной строки передать программе символическую ссылку?

4.2. Что произойдет, если маску режима создания файлов задать равной 777 (в восьмеричном представлении)? Проверьте результаты с помощью команды umask.

4.3. Убедитесь, что при сброшенном бите user-read вы не сможете прочитать свои собственные файлы.

4.4. Запустите программу, представленную листингом 4.3, после того как будут созданы файлы foo и bar. Что произойдет в этом случае?

4.5. В разделе 4.12 мы говорили, что нулевой размер для обычных файлов вполне допустим. Мы также говорили о том, что поле st_size имеет определенный смысл для каталогов и символических ссылок. Могут ли существовать каталоги или символические ссылки с нулевым размером?

4.6. Напишите утилиту, аналогичную cp(1), которая копировала бы файлы с дырками, не записывая байты со значением 0 в выходной файл.

4.7. Взгляните на вывод команды ls в разделе 4.12: файлы core и core.copy имеют различные права доступа. Объясните, как могли появиться такие различия, если исходить из предположения, что в промежутке времени между созданием этих файлов значение umask не изменялось.

4.8. При запуске программы из листинга 4.5 мы проверяли доступный объем дискового пространства с помощью команды df(1). Почему нельзя было воспользоваться командой du(1)?

4.9. Таблица 4.11 утверждает, что функция unlink воздействует на время последнего изменения статуса индексного узла. Как это может быть?

4.10. Как влияет системный предел количества одновременно открытых файлов на функцию myftw из раздела 4.21?

4.11. Наша версия функции myftw никогда не покидает текущий каталог. Измените эту функцию таким образом, чтобы она каждый раз, когда встречает каталог, вызывала функцию chdir для перехода в этот каталог, чтобы передавать функции lstat не полный путь к файлу, а только его имя. После обработки всех файлов в каталоге вызовите chdir("..."). Сравните время работы этих двух версий.

4.12. Для каждого процесса определен также корневой каталог, который используется в качестве отправной точки при разрешении абсолютных путей к файлам. Корневой каталог процесса может быть изменен с помощью функции `chroot`. Найдите описание этой функции в своем справочном руководстве. В каких случаях она может быть полезна?

4.13. Как с помощью функции `utime` изменить только один атрибут времени из двух?

4.14. Некоторые версии команды `finger`(1) выводят сообщения `"New mail received ..."` и `"unread since ..."`, где многоточием обозначено соответствующее время и дата. Как программа может определить эти время и дату?

4.15. Изучите различные форматы архивов, создаваемых командами `cpio`(1) и `tar`(1). (Описание их обычно можно найти в разделе 5 «UNIX Programmer's Manual».) Какие временные характеристики файлов могут быть сохранены в архиве? Какое значение времени последнего доступа к файлу будет установлено при его разархивировании и почему?

4.16. Существует ли в UNIX фундаментальное ограничение на количество вложенных каталогов? Чтобы узнать это, напишите программу, которая в цикле будет создавать новый каталог и сразу же выполнять переход в него. Убедитесь, что длина строки абсолютного пути к последнему каталогу превышает системный предел `PATH_MAX`. Есть ли возможность вызвать функцию `getcwd` из последнего каталога, чтобы получить абсолютный путь к нему? Как стандартные утилиты UNIX работают с такими длинными путями? Возможно ли заархивировать такое дерево каталогов с помощью `tar` или `cpio`?

4.17. В разделе 3.16 мы описали специальный каталог `/dev/fd/`. Чтобы любой пользователь смог обращаться к файлам в этом каталоге, для них должны быть установлены права доступа `rw-rw-rw-`. Некоторые программы перед созданием нового файла сначала удаляют его, если он уже существует, игнорируя при этом возвращаемое значение функции. Это делается следующим образом:

```
unlink(path);
if ((fd = creat(path, FILE_MODE)) < 0)
    err_sys(...);
```

Что произойдет, если в аргументе `path` передать путь `/dev/fd/1`?

5

Стандартная библиотека ввода-вывода

5.1. Введение

В этой главе мы рассмотрим стандартную библиотеку ввода-вывода. Эта библиотека определена стандартом ISO C, поскольку она реализована на многих операционных системах, не относящихся к семейству UNIX. Стандарт Single UNIX Specification определяет для нее дополнительные интерфейсы в качестве расширений стандарта ISO C.

Стандартная библиотека ввода-вывода сама производит размещение буферов и выполняет операции ввода-вывода блоками оптимального размера, что избавляет нас от необходимости задумываться о правильности выбора размера буфера (раздел 3.9). Таким образом, эта библиотека удобна в использовании, но в то же время неумелое обращение с ней может стать источником других проблем.

> Стандартная библиотека ввода-вывода была написана Деннисом Ритчи примерно в 1975 году. Это была генеральная ревизия библиотеки Portable I/O library Майка Леска (Mike Lesk). Удивительно, насколько несущественно изменилась библиотека за последние 30 лет.

5.2. Потоки и объекты FILE

Все функции, описанные в главе 3, работали с файлами посредством дескрипторов. При открытии файла возвращается файловый дескриптор, который затем используется во всех последующих операциях ввода-вывода. При обсуждении стандартной библиотеки ввода-вывода мы будем отталкиваться от термина *поток*. (Не путайте стандартный термин поток (stream) ввода-вывода с системой ввода-вывода STREAMS, которая является частью System V и стандартизирована в дополнении XSI STREAMS к стандарту Single UNIX Specification.) Открыв или создав файл средствами стандартной библиотеки ввода-вывода, мы говорим, что связали поток с файлом.

В наборе символов ASCII каждый символ представлен одним байтом. В национальных наборах символов один символ может быть представлен большим количеством байт. Стандартные файловые потоки ввода-вывода могут использоваться как с однобайтными, так и с многобайтными («wide» – «широкий») наборами символов. Ориентация потока определяет, являются ли читаемые и записываемые символы однобайтными или многобайтными. Изначально, когда поток создается, он не имеет ориентации. Если с неориентированным потоком ввода-вывода используется функция для работы с многобайтными символами (<wchar.h>), то для потока устанавливается ориентация на «широкие» символы. Если с неориентированным потоком ввода-вывода используется функция для работы с однобайтными символами, то для потока устанавливается ориентация на однобайтные символы. Изменить установленную ориентацию могут только две функции. Функция `freopen` (которую мы вскоре рассмотрим) сбрасывает ориентацию потока, а функция `fwide` может установить ориентацию потока.

```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *fp, int mode);
```

Возвращает положительное число, если поток ориентирован на многобайтные символы, отрицательное число, если поток ориентирован на однобайтные символы, или 0, если поток не имеет ориентации

Функция `fwide` может решать различные задачи в зависимости от значения аргумента *mode*.

- Если аргумент *mode* – отрицательное число, функция `fwide` попытается назначить потоку ориентацию на однобайтные символы.
- Если аргумент *mode* – положительное число, функция `fwide` попытается назначить потоку ориентацию на многобайтные символы.
- Если аргумент *mode* равен 0, функция `fwide` не будет менять ориентацию потока, она просто вернет значение, соотвутствующее текущей ориентации.

Обратите внимание: функция `fwide` не может изменить ориентацию уже ориентированного потока. Кроме того, она не возвращает признак ошибки. Единственное, что тут можно сделать, – это очистить переменную `errno` перед вызовом функции `fwide` и затем проверить значение этой переменной после вызова функции. На протяжении оставшейся части книги мы будем иметь дело только с потоками, ориентированными на однобайтные символы.

При открытии потока стандартная функция `fopen` возвращает указатель на объект `FILE`. Этот объект, как правило, является структурой, которая содержит всю информацию, необходимую для управления потоком средствами стандартной библиотеки ввода-вывода: дескриптор файла, используемый в операциях ввода-вывода, указатель на буфер потока, размер буфера, счетчик символов, находящихся в настоящий момент в буфере, флаг ошибки и т. д.

Прикладные программы никогда не работают с объектом FILE напрямую. Чтобы сослаться на поток, нужно просто передать указатель на объект FILE в виде аргумента любой стандартной функции ввода-вывода. Далее в тексте книги указатель на объект FILE, тип FILE *, мы будем называть указателем на файл.

В этой главе стандартная библиотека ввода-вывода будет обсуждаться в контексте ОС UNIX. Как мы уже упоминали, эта библиотека перенесена на самые разные платформы. Но чтобы вы получили некоторое представление о том, как эта библиотека может быть реализована, мы будем отталкиваться от типичной ее реализации в системе UNIX.

5.3. Стандартные потоки ввода, вывода и сообщений об ошибках

Для любого процесса автоматически создается три предопределенных потока: стандартный поток ввода, стандартный поток вывода и стандартный поток сообщений об ошибках. Эти потоки связаны с теми же файлами, что и дескрипторы STDIN_FILENO, STDOUT_FILENO и STDERR_FILENO, которые упоминались в разделе 3.2.

Доступ к этим трем потокам осуществляется посредством предопределенных указателей на файлы stdin, stdout и stderr. Определения файловых указателей находятся в заголовочном файле <stdio.h>.

5.4. Буферизация

Буферизация, предоставляемая стандартной библиотекой ввода-вывода, предназначена для того, чтобы минимизировать количество обращений к функциям read и write. (В табл. 3.2 мы приводили зависимость производительности операций ввода-вывода от размера буфера.) Кроме того, библиотека стремится произвести буферизацию потоков ввода-вывода автоматически, чтобы избавить приложения от необходимости беспокоиться о ней. К сожалению, буферизация – это тот самый аспект стандартной библиотеки ввода-вывода, который более всего смущает программистов.

Библиотека предоставляет три типа буферизации:

1. Полная буферизация. В этом случае фактический ввод-вывод осуществляется только тогда, когда будет заполнен стандартный буфер ввода-вывода. Обычно стандартная библиотека ввода-вывода использует полную буферизацию для файлов, расположенных на диске. Буфер, как правило, создается одной из стандартных функций ввода-вывода с помощью вызова malloc (раздел 7.8) во время первой операции ввода-вывода.

   Операция записи содержимого стандартного буфера ввода-вывода описывается термином *flush* (сбрасывать). Буфер может сбрасываться на диск автоматически одной из функций, например при его заполнении, или с помощью функции fflush. К сожалению, в UNIX термин *flush* имеет два различных смысла. В терминах стандартной библиотеки ввода-вывода он

означает запись содержимого буфера на диск. В терминах драйвера терминала, например для функции `tcflush` (глава 18), он означает удаление данных из буфера.

2. Построчная буферизация. В этом случае фактический ввод-вывод осуществляется тогда, когда в потоке будет встречен символ перевода строки. Это позволяет нам выводить по одному символу за раз (с помощью стандартной функции `fputc`), зная при этом, что фактическая запись произойдет только в момент, когда будет закончена строка. Построчная буферизация обычно используется для потоков, которые связаны с терминальными устройствами, например для стандартного ввода и стандартного вывода.

   Необходимо дать следующие предостережения по поводу построчной буферизации. Во-первых, буфер, используемый стандартной библиотекой ввода-вывода для сборки строки, имеет фиксированный размер, поэтому фактическая операция ввода-вывода может быть выполнена еще до того, как встретится символ перевода строки, если буфер будет заполнен раньше. Во-вторых, всякий раз, когда ввод производится средствами стандартной библиотеки ввода-вывода либо (а) из небуферизованного потока, либо (б) из потока с построчной буферизацией, который требует обращения к ядру за данными, все выходные потоки с построчной буферизацией сбрасываются. Уточнение для случая (б) необходимо, поскольку требуемые данные могут уже находиться в буфере и за ними не обязательно было бы обращаться к ядру. В случае (а) вполне очевидно, что требуемые данные могут быть получены только от ядра.

3. Отсутствие буферизации. Стандартная библиотека ввода-вывода не буферизует операции с символами. Если мы пишем в поток 15 символов (например, с помощью функции `fputs`), то рассчитываем, что эти 15 символов будут выведены как можно скорее, возможно, с помощью функции `write` (раздел 3.8).

   Так, например, стандартный поток сообщений об ошибках обычно не буферизуется. В результате сообщения выводятся максимально быстро, вне зависимости от того, содержат они символ перевода строки или нет.

Стандарт ISO C предъявляет следующие требования к буферизации.

- Стандартные потоки ввода и вывода буферизуются полностью, но только в том случае, если они не связаны с устройствами интерактивного взаимодействия.
- Стандартный поток сообщений об ошибках никогда не подвергается полной буферизации.

Однако эти требования ничего не говорят о том, могут ли стандартные потоки ввода и вывода быть небуферизованными или построчно буферизированными, если они связаны с устройствами интерактивного взаимодействия, и должен ли стандартный поток сообщений об ошибках быть небуферизованным или построчно буферизированным. В большинстве реализаций по умолчанию используются следующие виды буферизации.

- Стандартный поток сообщений об ошибках никогда не буферизуется.

- Все остальные потоки подвергаются построчной буферизации, если они связаны с терминальным устройством, и полной буферизации – в любом другом случае.

  Все четыре платформы, обсуждаемые в этой книге, следуют этим соглашениям: стандартный поток сообщений об ошибках не буферизуется, потоки, связанные с терминальными устройствами, подвергаются построчной буферизации, а все остальные потоки буферизуются полностью.

Более детально мы исследуем буферизацию стандартного ввода-вывода в разделе 5.12 на примере программы из листинга 5.3.

Если нас не устраивают принятые по умолчанию виды буферизации для какого-либо потока, то их можно изменить с помощью следующих функций.

```
#include <stdio.h>

void setbuf(FILE *restrict fp, char *restrict buf );

int setvbuf(FILE *restrict fp, char *restrict buf, int mode, size_t size);
```

Возвращают 0 в случае успеха, ненулевое значение в случае ошибки

Эти функции должны вызываться только *после* того, как поток будет открыт (это вполне очевидно, так как каждая из них требует передачи корректного указателя на файл в первом аргументе), но *перед* любой другой операцией, выполняемой над потоком.

С помощью функции `setbuf` можно разрешить или запретить буферизацию. Чтобы разрешить буферизацию, аргумент *buf* должен содержать указатель на буфер размером `BUFSIZ` (константа, значение которой определено в файле `<stdio.h>`). В этом случае поток обычно буферизуется полностью, но некоторые системы могут назначить потоку построчную буферизацию, если он связан с терминальным устройством. Чтобы запретить буферизацию, нужно в аргументе *buf* передать значение `NULL`.

При использовании функции `setvbuf` явно указывается желаемый режим буферизации. Делается это с помощью аргумента *mode*:

`_IOFBF`	полная буферизация
`_IOLBF`	построчная буферизация
`_IONBF`	буферизация отсутствует

Если мы запрещаем буферизацию, то значения аргументов *buf* и *size* игнорируются. Если мы задаем полную или построчную буферизацию, через аргументы *buf* и *size* можно передать указатель на буфер и его размер. Если же в аргументе *buf* передается значение `NULL`, то библиотека автоматически выделит для потока собственный буфер соответствующего размера. (Под «соответствующим размером» здесь подразумевается значение константы `BUFSIZ`).

Некоторые реализации библиотеки языка C используют значение из поля `st_blksize` структуры `stat` (раздел 4.2), чтобы определить оптимальный размер буфера ввода-вывода. Далее в этой главе мы увидим, что библиотека GNU C использует этот метод.

В табл. 5.1 перечислены действия, выполняемые этими двумя функциями в зависимости от значений их аргументов.

Таблица 5.1. Функции setbuf и setvbuf

Функция	mode	buf	Буфер и его размер	Тип буферизации
setbuf		Непустой указатель	Пользовательский буфер размером BUFSIZ	Полная или построчная буферизация
		NULL	Нет буфера	Буферизация отсутствует
setvbuf	_IOFBF	Непустой указатель	Пользовательский буфер размером *size*	Полная буферизация
		NULL	Системный буфер соответствующего размера	
	_IOLBF	Непустой указатель	Пользовательский буфер размером *size*	Построчная буферизация
		NULL	Системный буфер соответствующего размера	
	_IONBF	(Игнорируется)	Нет буфера	Буферизация отсутствует

Следует знать, что если стандартный буфер ввода-вывода размещен как автоматическая переменная внутри функции, то перед возвращением из функции необходимо закрыть поток. (Подробнее мы обсудим этот вопрос в разделе 7.8.) Кроме того, некоторые реализации используют часть буфера для своих внутренних целей, таким образом, фактический объем данных, которые могут быть сохранены в буфере, будет меньше его указанного размера. Вообще, лучше позволить системе самой выбирать размер буфера и автоматически размещать его в памяти. В этом случае стандартная библиотека ввода-вывода сама освободит память, занимаемую буфером в момент закрытия потока.

В любой момент содержимое буфера потока может быть сброшено.

```
#include <stdio.h>

int fflush(FILE *fp);
```

Возвращает 0 в случае успеха, EOF в случае ошибки

Эта функция передает ядру все незаписанные данные из буфера. В особом случае, когда в аргументе *fp* передается значение NULL, сбрасывается содержимое буферов всех потоков.

5.5. Открытие потока

Следующие три функции открывают потоки ввода-вывода.

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict type);

FILE *freopen(const char *restrict pathname, const char *restrict type,
              FILE *restrict fp);

FILE *fdopen(int filedes, const char *type);
```

Все три возвращают указатель на файл в случае успеха, NULL в случае ошибки

Перечислим различия между этими функциями.

1. Функция fopen открывает заданный файл.
2. Функция freopen открывает заданный файл и связывает его с заданным потоком, предварительно закрывая поток, если он уже был открыт. Если перед этим поток имел ориентацию, то функция сбрасывает ее. Как правило, эта функция используется для связывания открываемого файла с предопределенным стандартным потоком ввода, вывода или сообщений об ошибках.
3. Функция fdopen принимает открытый дескриптор файла, полученный от функций open, dup, dup2, fcntl, pipe, socket, socketpair или accept, и связывает его с потоком ввода-вывода. Часто эта функция вызывается с дескриптором, который был получен в результате создания канала или сетевого соединения. Поскольку эти типы файлов нельзя открыть стандартной функцией fopen, приходится сначала открывать их специальными функциями, чтобы получить дескриптор файла, а затем связывать дескриптор с потоком ввода-вывода, используя функцию fdopen.

Обе функции, fopen и freopen, являются частью стандарта ISO C. Функция fdopen определена стандартом POSIX.1, поскольку ISO C не имеет дела с дескрипторами.

Стандарт ISO C определяет 15 возможных значений аргумента *type*. Все они приводятся в табл. 5.2.

Таблица 5.2. Возможные значения аргумента type при открытии потока

type	Описание
r или rb	Открыть для чтения
w или wb	Усечь размер файла до 0 или создать и открыть на запись
a или ab	Открыть для записи в конец файла или создать для записи
r+, или r+b, или rb+	Открыть для чтения и для записи
w+, или w+b, или wb+	Усечь размер файла до 0 или создать и открыть для чтения и для записи
a+, или a+b, или ab+	Открыть или создать для чтения и для записи в конец файла

Использование символа b в аргументе *type* позволяет стандартной системе ввода-вывода различать текстовые и двоичные файлы. Так как ядро UNIX не различает эти типы файлов, указание символа b не оказывает никакого влияния.

Для функции fdopen значение аргумента *type* несколько отличается. Так как дескриптор уже был открыт, то открытие для записи не приводит к усечению файла. (Если дескриптор для существующего файла создается, например, функцией open, то усечение может быть выполнено с помощью флага O_TRUNC. Функция fdopen сама не может выполнить усечение файла, который она открывает для записи.) Кроме того, открытие для записи в конец файла не приводит к созданию нового файла (так как дескриптор может быть связан только с существующим файлом).

Когда файл открыт в режиме добавления в конец, каждая операция записи будет производиться в конец файла. Если несколько процессов откроют один и тот же файл в этом режиме, стандартная функция ввода-вывода будет корректно записывать данные каждого процесса.

> Версии функции fopen из Беркли, предшествовавшие 4.4BSD, и простейшая версия, которая приводится на странице 177 [Kernighan и Ritchie 1988], работают с режимом добавления в конец файла не совсем корректно. Эти версии вызывают функцию lseek для перехода в конец файла при его открытии. В случае, когда с файлом работают несколько процессов, он должен быть открыт с флагом O_APPEND, который мы рассматривали в разделе 3.3. Вызов функции lseek перед каждой операцией записи не даст желаемого эффекта;эту проблему мы также обсуждали в разделе 3.11.

Когда файл открывается для чтения и записи (символ «+» в аргументе *type*), применяются следующие ограничения:

- Вывод не может сразу же следовать за вводом без промежуточного вызова функций fflush, fseek, fsetpos или rewind.
- Ввод не может сразу следовать за выводом без вызова функций fseek, fsetpos или rewind или операции ввода, которая встречает конец файла.

Различные характеристики шести способов открытия потока, перечисленных в табл. 5.2, можно обобщить следующим образом.

Таблица 5.3. Шесть способов открытия потоков ввода-вывода

Ограничение	**r**	**w**	**a**	**r+**	**w+**	**a+**
Файл должен существовать	•			•		
Предыдущее содержимое файла будет утеряно		•			•	
Поток доступен для чтения	•			•	•	•
Поток доступен для записи		•	•	•	•	•
Запись возможна только в конец потока			•			•

Обратите внимание: используя в аргументе type символы a или w, можно создать новый файл, но при этом нельзя определить биты прав доступа к файлу, как мы делали это с помощью функции open или creat в главе 3.

По умолчанию поток открывается в режиме полной буферизации, если, конечно, он не связан с терминальным устройством, так как в этом случае используется режим построчной буферизации. Сразу же после открытия потока, но до выполнения первой операции ввода-вывода можно изменить режим буферизации с помощью функций `setbuf` и `setvbuf`, которые были описаны в предыдущем разделе.

Закрывается открытый поток с помощью функции `fclose`.

```
#include <stdio.h>

int fclose(FILE *fp);
```

Возвращает 0 в случае успеха, `EOF` в случае ошибки

Перед закрытием потока все данные, находящиеся в буфере вывода, сбрасываются. Все данные, которые находятся в буфере ввода, будут утеряны. Если память под буфер была выделена самой библиотекой ввода-вывода, она освобождается автоматически.

При нормальном завершении процесса, то есть при непосредственном вызове функции `exit` или при возврате из функции `main`, все незаписанные данные в буферах вывода сбрасываются на диск, после чего все потоки закрываются.

5.6. Чтение из потока и запись в поток

После открытия потока можно выбрать один из трех типов неформатированного ввода-вывода:

1. Посимвольный ввод-вывод. Можно читать или писать по одному символу за раз с помощью стандартных функций ввода-вывода, которые буферизуют данные, если поток буферизован.
2. Построчный ввод-вывод. Если необходимо читать или писать данные построчно, используются функции `fgets` и `fputs`. Каждая строка заканчивается символом перевода строки, а при использовании функции `fgets` нужно указать максимальную длину строки, которую мы можем принять. Эти две функции мы рассмотрим в разделе 5.7.
3. Прямой ввод-вывод. Этот тип ввода-вывода поддерживается функциями `fread` и `fwrite`. Каждая операция выполняет чтение или запись определенного количества объектов, имеющих заданный размер. Эти функции часто используются для работы с двоичными файлами, когда в каждой операции чтения или записи участвует одна структура данных. Эти функции мы рассмотрим в разделе 5.9.

   Термин *прямой ввод-вывод* из стандарта ISO C имеет также несколько синонимов: *двоичный ввод-вывод, ввод-вывод объектами, ввод-вывод записями* или *ввод-вывод структурами*.

Функции ввода

Три функции позволяют читать по одному символу за одно обращение.

```
#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(void);
```

Все три возвращают очередной символ в случае успеха, EOF в случае ошибки

Функция getchar определена как эквивалент getc(stdin). Разница между первыми двумя функциями заключается в том, что функция getc может быть реализована в виде макроса, тогда как fgetc – нет. Это означает следующее.

1. Аргумент функции getc не должен быть выражением с побочными эффектами.
2. Поскольку fgetc обязательно будет функцией, мы всегда можем узнать ее адрес. Это позволит передать адрес функции fgetc в виде аргумента другой функции.
3. Вызов функции fgetc скорее всего будет более длительным, чем вызов getc, так как обычно вызов функции занимает больше времени, чем обращение к макросу.

Все три функции возвращают очередной символ как unsigned char, преобразованный в int. Причина кроется в том, что функции должны возвращать положительное значение даже в том случае, когда старший бит символа установлен в 1. Преобразование в int связано с тем, что наряду с обычными символами функции могут возвращать признак ошибки или признак конца файла. Константа EOF, определяемая в файле <stdio.h>, должна иметь отрицательное значение. Чаще всего она имеет значение –1. Таким образом, мы не сможем сохранить возвращаемое значение любой из этих трех функций в переменной символьного типа и затем сравнить его с константой EOF.

Обратите внимание: эти функции возвращают одно и то же значение и в случае ошибки, и в случае достижения конца файла. Чтобы отличить один случай от другого, используются функции ferror или feof.

```
#include <stdio.h>

int ferror(FILE *fp);

int feof(FILE *fp);
```

Обе возвращают ненулевое значение (истина), если условие истинно, или 0 (ложь), если условие ложно

```
void clearerr(FILE *fp);
```

В большинстве реализаций в объекте FILE для каждого потока предусматриваются два флага:

- Флаг ошибки
- Флаг конца файла

Оба флага сбрасываются вызовом функции `clearerr`.

После чтения символа из потока можно вернуть символ обратно в поток вызовом функции `ungetc`.

```
#include <stdio.h>

int ungetc(int c, FILE *fp);
```

Возвращает значение аргумента *c* в случае успеха, `EOF` в случае ошибки

Символы, возвращенные в поток, будут заново прочитаны последующими операциями чтения в порядке, обратном порядку их возврата. Надо отметить, что хотя стандарт ISO C и позволяет возвращать в поток произвольное количество символов, реализации обязаны предоставлять возможность возврата только одного символа. Поэтому не следует рассчитывать более чем на один символ.

Возвращаемый символ не обязательно должен быть последним прочитанным символом. Невозможно вернуть в поток признак конца файла (`EOF`). Однако мы можем по достижении конца файла вернуть в поток один символ. Следующая операция чтения вернет нам этот символ, а следующая за ней – вернет `EOF`. Этот прием работает, потому что функция `ungetc` сбрасывает флаг конца файла у потока.

Возврат символов в поток чаще всего используется, когда необходимо прервать чтение на границе слова или лексемы определенного вида. Иногда нужно увидеть следующий символ, чтобы решить, как обрабатывать текущий. В этом случае мы просто возвращаем прочитанный символ в поток, и он будет получен при следующем вызове функции `getc`. Если бы стандартная библиотека ввода-вывода не предоставляла такой возможности, нам приходилось бы сохранять его в переменной, равно как и флаг, указывающий на то, что следующий символ следует взять из переменной, а не из потока.

> Когда мы возвращаем символы в поток с помощью функции `ungetc`, они на самом деле не записываются обратно в файл или в устройство. Возвращаемые символы просто сохраняются библиотекой ввода-вывода во внутреннем буфере потока.

Функции вывода

Каждой из описанных выше функций ввода соответствует функция вывода.

Как и в случае с функциями ввода, вызов `putchar(c)` эквивалентен вызову `putc(c, stdout)`, и `putc` также может быть реализована в виде макроса, тогда как `fputc` – нет.

```
#include <stdio.h>

int putc(int c, FILE *fp);

int fputc(int c, FILE *fp);

int putchar(int c);
```

Все три возвращают значение аргумента *c*
в случае успеха, EOF в случае ошибки

5.7. Построчный ввод-вывод

Построчный ввод выполняется следующими функциями.

```
#include <stdio.h>

char *fgets(char *restrict buf, int n, FILE *restrict fp);

char *gets(char *buf);
```

Обе возвращают *buf* в случае успеха,
NULL в случае ошибки или по достижении конца файла

Обеим функциям передается адрес буфера для размещения прочитанной строки. Функция gets читает из стандартного потока ввода, функция fgets – из указанного потока.

Для функции fgets указывается размер приемного буфера, *n*. Эта функция будет считывать входные данные в буфер до тех пор, пока не встретит символ перевода строки, но не более *n-1* символов. В конец прочитанной строки добавляется нулевой символ. Если длина строки, включая символ перевода строки, составляет более *n-1* символов, функция вернет только часть строки, но в конец буфера все равно будет добавлен завершающий нулевой символ. Последующие вызовы fgets вернут остаток строки.

Никогда не следует использовать функцию gets. Проблема с ней состоит в том, что она не позволяет определить размер приемного буфера. Если входная строка окажется длиннее буфера, это приведет к его переполнению и в результате будут перезаписаны данные, которые находятся в памяти сразу же после буфера. Описание того, как эта брешь в безопасности использовалась программой-червем в 1988 году, вы найдете в июньском номере «Communications of the ACM» за 1989 год (vol. 32 no. 6). Еще одно отличие gets от fgets заключается в том, что функция gets не сохраняет символ перевода строки в буфере, как это делает функция fgets.

> Это различие в обработке символа перевода строки корнями уходит в историю UNIX. Еще руководство к Version 7 (1979) гласит: «gets удаляет символ перевода строки, fgets оставляет его, и всё это ради сохранения обратной совместимости».

Хотя стандарт ISO C требует, чтобы реализация предоставляла функцию gets, используйте вместо нее функцию fgets.

Операции построчного вывода обеспечиваются функциями puts и fputs.

```
#include <stdio.h>

int fputs(const char *restrict str, FILE *restrict fp);

int puts(const char *str);
```

Обе возвращают неотрицательное значение в случае успеха, EOF в случае ошибки

Функция fputs записывает строку, завершающуюся нулевым символом, в указанный поток. Нулевой символ в поток не записывается. Примечательно, что это не построчный вывод в строгом смысле слова, поскольку строка может не содержать символ перевода строки в качестве последнего ненулевого символа. Обычно завершающему нулевому символу действительно предшествует символ перевода строки, но это совершенно не обязательно.

Функция puts записывает строку, завершающуюся нулевым символом, в поток стандартного вывода. Она не выводит завершающий нулевой символ, но добавляет символ перевода строки.

Функция puts достаточно безопасна, в отличие от ее пары – gets. Однако мы также рекомендуем не пользоваться ею, чтобы не задумываться постоянно о том, добавляет ли она символ перевода строки. Пользуясь только функциями fgets и fputs, мы всегда будем точно знать, что должны обрабатывать символ перевода строки.

5.8. Эффективность стандартных функций ввода-вывода

Используя функции из предыдущего раздела, мы можем оценить эффективность стандартной библиотеки ввода-вывода. Программа, представленная листингом 5.1, очень похожа на ту, что мы приводили в листинге 3.1 – она просто копирует данные со стандартного ввода на стандартный вывод с помощью функций getc и putc. Эти две функции могут быть реализованы в виде макросов.

Листинг 5.1. Копирование данных со стандартного ввода на стандартный вывод с помощью функций getc и putc

```
#include "apue.h"

int
main(void)
{
    int c;

    while ((c = getc(stdin)) != EOF)
```

```
        if (putc(c, stdout) == EOF)
            err_sys("ошибка вывода");

    if (ferror(stdin))
        err_sys("ошибка ввода");

    exit(0);
}
```

Можно также написать версию этой программы с использованием функций `fgetc` и `fputc`, которые всегда реализованы как функции, а не как макросы. (Не будем здесь приводить изменения в исходном коде, поскольку они достаточно тривиальны.)

В заключение приведем еще одну версию программы, которая выполняет чтение и запись построчно (листинг 5.2).

Листинг 5.2. Копирование данных со стандартного ввода на стандартный вывод с помощью функций fgets и fputs

```
#include "apue.h"

int
main(void)
{
    char buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("ошибка вывода");

    if (ferror(stdin))
        err_sys("ошибка ввода");

    exit(0);
}
```

Обратите внимание: программы из листингов 5.1 и 5.2 не закрывают потоки ввода-вывода явно. Мы уже знаем, что функция `exit` сбросит все незаписанные данные из буферов и закроет все открытые потоки. (Этот вопрос еще будет обсуждаться в разделе 8.5.) Интересно сравнить время, затраченное этими тремя программами, с результатами из табл. 3.2. Данные для сравнения приводятся в табл. 5.4 (все операции производились с одним и тем же файлом размером 98,5 Мб, содержащим 3 миллиона строк).

Для каждой из трех версий, использующих стандартные функции ввода-вывода, пользовательское время получилось больше, чем наилучший результат из табл. 3.2, потому что две версии выполняют 100 миллионов циклов для передачи данных по одному байту, а версия с построчным вводом-выводом выполняет 3 144 984 циклов. Версия программы, основанная на функции `read`, выполняет всего 12 611 циклов (для размера буфера 8 192 байта). Различия общего времени выполнения обусловлены различиями пользовательского времени, так как значения системного времени вполне сопоставимы.

Таблица 5.4. Время выполнения операций стандартными функциями ввода-вывода

Функция	Пользовательское время (секунды)	Системное время (секунды)	Общее время (секунды)	Размер программы (байты)
Лучшее время из табл. 3.2	0,01	0,18	6,67	
`fgets`, `fputs`	2,59	0,19	7,15	139
`getc`, `putc`	10,84	0,27	12,07	120
`fgetc`, `fputc`	10,44	0,27	11,42	120
Время из табл. 3.2, с размером буфера 1 байт	124,89	161,65	288,64	

Системное время практически одно и то же, потому что производится примерно одинаковое количество системных вызовов. Обратите внимание: преимущество стандартной библиотеки ввода-вывода состоит в том, что она избавляет нас от беспокойства по поводу буферизации или оптимальности выбранного размера буфера. Конечно, мы все-таки должны определиться с максимальным размером строки для версии программы, которая использует функцию `fgets`, но это гораздо проще, чем выбирать оптимальный размер буфера.

В последней колонке табл. 5.4 приводится размер сегмента кода программы, сгенерированного компилятором языка C. Здесь мы видим, что версия с функциями `fgetc`/`fputc` имеет тот же размер, что и версия с функциями `getc`/`putc`. Обычно функции `getc` и `putc` реализованы в виде макросов, но в библиотеке GNU C эти макросы просто разворачиваются в вызовы функций.

Версия программы с построчным вводом-выводом выполняется почти в два раза быстрее, чем версии с посимвольным вводом-выводом. Если бы `fgets` и `fputs` были реализованы через функции `getc` и `putc` (раздел 7.7 [Kernighan and Ritchie 1988]), то результаты совпадали бы с результатами версии, основанной на функции `getc`. На самом деле версия с построчным вводом-выводом выполнялась бы даже значительно дольше, так как в этом случае к существующим 6 миллионам вызовов функций добавились бы еще 200 миллионов. Таким образом, из полученных результатов можно сделать вывод, что функции построчного ввода-вывода реализованы с помощью функции `memccpy`(3). Часто для повышения эффективности функция `memccpy` пишется не на C, а на языке ассемблера.

И последнее, что представляет для нас интерес в этих результатах, – версия на основе функции `fgetc` выполняется намного быстрее, чем версия из листинга 3.1 с размером буфера `BUFFSIZE=1`. Обе версии производят одно и то же количество вызовов функций – приблизительно 200 миллионов, и все же выполнение версии на основе функции `fgetc` потребовало почти в 12 раз меньше пользовательского времени, а общее время выполнения получилось более чем

в 25 раз меньше. Эта разница обусловлена тем, что версия на основе функции open выполняет 200 миллионов вызовов функций, которые в свою очередь производят 200 миллионов системных вызовов. Версия на основе функции fgetc также выполняет 200 миллионов вызовов функций, но обращения к системным вызовам производятся всего 25222 раза. Обращение к системному вызову обычно намного дороже, чем обращение к обычной функции.

Тут мы должны оговориться: разумеется, результаты испытаний справедливы только для той системы, на которой они были получены. Результаты зависят от многих особенностей, которые различаются в разных реализациях UNIX. Тем не менее приводимые здесь числа и объяснения различий между версиями одной программы помогут нам лучше понять саму операционную систему. Сравнивая результаты из этого раздела и из раздела 3.9, мы узнали, что стандартная библиотека ввода-вывода не намного медленнее, чем прямое обращение к функциям read и write. Из полученных результатов видно, что приблизительная стоимость копирования одного мегабайта с использованием функций getc и putc составляет 0,11 секунды процессорного времени. В большинстве сложных приложений наибольшее количество пользовательского времени уходит на выполнение самого приложения, а не на обращения к стандартным функциям ввода-вывода.

5.9. Ввод–вывод двоичных данных

Функции из раздела 5.6 оперируют одним символом, функции из раздела 5.7 – одной строкой. При выполнении операций ввода-вывода двоичных данных предпочтительнее читать или записывать сразу целые структуры. Чтобы сделать это с помощью функций getc или putc, нам пришлось бы обходить структуру в цикле, выполняя чтение или запись по одному байту. Мы не можем воспользоваться функциями построчного ввода-вывода, поскольку функция fputs прекращает запись, встретив нулевой байт, а внутри структуры вполне могут содержаться нулевые байты. Точно так же и функция fgets не сможет корректно читать данные, которые содержат нулевые байты или символы перевода строки. Поэтому для ввода-вывода двоичных данных предоставляются следующие две функции.

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp);

size_t fwrite(const void *restrict ptr, size_t size, size_t nobj,
              FILE *restrict fp);
```

Обе возвращают количество прочитанных или записанных объектов

Два наиболее распространенных случая использования этих функций:

1. Чтение или запись массивов двоичных данных. Например, записать со 2-го по 5-й элементы массива чисел с плавающей точкой можно следующим образом:

```
float data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("ошибка вызова функции fwrite");
```

Здесь мы передали в аргументе *size* размер одного элемента массива, а в аргументе *nobj* – количество элементов.

2. Чтение или запись структур данных. Это делается следующим образом:

```
struct {
    short count;
    long total;
    char name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("ошибка вызова функции fwrite");
```

Здесь в аргументе *size* указан размер структуры, а в аргументе *nobj* – количество объектов для записи (один).

Очевидное обобщение этих двух случаев – чтение или запись массива структур. Для этого аргумент *size* должен содержать размер структуры (определяемый с помощью оператора `sizeof`), а аргумент *nobj* – количество элементов массива.

Обе функции возвращают количество прочитанных или записанных объектов. Для функции `fread` это число может быть меньше значения *nobj* в том случае, если произошла ошибка или был достигнут конец файла. В этой ситуации нужно вызывать функции `ferror` или `feof`. Если функция `fwrite` вернула число, которое меньше значения аргумента *nobj*, это свидетельствует об ошибке.

Фундаментальная проблема, связанная с вводом-выводом двоичных данных, заключается в том, что они могут быть корректно прочитаны только в той же системе, в которой они были записаны. Много лет назад, когда все версии UNIX работали на PDP-11, этой проблемы не существовало, но сегодня стало нормой объединение разнородных систем в сети. И нередко возникает желание записать данные на одной системе и обработать их на другой. В такой ситуации эти две функции не будут работать по двум причинам:

1. Смещение полей структур может отличаться для разных компиляторов и операционных систем из-за различных требований выравнивания. Некоторые компиляторы имеют возможность упаковывать структуры в целях экономии занимаемого пространства и возможно в ущерб производительности либо, наоборот, выполнять выравнивание полей для повышения скорости доступа во время выполнения. Это означает, что даже для одной и той же системы раскладка структуры может варьироваться в зависимости от параметров компиляции.
2. Форматы представления многобайтных целых чисел или чисел с плавающей точкой могут различаться на разных аппаратных платформах.

Мы коснемся некоторых из этих проблем, когда будем говорить о сокетах в главе 16. Решение проблемы обмена двоичными данными между различными системами заключается в использовании высокоуровневого протокола. За описанием различных приемов, используемых сетевыми протоколами для обмена двоичными данными, обращайтесь к разделу 8.2 [Rago 1993] или к разделу 5.18 [Stevens, Fenner, & Rudoff 2004].

Мы еще вернемся к функции `fread` в разделе 8.14, когда с ее помощью будем читать двоичные структуры данных учетной информации о процессах.

5.10. Позиционирование в потоке

Существует три способа позиционирования в потоке ввода-вывода:

1. С помощью функций `ftell` и `fseek`, которые впервые появились в Version 7. Они предполагают, что позиция в файле может быть представлена в виде длинного целого числа.
2. С помощью функций `ftello` и `fseeko`. Они определены стандартом Single UNIX Specification для случаев, когда длинного целого числа недостаточно для представления позиции в файле.
3. С помощью функций `fgetpos` и `fsetpos`. Они были определены стандартом ISO C. Для представления позиции в файле они используют абстрактный тип данных `fops_t`. Этот тип данных может быть увеличен настолько, насколько это будет необходимо для представления позиции в файле.

Переносимые приложения, которые предполагается портировать на операционные системы, отличные от UNIX, должны использовать функции `fgetpos` и `fsetpos`.

```
#include <stdio.h>

long ftell(FILE *fp);

                                    Возвращает текущую позицию файла
                                    в случае успеха, –1L в случае ошибки

int fseek(FILE *fp, long offset, int whence);

                                    Возвращает 0 в случае успеха,
                                    ненулевое значение в случае ошибки

void rewind(FILE *fp);
```

Для двоичных файлов текущая позиция измеряется в байтах относительно начала файла. Значение, возвращаемое функцией `ftell` для двоичных файлов, – это позиция данного байта. Чтобы установить позицию в двоичном файле с помощью функции `fseek`, нужно указать в аргументе *`offset`* смещение байта и то, как это смещение интерпретируется. Значение аргумента *`whence`* выбирается точно так же, как для функции `lseek` (раздел 3.6): `SEEK_SET` означает смещение от начала файла, `SEEK_CUR` – смещение от текущей позиции файла и `SEEK_END` – смещение от конца файла. Стандарт ISO C не требует, чтобы реа-

лизация поддерживала константу SEEK_END для двоичных файлов, поскольку некоторые системы требуют дополнения двоичных файлов в конце нулями, чтобы сделать размер файла кратным некоторому числу. Однако UNIX поддерживает использование константы SEEK_END для двоичных файлов.

В случае текстовых файлов текущая позиция может не соответствовать простому смещению байта. Опять же главным образом это относится к системам, отличным от UNIX, которые могут хранить текстовые данные в различных форматах. Чтобы установить текущую позицию в текстовом файле, аргумент *whence* должен иметь значение SEEK_SET, а для аргумента *offset* допускаются только два значения – 0, что означает возврат к началу файла, или значение, которое было получено с помощью функции ftell для этого файла. Кроме того, вернуться в начало файла можно с помощью функции rewind.

Функция ftello практически идентична функции ftell, а функция fseeko – функции fseek, за исключением того, что тип смещения у них не long, а off_t.

```
#include <stdio.h>

off_t ftello(FILE *fp);
                                    Возвращает текущую позицию файла
                                  в случае успеха, (off_t)-1 в случае ошибки

int fseeko(FILE *fp, off_t offset, int whence);
                                    Возвращает 0 в случае успеха,
                                  ненулевое значение в случае ошибки
```

О типе off_t мы говорили в разделе 3.6. Реализации могут определять тип off_t большего размера, чем 32-битное целое.

Как уже говорилось ранее, функции fgetpos и fsetpos определены стандартом ISO C.

```
#include <stdio.h>

int fgetpos(FILE *restrict fp, fpos_t *restrict pos);

int fsetpos(FILE *fp, const fpos_t *pos);
                                    Обе возвращают 0 в случае успеха,
                                  ненулевое значение в случае ошибки
```

Функция fgetpos записывает значение текущей позиции в объект, на который указывает аргумент *pos*. Это значение может использоваться в последующих вызовах fsetpos для переустановки текущей позиции файла.

5.11. Форматированный ввод-вывод

Форматированный вывод

Форматированный вывод производится с помощью четырех разновидностей функции printf.

```
#include <stdio.h>

int printf(const char *restrict format, ...);

int fprintf(FILE *restrict fp, const char *restrict format, ...);
```

Обе возвращают количество выведенных символов в случае успеха, отрицательное значение в случае ошибки

```
int sprintf(char *restrict buf, const char *restrict format, ...);

int snprintf(char *restrict buf, size_t n, const char *restrict format, ...);
```

Обе возвращают количество символов, записанных в массив, в случае успеха, отрицательное значение в случае ошибки

Функция `printf` осуществляет запись в стандартный поток вывода, `fprintf` — в заданный поток, а `sprintf` помещает форматированную строку в массив *buf*. Функция `sprintf` автоматически дополняет строку нулевым байтом, но он не учитывается в возвращаемом значении.

Обратите внимание: при использовании функции `sprintf` вполне возможно переполнение буфера, на который указывает аргумент `buf`. Таким образом, вызывающая программа должна гарантировать предоставление буфера достаточного размера. Во избежание проблем, связанных с переполнением буфера, была добавлена функция `snprintf`. Эта функция принимает размер буфера в виде дополнительного аргумента и игнорирует символы, которые могли бы быть записаны за пределы буфера. Функция `snprintf` возвращает количество символов, которое было бы записано в буфер, если бы он имел достаточно большой размер. Как и в случае с функцией `sprintf`, возвращаемое значение не учитывает завершающий нулевой байт. Если `snprintf` возвращает положительное значение, меньшее чем размер буфера *n*, это означает, что строка была полностью записана в буфер и не была усечена. Если возникает ошибка, функция `snprintf` возвращает отрицательное значение.

Спецификация формата управляет порядком интерпретации и в конечном счете отображением остальных аргументов. Каждый аргумент интерпретируется согласно спецификатору формата, который начинается с символа процента (`%`). Все символы строки формата, за исключением спецификаторов, копируются без изменений. Спецификатор формата включает четыре необязательных компонента, которые ниже показаны в квадратных скобках:

```
%[flags][fldwidth][precision][lenmodifier]convtype
```

Перечень возможных значений компонента `flags` приводится в табл. 5.5.

Компонент `fldwidth` определяет минимальную ширину поля для преобразования. Если в результате преобразования было получено меньшее количество символов, они будут дополнены пробелами. Ширина поля выражается положительным целым числом или звездочкой (`*`).

Компонент `precision` определяет минимальное количество цифр для отображения целых чисел, минимальное количество цифр, расположенных правее

десятичной точки, для чисел с плавающей точкой или максимальное количество символов для отображения строк. Компонент `precision` представляется в виде точки (`.`), за которой следует неотрицательное целое число или символ звездочки.

Таблица 5.5. Значения компонента flags строки спецификации формата

Флаг	Описание
-	Выравнивание по левому краю поля
+	Всегда отображать знак числа
пробел	Выводить пробел, если отсутствует знак числа
#	Преобразовать в альтернативную форму (например, включить префикс 0x при выводе чисел в шестнадцатеричном формате)
0	Заполнять нулями вместо пробелов при выравнивании по правому краю

В полях `fldwidth` и `precision` можно указать звездочку. В этом случае значение данного компонента определяется целочисленным аргументом функции. Этот аргумент должен стоять непосредственно перед аргументом, значение которого будет подвергнуто преобразованию.

Компонент `lenmodifier` определяет размер аргумента. Возможные значения приведены в табл. 5.6.

Таблица 5.6. Значения компонента lenmodifier строки спецификации формата

Модификатор длины	Описание
`hh`	`signed char` или `unsigned char`
`h`	`signed short` или `unsigned short`
`l`	`signed long`, `unsigned long` или многобайтный символ
`ll`	`signed long long` или `unsigned long long`
`j`	`intmax_t` или `uintmax_t`
`z`	`size_t`
`t`	`ptrdiff_t`
`L`	`long double`

Компонент `convtype` является обязательным. Он управляет интерпретацией аргумента. Различные виды преобразований приведены в табл. 5.7.

Таблица 5.7. Значения компонента convtype строки спецификации формата

Спецификатор	Описание
`d, i`	Десятичное число со знаком
`o`	Восьмеричное число без знака
`u`	Десятичное число без знака

Таблица 5.7 (продолжение)

Спецификатор	Описание
x, X	Шестнадцатеричное число без знака
f, F	Число с плавающей точкой двойной точности
e, E	Число с плавающей точкой двойной точности в экспоненциальной форме
g, G	Интерпретируется как f, F, e или E, в зависимости от значения интерпретируемого аргумента
a, A	Число с плавающей точкой двойной точности в шестнадцатеричной экспоненциальной форме
c	Символ (с модификатором длины l – многобайтный символ)
s	Строка (с модификатором длины l – строка многобайтных символов)
p	Указатель типа void
n	Указатель на целое со знаком, в которое записывается количество уже выведенных символов
%	Символ %
C	Многобайтный символ (расширение XSI, эквивалент lc)
S	Строка многобайтных символов (расширение XSI, эквивалент ls)

Следующие четыре разновидности функции printf очень похожи на предыдущие, но в них список аргументов переменной длины (...) заменен аргументом *arg*.

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *restrict format, va_list arg);

int vfprintf(FILE *restrict fp, const char *restrict format, va_list arg);

                    Обе возвращают количество выведенных символов
                  в случае успеха, отрицательное значение – в случае ошибки

int vsprintf(char *restrict buf, const char *restrict format, va_list arg);

int vsnprintf(char *restrict buf, size_t n,
              const char *restrict format, va_list arg);

                    Обе в случае успеха возвращают количество символов,
                записанных в массив, и отрицательное значение – в случае ошибки
```

Мы использовали функцию vsnprintf в функциях вывода сообщений об ошибках (приложение B).

Описание принципов обработки списков аргументов переменной длины в соответствии со стандартом ISO C вы найдете в разделе 7.3 [Kernighan and Ritchie 1988]. Следует помнить, что средства обработки списков аргументов пе-

ременной длины, определяемые стандартом ISO C, – заголовочный файл <stdarg.h> и связанные с ним функции – отличаются от функций <varargs.h>, которые предоставлялись старыми версиями UNIX.

Форматированный ввод

Форматированный ввод выполняется с помощью функций семейства scanf.

```
#include <stdio.h>

int scanf(const char *restrict format, ...);

int fscanf(FILE *restrict fp, const char *restrict format, ...);

int sscanf(const char *restrict buf, const char *restrict format, ...);
```

Все три возвращают количество введенных элементов или EOF, если возникла ошибка ввода или конец файла достигнут перед каким-либо преобразованием

Функции семейства scanf используются для анализа входной строки и преобразования последовательностей символов в переменные указанных типов. Аргументы, следующие за строкой формата, содержат адреса переменных, в которые будут записаны результаты преобразований.

Спецификация формата управляет порядком преобразования. Символ процента (%) обозначает начало спецификатора формата. Все символы в строке формата, за исключением спецификаторов формата и пробелов, должны совпадать с вводимыми символами. Если обнаруживается какое-либо несоответствие, обработка ввода останавливается, и остальная часть входной строки остается непрочитанной.

Спецификатор формата включает три необязательных компонента, которые ниже показаны в квадратных скобках:

```
%[*][fldwidth][lenmodifier]convtype
```

Необязательный первый компонент может содержать символ звездочки (*), который используется для подавления преобразования. Вводимая строка преобразуется согласно указанному формату, но результат преобразования не сохраняется.

Компонент fldwidth определяет максимальную ширину поля в символах. Компонент lenmodifier – размер аргумента, в который будет записан результат преобразования. Семейство функций scanf поддерживает те же самые модификаторы длины (lenmodifier), что и семейство функций printf (табл. 5.6).

Компонент convtype подобен соответствующему компоненту спецификатора формата функции printf, но между ними есть и некоторые отличия. Одно из них состоит в том, что результат преобразования, который сохраняется как беззнаковый тип, может на входе быть числом со знаком. Например, число –1 будет преобразовано в 4294967295 и записано в переменную беззнакового типа. В табл. 5.8 перечисляются типы преобразований, которые поддерживаются функциями семейства scanf.

Таблица 5.8. Значения компонента conv1ype строки спецификации формата

Спецификатор	Описание
d	Десятичное число со знаком, с основанием 10
i	Десятичное число со знаком, основание определяется форматом ввода
o	Восьмеричное число без знака (на входе может быть со знаком)
u	Десятичное число без знака, с основанием 10 (на входе может быть со знаком)
x	Шестнадцатеричное число без знака (на входе может быть со знаком)
a, A, e, E, f, F, g, G	Число с плавающей точкой
c	Символ (с модификатором длины l – многобайтный символ)
s	Строка (с модификатором длины l – строка многобайтных символов)
p	Указатель типа void
n	Указатель на целое со знаком, в которое записывается количество уже выведенных символов
%	Символ %
C	Многобайтный символ (расширение XSI, эквивалент lc)
S	Строка многобайтных символов (расширение XSI, эквивалент ls)
[	Начинает последовательность, состоящую только из указанных символов, ограниченную символом]
[^	Начинает последовательность любых символов, кроме указанных, ограниченную символом]

Аналогично семейству printf семейство scanf также включает функции, которые поддерживают передачу списка аргументов в виде переменной, как это определено в заголовочном файле <stdarg.h>.

```
#include <stdarg.h>
#include <stdio.h>

int vscanf(const char *restrict format, va_list arg);

int vfscanf(FILE *restrict fp, const char *restrict format,
            va_list arg);

int vsscanf(const char *restrict buf, const char *restrict format,
            va_list arg);
```

Все три возвращают количество введенных элементов или EOF, если возникла ошибка ввода или конец файла достигнут перед каким-либо преобразованием

За дополнительной информацией о функциях семейства scanf обращайтесь к справочному руководству вашей системы UNIX.

5.12. Подробности реализации

Как мы уже упоминали, функции стандартной библиотеки ввода-вывода в конечном итоге обращаются к функциям, описанным в главе 3. Каждому потоку ввода-вывода сопоставлен дескриптор файла, получить который можно, обратившись к функции `fileno`.

> Обратите внимание: функция `fileno` не определена стандартом ISO C – это расширение, поддерживаемое стандартом POSIX.1.

```
#include <stdio.h>

int fileno(FILE *fp);
```

Возвращает дескриптор файла, ассоциированный с потоком

Эта функция необходима в том случае, если мы собираемся, например, вызывать функции `dup` или `fcntl`.

Чтобы увидеть, как реализована стандартная библиотека ввода-вывода в вашей системе, начните с заголовочного файла `<stdio.h>`. Здесь вы найдете определение объекта `FILE`, флагов потока и всех стандартных функций ввода-вывода, таких как `getc`, которые определены как макросы. В разделе 8.5 [Kernighan and Ritchie 1988] приводится пример, который демонстрирует особенности большинства реализаций в UNIX. В главе 12 [Plauger 1992] вы найдете полные исходные тексты одной из реализаций стандартной библиотеки ввода-вывода. Кроме того, в свободном доступе имеется реализация стандартной библиотеки ввода-вывода GNU.

Пример

Программа, представленная в листинге 5.3, демонстрирует особенности буферизации для всех трех стандартных потоков ввода-вывода и для потока, ассоциированного с обычным файлом.

Листинг 5.3. Вывод сведений о буферизации для различных потоков ввода-вывода

```
#include "apue.h"

void pr_stdio(const char *, FILE *);

int
main(void)
{
    FILE *fp;

    fputs("введите любой символ\n", stdout);
    if (getchar() == EOF)
        err_sys("ошибка вызова функции getchar");
    fputs("эта строка выводится на стандартный вывод сообщений об ошибках\n",
           stderr);

    pr_stdio("stdin", stdin);
```

```
    pr_stdio("stdout", stdout);
    pr_stdio("stderr", stderr);

    if ((fp = fopen("/etc/motd", "r")) == NULL)
        err_sys("ошибка вызова функции fopen");
    if (getc(fp) == EOF)
        err_sys("ошибка вызова функции getc");
    pr_stdio("/etc/motd", fp);
    exit(0);
}

void
pr_stdio(const char *name, FILE *fp)
{
    printf("поток = %s, ", name);
    /*
     * Следующий код не является переносимым.
     */
    if (fp->_IO_file_flags & _IO_UNBUFFERED)
        printf("буферизация отсутствует");
    else if (fp->_IO_file_flags & _IO_LINE_BUF)
        printf("построчная буферизация");
    else /* ни один из вышеперечисленных режимов */
        printf("полная буферизация");
    printf(", размер буфера = %d\n", fp->_IO_buf_end - fp->_IO_buf_base);
}
```

Обратите внимание: для каждого потока, прежде чем вывести сведения о буферизации, мы выполняем операцию ввода-вывода, поскольку размещение буферов в памяти производится обычно во время первой операции ввода-вывода. Поля структуры `FILE` `_IO_file_flags`, `_IO_buf_base`, `_IO_buf_end` и константы `_IO_UNBUFFERED` и `_IO_LINE_BUFFERED` определены в стандартной библиотеке ввода-вывода GNU, используемой в ОС Linux. Помните, что реализации стандартной библиотеки ввода-вывода могут различаться в разных версиях UNIX.

Запустив программу дважды – один раз, когда все три стандартных потока ввода-вывода были связаны с терминалом, и второй раз, когда они были перенаправлены в файлы, мы получили следующие результаты:

```
$ ./a.out                   stdin, stdout и stderr связаны с терминалом
введите любой символ
                            здесь мы ввели символ перевода строки
эта строка выводится на стандартный вывод сообщений об ошибках
поток = stdin, построчная буферизация, размер буфера = 1024
поток = stdout, построчная буферизация, размер буфера = 1024
поток = stderr, буферизация отсутствует, размер буфера = 1
поток = /etc/motd, полная буферизация, размер буфера = 4096
$ ./a.out < /etc/termcap > std.out 2> std.err
                            запустим еще раз с перенаправлением
                            трех стандартных потоков в файлы
$ cat std.err
эта строка выводится на стандартный вывод сообщений об ошибках
```

```
$ cat std.out
введите любой символ
поток = stdin, полная буферизация, размер буфера = 4096
поток = stdout, полная буферизация, размер буфера = 4096
поток = stderr, буферизация отсутствует, размер буфера = 1
поток = /etc/motd, полная буферизация, размер буфера = 4096
```

Мы видим, что в данной системе стандартные потоки ввода и вывода буферизуются построчно, когда они связаны с терминалом. Размер буфера в этом случае составляет 1024 байта. Обратите внимание, что это не ограничивает размер вводимых и выводимых строк 1024 байтами – это лишь размер буфера. Для записи строки длиной 2048 байт в стандартный поток вывода потребуется два обращения к системному вызову `write`. Когда эти два потока перенаправляются в обычные файлы, они приобретают режим полной буферизации с размером буфера, равным предпочтительному размеру блока ввода-вывода (значение `st_blksize` структуры `stat`) для данной файловой системы. Также мы видим, что стандартный поток сообщений об ошибках не буферизуется ни в одном из случаев, как это и должно быть, и что по умолчанию для обычных файлов назначается режим полной буферизации.

5.13. Временные файлы

Стандарт ISO C определяет две вспомогательные функции, которые используются при создании временных файлов.

```
#include <stdio.h>

char *tmpnam(char *ptr);
```

Возвращает указатель на строку с уникальным именем файла

```
FILE *tmpfile(void);
```

Возвращает указатель на файл в случае успеха, `NULL` в случае ошибки

Функция `tmpnam` генерирует строку с уникальным полным именем файла, не существующего в данный момент в системе. При каждом вызове эта функция генерирует неповторяющиеся имена файлов до `TMP_MAX` раз. Константа `TMP_MAX` определена в файле `<stdio.h>`.

> Стандарт ISO C определяет эту константу и требует, чтобы ее значение было не меньше 25. Стандарт Single UNIX Specification требует, чтобы XSI-совместимые системы поддерживали константу `TMP_MAX` со значением, по меньшей мере, 10000. Хотя это минимальное значение и позволяет использовать четыре цифры (0000–9999) для создания уникальных имен файлов, тем не менее большинство реализаций UNIX используют для этих целей символы верхнего и нижнего регистра.

Если аргумент *ptr* содержит значение `NULL`, генерируемое имя файла сохраняется в статической области памяти и функция возвращает указатель на эту область. Последующие вызовы функции `tmpnam` могут затереть эту область памяти. (Это означает, что если мы вызываем функцию `tmpnam` более одного

раза, а имя временного файла необходимо сохранить, то мы должны полностью скопировать полученную строку, а не указатель на нее.) Если в аргументе *ptr* передается непустой указатель, то предполагается, что он содержит адрес буфера размером не менее L_tmpnam символов. (Константа L_tmpnam определена в файле <stdio.h>.) Сгенерированное имя файла сохраняется в этом буфере, и функция возвращает значение указателя ptr.

Функция tmpfile создает временный двоичный файл (wb+), который автоматически удаляется при его закрытии или по завершении процесса. В UNIX не имеет никакого значения тот факт, что файл двоичный.

Пример

Программа, представленная в листинге 5.4, демонстрирует работу с обеими функциями.

Листинг 5.4. Демонстрация функций tmpnam и tmpfile

```
#include "apue.h"

int
main(void)
{
    char name[L_tmpnam], line[MAXLINE];
    FILE *fp;

    printf("%s\n", tmpnam(NULL));      /* первое имя временного файла */
    tmpnam(name);                      /* второе имя временного файла */
    printf("%s\n", name);

    if ((fp = tmpfile()) == NULL)      /* создать временный файл */
        err_sys("ошибка вызова функции tmpfile");
    fputs("записанная строка\n", fp);  /* записать во временный файл */
    rewind(fp);                        /* затем прочитать */

    if (fgets(line, sizeof(line), fp) == NULL)
        err_sys("ошибка вызова функции fgets");
    fputs(line, stdout);               /* и вывести прочитанную строку */

    exit(0);
}
```

Запустив программу из листинга 5.4, мы получили следующие результаты:

```
$ ./a.out
/tmp/fileC1Icwc
/tmp/filemSkHSe
записанная строка
```

Как правило, функция tmpfile использует следующий алгоритм: сначала создается уникальное имя файла с помощью функции tmpnam, затем создается сам файл, для которого сразу же вызывается функция unlink. В разделе 4.15 мы уже говорили, что вызов функции unlink не приводит к немедленному удалению файла. Он будет удален системой автоматически в момент закрытия или по завершении процесса.

Стандарт Single UNIX Specification определяет в качестве расширений XSI две дополнительные функции для работы с временными файлами. Одна из них – функция tempnam.

```
#include <stdio.h>

char *tempnam(const char *directory, const char *prefix);
```

Возвращает указатель на строку с уникальным именем файла

Функция tempnam представляет собой разновидность функции tmpnam, которая позволяет вызывающей программе определить каталог и префикс имени временного файла. Существует четыре варианта выбора каталога, причем чаще всего используется первый из них.

1. Если определена переменная окружения TMPDIR, то в качестве имени каталога используется ее значение. (Переменные окружения мы опишем в разделе 7.9.)
2. Если в аргументе *directory* передается непустой указатель, то в качестве имени каталога используется заданная строка.
3. Если в файле <stdio.h> определена константа P_tmpdir, то она используется в качестве имени каталога.
4. В качестве имени каталога используется локальный каталог, обычно /tmp.

Если аргумент *prefix* не является пустым указателем, то он должен указывать на строку длиной до пяти байт, с которой будет начинаться имя временного файла.

Эта функция выделяет буфер для сгенерированного имени в динамической памяти с помощью функции malloc. По окончании работы со строкой пути можно освободить занимаемую память. (Функции malloc и free рассматриваются в разделе 7.8.)

Пример

Программа, представленная в листинге 5.5, демонстрирует работу функции tempnam.

Листинг 5.5. Использование функции tempnat

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    if (argc != 3)
        err_quit("Использование: a.out <каталог> <префикс>");
    printf("%s\n", tempnam(argv[1][0] != ' ' ? argv[1] : NULL,
           argv[2][0] != ' ' ? argv[2] : NULL));
    exit(0);
}
```

Обратите внимание: когда один из аргументов командной строки (имя каталога или префикс) начинается с пробела, в функцию вместо соответствующего указателя передается значение NULL. Теперь продемонстрируем различные возможности функции:

```
$ ./a.out /home/sar TEMP   задается имя каталога и префикс
/home/sar/TEMPsf0Ozi
$ ./a.out " " PFX          используется имя каталога по умолчанию: P_tmpdir
/tmp/PFXfBw7Gi
$ TMPDIR=/var/tmp ./a.out /usr/tmp " "   используется переменная
                                         окружения, префикса нет
/var/tmp/file8fVYNi                      используется переменная окружения,
                                         а не указанный каталог
$ TMPDIR=/no/such/dir ./a.out /home/sar/tmp QQQ
/home/sar/tmp/QQQ98s8Ui                  ошибочное значение переменной
                                         окружения игнорируется
```

Как видите, имя каталога определяется в последовательности, указанной выше, кроме того, функция проверяет, существует ли заданный каталог. Если каталог не существует (например, /no/such/dir), функция пытается применить следующий вариант из списка. Из этого примера мы также видим, что в данной реализации определена константа P_tmpdir со значением /tmp. Прием, который мы использовали для изменения значения переменной окружения, указав новое значение TMPDIR перед именем программы, применяется в командных оболочках Bourne shell, Korn shell и bash.

Вторая функция, определенная как расширение XSI, – это функция mkstemp. Она похожа на функцию tmpfile, но возвращает не указатель на файл, а открытый дескриптор временного файла.

```
#include <stdlib.h>

int mkstemp(char *template);
```

Возвращает дескриптор файла в случае успеха, –1 в случае ошибки

Возвращаемый дескриптор открыт для чтения и записи. Имя временного файла выбирается в соответствии с заданным шаблоном *template*, который представляет собой полный путь к файлу и должен содержать в конце шесть символов XXXXXX. Функция заменяет их случайными символами, создавая уникальное имя файла. В случае успеха функция mkstemp изменяет строку *template*, в которую записывается полученное имя временного файла.

В отличие от tmpfile, функция mkstemp не удаляет временный файл автоматически. Если временный файл необходимо удалить, то мы должны сделать это самостоятельно.

Функции tmpnam и tempnam имеют один недостаток: между моментом, когда будет сгенерировано уникальное имя файла, и моментом, когда приложение создаст файл с этим именем, существует некоторый промежуток времени. За это время другой процесс может создать файл с тем же самым именем.

Поэтому предпочтительнее использовать функции `tempfile` и `mkstemp`, которые лишены этого недостатка.

> Существует функция `mktemp`, которая напоминает `mkstemp`, за исключением того, что она создает имя временного файла. Функция `mktemp` не создает файл, таким образом она страдает тем же недостатком, что и функции `tmpnam` и `tempnam`. В стандарте Single UNIX Specification функция `mktemp` отмечена как унаследованный интерфейс. Унаследованные интерфейсы могут быть исключены из стандарта в будущем, поэтому новые приложения не должны использовать эту функцию.

5.14. Альтернативы стандартной библиотеке ввода-вывода

Стандартная библиотека ввода-вывода не совершенна. В книге [Korn and Vo 1991] перечисляются ее многочисленные недостатки, некоторые из которых присущи базовой архитектуре, но большая часть связана с различными аспектами реализации.

Один из врожденных недостатков стандартной библиотеки ввода-вывода, снижающий ее эффективность, заключается в том, что она выполняет большое количество операций копирования данных. При использовании функций построчного ввода-вывода, `fgets` и `fputs`, данные обычно копируются дважды: один раз – между ядром и буфером ввода-вывода (то есть при вызове функций `read` и `write`) и второй раз – между буфером ввода-вывода и строкой. Библиотека скоростного ввода-вывода (`fio`(3) в [AT&T 1990a]) обходит этот недостаток за счет того, что функция, которая считывает строку, возвращает указатель на нее вместо того, чтобы копировать строку в другой буфер. В [Hume 1988] говорится, что таким образом можно в три раза увеличить скорость работы утилиты `grep`(1).

В [Korn and Vo 1991] описывается другая альтернатива стандартной библиотеке ввода-вывода – *`sfio`*. Этот пакет почти не уступает в скорости библиотеке *`fio`* и обычно гораздо эффективнее, чем стандартная библиотека ввода-вывода. Кроме того, пакет *`sfio`* реализует некоторые функциональные возможности, недоступные в других библиотеках: потоки ввода-вывода более универсальны и могут представлять как файлы, так и области памяти, операции с потоками ввода-вывода могут быть изменены за счет подключения дополнительных модулей, улучшена обработка исключительных ситуаций.

В [Krieger, Stumm, and Unrau 1992] рассматривается другая альтернатива – с использованием файлов, отображаемых в память с помощью функции `mmap` (она будет описана в разделе 14.9). Этот пакет называется ASI, Alloc Stream Interface (интерфейс размещения потоков). Программный интерфейс очень напоминает функции распределения памяти, используемые в UNIX (`malloc`, `realloc` и `free`, которые мы рассмотрим в разделе 7.8). Как и пакет *`sfio`*, ASI старается минимизировать количество операций копирования за счет использования указателей.

Существуют реализации стандартной библиотеки ввода-вывода, спроектированные для систем с небольшими объемами памяти, таких как встраивае-

мые устройства. Эти реализации отличаются весьма скромными требованиями к памяти, в то время как переносимости, скорости или функциональным возможностям уделяется меньше внимания. В качестве примеров таких реализаций можно назвать библиотеку uClibc (за подробной информацией обращайтесь по адресу *http://www.uclibc.org*) и библиотеку newlibc (*http://sources.redhat.com/newlib*).

5.15. Подведение итогов

Стандартная библиотека ввода-вывода используется в большинстве приложений UNIX. Мы рассмотрели все функции, предоставляемые этой библиотекой, а также некоторые особенности ее реализации и вопросы эффективности. Следует всегда помнить самое главное – никогда не забывать о буферизации, поскольку именно в связи с нею возникает больше всего проблем и недопонимания.

Упражнения

5.1. Напишите реализацию функции `setbuf` с использованием функции `setvbuf`.

5.2. Измените программу из раздела 5.8, которая копирует файл с помощью функций построчного ввода-вывода (`fgets` и `fputs`), так, чтобы вместо константы `MAXLINE` использовалось значение 4. Что произойдет, если вы попытаетесь копировать файл, в котором длина строк превышает 4 байта? Объясните почему.

5.3. Что означает значение 0, возвращаемое функцией `printf`?

5.4. Следующий код корректно работает в одних системах и некорректно в других. В чем причина такого поведения?

```
#include <stdio.h>

int
main(void)
{
    char c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

5.5. Почему функция `tempnam` ограничивает длину префикса пятью символами?

5.6. Как можно использовать функцию `fsync` (раздел 3.13) с потоками ввода-вывода?

5.7. В программах из листингов 1.5 и 1.8 строка приглашения не содержит символа перевода строки, и мы не вызываем функцию `fflush`. Каким же образом она выводится на экран?

6

Информация о системе и файлы данных

6.1. Введение

Для нормальной работы любой UNIX-системы требуется наличие множества файлов данных. Файл паролей `/etc/passwd` и файл групп `/etc/group` очень часто используются в самых разных приложениях. Например, обращение к файлу паролей происходит всякий раз, когда пользователь входит в систему или когда кто-либо исполняет команду `ls -l`.

Исторически эти файлы представляют собой обычные текстовые файлы в формате ASCII, которые могут быть прочитаны с помощью стандартной библиотеки ввода-вывода. Но в больших системах последовательный просмотр записей в файле паролей оказывается довольно ресурсоемким. Имеет смысл хранить эти данные в формате, отличном от ASCII, но при этом необходим интерфейс для прикладных программ, который мог бы работать с любыми форматами файлов. Переносимые интерфейсы для доступа к этим файлам – тема данной главы. Кроме того, мы рассмотрим функции, предоставляющие информацию о системе, и функции даты и времени.

6.2. Файл паролей

Файл паролей UNIX, который стандарт POSIX.1 называет базой данных пользователей, содержит поля, перечисленные в табл. 6.1. Эти поля также имеются в структуре `passwd`, которая определена в заголовочном файле `<pwd.h>`.

> Обратите внимание: стандарт POSIX.1 определяет только пять из десяти полей структуры `passwd`. Большинство же платформ поддерживают как минимум семь полей. Системы, производные от BSD, поддерживают все десять.

Традиционно файл паролей хранится под именем `/etc/passwd` и представляет собой обычный текстовый файл в формате ASCII. Каждая строка файла состоит из полей (см. табл. 6.1), разделенных двоеточиями. Для примера приведем четыре строки из файла `/etc/passwd` в ОС Linux:

```
root:x:0:0:root:/root:/bin/bash
squid:x:23:23::/var/spool/squid:/dev/null
nobody:x:65534:65534:Nobody:/home:/bin/sh
sar:x:205:105:Stephen Rago:/home/sar:/bin/bash
```

Таблица 6.1. Поля файла /etc/passwd

Описание	Поле структуры passwd	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Имя пользователя	char *pw_name	•	•	•	•	•
Зашифрованный пароль	char *pw_passwd		•	•	•	•
Числовой идентификатор пользователя	uid_t pw_uid	•	•	•	•	•
Числовой идентификатор группы	gid_t pw_gid	•	•	•	•	•
Поле комментария	char *pw_gecos		•	•	•	•
Начальный рабочий каталог	char *pw_dir	•	•	•	•	•
Командный интерпретатор	char *pw_shell	•	•	•	•	•
Класс доступа пользователя	char *pw_class		•		•	
Следующее время изменения пароля	time_t pw_change		•		•	
Время истечения срока действия учетной записи	time_t pw_expire		•		•	

Обратите внимание на следующие обстоятельства:

- Как правило, в файле паролей имеется запись для пользователя с именем root. Этот пользователь имеет числовой идентификатор 0 (суперпользователь).
- Поле зашифрованного пароля содержит единственный символ-заполнитель. Старые версии UNIX хранили в этом поле зашифрованный пароль. Поскольку хранение даже зашифрованных паролей в файле, доступном для чтения всем пользователям, представляет собой угрозу безопасности, современные системы хранят зашифрованные пароли в другом месте. Мы подробнее рассмотрим этот вопрос в следующем разделе, когда будем обсуждать пароли.
- Некоторые поля могут быть пустыми. Так, если поле зашифрованного пароля не заполнено, это означает, что у пользователя нет пароля (что не рекомендуется). Запись для пользователя squid имеет одно пустое поле – поле комментария. Пустое поле комментария не оказывает никакого эффекта.

- Поле командного интерпретатора содержит имя исполняемого файла программы, которая используется в качестве командной оболочки при входе пользователя в систему. Если поле пустое, то значение по умолчанию для него – /bin/sh. Однако обратите внимание, что для пользователя squid в качестве командного интерпретатора используется устройство /dev/null. Очевидно, что оно не является исполняемым файлом – это предотвращает возможность входа в систему под именем squid для кого бы то ни было.

  Многие процессы-демоны, предоставляющие различные службы, имеют собственные идентификаторы пользователей (глава 13). Так, учетная запись squid предназначена для процессов, которые предоставляют услуги кэширующего прокси-сервера squid.

- Использование псевдоустройства /dev/null не единственный способ воспрепятствовать конкретному пользователю войти в систему. Не менее часто можно встретить /bin/false в качестве командного интерпретатора. Эта команда просто возвращает ненулевое значение (признак ошибки); оболочка расценивает этот код завершения как ложь. Также нередко для отключения учетной записи используется команда /bin/true. Она всегда возвращает признак успешного завершения (нулевое значение). В некоторых системах имеется команда nologin. Она выводит заданное сообщение об ошибке и возвращает код завершения, отличный от нуля.
- Имя пользователя nobody применяется с целью дать возможность кому-либо войти в систему, но с идентификатором пользователя 65534 и идентификатором группы 65534, которые не дают никаких привилегий. Он сможет получить доступ лишь к тем файлам, которые доступны на чтение или на запись для всех. (Это предполагает отсутствие в системе файлов, принадлежащих пользователю с идентификатором 65534 или группе с идентификатором 65534.)
- Некоторые системы, которые предоставляют команду finger(1), поддерживают включение дополнительной информации в поле комментария. Дополнительные поля в поле комментария отделяются друг от друга запятой и содержат реальное имя пользователя, место работы, номера рабочего и домашнего телефонов. Кроме того, некоторые утилиты заменяют амперсанд (&) в поле комментария именем пользователя (в верхнем регистре). Представим себе учетную запись следующего содержания:

  ```
  sar:x:205:105:Steve Rago, SF 5-121, 555-1111, 555-2222:/home/sar:/bin/sh
  ```

  В этом случае утилита finger выдала бы следующую информацию о пользователе Steve Rago:

  ```
  $ finger -p sar
  Login: sar                              Name: Steve Rago
  Directory: /home/sar                    Shell: /bin/sh
  Office: SF 5-121, 555-1111              Home Phone: 555-2222
  On since Mon Jan 19 03:57 (EST) on ttyv0 (messages off)
  No Mail.
  ```

Даже если система не поддерживает команду finger, ничто не мешает включать дополнительную информацию в поле комментария, поскольку это поле никак не интерпретируется системными утилитами.

В некоторых системах администратору доступна команда vipw, предназначенная для редактирования файла паролей. Команда vipw производит сериализацию изменений в файле паролей и обеспечивает согласование внесенных изменений с данными, хранящимися в дополнительных файлах. Многие системы предоставляют подобные возможности через графический интерфейс.

Стандарт POSIX.1 определяет всего две функции, с помощью которых можно получить доступ к отдельным учетным записям в файле паролей. Эти функции позволяют найти учетную запись по числовому идентификатору или имени пользователя.

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);
```

Обе возвращают указатель в случае успеха, NULL в случае ошибки

Функция getpwuid используется утилитой ls(1) для преобразования числового идентификатора, который хранится в индексном узле, в имя пользователя. Функция getpwnam используется утилитой login(1), когда пользователь вводит свое имя при входе в систему.

Обе функции возвращают указатель на заполненную структуру passwd. Эта структура обычно является статической переменной, расположенной в области памяти функции; в результате ее содержимое перезаписывается при каждом обращении к функции.

Эти две функции стандарта POSIX.1 удобны в том случае, когда нужно получить сведения о конкретном пользователе по его имени или числовому идентификатору, однако некоторым программам может потребоваться просмотреть весь файл паролей. Для этого предназначены следующие три функции.

```
#include <pwd.h>

struct passwd *getpwent(void);
```

Возвращает указатель в случае успеха, NULL в случае ошибки или по достижении конца файла

```
void setpwent(void);

void endpwent(void);
```

Эти три функции не включены в базовый стандарт POSIX.1. Они определены как расширения XSI в Single UNIX Specification. Таким образом, предполагается, что все версии UNIX должны их поддерживать.

Функция getpwent вызывается для получения очередной записи из файла паролей. Как и предыдущие две функции, getpwent возвращает указатель на заполненную структуру passwd. Обычно содержимое структуры перезаписывается при каждом вызове функции. При первом вызове функция открывает все необходимые файлы. Порядок, в котором возвращаются записи, заранее не определен – они могут следовать в любом порядке, потому что некоторые системы используют хешированную версию файла /etc/passwd.

Функция setpwent производит переход к началу каждого из используемых файлов, а функция endpwent закрывает файлы. По окончании работы с getpwent необходимо всегда закрывать файлы с помощью функции endpwent. Функция getpwent в состоянии определить, когда нужно открывать файлы (при первом обращении к ней), но она никак не сможет узнать, когда работа с файлом паролей закончена.

Пример

В листинге 6.1 показана реализация функции getpwnam.

Листинг 6.1. Функция getpwnam

```
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd *ptr;

    setpwent();
    while ((ptr = getpwent()) != NULL)
        if (strcmp(name, ptr->pw_name) == 0)
            break; /* совпадение найдено */

    endpwent();
    return(ptr);   /* в ptr будет значение NULL, если совпадение не найдено */
}
```

Обращение к функции setpwent в самом начале – это мера предосторожности; мы просто выполняем переход в начало файла на тот случай, если вызывающая программа уже открыла его вызовом функции getpwent. Функция endpwent вызывается по окончании работы по той причине, что ни getpwnam, ни getpwuid не должны оставлять файлы открытыми.

6.3. Теневые пароли

Зашифрованный пароль – это пароль пользователя, пропущенный через односторонний алгоритм шифрования. Поскольку алгоритм является односторонним, невозможно получить оригинальный пароль, имея его зашифрованную версию.

Традиционно этот алгоритм (см. [Morris and Thompson 1979]) генерирует 13 печатных символов из 64-символьного набора [a-zA-Z0-9./]. Некоторые современные системы используют для шифрования паролей алгоритм MD5, который генерирует последовательность из 31 символа. (Чем больше символов используется для хранения зашифрованного пароля, тем больше возможных комбинаций и тем сложнее будет подобрать истинный пароль простым перебором вариантов.) Если в поле зашифрованного пароля помещается единственный символ, это гарантирует, что пароль никогда не будет соответствовать этому значению.

Имея зашифрованный пароль, мы не можем применить к нему алгоритм, который преобразовал бы его обратно в простой текст. (Пароль в виде простого текста – это тот набор символов, который мы вводим в ответ на приглашение `Password:`.) Однако можно получить пароль, перебирая различные комбинации символов, пропуская их через односторонний алгоритм шифрования и сравнивая полученные результаты с зашифрованной версией пароля. Если бы пароли пользователей представляли собой набор случайных символов, то такой метод был бы не очень удачным решением. Однако пользователи стремятся выбирать пароли не случайным образом, нередко в качестве пароля они используют имена супругов, названия улиц или клички домашних животных. Это облегчает задачу любому, кто получил в свои руки копию файла паролей и пытается вычислить их. (Глава 4 книги [Garfinkel et al. 2003] содержит дополнительные сведения о паролях и о схеме шифрования пароля, используемой в UNIX.)

Чтобы затруднить доступ к зашифрованным паролям, современные системы хранят их в другом файле, который часто называют *теневым файлом паролей* (*shadow password file*). Этот файл должен содержать как минимум имена пользователей и зашифрованные пароли. Здесь также находится и другая информация, имеющая отношение к паролям (табл. 6.2).

Таблица 6.2. Поля записей в файле /etc/shadow

Описание	**Поле структуры** `spwd`
Имя пользователя	`char *sp_namp`
Зашифрованный пароль	`char *sp_pwdp`
Время последнего изменения пароля, в днях от начала Эпохи	`int sp_lstchg`
Минимальный период в днях между изменениями пароля	`int sp_min`
Максимальный период в днях между изменениями пароля	`int sp_max`
Количество дней до истечения срока действия пароля, в течение которых пользователь будет предупреждаться о необходимости его изменения	`int sp_warn`
Количество дней, в течение которых учетная запись не использовалась	`int sp_inact`
Время отключения учетной записи в днях от начала Эпохи	`int sp_expire`
Зарезервировано	`unsigned int sp_flag`

Обязательными являются только два поля – имя пользователя и зашифрованный пароль. Другие поля хранят информацию о том, как часто должен изменяться пароль и как долго будет оставаться активной учетная запись.

Теневой файл паролей не должен быть доступен на чтение для всех. Доступ к нему должны иметь лишь несколько программ, например `login`(1) и `passwd`(1). Часто владельцем таких программ является суперпользователь, и для них устанавливается бит set-user-ID. При использовании теневых паролей обычный файл паролей, `/etc/passwd`, может быть доступен для чтения любому.

В ОС Linux 2.4.22 и Solaris 9 для доступа к теневому файлу паролей предусмотрен отдельный набор функций, которые очень похожи на те, что используются для работы с обычным файлом паролей.

```
#include <shadow.h>
struct spwd *getspnam(const char *name);
struct spwd *getspent(void);
                Обе возвращают указатель в случае успеха, NULL в случае ошибки
void setspent(void);
void endspent(void);
```

В ОС FreeBSD 5.2.1 и Mac OS X 10.3 теневой файл паролей отсутствует[1], а вся дополнительная информация хранится в обычном файле паролей (табл. 6.1).

6.4. Файл групп

Файл групп UNIX, называемый в стандарте POSIX.1 базой данных групп, содержит поля, которые перечислены в табл. 6.3. Значения этих полей хранятся в структуре `group`, определенной в файле `<grp.h>`.

Таблица 6.3. Поля записей в файле /etc/group

Описание	Поле структуры group	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Имя группы	`char *gr_name`	•	•	•	•	•
Зашифрованный пароль	`char *gr_passwd`		•	•	•	•
Числовой идентификатор группы	`int gr_gid`	•	•	•	•	•
Массив указателей на отдельные имена пользователей	`char **gr_mem`	•	•	•	•	•

[1] В BSD-системах функцию файла теневых паролей выполняет файл `/etc/master.passwd`. По формату он аналогичен файлу `passwd`, но содержит зашифрованный пароль. Подробнее см. в разделе 6.6. – *Примеч. науч. ред.*

Поле gr_mem представляет собой массив указателей на имена пользователей, которые входят в состав группы. Этот массив завершается пустым указателем.

Для поиска имени группы или ее числового идентификатора в файле групп стандарт POSIX.1 определяет следующие две функции.

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
```

Обе возвращают указатель в случае успеха, NULL в случае ошибки

Как и в случае с файлом паролей, обе функции обычно возвращают указатель на статическую переменную, которая при каждом обращении к функциям перезаписывается.

Чтобы произвести поиск по всему файлу групп, потребуются некоторые дополнительные функции. Следующие три функции напоминают те, что используются для работы с файлом паролей.

```
#include <grp.h>
struct group *getgrent(void);
```

Возвращает указатель в случае успеха, NULL в случае ошибки или по достижении конца файла

```
void setgrent(void);
void endgrent(void);
```

Эти три функции не включены в базовый стандарт POSIX.1. Они определены как расширения XSI в Single UNIX Specification и предоставляются всеми версиями UNIX.

Функция setgrent открывает файл групп, если он еще не был открыт, и переходит в его начало. Функция getgrent считывает очередную запись из файла групп, предварительно открыв его, если он еще не был открыт. Функция endgrent закрывает файл групп.

6.5. Идентификаторы дополнительных групп

Правила использования групп в UNIX существенно изменились за долгое время. Так, в Version 7 каждый пользователь в конкретный момент времени мог принадлежать только к одной группе. После входа в систему пользователю назначался реальный идентификатор группы, соответствующий числовому идентификатору группы из записи в файле паролей. Пользователь в любой момент мог изменить свою принадлежность к группе с помощью утилиты newgrp(1). Если команда newgrp завершалась успехом (за информацией о правилах назначения прав доступа обращайтесь к справочному руководству), реальный идентификатор группы заменялся идентификатором новой группы, который затем использовался для всех последующих проверок

прав доступа к файлам. Пользователь всегда мог вернуться к первоначальной группе, запустив команду `newgrp` без параметров.

Такая практика сохранялась до тех пор, пока не была изменена в версии 4.2BSD (около 1983 г.). Начиная с версии 4.2BSD появилось понятие идентификаторов дополнительных групп. Теперь пользователь мог принадлежать как группе, идентификатор которой указан в учетной записи в файле паролей, так и входить в состав до 16 дополнительных групп. Проверки прав доступа к файлам были изменены таким образом, чтобы проверялся не только эффективный идентификатор группы, но и все идентификаторы дополнительных групп.

> Дополнительные группы – обязательная для реализации функциональная особенность в соответствии со стандартом POSIX.1. (В ранних версиях POSIX.1 она была необязательной.) Константа `NGROUPS_MAX` (табл. 2.8) определяет количество идентификаторов дополнительных групп. Наиболее часто встречающееся значение – 16 (табл. 2.12).

Преимущество дополнительных групп состоит в том, что пользователю больше не нужно явно изменять свою принадлежность к группе. Членство в нескольких группах одновременно стало самым обычным делом (например, участие в разработке нескольких проектов).

Для получения и изменения идентификаторов дополнительных групп предназначены следующие три функции.

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

Возвращает количество дополнительных групп в случае успеха, −1 в случае ошибки

```
#include <grp.h>    /* в Linux */
#include <unistd.h> /* в FreeBSD, Mac OS X и Solaris */

int setgroups(int ngroups, const gid_t grouplist[]);

#include <grp.h>    /* в Linux и Solaris */
#include <unistd.h> /* в FreeBSD и Mac OS X */

int initgroups(const char *username, gid_t basegid);
```

Обе возвращают 0 в случае успеха, −1 в случае ошибки

> Из этих трех функций только `getgroups` определена стандартом POSIX.1. Функции `setgroups` и `initgroups` не вошли в стандарт, поскольку они относятся к разряду привилегированных операций. Однако четыре платформы, рассматриваемые в данной книге, поддерживают все три функции.
>
> В Mac OS X 10.3 аргумент *basegid* объявлен с типом `int`.

Функция `getgroups` заполняет массив *grouplist* идентификаторами дополнительных групп. В массив будет записано до *gidsetsize* элементов. Количество идентификаторов, записанных в массив, передается в вызывающую программу в виде возвращаемого значения.

Если в аргументе *gidsetsize* передается 0, функция возвращает только количество дополнительных групп. Массив *grouplist* не изменяется. (Это позволяет вызывающей программе определить размер массива *grouplist* перед его размещением в динамической памяти.)

Функция setgroups может быть вызвана только суперпользователем для изменения списка идентификаторов дополнительных групп вызывающего процесса: в этом случае *grouplist* содержит массив идентификаторов групп, а *ngroups* – количество элементов в массиве. Значение *ngroups* не должно превышать константу NGROUPS_MAX.

Единственное практическое применение функции setgroups – вызов из функции initgroups, которая читает файл групп с помощью функций getgrent, setgrent и endgrent и определяет группы, к которым принадлежит *username*. После этого она вызывает setgroups, чтобы инициализировать список идентификаторов дополнительных групп пользователя. Чтобы вызвать функцию initgroups, процесс должен обладать привилегиями суперпользователя, так как она вызывает функцию setgroups. В дополнение ко всем найденным группам, к которым принадлежит *username*, initgroups также включает в список идентификаторов дополнительных групп и значение *basegid*, где *basegid* – идентификатор группы, взятый из записи для *username* в файле паролей.

Функция initgroups используется немногими программами, в качестве примера можно упомянуть утилиту login(1), которая вызывает initgroups, когда пользователь входит в систему.

6.6. Различия реализаций

Мы уже рассмотрели теневой файл паролей, который поддерживается в ОС Linux и Solaris. FreeBSD и Mac OS X хранят зашифрованные пароли иным способом. В табл. 6.4 обобщены сведения о том, как четыре платформы, обсуждаемые в данной книге, хранят информацию о пользователях и группах.

Таблица 6.4. Различия в реализации хранения учетных записей

Информация	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Учетные записи	/etc/passwd	/etc/passwd	netinfo	/etc/passwd
Зашифрованные пароли	/etc/master.passwd	/etc/shadow	netinfo	/etc/shadow
Наличие хешированных файлов паролей	Да	Нет	Нет	Нет
Информация о группах	/etc/group	/etc/group	netinfo	/etc/group

В ОС FreeBSD теневой файл паролей хранится под именем /etc/master.passwd. Для редактирования этого файла используются специальные команды, которые в свою очередь генерируют файл /etc/passwd на основе теневого файла паролей. Кроме того, создаются хешированные версии файлов: /etc/pwd.db – хешированная версия файла /etc/passwd и /etc/spwd.db – хешированная вер-

сия файла /etc/master.passwd. Они обеспечивают более высокую производительность в крупных системах.

Однако в ОС Mac OS X файлы /etc/passwd и /etc/master.passwd используются только в однопользовательском режиме (когда администратор проводит обслуживание системы; однопользовательский режим обычно означает, что системные службы не запущены). В нормальном, т. е. многопользовательском, режиме работы системы доступ к информации о пользователях и группах обеспечивает служба каталогов netinfo.

Хотя операционные системы Linux и Solaris поддерживают схожие интерфейсы теневых паролей, в их реализации все же имеются некоторые отличия. Например, в Solaris целочисленные поля из табл. 6.2 определены с типом int, а в Linux – с типом long int. Другое отличие – поле, где подсчитывается период, в течение которого учетная запись оставалась неактивной. В Solaris это поле обозначает количество дней, прошедших с момента последнего входа пользователя в систему, тогда как в Linux это количество дней, оставшихся до окончания максимального срока действия существующего пароля.

Во многих системах базы данных пользователей и групп реализованы с использованием Network Information Service (NIS – сетевая информационная служба). Это позволяет администраторам редактировать эталонные копии баз данных и распространять их автоматически по всем серверам в организации. Клиентские системы соединяются с серверами и получают от них необходимые сведения о пользователях и группах. NIS+ и Lightweight Directory Access Protocol (LDAP – облегченный протокол доступа к каталогам) предоставляют аналогичные функциональные возможности. В большинстве систем метод администрирования каждого вида информации контролируется с помощью конфигурационного файла /etc/nsswitch.conf.

6.7. Прочие файлы данных

До сих пор обсуждались только два файла системных данных: файл паролей и файл групп. В своей повседневной работе UNIX-системы используют множество других файлов. Например, для обеспечения поддержки сетевых взаимодействий на платформе BSD необходимы файлы /etc/services (перечень служб, предоставляемых серверами сети), /etc/protocols (перечень сетевых протоколов) и /etc/networks (список сетей). К счастью, интерфейсы для работы с этими файлами очень напоминают те, что мы уже обсудили в связи с файлом паролей и файлом групп.

Эти интерфейсы следуют одному общему принципу – для работы с каждым файлом предоставляется по меньшей мере три функции:

1. Функция get считывает следующую запись, открывая файл в случае необходимости. Эти функции обычно возвращают указатель на структуру. Если достигнут конец файла, возвращается пустой указатель. Большинство функций get возвращают указатель на структуру, размещенную статически, поэтому мы должны скопировать ее содержимое, чтобы сохранить его для последующего использования.

2. Функция set открывает файл, если он еще не открыт, и переходит в начало файла. Эта функция используется, если по каким-то причинам необходимо вернуться в начало файла.
3. Функция end закрывает файл данных. Как уже упоминалось ранее, она всегда должна вызываться по завершении работы с файлом.

Кроме того, если файл данных поддерживает тот или иной вид поиска по ключу, то предусматриваются функции, которые выполняют поиск записи по заданному ключу. Например, для файла паролей имеются две функции: getpwnam ищет запись с определенным именем пользователя, а getpwuid ищет запись с определенным идентификатором пользователя.

В табл. 6.5 приводятся некоторые функции для работы с файлами данных, обычные для систем UNIX. В эту таблицу включены функции, предназначенные для работы с файлами паролей и групп, которые рассматривались ранее в этой главе, и некоторые из функций поддержки сети. Перечислены все функции get, set и end для всех файлов данных, упомянутых в таблице.

Таблица 6.5. Функции для работы с системными файлами данных

Описание	Файл	Заголовочный файл	Структура	Дополнительные функции поиска
Пароли	/etc/passwd	<pwd.h>	passwd	getpwnam, getpwuid
Группы	/etc/group	<grp.h>	group	getgrnam, getgrgid
Теневой файл паролей	/etc/shadow	<shadow.h>	spwd	getspnam
Сетевые узлы	/etc/hosts	<netdb.h>	hostent	gethostbyname, gethostbyaddr
Сети	/etc/networks	<netdb.h>	netent	getnetbyname, getnetbyaddr
Протоколы	/etc/protocols	<netdb.h>	protoent	getprotobyname, getprotobynumber
Службы	/etc/services	<netdb.h>	servent	getservbyame, getservbyprot

В ОС Solaris последние четыре файла из табл. 6.5 являются символическими ссылками на файлы с теми же именами, расположенные в каталоге /etc/inet. Большинство реализаций UNIX предоставляют дополнительные функции, подобные перечисленным, но предназначенные для задач системного администрирования и специфичные для каждой конкретной реализации.

6.8. Учет входов в систему

В большинстве систем UNIX имеется два файла данных: utmp, в котором хранится информация о всех работающих в системе пользователях, и wtmp, в котором отслеживается информация о всех попытках входа в систему и выхода из нее. В Version 7 в оба файла записывалась в двоичном виде информация, представляемая одной и той же структурой:

```
struct utmp {
    char ut_line[8]; /* имя устройства: "ttyh0", "ttyd0", "ttyp0", ... */
```

```
    char ut_name[8]; /* пользователь */
    long ut_time;    /* количество секунд от начала Эпохи */
};
```

Во время входа пользователя в систему программа `login` заполняла одну такую структуру и записывала ее в файл `utmp`, и та же самая структура добавлялась к файлу `wtmp`. При выходе из системы процесс `init` стирал запись в файле `utmp`, заполняя ее нулевыми байтами, и в файл `wtmp` добавлялась новая запись. В этой записи, соответствующей выходу из системы, поле `ut_name` очищалось. Во время перезагрузки системы, а также до и после изменения системной даты и времени в файл `wtmp` добавлялись специальные записи. Утилита `who`(1) извлекала данные из файла `utmp` и выводила их в удобочитаемом виде. В более поздних версиях UNIX имеется команда `last`(1), которая читает данные из файла `wtmp` и выводит выбранные записи.

Большинство версий UNIX все еще поддерживают файлы `utmp` и `wtmp`, но, как и следовало ожидать, объем информации в этих файлах вырос. Так, 20-байтная структура, которая использовалась в Version 7, выросла до 36 байт в SVR2, а в SVR4 расширенная структура `utmp` занимает уже 350 байт!

> Детальное описание формата этих записей в ОС Solaris приводится на странице справочного руководства `utmpx`(4). В Solaris 9 оба файла находятся в каталоге `/var/adm`. ОС Solaris предоставляет много функций для работы с этими файлами, описание которых можно найти в `getutx`(3).
>
> На странице справочного руководства `utmp`(5) в Linux 2.4.22, FreeBSD 5.2.1 и Mac OS X 10.3 дается описание формата записей для этих версий. Полные имена этих двух файлов – `/var/run/utmp` и `/var/log/wtmp`.

6.9. Информация о системе

Стандарт POSIX.1 определяет функцию `uname`, которая возвращает сведения о текущем хосте и операционной системе.

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

Возвращает неотрицательное зачение в случае успеха, –1 в случае ошибки

В качестве аргумента передается адрес структуры `utsname`, и функция заполняет ее. Стандартом POSIX.1 определен лишь минимальный набор полей в этой структуре, каждое из которых является массивом символов, а размеры этих массивов определяются конкретными реализациями. Некоторые реализации добавляют в эту структуру дополнительные поля.

```
struct utsname {
    char sysname[];  /* имя операционной системы */
    char nodename[]; /* имя узла сети */
    char release[];  /* номер выпуска операционной системы */
    char version[];  /* номер версии этого выпуска */
```

```
    char machine[];  /* тип аппаратной архитектуры */
};
```

Каждая строка заканчивается нулевым символом. В табл. 6.6 приводятся максимальные размеры массивов для всех четырех платформ, обсуждаемых в этой книге. Информация, которая содержится в структуре `utsname`, обычно выводится командой `uname`(1).

> Стандарт POSIX.1 предупреждает, что поле `nodename` может не соответствовать действительному сетевому имени хоста. Эта поле пришло из System V, и ранее оно хранило имя хоста в сети, работающей по протоколу UUCP.
>
> Кроме того, помните, что содержимое этой структуры не дает никакой информации о версии POSIX.1. Эти сведения можно получить при помощи константы `_POSIX_VERSION` (обсуждалось ранее в разделе 2.6).
>
> Наконец, эта функция дает лишь возможность получить информацию в виде структуры; стандарт POSIX.1 никак не оговаривает порядок инициализации этой информации.

Системы, производные от BSD, традиционно предоставляют функцию `gethostname`, которая возвращает только имя хоста. Как правило, это имя соответствует сетевому имени хоста в сети TCP/IP.

```
#include <unistd.h>

int gethostname(char *name, int namelen);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Аргумент *namelen* определяет размер буфера *name*. Если в буфере предусмотрено достаточно пространства, то возвращаемая в *name* строка будет завершаться нулевым символом. Если места в буфере недостаточно, то не оговаривается, будет ли полученная строка завершаться нулевым символом.

В настоящее время функция `gethostname` является частью стандарта POSIX.1, который указывает, что максимальная длина имени хоста равна `HOST_NAME_MAX`. Значения максимальной длины имен хостов для всех четырех обсуждаемых платформ приводятся в табл. 6.6.

Таблица 6.6. Ограничения на строки идентификации системы

Функция	Максимальная длина аргумента name			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
uname	256	65	256	257
gethostname	256	64	256	256

Если хост подключен к сети TCP/IP, то имя хоста обычно представляет собой полное корректное доменное имя (fully qualified domain name).

Кроме того, существует команда `hostname`(1), которая может выводить или изменять имя хоста. (Имя хоста устанавливается суперпользователем с помощью аналогичной функции `sethostname`.) Как правило, имя хоста устанав-

ливается во время загрузки системы в одном из файлов инициализации, вызываемых /etc/rc или init.

6.10. Функции даты и времени

В качестве основной службы времени ядро UNIX предоставляет счетчик секунд, прошедших от начала Эпохи – 00:00:00 1 января 1970 года по всеобщему скоординированному времени (UTC). В разделе 1.10 мы уже говорили, что значение счетчика представлено типом данных time_t и называется *календарным временем*. С помощью календарного времени можно представить как дату, так и время суток. ОС UNIX всегда отличалась от других систем тем, что она (а) хранит время UTC, а не местное время, (б) автоматически выполняет преобразования, такие как переход на летнее время, и (в) хранит дату и время как единое целое.

Функция time возвращает текущее время и дату.

```
#include <time.h>

time_t time(time_t *calptr);
```

Возвращает значение времени в случае успеха, –1 в случае ошибки

Функция всегда возвращает значение времени. Если в качестве аргумента *calptr* передается непустой указатель, то значение времени дополнительно записывается по указанному адресу.

> Мы ничего не сказали о том, как ядро инициализирует текущее время. Исторически сложилось так, что реализации, производные от System V, вызывали функцию stime(2), тогда как системы, производные BSD, использовали функцию settimeofday(2).
>
> Стандарт Single UNIX Specification не оговаривает, как система должна устанавливать текущее время.

Функция gettimeofday дает более высокую точность, чем функция time (до микросекунд). Для некоторых приложений это очень важно.

```
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```

Всегда возвращает значение 0

Эта функция определена стандартом Single UNIX Specification как расширение XSI. Единственное допустимое значение аргумента *tzp* – NULL; любые другие значения могут привести к непредсказуемым результатам. Некоторые платформы поддерживают указание часового пояса через аргумент *tzp*, но это зависит от конкретной реализации и не определено в Single UNIX Specification.

Функция gettimeofday сохраняет время, прошедшее от начала Эпохи до настоящего момента, по адресу *tp*. Это время представлено в виде структуры *timeval*, которая хранит секунды и микросекунды:

```
struct timeval {
    time_t tv_sec; /* секунды */
    long tv_usec;  /* микросекунды */
};
```

Как только получено целочисленное значение количества секунд, прошедших с начала Эпохи, как правило, вызывается одна из функций преобразования, которая переведет числовое значение в удобочитаемые время и дату. На рис. 6.1 показаны взаимоотношения между различными функциями преобразования времени.

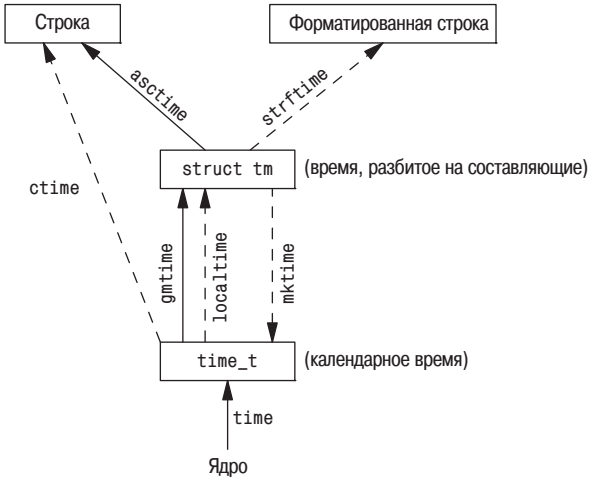


***Рис. 6.1.** Взаимоотношения между различными функциями представления времени*

(Четыре функции, которые показаны на этом рисунке пунктирными линиями, – localtime, mktime, ctime и strftime – учитывают значение переменной окружения TZ, которую мы рассмотрим далее в этом разделе.)

Две функции, localtime и gmtime, преобразуют календарное время в структуру tm, состоящую из следующих элементов:

```
struct tm {       /* время, разбитое на составляющие */
    int tm_sec;   /* секунды от начала минуты: [0 - 60] */
    int tm_min;   /* минуты от начала часа: [0 - 59] */
    int tm_hour;  /* часы от полуночи: [0 - 23] */
    int tm_mday;  /* дни от начала месяца: [1 - 31] */
    int tm_mon;   /* месяцы с января: [0 - 11] */
    int tm_year;  /* годы с 1900 года */
    int tm_wday;  /* дни с воскресенья: [0 - 6] */
    int tm_yday;  /* дни от начала года (1 января): [0 - 365] */
```

```
    int tm_isdst; /* флаг перехода на летнее время: <0, 0, >0 */
};
```

Количество секунд может превышать 59, когда для коррекции времени вставляется дополнительная секунда. Обратите внимание, что отсчет всех компонентов, кроме дня месяца, начинается с 0. Флаг перехода на летнее время представлен положительным числом, если действует летнее время, 0 – если нет и отрицательным числом, если данная информация недоступна.

> В предыдущих версиях Single UNIX Specification допускалась вставка двух дополнительных секунд. Таким образом, диапазон значений поля `tm_sec` составлял 0–61. Формальное определение UTC не допускает вставки двух дополнительных секунд, поэтому сейчас диапазон представления секунд определяется как 0–60.

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);

struct tm *localtime(const time_t *calptr);
```

Обе возвращают указатель на структуру `tm`

Функции `localtime` и `gmtime` отличаются тем, что первая преобразует календарное время в местное, учитывая при этом часовой пояс и переход на летнее время, а вторая разбивает календарное время UTC на составляющие.

Функция `mktime` принимает местное время в виде структуры `tm` и преобразует его в значение `time_t`.

```
#include <time.h>

time_t mktime(struct tm *tmptr);
```

Возвращает календарное время в случае успеха, –1 в случае ошибки

Функции `asctime` и `ctime` возвращают строку длиной 26 байт, которая напоминает вывод команды `date`(1):

```
Tue Feb 10 18:27:38 2004\n\0

#include <time.h>

char *asctime(const struct tm *tmptr);

char *ctime(const time_t *calptr);
```

Обе возвращают указатель на строку, завершающуюся нулевым символом

Функции `asctime` в качестве аргумента передается структура `tm`, тогда как функции `ctime` – календарное время.

И наконец, последняя и самая сложная функция времени – `strftime`. Это `printf`-подобная функция для представления временных значений.

```
#include <time.h>

size_t strftime(char *restrict buf, size_t maxsize,
                const char *restrict format,
                const struct tm *restrict tmptr);
```

Возвращает количество символов, записанных в массив, если в нем достаточно места, в противном случае возвращает 0

Последний аргумент функции – указатель на структуру `tm`, содержащую время, которое должно быть представлено в виде отформатированной строки. Результат форматирования сохраняется в буфере *buf*, размер которого определяется аргументом *maxsize*. Если полученная в результате преобразования строка, включая завершающий нулевой символ, умещается в буфере, то функция возвращает длину полученной строки без завершающего нулевого символа. В противном случае возвращается 0.

Аргумент *format* управляет форматированием значения времени. Как и в случае с функцией `printf`, спецификаторы формата начинаются с символа процента, за которым следуют служебные символы. Все остальные символы в строке `format` выводятся без изменений. Два символа процента, следующие друг за другом, будут отображаться как один символ процента. В отличие от функции `printf`, каждый спецификатор формата генерирует на выходе строки фиксированного размера – спецификаторы ширины поля вывода не предусмотрены. В табл. 6.7 перечислены 37 спецификаторов формата, определяемых стандартом ISO C. В третьей колонке таблицы приводится вывод функции `strftime` в ОС Linux, соответствующий времени `Tue Feb 10 18:27:38 EST 2004`.

Таблица 6.7. Спецификаторы формата функции strftime

Спецификатор формата	Описание	Пример
%a	Краткое название дня недели	Tue
%A	Полное название дня недели	Tuesday
%b	Краткое название месяца	Feb
%B	Полное название месяца	February
%c	Дата и время	Tue Feb 10 18:27:38 2004
%C	Две первые цифры года	20
%d	День месяца: [01-31]	10
%D	Дата: [MM/DD/YY]	02/10/04
%e	День месяца (начальный 0 замещается пробелом): [1-31]	10
%F	Дата в формате ISO 8601: [YYYY-MM-DD]	2004-02-10
%g	Последние две цифры года в формате ISO 8601, с учетом номера недели: [00-99]	04

Спецификатор формата	Описание	Пример
%G	Год в формате ISO 8601, с учетом номера недели	2004
%h	То же, что %b	Feb
%H	Час (в 24-часовом формате): [00-23]	18
%I	Час (в 12-часовом формате): [00-12]	06
%j	День года: [001-366]	041
%m	Номер месяца: [01-12]	02
%M	Минуты: [00-59]	27
%n	Символ перевода строки	
%p	AM или PM (до или после полудня)	PM
%r	Местное время в 12-часовом формате	06:27:38 PM
%R	То же, что %H:%M	18:27
%S	Секунды: [00-60]	38
%t	Символ горизонтальной табуляции	
%T	То же, что %H:%M:%S	18:27:38
%u	Номер дня недели в формате ISO 8601 [понедельник=1, 1-7]	2
%U	Номер недели в году, воскресенье – первый день недели: [01-53]	06
%V	Номер недели в году в формате ISO 8601: [01-53]	07
%w	Номер дня недели [воскресенье=0, 0-6]	2
%W	Номер недели в году, понедельник – первый день недели: [00-53]	06
%x	Дата	02/10/04
%X	Время	18:27:38
%y	Последние две цифры года: [00-99]	04
%Y	Год	2004
%z	Разница между поясным временем и UTC	-0500
%Z	Название часового пояса	EST
%%	Символ процента	%

Единственные спецификаторы, смысл которых не очевиден: %U, %V и %W. Спецификатор %U представляет номер недели в году, начиная с недели, на которую выпадает первое воскресенье года. Спецификатор %W представляет номер недели в году, начиная с недели, на которую выпадает первый понедельник года. Действие спецификатора %V зависит от конкретного года. Если неделя, на которую выпадает 1 января, содержит 4 или более дней нового года,

то она считается первой неделей года, в противном случае – последней неделей предыдущего года. В обоих случаях первым днем недели считается понедельник.

Как и `printf`, функция `strftime` поддерживает модификаторы для некоторых спецификаторов формата. Модификаторы `O` и `E` можно использовать для генерации строки в альтернативном формате, если таковой поддерживается языковыми настройками системы.

> Некоторые системы поддерживают дополнительные, нестандартные спецификаторы формата для функции `strftime`.

Мы уже упоминали, что четыре функции, которые на рис. 6.1 обозначены пунктирными линиями, учитывают значение переменной окружения `TZ` – это функции `localtime`, `mktime`, `ctime` и `strftime`. Если эта переменная определена, то ее значение используется вместо значения часового пояса по умолчанию. Если переменная определена как пустая строка, например как `TZ=`, то обычно в качестве часового пояса используется UTC. Эта переменная часто содержит нечто вроде `TZ=EST5EDT`, но стандарт POSIX.1 допускает указание более детальной информации. За дополнительной информацией о переменной окружения `TZ` обращайтесь к главе «Environment Variables» стандарта Single UNIX Specification [Open Group 2004].

> Все описанные в этом разделе функции даты и времени, за исключением функции `gettimeofday`, определены стандартом ISO C. Однако переменная окружения `TZ` добавлена стандартом POSIX.1. В системах FreeBSD 5.2.1, Linux 2.4.22 и Mac OS X 10.3 дополнительную информацию об этой переменной можно найти на странице справочного руководства `tzset`(3). В Solaris 9 – на странице справочного руководства `environ`(5).

6.11. Подведение итогов

Все системы UNIX используют файл паролей и файл групп. Мы рассмотрели различные функции для работы с этими файлами. Мы также обсудили теневые файлы паролей, использование которых повышает уровень безопасности системы. Дополнительные группы предоставляют способ включения пользователя сразу в несколько групп. Кроме того, мы рассмотрели ряд функций, предусмотренных большинством систем для работы с другими файлами данных. Мы обсудили определяемые стандартом POSIX.1 функции, которые могут использоваться приложениями для идентификации системы, в которой они запущены. Глава заканчивается обзором функций, предназначенных для работы с датой и временем и определяемых стандартами ISO C и Single UNIX Specification.

Упражнения

6.1. Представьте, что система использует теневой файл паролей и вам необходимо получить пароль в зашифрованном виде. Как это можно сделать?

6.2. Реализуйте предыдущее упражнение в виде функции, исходя из предположения, что в системе имеется теневой файл паролей и вы обладаете привилегиями суперпользователя.

6.3. Напишите программу, которая вызывает функцию `uname` и выводит содержимое всех полей структуры `utsname`. Сравните получившиеся результаты с тем, что выводит команда `uname`(1).

6.4. Вычислите максимально возможное значение времени, которое может быть представлено с помощью типа `time_t`. Что произойдет, когда счетчик времени переполнится?

6.5. Напишите программу, которая будет получать текущее время и выводить его с помощью функции `strftime` так, чтобы результат выглядел так же, как вывод команды `date`(1) по умолчанию. Присвойте переменной окружения `TZ` другое значение и проверьте, что произойдет.

7

Среда окружения процесса

7.1. Введение

Прежде чем перейти к функциям управления процессами, которые будут обсуждаться в следующей главе, мы должны исследовать среду окружения отдельного процесса. В этой главе мы рассмотрим, как вызывается функция `main` при запуске программы, как программе передаются аргументы командной строки, как выглядит типичная раскладка памяти, как распределяется дополнительная память, как процесс может использовать переменные окружения и как завершается работа процесса. Дополнительно мы исследуем функции `longjmp` и `setjmp` и их взаимодействие со стеком. И наконец, рассмотрим ограничения на ресурсы процесса.

7.2. Функция main

Программы на языке C начинают свою работу с вызова функции `main`. Прототип этой функции:

```
int main(int argc, char *argv[]);
```

где *argc* – это количество аргументов командной строки, а *argv* – массив указателей на аргументы. Мы подробно рассмотрим эти аргументы в разделе 7.4.

Когда ядро запускает программу на языке C (с помощью одной из функций семейства `exec`, которые будут описаны в разделе 8.10), перед вызовом функции `main` выполняется специальная процедура начального запуска. Адрес этой процедуры указывается в исполняемом файле программы как точка входа. Этот адрес определяется редактором связей, который вызывается компилятором языка C. Процедура начального запуска принимает от ядра аргументы командной строки и значения переменных окружения, после чего выполняет обращение к функции `main`.

7.3. Завершение работы процесса

Существует восемь способов завершения работы процесса. Нормальными считаются пять из них:

1. Возврат из функции `main`.
2. Вызов функции `exit`.
3. Вызов функции `_exit` или `_Exit`.
4. Возврат из функции запуска последнего потока (раздел 11.5).
5. Вызов функции `pthread_exit` (раздел 11.5) из последнего потока.

Ненормальное завершение процесса происходит в следующих случаях:

6. При вызове функции `abort` (раздел 10.17).
7. При получении сигнала (раздел 10.2).
8. По запросу на завершение последнего потока (разделы 11.5 и 12.7).

> Мы не рассматриваем способы завершения процесса, связанные с потоками, до глав 11 и 12, где обсуждаются потоки.

Процедура начального запуска, о которой мы говорили в предыдущем разделе, спроектирована так, что она вызывает функцию `exit`, когда происходит возврат из функции `main`. Если процедура начального запуска написана на C (хотя чаще всего она написана на языке ассемблера), то запуск функции `main` выглядит примерно так:

```
exit(main(argc, argv));
```

Функции семейства exit

Нормальное завершение программы осуществляется тремя функциями: `_exit` и `_Exit`, которые сразу же возвращают управление ядру, и `exit`, которая производит ряд дополнительных операций и только после этого возвращает управление ядру.

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

В разделе 8.5 мы рассмотрим влияние этих функций на другие процессы, например, на родителя и потомков завершаемого процесса.

> Прототипы функций определены в разных заголовочных файлах по той причине, что функции `_exit` и `_Exit` определены стандартом ISO C, тогда как функция `exit` – стандартом POSIX.1.

Функция exit всегда выполняла корректное закрытие стандартной библиотеки ввода-вывода, вызывая функцию fclose для всех открытых потоков. В разделе 5.5 мы уже говорили, что это приводит к сбросу всех буферов (то есть к записи их содержимого на диск).

Все три функции завершения принимают единственный аргумент целочисленного типа, который называется *кодом завершения*. Большинство командных оболочек UNIX позволяют узнать код завершения процесса. Код завершения процесса считается неопределенным, если (а) любая из этих функций вызвана без указания кода завершения процесса, (б) функция main вызывает оператор return без аргумента или (в) функция main объявлена как не возвращающая целочисленное значение. Однако если функция main, объявленная как возвращающая целочисленное значение, «неожиданно завершается» (неявный возврат из функции), то код завершения процесса считается равным 0.

> Такое поведение было определено стандартом ISO C достаточно недавно – в 1999 году. Исторически код завершения считался неопределенным, если выход из функции main осуществлялся не через явное обращение к оператору return или к функции exit.

Возврат целочисленного значения из функции main эквивалентен вызову функции exit с тем же самым значением. Таким образом,

```
exit(0);
```

означает то же самое, что

```
return(0);
```

из функции main.

Пример

В листинге 7.1 представлен пример классической программы «Привет, МИР!»

Листинг 7.1. Классический пример программы на языке C

```
#include <stdio.h>

main()
{
    printf("Привет, МИР!\n");
}
```

Если скомпилировать и запустить эту программу, то окажется, что она возвращает случайный код завершения. Если мы скомпилируем программу в разных системах, то, скорее всего, получим различные коды завершения в зависимости от содержимого стека и регистров процессора в момент выхода из функции main:

```
$ cc hello.c
$ ./a.out
Привет, МИР!
$ echo $?            вывести код завершения
13
```

Теперь, если мы включим режим совместимости компилятора с расширением 1999 ISO C, то увидим, что код завершения изменился:

```
$ cc -std=c99 hello.c          включить расширения 1999 ISO C компилятора gcc
hello.c:4: warning: return type defaults to `intґ
$ ./a.out
Привет, МИР!
$ echo $?                      вывести код завершения
0
```

Обратите внимание на предупреждение компилятора, которое появилось при включенном режиме совместимости с расширением 1999 ISO C. Оно появилось потому, что функция `main` не объявлена явно как возвращающая целочисленное значение. Если добавить это объявление, то предупреждение выводиться не будет. Однако, если включить вывод всех предупреждений компилятора (с помощью флага `-Wall`), то появится еще одно предупреждение, примерно такого содержания: «control reaches end of nonvoid function» («достигнут конец функции, имеющей возвращаемое значение»).

Объявление функции `main` как возвращающей целочисленное значение и использование функции `exit` вместо оператора `return` приводит к появлению бесполезных предупреждений от некоторых компиляторов и от программы `lint`(1). Эти компиляторы не понимают, что выход из `main` с помощью функции `exit` по сути то же самое, что и обращение к оператору `return`. Один из способов избавиться от таких предупреждений, которые через некоторое время становятся раздражающими, заключается в использовании оператора `return` вместо вызова функции `exit`. Но, сделав это, мы лишимся возможности легко и просто находить все точки выхода из программы с помощью утилиты `grep`. Другое возможное решение – объявить функцию `main` с типом возвращаемого значения `void` вместо `int` и продолжать использовать функцию `exit`. Это избавляет нас от надоедливых предупреждений компилятора, но выглядит не очень правильно (особенно в исходных текстах программы) и, кроме того, может вызвать появление других предупреждений, поскольку предполагается, что функция `main` должна возвращать целочисленное значение. В этой книге мы будем определять функцию `main` как возвращающую целочисленное значение, поскольку это определено стандартами ISO C и POSIX.1.

В зависимости от компилятора выводимые предупреждения могут быть более или менее подробными. Обратите внимание: компилятор GNU C обычно не выдает эти ненужные сообщения, если не используются дополнительные опции, управляющие выводом предупреждений.

В следующей главе мы увидим, как один процесс может запустить другой процесс, дождаться его завершения и получить код завершения.

Функция atexit

В соответствии со стандартом ISO C процесс может зарегистрировать до 32 функций, которые будут автоматически вызываться функцией `exit`. Они называются *обработчиками выхода* и регистрируются с помощью функции `atexit`.

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Возвращает 0 в случае успеха, ненулевое значение – в случае ошибки

Это объявление говорит о том, что функции `atexit` в качестве аргумента передается адрес функции обработчика. Функции-обработчику при вызове не передается никаких аргументов, и от нее не ожидается возврата значения. Функция `exit` вызывает обработчики в последовательности, обратной порядку их регистрации. Каждый обработчик вызывается столько раз, сколько он был зарегистрирован.

> Обработчики выхода впервые появились в стандарте ANSI C в 1989 году. Системы, предшествовавшие этому стандарту, такие как SVR3 и BSD4.3, не предоставляли функций обработки выхода.
>
> Стандарт ISO C требует, чтобы системы поддерживали возможность регистрации как минимум 32 обработчиков. Максимально возможное количество обработчиков для заданной системы можно определить с помощью функции `sysconf` (табл. 2.10).

В соответствии со стандартами ISO C и POSIX.1, функция `exit` сначала должна вызвать все зарегистрированные функции-обработчики и затем закрыть все открытые потоки (с помощью функции `fclose`). Стандарт POSIX.1 расширил положения ISO C, указав, что регистрация всех функций-обработчиков аннулируется, если процесс вызывает одну из функций семейства `exec`. На рис. 7.1 показано, как запускается и завершается программа, написанная на языке C.

Обратите внимание: ядро может запустить программу единственным способом – через вызов одной из функций семейства `exec`. Процесс может добро-

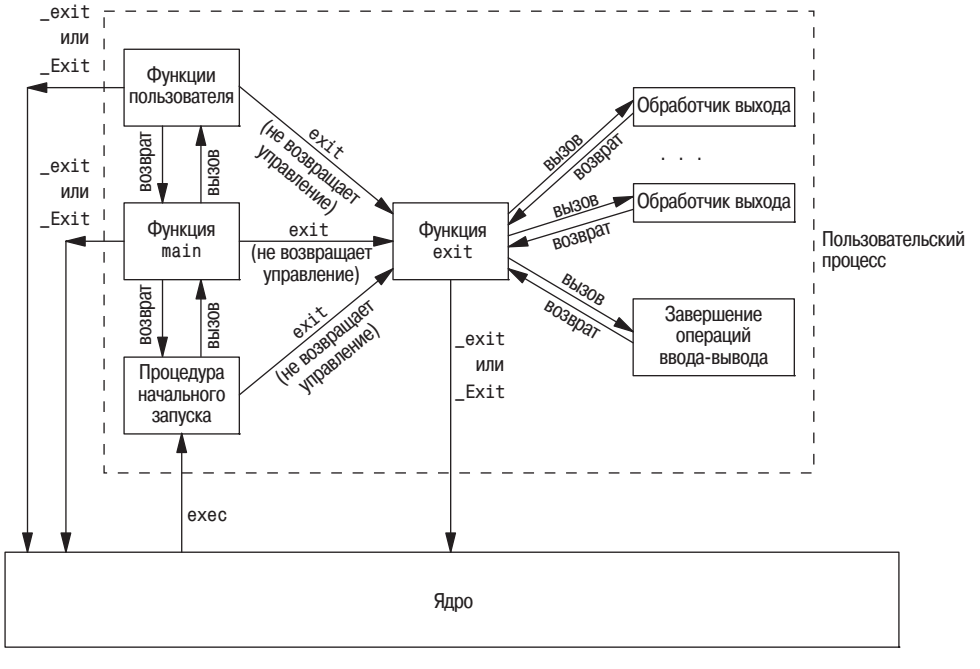


***Рис. 7.1.** Процесс запуска и завершения программы, написанной на языке C*

вольно завершить свою работу только через вызов функции _exit или _Exit, явный или неявный (с помощью функции exit). Процесс может также непреднамеренно прекратить работу по сигналу (что не показано на рис. 7.1).

Пример

Программа, представленная листингом 7.2, демонстрирует использование функции atexit.

Листинг 7.2. Пример использования обработчиков выхода

```
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("невозможно зарегистрировать my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("невозможно зарегистрировать my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("невозможно зарегистрировать my_exit1");

    printf("функция main завершила работу\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("первый обработчик выхода\n");
}

static void
my_exit2(void)
{
    printf("второй обработчик выхода\n");
}
```

У нас эта программа дала следующие результаты:

```
$ ./a.out
функция main завершила работу
первый обработчик выхода
первый обработчик выхода
второй обработчик выхода
```

Обработчик выхода вызывается столько раз, сколько он был зарегистрирован. В программе из листинга 7.2 первый обработчик был зарегистрирован дважды, поэтому он был вызван два раза. Обратите внимание, что вызов функции exit не используется, вместо этого выполняется оператор return.

7.4. Аргументы командной строки

Процесс, запускающий программу при помощи функции exec, может передать ей аргументы командной строки. Это обычная практика для командных оболочек UNIX, как мы уже видели на многочисленных примерах из предыдущих глав.

Пример

Программа, представленная листингом 7.3, выводит все аргументы командной строки на стандартный вывод. Обратите внимание: стандартная утилита echo(1) не выводит нулевой аргумент.

Листинг 7.3. Вывод всех аргументов командной строки

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int i;

    for (i = 0; i < argc; i++) /* вывести все аргументы командной строки */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

Если скомпилировать эту программу и дать исполняемому файлу имя echoarg, то результат будет таков:

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

Согласно стандартам ISO C и POSIX.1, элемент массива argv[argc] должен быть представлен пустым указателем. Учитывая это обстоятельство, цикл обработки аргументов командной строки можно оформить несколько иначе:

```
for (i = 0; argv[i] != NULL; i++)
```

7.5. Список переменных окружения

Каждой программе, помимо аргументов командной строки, передается также *список переменных окружения*. Подобно списку аргументов командной строки, список переменных окружения представляет собой массив указателей, каждый из которых указывает на строку символов, завершающуюся нулевым символом. Адрес массива указателей хранится в глобальной переменной environ:

```
extern char **environ;
```

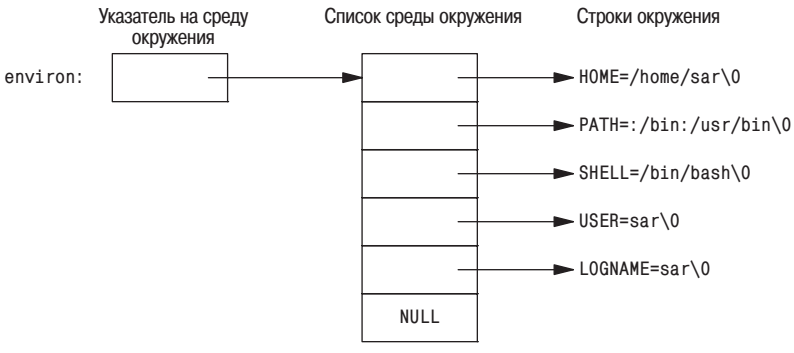


***Рис. 7.2.** Среда окружения, состоящая из пяти строк*

Например, среда окружения, которая насчитывает пять переменных, будет похожа на то, что изображено на рис. 7.2. Здесь явно показаны нулевые символы, которыми завершаются строки. Переменная `environ` отмечена как *указатель на среду окружения*, массив указателей – как *список среды окружения*, а строки, на которые они указывают, как *строки окружения*.

В соответствии с принятыми соглашениями окружение состоит из строк формата (рис. 7.2):

```
name=value
```

Большинство предопределенных имен состоят из символов верхнего регистра, но это всего лишь традиция.

Исторически в большинстве версий UNIX функция `main` имеет третий аргумент, который является указателем на среду окружения:

```
int main(int argc, char *argv[], char *envp[]);
```

Стандарт ISO C определяет только два аргумента функции `main`, а передача среды окружения через третий аргумент не имеет никаких преимуществ перед передачей той же информации через глобальную переменную `environ`. Поэтому стандарт POSIX.1 указывает, что передача среды окружения должна осуществляться не через третий аргумент функции `main`, а через глобальную переменную `environ` (если это возможно). Доступ к конкретным переменным окружения обычно осуществляется с помощью функций `getenv` и `putenv`, которые будут описаны в разделе 7.9, а не через глобальную переменную `environ`. Однако для просмотра всех переменных окружения следует использовать указатель `environ`.

7.6. Раскладка памяти программы на языке C

Традиционно программы на языке C всегда состояли из следующих частей:

- Сегмент кода, машинные инструкции, которые выполняются центральным процессором. Обычно сегмент кода является разделяемым, чтобы

в памяти располагалась только одна копия сегмента для часто используемых программ, таких как текстовые редакторы, компиляторы языка C, командные оболочки и некоторые другие. Кроме того, сегмент кода зачастую доступен только для чтения, чтобы предотвратить возможность случайного изменения расположенных в нем инструкций.

- Сегмент инициализированных данных, который обычно называют просто сегментом данных. Он содержит переменные, которые инициализированы определенными значениями в тексте программы. Например, если объявление

```
int maxcount = 99;
```

расположено где-либо за пределами функции, то указанная переменная будет сохранена вместе со своим значением в сегменте инициализированных данных.

- Сегмент неинициализированных данных, часто называемый сегментом «bss». Это название происходит от древнего оператора языка ассемблера, который расшифровывается как «block started by symbol» (блок, начинающийся с символа). Ядро инициализирует данные в этом сегменте арифметическим нулем или нулевыми указателями перед запуском программы. Если объявление

```
long sum[1000];
```

расположено за пределами функции, то указанная переменная будет сохранена вместе со своим значением в сегменте неинициализированных данных.

- Сегмент стека (stack), где хранятся переменные с автоматическим классом размещения, а также информация, которая сохраняется при каждом вызове функции. При каждом вызове функции в стеке сохраняется адрес возврата из функции и определенная информация о среде окружения вызывающей программы, например регистры процессора. После этого вызванная функция резервирует на стеке дополнительное место для автоматических и временных переменных. Благодаря такой организации в языке C возможны рекурсивные вызовы функций. Всякий раз, когда функция рекурсивно вызывает сама себя, создается новый фрейм стека, благодаря чему один набор локальных переменных не накладывается на другой.
- Куча (heap), или область динамической памяти. Традиционно куча располагалась в пространстве между сегментом неинициализированных данных и стеком.

На рис. 7.3 показано типичное размещение этих сегментов. Это логическое представление того, как выглядит программа; в конкретной системе раскладка памяти программы не обязательно будет выглядеть именно так. Тем не менее этот рисунок показывает типичный пример раскладки памяти, которую мы будем обсуждать. В ОС Linux на микропроцессорах Intel x86 сегмент кода начинается с адреса `0x8048000`, а дно стека расположено ниже адреса `0xC0000000` (на данной аппаратной архитектуре стек растет вниз – от стар-

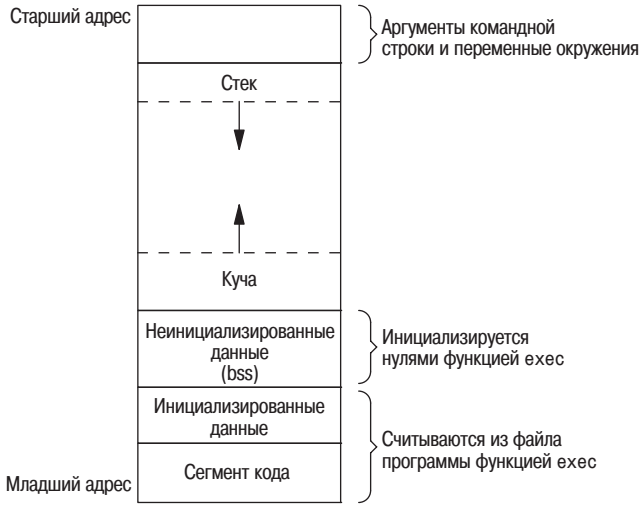


***Рис. 7.3.** Типичное размещение сегментов в памяти*

ших адресов к младшим). Неиспользуемое виртуальное адресное пространство между вершиной кучи и вершиной стека очень велико.

> В исполняемом файле `a.out` существует несколько сегментов дополнительных типов, которые содержат таблицу идентификаторов, информацию для отладчика, таблицы связи с динамическими разделяемыми библиотеками и тому подобное. Эти дополнительные сегменты не загружаются как часть образа программы, исполняемой процессом.

Обратите внимание, что сегмент неинициализированных данных на рис. 7.3 не хранится в файле программы на диске. По этой причине ядро обнуляет этот сегмент, прежде чем запустить программу. Единственные сегменты, которые должны быть сохранены в файле программы, – это сегмент кода и сегмент инициализированных данных.

Команда `size`(1) выводит размеры (в байтах) сегментов кода, данных и bss. Например:

```
$ size /usr/bin/cc /bin/sh
   text    data     bss     dec     hex filename
  79606    1536     916   82058   1408a /usr/bin/cc
 619234   21120   18260  658614   a0cb6 /bin/sh
```

В четвертой и пятой колонках выводится общий размер всех трех сегментов в десятичном и шестнадцатеричном представлении соответственно.

7.7. Разделяемые библиотеки

Большинство современных версий UNIX поддерживают разделяемые библиотеки. В [Arnold 1986] описана ранняя реализация разделяемых библиотек в System V, в [Gingel et al. 1987] – реализация в SunOS. Разделяемые

библиотеки позволяют изъять из исполняемого файла библиотечные функции; в результате в памяти системы хранится единственная копия библиотеки, к которой обращаются все процессы. Это заметно уменьшает размер исполняемых файлов, но может несколько увеличить нагрузку, когда приложение запускается в первый раз или когда происходит первое обращение к библиотечной функции. Еще одно преимущество разделяемых библиотек заключается в том, что при обновлении библиотеки не требуется исправлять связи с библиотекой в каждой программе, которая использует эту библиотеку. (Здесь мы исходим из предположения, что количество и типы аргументов библиотечных функций не изменились.)

Разные системы предоставляют различные способы, чтобы указать, использовать ли динамические библиотеки в программе или нет. Опции для команд `cc`(1) и `ld`(1) типичны. Чтобы продемонстрировать различия в размерах, попробуем собрать исполняемый файл – классическую программу `hello.c` – сначала без разделяемых библиотек:

```
$ cc -static hello1.c       укажем явно, что разделяемые библиотеки
                            не должны использоваться
$ ls -l a.out
-rwxrwxr-x 1 sar        475570 Feb 18 23:17 a.out
$ size a.out
   text    data     bss     dec     hex    filename
 375657    3780    3220  382657   5d6c1    a.out
```

Если теперь скомпилировать программу с использованием разделяемых библиотек, то размеры сегмента кода и данных существенно уменьшатся:

```
$ cc hello1.c          по умолчанию gcc использует разделяемые библиотеки
$ ls -l a.out
-rwxrwxr-x 1 sar 11410 Feb 18 23:19 a.out
$ size a.out
   text    data     bss     dec     hex    filename
    872     256       4    1132    46c    a.out
```

7.8. Распределение памяти

Стандарт ISO C определяет три функции, с помощью которых производится распределение памяти:

1. Функция `malloc` выделяет заданное количество байт памяти. Выделенный объем памяти не очищается.
2. Функция `calloc` выделяет пространство для заданного количества объектов определенного размера. Выделенный объем памяти заполняется нулевыми байтами.
3. Функция `realloc` перераспределяет выделенную ранее память, увеличивая или уменьшая ее объем. Увеличение выделенного ранее объема может сопровождаться перемещением участка памяти в новое место. Кроме того, участок памяти, который оказывается между концом ранее выделенного блока и новым концом, не инициализируется.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
                                    Все три возвращают непустой указатель
                                      в случае успеха, NULL в случае ошибки
void free(void *ptr);
```

Эти три функции гарантируют возврат указателей, которые будут иметь выравнивание, подходящее для сохранения любого объекта данных. Например, если ограничение для конкретной системы состоит в том, что объекты типа `double` должны размещаться в адресах, кратных числу 8, то все указатели, возвращаемые этими функциями, будут содержать адреса, кратные 8.

Все три функции возвращают нетипизированный указатель `void*`, поэтому, подключая к программе заголовочный файл `<stdlib.h>` (где находятся прототипы функций), мы не должны выполнять явное приведение типов при присваивании значений функций типизированным указателям.

Функция `free` освобождает выделенную ранее память, на которую указывает аргумент *ptr*. Освобожденное пространство, как правило, помещается в пул свободной памяти и может быть снова распределено при последующем обращении к одной из трех функций.

Функция `realloc` позволяет увеличивать или уменьшать размер ранее выделенной области памяти. (Наиболее часто производится увеличение размера области.) Например, если мы выделяем участок памяти для массива из 512 элементов и затем в процессе его заполнения вдруг обнаруживаем, что нам потребуется память для хранения более 512 элементов, то мы можем вызвать функцию `realloc`. В этом случае, если позади выделенной ранее области имеется блок свободной памяти достаточного объема, функция `realloc` ничего никуда не перемещает, она просто добавляет область требуемого объема в конец и возвращает тот же самый указатель, который был ей передан. Однако, если позади выделенной ранее области нет свободного участка памяти, функция `realloc` выделяет другую область памяти достаточного объема и копирует существующий массив из 512 элементов в новое место, после чего освобождает старую область памяти и возвращает указатель на новую область. Поскольку ранее выделенный объем памяти может перемещаться, мы не должны использовать другие указатели на эту область. Упражнение 4.16 показывает, как можно использовать функцию `realloc` совместно с `getcwd`, чтобы работать с именами файлов любой длины. В листинге 17.28 приводится пример использования функции `realloc` для организации динамических массивов, чтобы не указывать их размер во время компиляции.

Обратите внимание, что последний аргумент функции `realloc` определяет новый размер требуемой области, а не разницу между новым и старым размерами. В особом случае, когда в аргументе *ptr* передается пустой указатель, `realloc` ведет себя как функция `malloc` и выделяет область размером *newsize*.

> Ранние версии этих функций позволяли снова получить с помощью функции realloc блок, освобожденный функцией free после последнего обращения к функциям malloc, realloc или calloc. Эта хитрость существовала еще в Version 7 и использовала свойство стратегии поиска, реализованной в функции malloc для уплотнения памяти. В ОС Solaris эта особенность сохранилась и поныне, но в других системах – нет. Она не документирована и не должна использоваться.

Функции распределения памяти обычно реализуются на основе системного вызова sbrk(2). Этот системный вызов расширяет (или усекает) область динамической памяти (кучи) процесса (рис. 7.3). Пример типичной реализации функций malloc и free приводится в разделе 8.7 [Kernighan and Ritchie 1988].

Хотя системный вызов sbrk(2) может как увеличивать, так и уменьшать объем памяти процесса, большинство версий malloc и free никогда не уменьшают его. Освобождаемое пространство становится доступным для последующего распределения и, как правило, не возвращается ядру – оно помещается в пул свободной памяти функции malloc.

Важно понимать, что большинство реализаций выделяют несколько больший объем памяти, чем требуется, и используют дополнительное пространство для хранения служебной информации: размера распределенного блока, указателя на следующий распределенный блок и тому подобного. Это означает, что запись за пределы выделенной области может уничтожить служебную информацию в следующем блоке. Подобного рода ошибки часто носят катастрофический характер и найти их чрезвычайно трудно, потому что они могут не проявлять себя достаточно длительное время. Кроме того, существует возможность уничтожить служебную информацию блока памяти, если записать данные перед началом распределенной области.

Запись за пределы выделенного блока памяти может уничтожить не только служебную информацию. Память до и после такого блока может использоваться для хранения других динамических объектов. Эти объекты могут быть не связаны с разрушающим их участком программы, что еще больше осложняет поиск источника повреждений.

Другие возможные ошибки, которые могут оказаться фатальными – попытка освобождения блока памяти, уже освобожденного ранее, и передача функции free указателя, который не был получен от одной из трех функций распределения памяти. Если процесс вызывает функцию malloc, но забывает вызвать функцию free, то объем используемой памяти начинает непрерывно увеличиваться; это называют утечкой памяти. Если процесс не будет возвращать ставшие ненужными блоки памяти с помощью функции free, объем адресного пространства, занимаемого процессом, будет медленно увеличиваться до тех пор, пока свободное пространство не закончится. Это может привести к снижению производительности системы из-за лишних обращений к файлу подкачки.

Поскольку очень сложно отыскать ошибки, связанные с распределением памяти, некоторые системы предоставляют версии функций распределения памяти, выполняющие дополнительную проверку наличия ошибок при каждом вызове. Эти версии функций часто характеризуются включением спе-

циальной библиотеки редактора связей. Кроме того, существуют общедоступные исходные тексты, которые можно скомпилировать со специальными флагами, разрешающими проведение дополнительных проверок во время выполнения.

> Операционные системы FreeBSD, Mac OS X и Linux поддерживают дополнительные возможности отладки через установку переменных среды. Кроме того, библиотеке FreeBSD можно передать дополнительные параметры через символическую ссылку `/etc/malloc.conf`.

Альтернативные функции распределения памяти

Существует большое количество функций, которые могут служить заменой для `malloc` и `free`. Некоторые системы уже включают библиотеки, предоставляющие альтернативные реализации функций распределения памяти. Другие системы предоставляют только стандартные функции, оставляя программистам право скачивать и использовать альтернативные библиотеки, если они того пожелают. Здесь мы упомянем некоторые из альтернатив.

libmalloc

Операционные системы, основанные на SVR4, такие как Solaris, включают библиотеку `libmalloc`, которая предоставляет ряд интерфейсов, соответствующих функциям распределения памяти стандарта ISO C. Библиотека `libmalloc` включает в себя функцию `mallopt`, позволяющую процессу устанавливать определенные переменные, которые контролируют действия функций распределения памяти. Кроме того, в библиотеке имеется функция `mallinfo`, с помощью которой можно получить статистику по функциям распределения памяти.

vmalloc

В [Vo 1996] описывается библиотека функций распределения памяти, которая позволяет использовать различные приемы для различных областей памяти. В дополнение к специфичным функциям, библиотека `vmalloc` предоставляет функции, эмулирующие стандартные функции распределения памяти стандарта ISO C.

quick-fit

Традиционно в качестве стандартного алгоритма выделения памяти используется либо метод наилучшего приближения (best-fit), либо метод первого подходящего (first-fit). Алгоритм quick-fit (быстрого приближения) превосходит по скорости любой из них, но использует больше памяти. В [Weinstock and Wulf 1988] описывается этот алгоритм, в основе которого лежит принцип разделения памяти на блоки различных размеров и размещения их в различных списках свободных блоков в зависимости от размера. Свободные реализации функций `free` и `malloc` на основе алгоритма quick-fit доступны на различных FTP-серверах.

Функция alloca

Это еще одна функция, которая заслуживает внимания. Функция `alloca` вызывается точно так же, как функция `malloc`, однако вместо того чтобы распределять память из кучи, она выделяет память во фрейме стека текущей функции. Преимущество такого выделения памяти состоит в том, что нет необходимости освобождать выделенное пространство – это происходит автоматически после выхода из функции. Функция `alloca` увеличивает размер фрейма стека. Недостаток этой функции состоит в том, что она не может использоваться в системах, в которых невозможно увеличить фрейм стека после вызова функции. Тем не менее она используется во многих программных пакетах, и существуют ее реализации для большого количества систем.

Все четыре платформы, обсуждаемые в этой книге, предоставляют функцию `alloca`.

7.9. Переменные окружения

Как мы уже говорили ранее, строка окружения обычно записывается в формате

```
name=value
```

Ядро UNIX никогда не обращается к этим строкам; их интерпретация полностью зависит от самих приложений. Так, например, командные оболочки используют в своей работе многочисленные переменные окружения. Некоторые из них, такие как `HOME` и `USER`, устанавливаются автоматически при входе в систему, другие определяются пользователем. Обычно инициализация переменных окружения производится в файле начального запуска командной оболочки. Если, например, установить переменную среды окружения `MAILPATH`, то она будет сообщать командным оболочкам Bourne shell, GNU Bourne-again shell и Korn shell имя каталога, в котором находится электронная почта.

Стандарт ISO C определяет функцию, с помощью которой можно получить значение любой переменной окружения, но оговаривает, что содержимое среды окружения зависит от реализации.

```
#include <stdlib.h>

char *getenv(const char *name);
```

Возвращает указатель на значение переменной с именем *name* или `NULL`, если переменная не найдена

Обратите внимание, что эта функция возвращает указатель на подстроку *value* в строке *name=value*. Когда нужно получить значение конкретной переменной окружения, всегда следует использовать функцию `getenv` вместо прямого обращения к массиву `environ`.

Некоторые переменные окружения в Single UNIX Specification определяются стандартом POSIX.1, тогда как другие определены только в системах, ко-

торые поддерживают расширения XSI. В табл. 7.1 перечислены переменные окружения, определяемые в Single UNIX Specification, а также отмечено, какими реализациями они поддерживаются. Переменные окружения, определяемые стандартом POSIX.1, отмечены точкой, остальные являются расширениями XSI. В четырех реализациях, обсуждаемых в данной книге, поддерживается много дополнительных переменных окружения. Обратите внимание, что стандарт ISO C не определяет никаких переменных окружения.

Таблица 7.1. Переменные окружения, определяемые стандартом Single UNIX Specification

Переменная	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Описание
COLUMNS	•	•	•	•	•	Ширина строки терминала
DATEMSK	XSI		•		•	Полный путь к файлу шаблона для функции getdate(3)
HOME	•	•	•	•	•	Домашний каталог
LANG	•	•	•	•	•	Название локали (языковых настроек)
LC_ALL	•	•	•	•	•	Название локали (языковых настроек)
LC_COLLATE	•	•	•	•	•	Название локали (языковых настроек) для выполнения сравнения
LC_CTYPE	•	•	•	•	•	Название локали (языковых настроек) для классификации символов языка
LC_MESSAGES	•	•	•	•	•	Название локали (языковых настроек) для вывода сообщений
LC_MONETARY	•	•	•	•	•	Название локали (языковых настроек) для представления денежных величин
LC_NUMERIC	•	•	•	•	•	Название локали (языковых настроек) для представления чисел
LC_TIME	•	•	•	•	•	Название локали (языковых настроек) для форматирования даты и времени
LINES	•	•	•	•	•	Количество строк терминала
LOGNAME	•	•	•	•	•	Имя пользователя
MSGVERB	XSI	•			•	Определяет компонент сообщения, который будет выводиться функцией fmtmsg(3)

Таблица 7.1 (продолжение)

Переменная	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Описание
NLSPATH	XSI	•	•	•	•	Шаблон имени каталога с сообщениями
PATH	•	•	•	•	•	Список каталогов, в которых будет выполняться поиск исполняемых файлов
PWD	•	•	•	•	•	Абсолютный путь к текущему каталогу
SHELL	•	•	•	•	•	Имя командной оболочки, предпочитаемой пользователем
TERM	•	•	•	•	•	Тип терминала
TMPDIR	•	•	•	•	•	Путь к каталогу для создания временных файлов
TZ	•	•	•	•	•	Информация о часовом поясе

Иногда может потребоваться не только получить, но и изменить значение существующей переменной или даже добавить новую. (В следующей главе мы рассмотрим, как можно оказывать влияние на среду окружения текущего и любых порожденных им процессов. Мы не можем изменить среду окружения родительского процесса, который зачастую является командной оболочкой. Тем не менее было бы удобно иметь возможность изменять среду окружения текущего процесса.) К сожалению, не все системы поддерживают эту возможность. В табл. 7.2 приводится список функций, которые поддерживаются различными стандартами и реализациями.

Таблица 7.2. Различные функции для работы со средой окружения

Функция	ISO C	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
getenv	•	•	•	•	•	•
putenv		XSI	•	•	•	•
setenv		•	•	•	•	
unsetenv		•	•	•	•	
clearenv				•		

Функция clearenv не входит в стандарт Single UNIX Specification. Она используется для удаления всех записей из списка строк окружения.

Ниже приводятся прототипы второй, третьей и четвертой функций из табл. 7.2.

```
#include <stdlib.h>

int putenv(char *str);

int setenv(const char *name, const char *value, int rewrite);

int unsetenv(const char *name);
```

Все функции возвращают 0 в случае успеха, ненулевое значение в случае ошибки

Работают эти функции следующим образом.

- Функция putenv принимает строку в формате *name=value* и помещает ее в список переменных окружения. Если переменная с именем *name* уже существует, то она будет удалена перед вставкой новой строки.
- Функция setenv присваивает переменной *name* значение *value*. Если переменная *name* уже существует в среде окружения, тогда: (а) если аргумент *rewrite* не равен нулю, то существующее определение переменной сначала удаляется из списка; (б) если аргумент *rewrite* равен нулю, то существующее определение переменной не удаляется, новое значение *value* не запоминается и функция возвращает управление без признака ошибки.
- Функция unsetenv удаляет определение переменной с именем *name*. Если заданной переменной не существует, то это не считается ошибкой.

> Обратите внимание на различия между функциями putenv и setenv. Функция setenv выделяет память, чтобы создать строку *name=value* из своих аргументов, а putenv просто вставляет переданную ей строку непосредственно в список переменных окружения. Действительно в ОС Linux и Solaris функция putenv помещает адрес строки, который передается ей в качестве аргумента, непосредственно в список переменных окружения. В этом случае было бы ошибкой передавать строки, размещенные в стеке, так как память под стеком еще не раз будет использована после того, как текущая функция вернет управление в вызывающую программу.

Было бы интересно узнать, какие действия выполняются этими функциями при изменении списка переменных окружения. Вспомните рис. 7.3: список переменных окружения – это массив указателей на строки формата *name=value*, и эти строки обычно хранятся в верхней части адресного пространства процесса – над стеком. Операция удаления строки выглядит достаточно просто; мы находим нужный указатель в массиве и перемещаем все последующие указатели на один вниз. Но операция добавления новой строки или изменения существующей оказывается более сложной. Пространство над стеком не может быть раздвинуто, потому что эта область часто находится в старших адресах адресного пространства процесса и не может расти вверх; она также не может расти вниз, потому что ниже находится стек, который не может быть перемещен.

1. Если необходимо изменить значение существующей переменной *name*:
   a. Если длина подстроки *value* меньше или равна существующей, новая подстрока просто копируется поверх существующей.

б. Если длина подстроки *value* больше существующей, то необходимо выделить память для новой строки с помощью функции malloc, скопировать туда новую строку и затем заменить в массиве старый указатель на новый.

2. Если добавляется новая переменная, то это значительно осложняет дело. Прежде всего нужно вызвать функцию malloc, выделить память для строки формата *name=value* и скопировать ее туда.

   а. Затем, если новая переменная добавляется впервые, следует опять вызвать malloc для выделения памяти под новый массив указателей. Далее нужно скопировать список указателей в новое место, добавить к нему указатель на новую строку и, разумеется, поместить в конец списка пустой указатель. Наконец, надо записать в переменную environ адрес нового списка. Обратите внимание: если изначально список переменных окружения размещался выше стека (рис. 7.3), то теперь он переместится в область динамической памяти (в кучу). Однако большинство указателей в этом списке по-прежнему будут указывать на строки, размещенные выше стека.

   б. Если добавление переменной производится не в первый раз, то мы уже знаем, что список указателей располагается в куче. Таким образом, нужно вызвать realloc, чтобы выделить место для еще одного указателя, добавить в список новую строку *name=value* (на место пустого указателя) и затем записать пустой указатель в самый конец списка.

7.10. Функции setjump и longjump

В языке C нельзя выполнить безусловный переход (оператор goto) к метке, которая находится в теле другой функции. В таким случаях необходимо использовать функции setjmp и longjmp. Позднее мы увидим, что эти функции очень удобны для обработки ошибочных ситуаций, когда ошибка происходит в глубоко вложенном вызове функции.

Рассмотрим программу-заготовку из листинга 7.4. Здесь имеется главный цикл, который читает строки со стандартного ввода и вызывает для обработки каждой из них функцию do_line. Эта функция в свою очередь обращается к функции get_token, которая выбирает из входной строки очередную лексему. Предполагается, что первая лексема строки является некоторой командой, и оператор switch определяет, как обрабатывать каждую из команд. Для единственной показанной команды вызывается функция cmd_add.

Программа из листинга 7.4 представляет собой типичную заготовку для программ, которые читают команды, определяют их тип и затем вызывают функции, соответствующие командам. На рис. 7.4 показано, как мог бы выглядеть стек после вызова функции cmd_add.

Листинг 7.4. Типичная заготовка программы обработки команд

```
#include "apue.h"

#define TOK_ADD 5
```

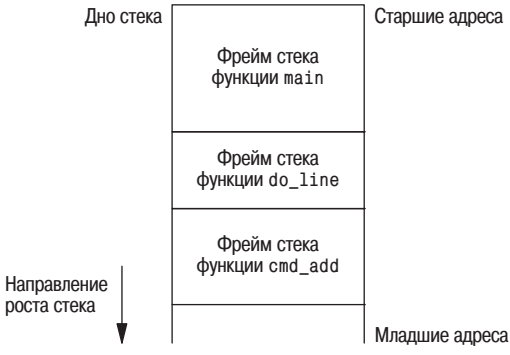


***Рис. 7.4.** Состояние стека после вызова функции cmd_add*

```
void do_line(char *);
void cmd_add(void);
int get_token(void);

int
main(void)
{
    char line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char *tok_ptr;     /* глобальный указатель для get_token() */

void
do_line(char *ptr) /* обработка одной строки ввода */
{
    int cmd;

    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
        switch (cmd) {    /* для каждой команды свой оператор case */
        case TOK_ADD:
            cmd_add();
            break;
        }
    }
}

void
cmd_add(void)
{
    int token;

    token = get_token();
    /* остальные действия по обработке этой команды */
```

```
}

int
get_token(void)
{
    /* получить очередную лексему из строки, на которую указывает tok_ptr */
}
```

Переменные с автоматическим классом размещения хранятся в пределах фреймов стека каждой из функций. Массив `line` хранится во фрейме стека функции `main`, целочисленная переменная `cmd` – во фрейме стека функции `do_line`, целочисленная переменная `token` – во фрейме стека функции `cmd_add`.

Как мы уже говорили, такой тип раскладки стека достаточно типичен, но совсем не обязателен. Стеки не обязательно должны расти в направлении младших адресов памяти. В системах, которые не имеют аппаратной поддержки механизма стека, его реализация на языке C могла бы использовать связанный список фреймов стека.

При разработке программ, подобных представленной в листинге 7.4, часто возникает вопрос, как обрабатывать нефатальные ошибки. Например, если функция `cmd_add` встречает ошибку, скажем недопустимое число, то может потребоваться вывести сообщение об ошибке, проигнорировать остальную часть входной строки и вернуться в функцию `main`, чтобы перейти к обработке следующей строки. Но когда ошибка возникает в глубоко вложенной функции, сделать это на C достаточно трудно. (В этом примере функция `cmd_add` находится на втором уровне вложенности относительно функции `main`, но очень часто точка, из которой мы хотим вернуться, находится на пятом уровне вложенности и даже глубже.) Если в каждую функцию добавлять код, который будет возвращать признак ошибки на один уровень вверх, то исходные тексты станут неудобочитаемыми.

Решение этой проблемы заключается в использовании нелокальных переходов: функций `setjmp` и `longjmp`. Определение «нелокальный» означает, что мы не используем обычный оператор перехода языка C, вместо этого мы выполняем обратный переход через фреймы стека, созданные цепочкой вызовов на пути к текущей функции.

```
#include <setjmp.h>

int setjmp(jmp_buf env);

        Возвращает 0, если вызвана непосредственно, или ненулевое значение,
            если возврат произошел в результате обращения к функции longjmp

void longjmp(jmp_buf env, int val);
```

Функция `setjmp` вызывается из точки, в которую мы хотим возвратиться; в нашем примере она находится в функции `main`. В этом случае `setjmp` возвращает значение 0, потому что это непосредственный вызов функции. Аргумент *env* функции `setjmp` имеет специальный тип `jmp_buf`. Этот тип данных – своего рода массив, который может хранить информацию, необходимую для

восстановления состояния стека, когда будет произведен вызов функции longjmp. Обычно переменная *env* является глобальной, так как она должна быть доступна из других функций.

Когда возникает ошибка, например, в функции cmd_add, мы вызываем longjmp с двумя аргументами. Первый – тот самый *env*, который использовался при обращении к setjmp, а второй – *val*, значение, отличное от нуля, которое становится возвращаемым значением функции setjmp. Второй аргумент позволяет вызывать longjmp более одного раза для каждого setjmp. Например, можно было бы выполнить переход с помощью longjmp из cmd_add со значением аргумента *val*, равным 1, а из get_token – со значением *val*, равным 2. В таком случае setjmp в функции main будет возвращать либо 1, либо 2, что позволит определить, откуда был произведен переход – из cmd_add или из get_token.

Теперь вернемся к нашему примеру. Листинг 7.5 показывает функции cmd_add и main. (Две другие функции, do_line и get_token, остались без изменений.)

Листинг 7.5. Пример использования функций setjmp и longjmp

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD 5

jmp_buf jmpbuffer;

int
main(void)
{
    char line[MAXLINE];
    if (setjmp(jmpbuffer) != 0)
        printf("ошибка");

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

...

void
cmd_add(void)
{
    int token;

    token = get_token();
    if (token < 0) /* проверка наличия ошибки */
        longjmp(jmpbuffer, 1);
    /* остальные действия по обработке этой команды */
}
```

Когда начинается выполнение функции main, функция setjmp записывает всю необходимую информацию в переменную jmpbuffer и возвращает значение 0. Затем вызывается функция do_line, которая в свою очередь вызывает

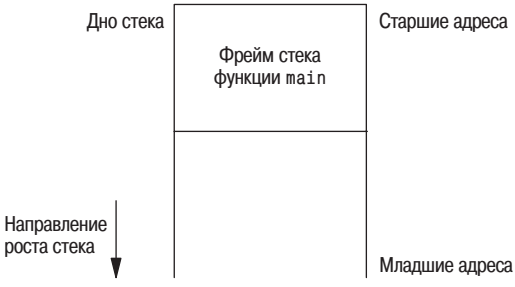


***Рис. 7.5.** Состояние стека после вызова функции longjmp*

функцию cmd_add. Теперь предположим, что была обнаружена некая ошибка. Перед вызовом функции longjmp из cmd_add стек выглядит так, как показано на рис. 7.4. Функция longjmp «раскручивает» стек в обратную сторону – до фрейма функции main, выбрасывая фреймы, созданные во время вызова функций cmd_add и do_line (рис. 7.5). В результате вызова функции longjmp происходит возврат из функции setjmp в main, но на этот раз возвращаемое значение равно 1 (второй параметр longjmp).

Переменные с классами размещения register, automatic и volatile

Мы увидели, как выглядит стек после вызова функции longjmp. Следующий вопрос, на который нам нужно ответить, – каково состояние автоматических и регистровых переменных в функции main? Когда в результате вызова функции longjmp управление возвращается в функцию main, получают ли эти переменные значения, которые они имели на момент вызова функции setjmp (то есть «откручиваются» ли их значения назад), или их значения остаются без изменения с момента вызова функции do_line (вызвавшей функцию cmd_add, которая в свою очередь вызвала функцию longjmp)? К сожалению, ответ на этот вопрос таков: «Это зависит от реализации». Большинство реализаций не «откручивают» назад автоматические и регистровые переменные, а стандарты утверждают, что их значения в этом случае не определены. Если у вас есть автоматические переменные, значения которых не должны «откручиваться» назад, определите их со спецификатором volatile. Вызов функции longjmp не оказывает влияния на глобальные или статические переменные.

Пример

Программа, представленная листингом 7.6, демонстрирует различия в поведении автоматических, регистровых, глобальных, статических и volatile переменных, наблюдаемые после вызова функции longjmp.

Листинг 7.6. Влияние longjmp на переменные с различными классами размещения

```
#include "apue.h"
#include <setjmp.h>
```

```
static void f1(int, int, int, int);
static void f2(void);

static jmp_buf jmpbuffer;
static int globval;

int
main(void)
{
    int autoval;
    register int regival;
    volatile int volaval;
    static int statval;

    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("после вызова longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
               " volaval = %d, statval = %d\n",
               globval, autoval, regival, volaval, statval);
        exit(0);
    }

    /*
     * Изменить переменные после обращения к setjmp, но до вызова longjmp.
     */
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* никогда не вернет */
                                            /* управление в эту точку */
    exit(0);
}

static void
f1(int i, int j, int k, int l)
{
    printf("в функции f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
           " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}

static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

Если скомпилировать эту программу с включенной оптимизацией и без оптимизации, то мы получим разные результаты:

```
$ cc testjmp.c      скомпилировать без оптимизации
$ ./a.out
```

```
в функции f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
после вызова longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
$ cc -O testjmp.c    скомпилировать с полной оптимизацией
$ ./a.out
в функции f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
после вызова longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

Обратите внимание: оптимизация не оказывает никакого влияния на глобальные, статические переменные и на переменные, объявленные со спецификатором `volatile`; эти переменные после вызова `longjmp` сохраняют последние присвоенные им значения. Страница справочного руководства `setjmp`(3) в одной из систем заявляет, что переменные, хранящиеся в памяти, будут иметь те же значения, что и в момент вызова `longjmp`, тогда как переменные, находящиеся в регистрах центрального процессора и арифметического сопроцессора, будут восстановлены в состояние, соответствующее первому вызову функции `setjmp`. Это в точности соответствует тому, что мы наблюдали в экспериментах с программой из листинга 7.6. Когда оптимизация отключена, все пять переменных сохраняются в памяти (спецификатор `register` для переменной `regival` игнорируется). Когда оптимизация включена, переменные `autoval` и `regival` перемещаются в регистры (даже несмотря на то, что первая из них не была объявлена как `register`), а переменная, объявленная со спецификатором `volatile`, остается в памяти. Из этого примера следует вывод – если вы пишете переносимый код, в котором выполняются нелокальные переходы, используйте спецификатор `volatile`. В зависимости от системы, могут обнаружиться и другие отличия.

Некоторые строки в листинге 7.6, содержащие обращения к функции `printf`, не умещаются по ширине экрана, что несколько неудобно. Вместо того чтобы многократно повторять вызовы `printf`, мы полагаемся на возможность конкатенации строк, предусмотренную стандартом ISO C, когда последовательность

```
"string1" "string2"
```

эквивалентна последовательности

```
"string1string2"
```

Мы еще вернемся к функциям `setjmp` и `longjmp` в главе 10, когда будем обсуждать обработчики сигналов и версии этих функций для работы с сигналами: `sigsetjmp` и `siglongjmp`.

Возможные проблемы с автоматическими переменными

Рассмотрев порядок работы с фреймами стека, мы должны обратить ваше внимание на одну потенциальную ошибку, связанную с автоматическими

переменными. Всегда следует придерживаться основного правила – не обращаться к автоматической переменной после того, как функция, в которой она была объявлена, вернула управление. Многочисленные предупреждения об этом встречаются повсюду в справочном руководстве UNIX.

В листинге 7.7 показана функция `open_data`, которая открывает поток ввода-вывода и выполняет настройку режима его буферизации.

Листинг 7.7. Неправильное использование автоматической переменной

```
#include <stdio.h>
#define DATAFILE "datafile"

FILE *
open_data(void)
{
    FILE *fp;
    char databuf[BUFSIZ]; /*setvbuf сделает этот массив буфером ввода-вывода*/

    if ((fp = fopen(DATAFILE, "r")) == NULL)
        return(NULL);
    if (setvbuf(fp, databuf, _IOLBF, BUFSIZ) != 0)
        return(NULL);
    return(fp); /* ошибка */
}
```

Проблема в том, что, когда функция `open_data` вернет управление, пространство на стеке, которое она использовала, будут отдано под фрейм стека следующей вызываемой функции. Однако стандартная библиотека ввода-вывода по-прежнему будет использовать эту часть памяти под буфер потока ввода-вывода. Хаос будет неизбежен. Чтобы исправить эту проблему, следует разместить массив `databuf` в глобальной памяти, статически (`static` или `extern`) или динамически (с помощью одной из функций распределения памяти).

7.11. Функции getrlimit и setrlimit

Любой процесс имеет ряд ограничений на использование ресурсов. Значения некоторых из этих ограничений можно запросить и изменить с помощью функций `getrlimit` и `setrlimit`.

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);
```

Обе возвращают 0 в случае успеха, ненулевое значение в случае ошибки

Эти две функции определены стандартом Single UNIX Specification как расширения XSI. Ограничения на ресурсы для процесса обычно устанавливаются процессом с идентификатором 0 во время инициализации системы и затем наследуются остальными процессами. Каждая реализация предлагает свой собственный способ настройки различных ограничений.

При обращении к этим функциям им передается наименование ресурса (*resource*) и указатель на следующую структуру:

```
struct rlimit {
    rlim_t rlim_cur; /* мягкий предел: текущий предел */
    rlim_t rlim_max; /* жесткий предел: максимальное значение для rlim_cur */
};
```

Изменение пределов ресурсов производится в соответствии со следующими тремя правилами:

1. Процесс может изменять значение мягкого предела, при условии, что оно не будет превышать значения жесткого предела.
2. Процесс может понизить значение жесткого предела вплоть до значения мягкого предела. Операция понижения жесткого предела необратима для рядовых пользователей.
3. Только процесс, обладающий привилегиями суперпользователя, может поднять значение жесткого предела.

Бесконечность предела определяется константой `RLIM_INFINITY`.

В аргументе `resource` передается одно из следующих значений.

`RLIMIT_AS` Максимальный размер доступной процессу памяти (в байтах). Этот предел оказывает влияние на функции `sbrk` (раздел 1.11) и `mmap` (раздел 14.9).

`RLIMIT_CORE` Максимальный размер файла дампа памяти (core). Значение 0 отключает создание файлов core.

`RLIMIT_CPU` Максимальное количество процессорного времени в секундах. По достижении мягкого предела процессу будет послан сигнал `SIGXCPU`.

`RLIMIT_DATA` Максимальный размер сегмента данных в байтах: сумма размеров сегментов инициализированных данных, неинициализированных данных и кучи (рис. 7.3).

`RLIMIT_FSIZE` Максимальный размер создаваемого файла в байтах. По достижении мягкого предела процессу будет послан сигнал `SIGXFSZ`.

`RLIMIT_LOCKS` Максимальное количество блокировок, которое процесс может наложить на файл. (Это число также включает количество оповещений об операциях над файлами, производимых другими процессами (leases), – функциональная особенность ОС Linux. Дополнительные сведения вы найдете на странице справочного руководства `fcntl`(2) в Linux.)

`RLIMIT_MEMLOCK` Максимальный объем памяти, которую процесс может заблокировать с помощью функции `mlock`(2), в байтах.

`RLIMIT_NOFILE` Максимальное количество одновременно открытых файлов. Изменение этого предела оказывает влияние на значение, возвращаемое функцией `sysconf` для аргумента `_SC_OPEN_MAX` (раздел 2.5.4 и листинг 2.4).

`RLIMIT_NPROC` Максимальное количество дочерних процессов на реальный идентификатор пользователя. Изменение этого предела оказывает влияние на значение, возвращаемое функцией `sysconf` для аргумента `_SC_CHILD_MAX` (раздел 2.5.4).

RLIMIT_RSS Максимальный объем страниц виртуальной памяти процесса, размещаемых резидентно в оперативной памяти, в байтах. Если физической памяти недостаточно, ядро будет «отнимать» память у процессов, которые превысили этот предел.

RLIMIT_SBSIZE Максимальный объем буферов сокетов, который может быть использован в каждый конкретный момент времени, в байтах.

RLIMIT_STACK Максимальный размер стека в байтах (рис. 7.3).

RLIMIT_VMEM Синоним RLIMIT_AS.

В табл. 7.3 указано, какие ограничения на ресурсы определены стандартом Single UNIX Specification и какие из них поддерживаются каждой из реализаций.

Таблица 7.3. Поддерживаемые ограничения на ресурсы

Предел	XSI	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
RLIMIT_AS	•		•		•
RLIMIT_CORE	•	•	•	•	•
RLIMIT_CPU	•	•	•	•	•
RLIMIT_DATA	•	•	•	•	•
RLIMIT_FSIZE	•	•	•	•	•
RLIMIT_LOCKS			•		
RLIMIT_MEMLOCK		•	•	•	
RLIMIT_NOFILE	•	•	•	•	•
RLIMIT_NPROC		•	•	•	
RLIMIT_RSS		•	•	•	
RLIMIT_SBSIZE		•			
RLIMIT_STACK	•	•	•	•	•
RLIMIT_VMEM		•			•

Предельные значения для ресурсов оказывают влияние на вызывающий процесс и наследуются всеми его дочерними процессами. Это означает, что команда изменения ограничений на ресурсы должна быть встроена в командную оболочку, чтобы воздействовать на все процессы, запускаемые из нее. И действительно, в командных оболочках Bourne shell, GNU Bourne-again shell и Korn shell имеется встроенная команда ulimit, а в командной оболочке C shell – команда limit. (Команды umask и chdir также должны быть встроенными.)

Пример

Программа, представленная листингом 7.8, выводит текущие мягкие и жесткие значения для всех пределов, поддерживаемых системой.

Чтобы скомпилировать эту программу в различных реализациях UNIX, мы использовали директивы условной компиляции для подключения заголовочных файлов и обработки констант, имена которых могут различаться в разных системах. Обратите также внимание на то, что в функции `printf` нужно использовать различные спецификаторы формата, поскольку некоторые платформы определяют тип `rlim_t` как `unsigned long long` вместо `unsigned long`.

Листинг 7.8. Вывод значений пределов ресурсов

```
#include "apue.h"
#if defined(BSD) || defined(MACOS)
#include <sys/time.h>
#define FMT "%10lld "
#else
#define FMT "%10ld "
#endif
#include <sys/resource.h>

#define doit(name) pr_limits(#name, name)

static void pr_limits(char *, int);

int
main(void)
{
#ifdef RLIMIT_AS
    doit(RLIMIT_AS);
#endif
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);
#ifdef RLIMIT_LOCKS
    doit(RLIMIT_LOCKS);
#endif
#ifdef RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
    doit(RLIMIT_NOFILE);
#ifdef RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
#ifdef RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
#endif
    doit(RLIMIT_STACK);
#ifdef RLIMIT_VMEM
    doit(RLIMIT_VMEM);
```

```
#endif
    exit(0);
}

static void
pr_limits(char *name, int resource)
{
    struct rlimit limit;

    if (getrlimit(resource, &limit) < 0)
        err_sys("ошибка вызова функции getrlimit для %s", name);
    printf("%-19s ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(бесконечность)");
    else
        printf(FMT, limit.rlim_cur);

    if (limit.rlim_max == RLIM_INFINITY)
        printf("(бесконечность)");
    else
        printf(FMT, limit.rlim_max);
    putchar((int)'\n');
}
```

Обратите внимание, что в макросе `doit` для генерации строки, содержащей имя ресурса, мы воспользовались оператором создания строки (`#`), который предусматривается стандартом ISO C. Таким образом, вызов макроса

```
doit(RLIMIT_CORE)
```

препроцессор C развернет в строку

```
pr_limits("RLIMIT_CORE", RLIMIT_CORE);
```

Запустив программу в ОС FreeBSD, мы получили следующие результаты:

```
$ ./a.out
RLIMIT_CORE    (бесконечность)    (бесконечность)
RLIMIT_CPU     (бесконечность)    (бесконечность)
RLIMIT_DATA          536870912          536870912
RLIMIT_FSIZE   (бесконечность)    (бесконечность)
RLIMIT_MEMLOCK (бесконечность)    (бесконечность)
RLIMIT_NOFILE             1735               1735
RLIMIT_NPROC               867                867
RLIMIT_RSS     (бесконечность)    (бесконечность)
RLIMIT_SBSIZE  (бесконечность)    (бесконечность)
RLIMIT_STACK           67108864           67108864
RLIMIT_VMEM    (бесконечность)    (бесконечность)
```

В ОС Solaris:

```
$ ./a.out
RLIMIT_AS      (бесконечность)    (бесконечность)
RLIMIT_CORE    (бесконечность)    (бесконечность)
RLIMIT_CPU     (бесконечность)    (бесконечность)
```

```
RLIMIT_DATA     (бесконечность)    (бесконечность)
RLIMIT_FSIZE    (бесконечность)    (бесконечность)
RLIMIT_NOFILE              256              65536
RLIMIT_STACK           8388608    (бесконечность)
RLIMIT_VMEM     (бесконечность)   (бесконечность)
```

После знакомства с сигналами мы продолжим обсуждение пределов ресурсов в упражнении 10.11.

7.12. Подведение итогов

Понимание среды окружения программ UNIX, написанных на C, совершенно необходимо для понимания особенностей управления процессами в UNIX. Из материала этой главе мы узнали, как процесс запускается, как он может завершиться и каким образом процессу передаются списки аргументов и переменных окружения. Несмотря на то, что ядро никак не анализирует ни тот, ни другой список, тем не менее именно оно передает их новому процессу от вызывающей функцию `exec` программы.

Мы также исследовали типичную раскладку памяти программ, написанных на C, и коснулись вопроса динамического распределения и освобождения памяти. Детально рассматривались функции управления средой окружения, так как они связаны с распределением памяти. Были описаны функции `setjmp` и `longjmp`, которые обеспечивают выполнение нелокальных переходов в пределах процесса. И в завершение мы рассказали об ограничениях ресурсов, которые накладываются различными реализациями.

Упражнения

7.1. Если в операционных системах FreeBSD и Linux на аппаратной архитектуре x86 запустить программу «Привет, МИР!», которая не вызывает функцию `exit` и не использует оператор `return` для выхода из функции `main`, то код завершения программы окажется равным 13 (это легко проверить средствами командной оболочки). Почему?

7.2. Когда фактически происходит отображение строк, которые выводятся с помощью функции `printf` в листинге 7.2?

7.3. Существует ли способ получить доступ к аргументам командной строки из функций, вызываемых из функции `main`, при условии, что (а) аргументы *argc* и *argv* в вызываемую функцию передаваться не будут и (б) содержимое этих аргументов не будет копироваться в глобальные переменные.

7.4. В некоторых реализациях UNIX нулевой адрес в сегменте данных программы преднамеренно делается недоступным. Почему?

7.5. Попробуйте определить с помощью `typedef` новый тип данных `Exitfunc` для функции-обработчика выхода. Измените прототип функции `atexit` с использованием этого типа.

7.6. Если разместить массив значений типа long с помощью функции `calloc`, то будут ли элементы массива инициализированы нулевыми значениями? Если разместить массив указателей с помощью функции `calloc`, то будут ли элементы массива инициализированы как пустые указатели?

7.7. Почему в конце раздела 7.6 мы не получили размеры стека и кучи от команды `size`?

7.8. Почему в разделе 7.7 размеры файлов (475570 и 11410) не совпадают с суммой размеров их сегментов кода и данных?

7.9. Почему в разделе 7.7 при использовании разделяемых библиотек получается настолько большая разница в размерах файлов такой простенькой программы?

7.10. В конце раздела 7.10 мы показали, что функция не может возвращать указатель на локальную переменную с автоматическим классом размещения. Как вы думаете, будет ли следующий код работать корректно?

```
int
f1(int val)
{
    int *ptr;

    if (val == 0) {
        int val;

        val = 5;
        ptr = &val;
    }
    return(*ptr + 1);
}
```

8

Управление процессами

8.1. Введение

Теперь мы перейдем к обсуждению управления процессами в UNIX. Сюда относится создание новых процессов, запуск программ и завершение процессов. Мы также рассмотрим различные идентификаторы, определяющие принадлежность процесса – реальные, эффективные и сохраненные идентификаторы пользователя и группы – и их влияние на элементарные функции управления процессами. Также будут обсуждаться интерпретируемые файлы и функция `system`. В завершение речь пойдет о средствах, предоставляемых большинством UNIX-систем для учета использования ресурсов процессами. Это позволит нам взглянуть на функции управления процессами под другим углом.

8.2. Идентификаторы процесса

Любой процесс обладает уникальным идентификатором процесса, который представляет собой целое положительное число. Поскольку идентификатор процесса – это единственный широко используемый идентификатор, уникальность которого гарантируется системой, он часто присоединяется к другим идентификаторам для придания им уникальности. Например, приложения иногда включают идентификатор процесса в имена файлов, чтобы обеспечить их уникальность.

Но несмотря на свою уникальность идентификаторы процесса могут использоваться многократно. По завершении процесса его идентификатор может быть использован повторно для другого процесса. Однако в большинстве версий UNIX реализованы специальные алгоритмы, позволяющие отложить повторное использование идентификатора на более позднее время, чтобы вновь создаваемый процесс не мог получить идентификатор процесса, который завершился совсем недавно. Это помогает избежать ситуации, когда новый процесс по ошибке может быть принят за предыдущий при использовании того же самого идентификатора.

Существует ряд специальных процессов, определяемых конкретной реализацией. Процесс с идентификатором 0 – это, как правило, процесс-планировщик, который часто называют *swapper* (программа подкачки). Этому процессу не соответствует никакая программа на диске, поскольку он является частью ядра и известен как системный процесс. Процесс с идентификатором 1 – это обычно процесс `init`, который запускается ядром в конце процедуры начальной загрузки. В старых версиях UNIX этому процессу соответствует программа `/etc/init`, в более новых версиях – `/sbin/init`. Этот процесс отвечает за запуск операционной системы после загрузки ядра. Обычно `init` читает системные файлы инициализации – `/etc/rc*` или `/etc/inittab`, а также файлы, расположенные в каталоге `/etc/init.d`, и переводит систему в некоторое состояние, например в многопользовательский режим. Процесс `init` никогда не «умирает». Это обычный пользовательский процесс, он не является системным процессом ядра, как *swapper*, хотя и обладает привилегиями суперпользователя. Далее в этой главе мы увидим, как процесс `init` становится родительским процессом любого осиротевшего дочернего процесса.

Каждая версия UNIX имеет свой собственный набор процессов ядра, отвечающих за работу системных служб. Например, в некоторых реализациях виртуальной памяти UNIX идентификатор 2 соответствует процессу *pagedaemon*. Этот процесс отвечает за поддержку страничного обмена системы виртуальной памяти.

В дополнение к идентификатору процесса, каждый процесс обладает еще целым рядом идентификаторов. Вот функции, которые возвращают эти идентификаторы:

```
#include <unistd.h>

pid_t getpid(void);
```

Возвращает идентификатор вызывающего процесса

```
pid_t getppid(void);
```

Возвращает идентификатор родительского процесса

```
uid_t getuid(void);
```

Возвращает реальный идентификатор пользователя вызывающего процесса

```
uid_t geteuid(void);
```

Возвращает эффективный идентификатор пользователя вызывающего процесса

```
gid_t getgid(void);
```

Возвращает реальный идентификатор группы вызывающего процесса

```
gid_t getegid(void);
```

Возвращает эффективный идентификатор группы вызывающего процесса

Обратите внимание, что ни одна из этих функций не возвращает признак ошибки. К идентификатору родительского процесса мы еще вернемся, когда будем обсуждать функцию fork. Реальный и эффективный идентификаторы пользователя и группы мы уже рассматривали в разделе 4.4.

8.3. Функция fork

Любой существующий процесс может создать новый процесс, обратившись к функции fork.

```
#include <unistd.h>

pid_t fork(void);
```

Возвращает 0 в дочернем процессе, идентификатор дочернего процесса – в родительском, –1 в случае ошибки

Новый процесс, создаваемый функцией fork, называется *дочерним процессом, или процессом-потомком*. Эта функция вызывается один раз, а управление возвращает дважды, с единственным отличием – в дочернем процессе она возвращает 0, в то время как в родительском – идентификатор созданного дочернего процесса. Последнее обстоятельство объясняется тем, что у процесса может быть целое множество потомков, а система не предусматривает функций, с помощью которых можно было бы получить идентификаторы дочерних процессов. В дочернем процессе функция fork возвращает 0, поскольку дочерний процесс имеет только одного родителя и всегда может получить его идентификатор с помощью функции getppid. (Идентификатор процесса 0 зарезервирован за ядром, поэтому невозможно получить 0 в качестве идентификатора дочернего процесса.)

И родительский, и дочерний процессы продолжают выполнение инструкций программы, следующих за вызовом функции fork. Процесс-потомок представляет собой копию родительского процесса. Таким образом, потомок получает в свое распоряжение копии сегмента данных, кучи и стека родителя. Обратите внимание, что это именно копии; родительский и дочерний процессы не используют совместно одни и те же области памяти. Они совместно используют сегмент кода (раздел 7.6).

Современные версии UNIX не производят немедленного полного копирования сегмента данных, стека и кучи, поскольку очень часто вслед за вызовом fork сразу же следует вызов exec. Вместо этого используется метод, который получил название *копирование при записи* (*copy-on-write, COW*). Указанные выше области памяти используются совместно обоими процессами, но ядро делает их доступными только для чтения. Если один из процессов попытается изменить данные в этих областях, ядро немедленно сделает только копию конкретного участка памяти; обычно это «страница» виртуальной памяти. Более подробно об этом можно прочитать в разделе 9.2 [Bach 1986] и в разделах 5.6 и 5.7 [McKusik et al. 1996].

Некоторые платформы предоставляют несколько версий функции fork. Все четыре платформы, обсуждаемые в данной книге, поддерживают функцию vfork(2), которую мы рассмотрим в следующем разделе.

Кроме того, ОС Linux 2.4.22 предоставляет возможность создания новых процессов с помощью системного вызова clone(2). Это более универсальный вариант функции fork, который позволяет вызывающему процессу определить, что будет совместно использоваться дочерним и родительским процессами.

FreeBSD 5.2.1 предоставляет системный вызов rfork(2), который очень напоминает системный вызов clone в Linux и позаимствован из ОС Plan 9 ([Pike et al. 1995]).

Solaris 9 предоставляет две библиотеки для работы с потоками: одна для потоков POSIX (pthreads) и другая – для потоков Solaris. Поведение функции fork в этих библиотеках различно. В случае потоков POSIX функция fork создает процесс, содержащий только вызывающий поток, а в случае потоков Solaris – процесс, содержащий копии всех потоков вызывающего процесса. Для предоставления семантики, аналогичной случаю потоков POSIX, Solaris поддерживает функцию fork1, которая может использоваться для создания процесса, состоящего только из копии вызывающего потока, независимо от используемой библиотеки. Более подробно потоки обсуждаются в главах 11 и 12.

Пример

Программа, представленная листингом 8.1, демонстрирует работу с функцией fork и показывает, что изменение переменных в дочернем процессе никак не сказывается на переменных в родительском процессе.

Листинг 8.1. Пример работы с функцией fork

```
#include "apue.h"

int glob = 6; /* глобальная переменная в сегменте инициализированных данных */
char buf[] = "запись в stdout\n";

int
main(void)
{
    int var;  /* переменная, размещаемая на стеке */
    pid_t pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("ошибка вызова функции write");
    printf("перед вызовом функции fork\n"); /* мы не сбрасываем */
                                            /* буферы stdout */
    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) {                  /* дочерний процесс */
        glob++;                             /* изменить переменные */
        var++;
    } else {
        sleep(2);                           /* родительский процесс */
    }

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
```

```
    exit(0);
}
```

После запуска программы мы получим:

```
$ ./a.out
запись в stdout
перед вызовом функции fork
pid = 430, glob = 7, var = 89   переменные в дочернем процессе были изменены
pid = 429, glob = 6, var = 88   родительская копия осталась без изменений
$ ./a.out > temp.out
$ cat temp.out
запись в stdout
перед вызовом функции fork
pid = 432, glob = 7, var = 89
перед вызовом функции fork
pid = 431, glob = 6, var = 88
```

Вообще, никогда нельзя сказать точно, какой из двух процессов первым получит управление после вызова функции `fork` – дочерний или родительский. Это во многом зависит от алгоритма планирования, используемого ядром. Если необходимо синхронизировать работу родительского и дочернего процессов, то потребуется организовать некоторое взаимодействие между ними. В программе из листинга 8.1 мы просто приостановили родительский процесс на 2 секунды, чтобы позволить дочернему процессу выполняться первым. Но нет никакой гарантии, что этот прием сработает при любых условиях. Об этом и некоторых других видах синхронизации мы поговорим в разделе 8.9, когда будем обсуждать гонки за ресурсами. В разделе 10.6 мы покажем, как можно использовать сигналы для синхронизации родительского и дочернего процессов после возврата управления из функции `fork`.

Записывая данные на стандартный вывод, мы вычитаем 1 из размера `buf`, чтобы избежать записи завершающего нулевого байта. Несмотря на то, что функция `strlen` вычисляет длину строки без учета завершающего нулевого байта, `sizeof` вычисляет размер буфера, который включает завершающий нулевой байт. Другое отличие: обращение к `strlen` представляет собой полноценный вызов функции, тогда как `sizeof` вычисляет размер буфера на этапе компиляции, поскольку буфер инициализирован известной строкой фиксированной длины.

Обратите внимание на то, как функция `fork` в программе из листинга 8.1 взаимодействует с функциями ввода-вывода. В главе 3 мы уже говорили, что функция `write` не буферизуется. Так как функция `write` вызывается перед `fork`, она выведет данные на стандартный вывод только один раз. С другой стороны, стандартная библиотека ввода-вывода буферизуется. В разделе 5.12 мы говорили, что стандартному потоку вывода назначается режим построчной буферизации, если он связан с терминалом, и режим полной буферизации – в любом другом случае. Запуская программу в интерактивном режиме, мы получаем только одну копию строки, выводимой функцией `printf`, потому что буфер стандартного потока вывода сбрасывается, если встречен символ перевода строки. Но когда стандартный поток вывода пере-

направляется в файл, мы получаем две копии строки. В этом случае перед обращением к `fork` функция `printf` вызывается один раз, но строка в момент вызова функции `fork` еще находится в буфере. В результате этот буфер будет скопирован в адресное пространство дочернего процесса при копировании сегмента данных родителя. Теперь и у родительского, и у дочернего процессов есть буферы стандартного потока ввода, в которых находится одна и та же строка. Второй вызов функции `printf`, который происходит непосредственно перед вызовом функции `exit`, лишь добавляет свои данные в конец существующего буфера. По завершении каждого из процессов его копия буфера наконец сбрасывается на диск.

Совместное использование файлов

Когда при запуске программы из листинга 8.1 мы перенаправляем стандартный поток вывода родительского процесса в файл, стандартный поток вывода дочернего процесса также оказывается перенаправленным. Действительно, одна из особенностей функции `fork` заключается в том, что она передает дочернему процессу дубликаты всех дескрипторов, открытых в родительском процессе. Мы говорим «дубликаты», потому что они представляют собой копии дескрипторов, как если бы для каждого из них была вызвана функция `dup`. Родительский и дочерний процессы совместно используют одни и те же записи в таблице файлов для каждого из открытых дескрипторов (вспомните рис. 3.3).

Представим себе процесс, который открыл три файла: стандартного ввода, стандартного вывода и стандартного вывода сообщений об ошибках. По возвращении из функции `fork` мы получим распределение дескрипторов, показанное на рис. 8.1.

Важно заметить, что и родительский, и дочерний процессы совместно используют текущую позицию файла. Рассмотрим процесс, который запустил дочерний процесс и ожидает его завершения. Допустим, что оба процесса в ходе своей работы производят запись в стандартный поток вывода. Если стандартный поток вывода родительского процесса будет перенаправлен в файл (например, командной оболочкой), то текущая позиция файла, установленная родительским процессом, неизбежно будет изменена дочерним процессом, когда он выполнит запись в стандартный поток вывода. В этом случае дочерний процесс может записывать данные в стандартный поток вывода, пока родительский процесс ожидает его завершения. По окончании работы дочернего процесса родительский процесс сможет продолжить запись в стандартный поток вывода, зная, что его данные будут записаны после тех, что записал дочерний процесс. Если бы текущая позиция файла различалась у родительского и дочернего процессов, то подобного эффекта достичь было бы гораздо сложнее, и это потребовало бы дополнительных усилий со стороны родительского процесса.

Если и родительский, и дочерний процессы пишут в один и тот же дескриптор без какой-либо синхронизации, например, когда родительский процесс не ожидает завершения дочернего процесса, то данные их вывода будут пе-

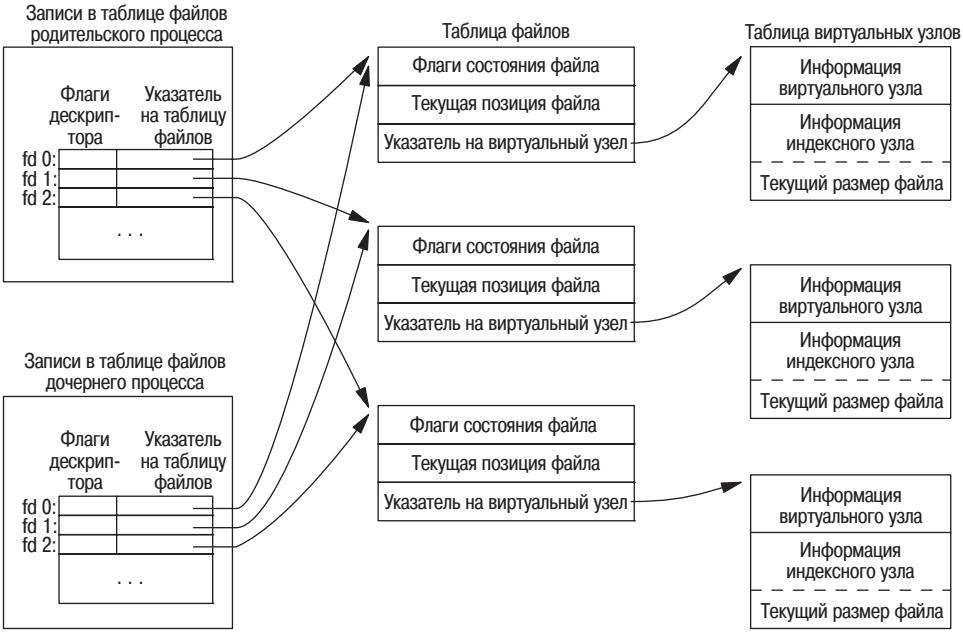


Рис. 8.1. *Совместное использование открытых файлов родительским и дочерним процессами после вызова функции fork*

ремешаны (если этот дескриптор был открыт до вызова функции `fork`). Хотя это и возможно согласно рис. 8.1, тем не менее такой режим работы не является нормальным.

Существуют два стандартных способа обслуживания дескрипторов после вызова функции `fork`.

1. Родительский процесс ожидает, когда потомок завершит свою работу. В этом случае родительскому процессу ничего не нужно делать со своими дескрипторами. Когда потомок завершится, текущая позиция файла любого из разделяемых дескрипторов, которые использовались им для чтения или записи, изменится надлежащим образом.
2. И родительский, и дочерний процессы продолжают работу независимо друг от друга. В этом случае после вызова функции `fork` родительский процесс закрывает дескрипторы, которые ему больше не потребуются, дочерний процесс делает то же самое. Таким образом, они более не будут совместно использовать одни и те же дескрипторы. Этот сценарий часто используется в сетевых серверах.

Помимо открытых файлов есть много других характеристик родительского процесса, которые наследуются дочерним:

- Реальный идентификатор пользователя, реальный идентификатор группы, эффективный идентификатор пользователя, эффективный идентификатор группы.

- Идентификаторы дополнительных групп.
- Идентификатор группы процессов.
- Идентификатор сессии.
- Управляющий терминал.
- Флаги set-user-ID и set-group-ID.
- Текущий рабочий каталог.
- Корневой каталог.
- Маска режима создания файлов.
- Маска сигналов и их диспозиция.
- Флаги close-on-exec для открытых дескрипторов.
- Среда окружения.
- Присоединенные сегменты разделяемой памяти.
- Отображения в память.
- Ограничения на ресурсы.

Существуют следующие отличия между родительским и дочерним процессами:

- Функция `fork` возвращает различные значения.
- Различные идентификаторы процессов.
- Различные идентификаторы родительских процессов: идентификатор родительского процесса в потомке соответствует идентификатору процесса в родительском процессе, идентификатор родительского процесса в родительском процессе остается без изменений.
- Значения `tms_utime`, `tms_stime`, `tms_cutime` и `tms_cstime` в дочернем процессе устанавливаются равными 0.
- Блокировки файлов, установленные в родительском процессе, не наследуются.
- Таймеры, ожидающие срабатывания, в дочернем процессе сбрасываются.
- Набор сигналов, ожидающих обработки, в дочернем процессе очищается.

Многие из этих характеристик еще не обсуждались; мы поговорим о них в следующих главах.

Ошибка вызова функции `fork` происходит обычно в двух случаях: (а) когда в системе слишком много работающих процессов, что обычно свидетельствует о неполадках, или (б) когда общее количество процессов превысило максимальное значение для заданного реального идентификатора пользователя. В табл. 2.10 мы уже указывали, что максимальное количество одновременно работающих процессов на один реальный идентификатор пользователя определяется константой `CHILD_MAX`.

Два основных случая, когда используется функция `fork`:

1. Когда процесс хочет продублировать себя, чтобы родительский и дочерний процессы могли выполнять различные участки программы одновре-

менно. Это обычно используется в сетевых серверах. Родительский процесс ожидает запроса на обслуживание от клиента, по его получении он вызывает fork и передает обслуживание запроса дочернему процессу, после чего возвращается к ожиданию следующего запроса.

2. Когда процесс хочет запустить другую программу. Это обычно используется в командных оболочках. В этом случае дочерний процесс вызывает функцию exec (которую мы рассмотрим в разделе 8.10), как только функция fork вернет управление.

Некоторые операционные системы объединяют вызов fork и следующий за ним вызов exec в одну операцию, которая называется *spawn*. UNIX разделяет две операции по той простой причине, что достаточно часто вызов функции fork не сопровождается вызовом функции exec. Кроме того, такое разделение позволяет дочернему процессу между вызовами fork и exec изменить некоторые характеристики процесса, например, перенаправление ввода-вывода, идентификатор пользователя, диспозицию сигналов и т. д. Многочисленные примеры, иллюстрирующие это, мы увидим в главе 15.

> Впрочем, стандарт Single UNIX Specification включает интерфейсы spawn в группу расширений реального времени. Однако эти интерфейсы не служат заменой fork и exec. Они предназначены для систем, в которых имеются определенные сложности с эффективной реализацией функции fork, особенно для тех, в которых отсутствует аппаратная поддержка управления памятью.

8.4. Функция vfork

Порядок вызова и возвращаемые значения функций vfork и fork одинаковы, но семантика этих двух функций различается.

> Функция vfork впервые появилась в 2.9BSD. Некоторые считают эту функцию пятном на репутации UNIX, однако все платформы, обсуждаемые в этой книге, поддерживают ее. Фактически разработчики удалили vfork из версии 4.4BSD, но все дистрибутивы BSD с открытыми исходными текстами, происходящие от 4.4BSD, восстановили ее поддержку. В третьей версии Single UNIX Specification функция vfork отмечена как устаревший интерфейс.

Функция vfork предназначена для создания новых процессов, когда целью нового процесса является запуск новой программы с помощью функции exec (пункт 2 в конце предыдущего раздела). Программа из листинга 1.5 также относится к программам этого типа. Функция vfork создает новый процесс точно так же, как fork, но не копирует адресное пространство родительского процесса в адресное пространство потомка, поскольку потомок не будет работать с этим адресным пространством – он просто вызывает функцию exec (или exit) сразу же после того, как vfork вернет управление. Таким образом, до вызова exec или exit дочерний процесс исполняется в адресном пространстве родительского процесса. Такой подход более эффективен для некоторых реализаций виртуальной памяти в UNIX. (Как мы уже упоминали в предыдущем разделе, для повышения эффективности работы связки fork – exec многие реализации используют технику копирования при записи, но полное

отсутствие копирования все же гораздо эффективнее, чем копирование даже небольших объемов данных.)

Еще одно различие между этими функциями – vfork гарантирует, что дочерний процесс получит управление первым и будет удерживать его, пока не вызовет функцию exec или exit. Когда дочерний процесс вызывает любую из этих функций, родительский процесс возобновляет свою работу. (Это может привести к тупиковой ситуации, если процесс-потомок зависит от дальнейших действий родительского процесса, которые должны быть выполнены до вызова любой из этих функций.)

Пример

Программа, представленная листингом 8.2, – это измененная версия программы из листинга 8.1. Мы заменили функцию fork на vfork и убрали запись на стандартный вывод. Теперь нет необходимости приостанавливать родительский процесс с помощью функции sleep, поскольку vfork гарантирует, что он будет приостановлен ядром до тех пор, пока дочерний процесс не вызовет exec или exit.

Листинг 8.2. Пример работы с функцией vfork

```
#include "apue.h"

int glob = 6; /* глобальная переменная в сегменте инициализированных данных */

int
main(void)
{
    int var;  /* локальная переменная в стеке */
    pid_t pid;

    var = 88;
    printf("перед вызовом функции vfork\n"); /* мы не сбрасываем буферы stdout */
    if ((pid = vfork()) < 0) {
        err_sys("ошибка вызова функции vfork");
    } else if (pid == 0) {                  /* дочерний процесс */
        glob++;                             /* изменить переменные */
        var++;
        _exit(0);                           /* завершение дочернего процесса */
    }

    /*
     * Родительский процесс продолжит работу отсюда.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Запуск этой программы дает следующие результаты:

```
$ ./a.out
перед вызовом функции vfork
pid = 29039, glob = 7, var = 89
```

Здесь значения переменных, увеличенные в дочернем процессе, изменились и в родительском процессе. Поскольку известно, что дочерний процесс продолжает работу в адресном пространстве родительского процесса, это не стало для нас сюрпризом. Однако это поведение отличается от того, что мы видели при работе с функцией `fork`.

Обратите внимание, в программе из листинга 8.2 вместо функции `exit` используется `_exit`. Как уже говорилось в разделе 7.3, функция `_exit` не производит сброс буферов ввода-вывода. Вызвав функцию `exit`, мы получили бы несколько иные результаты. В зависимости от реализации стандартной библиотеки ввода-вывода мы могли бы и не заметить никаких различий или увидели бы, что пропали данные, выводимые функцией `printf` в родительском процессе.

Когда дочерний процесс завершает свою работу через вызов функции `exit`, содержимое всех буферов ввода-вывода сбрасывается. Если это единственное действие, которое производится библиотекой, то мы не увидим никаких различий по сравнению с вызовом функции `_exit`. Однако, если реализация дополнительно закрывает потоки ввода-вывода, то память, в которой размещается объект `FILE` стандартного потока вывода, будет очищена. Поскольку дочерний процесс заимствует адресное пространство родительского процесса, то когда родительский процесс возобновит работу и вызовет функцию `printf`, она ничего не сможет вывести и вернет признак ошибки (–1). Обратите внимание, что дескриптор `STDOUT_FILENO` родительского процесса все еще является допустимым, поскольку дочерний процесс получает копию массива файловых дескрипторов родительского процесса (рис. 8.1).

> В большинстве современных реализаций функция `exit` не закрывает потоки ввода-вывода. Поскольку процесс собирается завершить свою работу, ядро все равно закроет все дескрипторы файлов, открытые процессом. Закрытие их в библиотеке только увеличивает нагрузку и не несет никакой выгоды.

Дополнительные сведения о реализации функций `fork` и `vfork` можно найти в [McKusik et al. 1996], раздел 5.6. К изучению функции `vfork` мы вернемся в упражнениях 8.1 и 8.2.

8.5. Функция exit

В разделе 7.3 мы упоминали пять способов нормального завершения работы процесса:

1. Возврат из функции `main`. Как уже говорилось в разделе 7.3, это эквивалентно вызову функции `exit`.
2. Вызов функции `exit`. Эта функция определена стандартом ISO C, она производит вызов всех функций-обработчиков выхода, зарегистрированных функцией `atexit`, и закрывает все стандартные потоки ввода-вывода. Поскольку стандарт ISO C не затрагивает дескрипторы файлов, многозадачность (родительский и дочерний процессы) и управление заданиями, определение этой функции для UNIX является неполным.

3. Вызов функции _exit или _Exit. Стандарт ISO C определяет функцию _Exit как способ завершения процесса без запуска функций-обработчиков выхода или обработчиков сигналов. Однако от конкретной реализации зависит, будут ли буфера ввода-вывода сбрасываться на диск или нет. В системе UNIX имена _exit и _Exit являются синонимами, обе эти функции не сбрасывают буферы ввода-вывода. Функция _exit вызывается из exit и производит действия, специфичные для UNIX. Функция _exit определена стандартом POSIX.1.

   В большинстве реализаций UNIX exit(3) представляет собой библиотечную функцию, в то время как _exit(2) – системный вызов.

4. Возврат из стартовой процедуры последнего потока в процессе. Однако код выхода потока при этом не будет использоваться в качестве кода выхода процесса. Когда последний поток в процессе возвращается из своей стартовой процедуры, процесс завершается с кодом выхода 0.
5. Вызов функции pthread_exit из последнего потока в процессе. Как и в предыдущем случае, код выхода процесса будет равен 0, аргумент функции pthread_exit при этом игнорируется. Более подробно об этой функции мы поговорим в разделе 11.5.

Три способа ненормального завершения процесса:

1. Вызов функции abort. Это особый случай следующего способа, так как эта функция генерирует сигнал SIGABRT.
2. При получении процессом некоторых сигналов. (Более подробно о сигналах рассказывается в главе 10.) Сигнал может быть сгенерирован самим процессом (например, с помощью функции abort), другими процессами или ядром. К последним относятся сигналы, передаваемые при попытке обращения к памяти, расположенной вне адресного пространства процесса, или при попытке деления на ноль.
3. По запросу на завершение последнего потока. По умолчанию завершение потока происходит с некоторой задержкой: один поток запрашивает завершение другого потока, и через какое-то время указанный поток завершается. Мы обсудим запросы на завершение в разделах 11.5 и 12.7.

Независимо от того, как именно завершается процесс, в конечном итоге ядро выполняет один и тот же код. Этот код закрывает все открытые процессом дескрипторы, освобождает занимаемую процессом память и т. д.

Для любого из перечисленных способов необходимо, чтобы завершающийся процесс был в состоянии известить родительский процесс о том, как он завершился. В случае трех функций выхода (exit, _exit и _Exit) родительскому процессу через аргумент функции передается код выхода. А в случае ненормального завершения ядро генерирует код завершения, который указывает причину завершения процесса. В любом случае родительский процесс может получить код завершения от функции wait или waitpid (описание этих функций будет дано в следующем разделе).

Обратите внимание на различие между кодом выхода, который является аргументом одной из трех функций выхода или возвращаемым значением функции `main`, и кодом завершения. Ядро преобразует код выхода в код завершения, когда в заключение вызывается функция `_exit` (рис. 7.1). В табл. 8.1 перечислены способы, с помощью которых родительский процесс может получить код завершения дочернего процесса. Если дочерний процесс завершился нормально, родительский процесс может получить его код выхода.

Когда мы описывали функцию `fork`, было очевидно, что родительский процесс продолжает существовать после вызова функции `fork`. Сейчас мы говорим о возврате кода завершения родительскому процессу. Но что произойдет, если родительский процесс завершится раньше дочернего? Ответ таков: родителем любого процесса, родительский процесс которого завершился раньше его самого, становится процесс `init`. В таком случае мы говорим, что процесс был унаследован процессом `init`. Обычно при завершении какого-либо процесса ядро проверяет все активные процессы, чтобы узнать, не является ли завершившийся процесс чьим-либо родителем. Если это так, то для процесса, оставшегося активным, идентификатором родительского процесса назначается 1 (идентификатор процесса `init`). Таким образом удается гарантировать существование родителя у любого процесса.

Еще один момент, который мы должны рассмотреть – это когда дочерний процесс заканчивает работу раньше родительского. Если дочерний процесс полностью исчезнет, то родительский процесс будет не в состоянии получить его код завершения, когда это ему потребуется. Ядро сохраняет некоторый объем информации о каждом завершившемся процессе, чтобы она была доступна, когда родительский процесс вызовет функцию `wait` или `waitpid`. В простейшем случае эта информация состоит из идентификатора процесса, кода завершения процесса и количества процессорного времени, затраченного процессом. Ядро может освободить всю память, занимаемую процессом, и закрыть его открытые файлы. В терминологии UNIX процесс, который завершился, но его родительский процесс этого не ждал, называют *зомби*. Команда `ps`(1) выводит в поле состояния процесса-зомби символ Z. Если написать долго работающую программу, которая порождает множество дочерних процессов, они будут превращаться в зомби, если программа не станет дожидаться получения от них кодов завершения.

> В некоторых системах существует возможность предотвратить появление зомби; в 300 разделе 10.7 будет описано, как именно это сделать.

И наконец, рассмотрим случай, когда заканчивается процесс, унаследованный процессом `init`. Превращается ли он в зомби? Нет, потому что `init` создан так, что всякий раз, когда один из его потомков завершается, `init` вызывает одну из функций `wait`, чтобы забрать код завершения. Таким способом `init` препятствует засорению системы процессами-зомби. Под «потомками процесса `init`» мы подразумеваем как процессы, запущенные непосредственно процессом `init` (например, `getty`, который будет описан в разделе 9.2), так и унаследованнные процессы, родители которых завершили работу.

8.6. Функции wait и waitpid

Когда процесс завершается, обычным образом или аварийно, ядро извещает об этом родительский процесс с помощью сигнала SIGCHLD. Поскольку завершение дочернего процесса есть событие асинхронное (оно может произойти в любой момент времени), то и сигнал представляет собой асинхронное извещение, посылаемое ядром родительскому процессу. Родительский процесс может проигнорировать сигнал или предоставить функцию, которая будет вызвана по прибытии сигнала – обработчик сигнала. По умолчанию процессы игнорируют этот сигнал. Мы обсудим возможные варианты поведения в главе 10. А сейчас мы должны знать, что функции wait или waitpid, вызванные родительским процессом, могут:

- Заблокировать процесс, если все его дочерние процессы продолжают работу.
- Сразу же вернуть управление с кодом завершения дочернего процесса, если он уже закончил работу и ожидает, пока родительский процесс заберет код завершения.
- Сразу же вернуть управление с признаком ошибки, если у вызвавшего процесса нет ни одного дочернего процесса.

Если процесс вызывает функцию wait по получении сигнала SIGCHLD, то функция сразу же вернет управление. Но если функция wait была вызвана в любой произвольный момент времени, то она может заблокировать родительский процесс.

```
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Обе возвращают идентификатор процесса в случае успеха, –1 в случае ошибки

Различия между этими функциями следующие:

- Функция wait может заблокировать вызывающий процесс до тех пор, пока дочерний процесс не завершит свою работу, в то время как функция waitpid предоставляет возможность предотвратить блокировку.
- Функция waitpid не ждет первого завершившегося дочернего процесса – можно указать, завершения какого процесса она должна ожидать.

Если дочерний процесс уже завершился и находится в состоянии зомби, функция wait сразу же возвращает управление и передает код завершения этого процесса. В противном случае она блокирует вызывающий процесс до тех пор, пока дочерний процесс не завершит свою работу. Если у вызывающего процесса имеется несколько дочерних процессов, то функция wait вернет управление, когда завершит работу один из них. Мы всегда можем узнать, какой из потомков завершился, поскольку функция возвращает идентификатор процесса.

В обеих функциях аргумент *statloc* представляет собой указатель на целое число. Если в аргументе передается непустой указатель, то по заданному адресу будет записан код завершения дочернего процесса. Если код завершения нас не интересует, то можно просто передать в этом аргументе пустой указатель.

Целочисленный код завершения, возвращаемый этими двумя функциями, традиционно определяется реализацией. В нем несколько бит отводится под код выхода (в случае нормального завершения работы), несколько бит – под номер сигнала (в случае аварийного завершения), один бит указывает, был ли создан файл дампа памяти (файл core), и т. д. Согласно стандарту POSIX.1 в файле `<sys/wait.h>` определяются различные макросы, с помощью которых производится извлечение кодов выхода. Определить, как завершился процесс, можно с помощью четырех взаимоисключающих макросов, имена которых начинаются с префикса WIF. В зависимости от того, какой из этих четырех макросов возвращает истину, можно использовать другие макросы, чтобы получить код выхода, номер сигнала и другую информацию. Все четыре макроопределения приводятся в табл. 8.1.

Таблица 8.1. Макроопределения для проверки кода завершения, возвращаемого функциями wait и waitpid

Макроопределение	Описание
`WIFEXITED (`*status*`)`	Возвращает `true`, если *status* был получен от дочернего процесса при нормальном его завершении. В этом случае можно извлечь младшие 8 бит из аргумента, переданного функции `exit`, `_exit` или `_Exit`, следующим образом: `WEXITSTATUS(`*status*`)`

`WIFSIGNALED (`*status*`)`	Возвращает `true`, если *status* был получен от дочернего процесса при ненормальном (аварийном) его завершении, в результате получения сигнала, который не был перехвачен. В этом случае можно узнать номер сигнала, вызвавшего завершение дочернего процесса, следующим образом:
	`WTERMSIG(`*status*`)`
	Кроме того, в некоторых реализациях (но не в Single UNIX Specification) определен макрос
	`WCOREDUMP(`*status*`)`
	который возвращает `true`, если в результате аварийного завершения процесса был создан файл с дампом памяти (core)

`WIFSTOPPED (`*status*`)`	Возвращает `true`, если *status* был получен в результате остановки дочернего процесса по сигналу. В этом случае можно узнать номер сигнала, вызвавшего остановку процесса, с помощью макроса `WSTOPSIG(`*status*`)`
`WIFCONTINUED (`*status*`)`	Возвращает `true`, если *status* был получен для дочернего процесса, который продолжил работу после остановки (расширение XSI в стандарте POSIX.1 – только для функции `waitpid`)

В разделе 9.8, когда речь пойдет об управлении заданиями, мы увидим, как можно остановить процесс.

Пример

Функция pr_exit, представленная в листинге 8.3, использует макросы из табл. 8.1 для вывода сведений, полученных из кода завершения. В этой книге мы будем использовать ее во многих примерах. Обратите внимание: эта функция обращается к макросу WCOREDUMP, если он определен в системе.

Листинг 8.3. Вывод сведений, полученных из кода завершения

```
#include "apue.h"
#include <sys/wait.h>

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("нормальное завершение, код выхода = %d\n",
               WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("аварийное завершение, номер сигнала = %d%s\n",
               WTERMSIG(status),
#ifdef WCOREDUMP
               WCOREDUMP(status) ? " (создан файл core)" : "");
#else
               "");
#endif
    else if (WIFSTOPPED(status))
        printf("дочерний процесс остановлен, номер сигнала = %d\n",
               WSTOPSIG(status));
}
```

FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3 и Solaris 9 поддерживают макроопределение WCOREDUMP.

Программа из листинга 8.4 демонстрирует интерпретацию различных значений кода завершения с помощью функции pr_exit. Запустив эту программу, мы получим следующие результаты:

```
$ ./a.out
нормальное завершение, код выхода = 7
аварийное завершение, номер сигнала = 6 (создан файл core)
аварийное завершение, номер сигнала = 8 (создан файл core)
```

К сожалению, не существует переносимого способа преобразования номеров сигналов, полученных от WTERMSIG, в описательные имена. (Один из возможных способов приводится в разделе 10.21.) Чтобы убедиться в том, что сигнал SIGABRT имеет значение 6, а сигнал SIGFPE – значение 8, приходится заглядывать в файл <signal.h>.

Как уже говорилось ранее, если родительский процесс имеет несколько процессов-потомков, то функция wait вернет управление по завершении любого из них. А что делать, если мы хотим дождаться завершения конкретного дочернего процесса (при условии, что нам известен его идентификатор)? В ранних версиях UNIX приходилось вызывать функцию wait и сравнивать возвращаемый ею идентификатор процесса с тем, который нас интересует. Если завершившийся процесс оказывался не тем, который мы ожидали, приходилось сохранять идентификатор процесса и код его завершения в отдельном списке и снова вызывать функцию wait. Эту операцию надо было повторять до тех пор, пока не завершится желаемый процесс. Если после этого нужно было дождаться завершения другого процесса, мы вынуждены были сначала просмотреть список уже завершившихся процессов, и если его в этом списке не было, вызывать функцию wait. Таким образом, возникла потребность в функции, которая ожидала бы завершения конкретного процесса. Эта функциональность (и даже больше) заложена в функцию waitpid, которая определена стандартом POSIX.1.

Листинг 8.4. Интерпретация различных кодов завершения

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t pid;
    int status;

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid == 0)          /* дочерний процесс */
        exit(7);

    if (wait(&status) != pid)   /* дождаться завершения дочернего процесса */
        err_sys("ошибка вызова функции wait");
    pr_exit(status);            /* и вывести код завершения */

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid == 0)          /* дочерний процесс */
        abort();                /* послать сигнал SIGABRT */

    if (wait(&status) != pid)   /* дождаться завершения дочернего процесса */
        err_sys("ошибка вызова функции wait");
    pr_exit(status);            /* и вывести код завершения */

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid == 0)          /* дочерний процесс */
        status /= 0;            /* деление на 0 сгенерирует сигнал SIGFPE */

    if (wait(&status) != pid)   /* дождаться завершения дочернего процесса */
        err_sys("ошибка вызова функции wait");
```

```
        pr_exit(status);            /* и вывести код завершения */
        exit(0);
    }
```

Интерпретация функцией `waitpid` аргумента *pid* зависит от его значения:

`pid == -1` Ожидает завершения любого дочернего процесса. В данном случае функция `waitpid` эквивалентна функции `wait`.

`pid > 0` Ожидает завершения процесса с идентификатором *pid*.

`pid == 0` Ожидает завершения любого дочернего процесса, идентификатор группы процессов которого тот же, что у вызывающего процесса (группы процессов обсуждаются в разделе 9.4).

`pid < -1` Ожидает завершения любого дочернего процесса, идентификатор группы процессов которого совпадает с абсолютным значением *pid*.

Функция `waitpid` возвращает идентификатор завершившегося дочернего процесса и сохраняет его код завершения по адресу, на который указывает аргумент *statloc*. Ошибка функции `wait` возникает только в том случае, когда у процесса нет потомков. (Еще одна ошибочная ситуация возможна, если выполнение функции было прервано сигналом. Мы обсудим этот вариант в главе 10.) Функция `waitpid`, кроме того, может завершиться ошибкой, если заданный процесс или группа процессов не существуют или не являются потомками вызывающего процесса.

Аргумент *options* позволяет управлять поведением функции `waitpid`. Он может содержать 0 или значение, полученное в результате поразрядной операции ИЛИ (OR) из констант, приведенных в табл. 8.2.

Таблица 8.2. Константы для аргумента options функции waitpid

Константа	Описание
`WCONTINUED`	Если реализация поддерживает управление заданиями, функция `waitpid` вернет код состояния потомка, определяемого аргументом *pid*, который возобновил работу после остановки и код состояния которого еще не был получен (расширение XSI стандарта POSIX.1).
`WNOHANG`	Функция `waitpid` не будет блокировать вызывающий процесс, если потомок, определяемый аргументом *pid*, еще не изменил свое состояние. В этом случае функция возвращает значение 0.
`WUNTRACED`	Если реализация поддерживает управление заданиями, функция `waitpid` вернет код состояния дочернего процесса, определяемого аргументом *pid*, который был остановлен и код состояния которого еще не был получен с момента остановки. Макрокоманда `WIFSTOPPED` позволяет определить, соответствует ли возвращаемое значение остановленному дочернему процессу.

Функция `waitpid` предоставляет три функциональные возможности, которых лишена функция `wait`.

1. Функция waitpid позволяет указать, завершения какого именно процесса необходимо дождаться, в то время как функция wait возвращает код состояния первого процесса-потомка, который завершит работу. Мы вернемся к обсуждению этой возможности, когда будем рассказывать о функции popen.
2. Функция waitpid предоставляет возможность отключения блокировки для тех случаев, когда мы хотим узнать состояние дочернего процесса, но не хотим, чтобы вызывающий процесс был заблокирован.
3. Функция waitpid поддерживает управление заданиями с помощью констант WUNTRACED и WCONTINUED.

Пример

Вернемся к обсуждению процессов-зомби в разделе 8.5. Если мы хотим, чтобы процесс, который создает потомка с помощью функции fork, не дожидался завершения дочернего процесса и при этом процесс-потомок не превращался бы в зомби до тех пор, пока родительский процесс не завершится, то мы должны вызвать функцию fork дважды. Этот прием использует программа, представленная листингом 8.5.

Листинг 8.5. Предотвращение появления зомби за счет двойного вызова функции fork

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t pid;

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) {    /* первый потомок */
        if ((pid = fork()) < 0)
            err_sys("ошибка вызова функции fork");
        else if (pid > 0)
            exit(0);          /* первый потомок, он же */
                              /* родительский процесс для второго потомка */
        /*
         * Здесь продолжает работу второй потомок, для которого
         * родительским стал процесс init, поскольку настоящий
         * родительский процесс вызвал функцию exit() чуть выше.
         * Теперь мы можем продолжать работу, зная,
         * что когда процесс завершится, его код завершения
         * будет получен процессом init.
         */
        sleep(2);
        printf("второй потомок, идентификатор родительского процесса = %d\n",
                getppid());
        exit(0);
```

```
    }

    if (waitpid(pid, NULL, 0) != pid) /* ожидать завершения первого потомка */
        err_sys("ошибка вызова функции waitpid");
    /*
     * Здесь родительский (первоначальный) процесс продолжает работу,
     * поскольку он не является родительским
     * процессом для второго потомка.
     */
    exit(0);
}
```

Мы приостановили работу второго потомка на две секунды, чтобы гарантировать, что первый потомок завершит свою работу до того, как будет выведен идентификатор родительского процесса. Функция `fork` вернет управление как родительскому, так и дочернему процессу, но мы никогда заранее не знаем, кто из них будет первым. Если бы второй дочерний процесс не был приостановлен и после вызова функции `fork` получил бы управление первым, то идентификатор процесса, выведенный функцией `printf`, был бы идентификатором первичного родительского процесса, а не процесса `init`.

В результате запуска программы из листинга 8.5 мы получили

```
$ ./a.out
$ второй потомок, идентификатор родительского процесса = 1
```

Обратите внимание: командная оболочка вывела приглашение (символ `$`), как только первичный процесс завершил свою работу, то есть еще до того, как второй потомок вывел идентификатор своего родительского процесса.

8.7. Функция waitid

Расширения XSI стандарта Single UNIX Specification включают дополнительную функцию, которая может получить код выхода процесса. Функция `waitid` очень похожа на функцию `waitpid`, но предоставляет дополнительные возможности.

```
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Подобно функции `waitpid`, функция `waitid` позволяет процессу указать, завершения какого из потомков необходимо дождаться. Вместо того чтобы передавать эту информацию вместе с идентификатором процесса или группы процессов в закодированном виде через единственный аргумент, функция *`waitid`* предоставляет два отдельных аргумента. Значение аргумента *`id`* интерпретируется в зависимости от значения аргумента *`idtype`*. Возможные значения этого аргумента приводятся в табл. 8.3.

Таблица 8.3 Возможные значения аргумента idtype функции waitid

Константа	Описание
P_PID	Ожидать завершения конкретного дочернего процесса. Аргумент *id* содержит его идентификатор.
P_PGID	Ожидать завершения любого дочернего процесса, принадлежащего к указанной группе процессов. Аргумент *id* содержит идентификатор группы процессов.
P_ALL	Ожидать завершения любого дочернего процесса. Содержимое аргумента *id* игнорируется.

Аргумент *options* содержит значение, полученное в результате поразрядной операции ИЛИ (OR) из констант, приведенных в табл. 8.4. Эти константы определяют, какие изменения состояния дочернего процесса интересуют вызывающий процесс.

Таблица 8.4. Константы для аргумента options функции waitid

Константа	Описание
WCONTINUED	Ожидать завершения процесса, который возобновил работу после остановки и код состояния которого еще не был получен.
WEXITED	Получить информацию о состоянии завершившегося процесса.
WNOHANG	Сразу же возвращать управление и не блокировать вызывающий процесс, если код выхода дочернего процесса недоступен.
WNOWAIT	Не уничтожать информацию о состоянии дочернего процесса, чтобы ее можно было затем получить с помощью функции wait, waitpid или waitid.
WSTOPPED	Ожидать завершения процесса, который был остановлен и код состояния которого еще не был получен.

Аргумент *infop* – это указатель на структуру siginfo. Эта структура содержит подробную информацию о сигнале, вызвавшем изменение состояния дочернего процесса. Структура siginfo будет рассмотрена в разделе 10.14.

> Из четырех платформ, обсуждаемых в данной книге, только ОС Solaris поддерживает функцию waitid.

8.8. Функции wait3 и wait4

Большинство реализаций UNIX предоставляют две дополнительные функции: wait3 и wait4. Эти функции впервые появились в ветке BSD. Единственное их преимущество перед функциями wait, waitid и waitpid заключается в том, что они предоставляют дополнительный аргумент, через который ядро может вернуть краткую справку о ресурсах, использованных завершившимся процессом и всеми его дочерними процессами.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Обе возвращают идентификатор процесса
в случае успеха, 0 или −1 в случае ошибки

Информация об использованных ресурсах включает такие сведения, как количество процессорного времени (пользовательского и системного), неудачных попыток обращений к страницам виртуальной памяти, принятых сигналов и тому подобное. За подробностями обращайтесь к странице справочного руководства `getrusage`(2). (Эта информация о ресурсах отличается от ограничений на ресурсы, которые обсуждались в разделе 7.11.) В табл. 8.5 приводится информация о различных аргументах, поддерживаемых функциями семейства `wait`.

Таблица 8.5. Аргументы, поддерживаемые функциями семейства wait на различных платформах

Функция	*pid*	*options*	*rusage*	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
wait				•	•	•	•	•
waitid	•	•		XSI				•
waitpid	•	•		•	•	•	•	•
wait3		•	•		•	•	•	•
wait4	•	•	•		•	•	•	•

Функция `wait3` была включена в ранние версии стандарта Single UNIX Specification. Во второй версии стандарта она была переведена в разряд устаревших, а в третьей версии вообще исключена из стандарта.

8.9. Гонка за ресурсами

Мы будем называть гонкой за ресурсами состояние, которое возникает в том случае, когда несколько процессов пытаются одновременно производить некоторые действия с данными, находящимися в совместном использовании, и конечный результат зависит от порядка, в котором эти процессы выполняются. Функция `fork` являет собой яркий пример потенциального источника проблем, связанных с гонкой за ресурсами, если логика выполнения программы явно или неявно зависит от того, кто первым получит управление – родительский процесс или дочерний. Вообще, это невозможно заранее предсказать. Но даже если бы мы и знали наверняка, какой из процессов первым

получит управление, то все равно дальнейшая работа процесса зависит от степени нагрузки на систему и алгоритма планирования, заложенного в ядре.

Мы уже встречались с потенциальной ситуацией гонки за ресурсами в программе из листинга 8.5, когда второй потомок выводил идентификатор родительского процесса. Если второй потомок получит управление раньше первого, то его родительским процессом будет первый потомок. Но если сначала получит управление первый потомок и у него будет достаточно времени, чтобы успеть завершиться, то родительским процессом для второго потомка станет процесс `init`. Даже вызов функции `sleep`, который использовался в нашем примере, не может гарантировать, что процессы будут выполняться в заданном порядке. Если система сильно загружена, то даже после двухсекундной задержки второй потомок может получить управление раньше, чем первому потомку удастся завершить свою работу. Проблемы такого рода очень сложны в отладке, потому что в большинстве случаев они себя не проявляют.

Процесс, который желает дождаться завершения дочернего процесса, должен вызвать одну из функций семейства `wait`. Если же процесс хочет дождаться завершения родительского процесса, как в программе из листинга 8.5, то можно было бы воспользоваться примерно таким циклом:

```
while (getppid() != 1)
    sleep(1);
```

Однако этот цикл, который называется *опросом* (*polling*), порождает еще одну проблему. Дело в том, что процесс непроизводительно расходует процессорное время, так как вызывающий процесс возобновляет работу каждую секунду, чтобы проверить истинность условия.

Чтобы не попасть в гонку за ресурсами и избежать применения опроса, необходимо нечто вроде обмена сигналами между процессами. Для этой цели можно использовать сигналы, и один такой способ будет описан в разделе 10.16. Также могут использоваться различные виды межпроцессного взаимодействия (interprocess communication – IPC). Некоторые из них мы рассмотрим в главах 15 и 17.

Для организации взаимоотношений между родительским и дочерним процессами часто используется следующий сценарий. После вызова функции `fork` оба процесса, родительский и дочерний, выполняют некоторые действия. Например, родительский процесс может добавить запись с идентификатором потомка в файл журнала, а потомок может создать файл для родительского процесса. В таком случае требуется, чтобы каждый из процессов имел возможность известить другой процесс о завершении определенных начальных операций и дождался бы завершения этих операций другим процессом, прежде чем продолжить работу. Следующий код иллюстрирует этот сценарий:

```
#include "apue.h"

TELL_WAIT(); /* выполнить подготовительные операции для TELL_xxx и WAIT_xxx */
```

```
if ((pid = fork()) < 0) {
    err_sys("ошибка вызова функции fork");
} else if (pid == 0) {      /* дочерний процесс */

    /* дочерний процесс выполняет необходимые действия ... */

    TELL_PARENT(getppid()); /* сообщить родительскому процессу */
                            /* о завершении подготовительных операций */
    WAIT_PARENT();          /* и дождаться ответа родительского процесса */

    /* потомок продолжает работу самостоятельно ... */

    exit(0);
}

/* родительский процесс выполняет необходимые действия ... */

TELL_CHILD(pid);            /* сообщить дочернему процессу */
                            /* о завершении подготовительных операций */
WAIT_CHILD();               /* и дождаться ответа дочернего процесса */

/* родительский процесс продолжает работу самостоятельно ... */

exit(0);
```

Здесь мы исходим из предположения, что все необходимые определения находятся в заголовочном файле `apue.h`. Пять процедур – `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT` и `WAIT_CHILD` – должны быть оформлены в виде функций или макроопределений.

Мы покажем различные варианты реализации процедур `TELL` и `WAIT` в последующих главах: в разделе 10.16 продемонстрируем реализацию на основе сигналов, а в листинге 15.3 – на основе неименованных каналов. А теперь рассмотрим пример, в котором используются эти пять процедур.

Пример

Программа, представленная листингом 8.6, выводит две строки: одна формируется дочерним процессом, а другая – родительским. Программа содержит гонку за ресурсами, поскольку порядок вывода символов строк зависит от того, какой процесс получает управление и как долго он работает.

Листинг 8.6. Программа, содержащая гонку за ресурсами

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t pid;

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) {
        charatatime("от дочернего процесса\n");
```

```
    } else {
        charatatime("от родительского процесса\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char *ptr;
    int c;

    setbuf(stdout, NULL); /* установить небуферизованный режим */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Мы установили небуферизованный режим для стандартного потока вывода, поэтому запись каждого символа сопровождается вызовом функции `write`. Это сделано для того, чтобы ядро могло производить переключение процессов настолько часто, насколько это возможно. Таким образом создается ситуация гонки за ресурсами. (Если этого не сделать, то, быть может, мы никогда и не увидим результаты, показанные ниже. Но если мы их не видим, это не значит, что гонки за ресурсами не существует; это лишь означает, что мы не наблюдаем ее в данной конкретной системе.) Ниже приводится вывод, действительно полученный от программы:

```
$ ./a.out
оот дочернего процесса
т родительского процесса
$ ./a.out
оот дочернего процесса
т родительского процесса
$ ./a.out
от дочернего процесса
от родительского процесса
```

А теперь изменим программу из листинга 8.6 так, чтобы она использовала функции `TELL` и `WAIT`. Эти изменения представлены в листинге 8.7. Добавленные строки отмечены символом «+».

Листинг 8.7. Модификация программы из листинга 8.6, позволяющая избежать гонки за ресурсами

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t pid;

+   TELL_WAIT();
```

```
+
    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) {
+       WAIT_PARENT();  /* родительский процесс стартует первым */
        charatatime("от дочернего процесса\n");
    } else {
        charatatime("от родительского процесса\n");
+       TELL_CHILD(pid);
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char *ptr;
    int c;

    setbuf(stdout, NULL); /* установить небуферизованный режим */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Запустив эту программу, мы получим то, что и ожидали – символы, выводимые двумя процессами, более не смешиваются.

В программе из листинга 8.7 родительский процесс стартует первым. Если требуется, чтобы первым стартовал дочерний процесс, то нужно изменить строки, которые следуют после вызова функции `fork`:

```
} else if (pid == 0) {
    charatatime("от дочернего процесса\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD(); /* дочерний процесс стартует первым */
    charatatime("от родительского процесса\n");
}
```

В упражнении 8.3 будет продолжено обсуждение этого примера.

8.10. Функция exec

Мы уже говорили в разделе 8.3, что функция `fork` часто используется для создания нового процесса, который затем запускает другую программу с помощью одной из функций семейства `exec`. Когда процесс вызывает одну из функций `exec`, то он полностью замещается другой программой, и эта новая программа начинает выполнение собственной функции `main`. Идентификатор процесса при этом не изменяется, поскольку функция `exec` не создает новый процесс, она просто замещает текущий процесс – его сегмент кода, сегмент данных, динамическую область памяти и сегмент стека – другой программой.

Существует шесть различных функций exec, но мы обычно будем говорить просто о «функции exec», подразумевая любую из них. Эти шесть функций завершают список примитивов UNIX, предназначенных для управления процессами. С помощью функции fork можно создавать новые процессы, с помощью функций exec – запускать новые программы. Функция exit и функции семейства wait обслуживают процедуры выхода и ожидания завершения. Эти примитивы – единственное, что необходимо для управления процессами. Мы будем использовать их в последующих разделах для создания дополнительных функций, таких как popen и system.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);
```

Все шесть функций возвращают –1 в случае ошибки, не возвращают управление в случае успеха

Одно из отличий между этими функциями заключается в том, что первые четыре принимают в качестве аргумента полный путь к файлу, а последние две – только имя файла. Аргумент *filename* интерпретируется следующим образом:

- Если аргумент *filename* содержит символ слэша, он интерпретируется как полный путь к файлу.
- В противном случае производится поиск исполняемого файла в каталогах, перечисленных в переменной окружения PATH.

Переменная окружения PATH содержит список каталогов, разделенных двоеточиями; они называются префиксами пути. Например, строка окружения в формате *name=value*

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

определяет четыре каталога, в которых будет производиться поиск исполняемых файлов. Последним указан текущий каталог. (Пустой префикс также означает текущий каталог. Он может быть определен двоеточием в начале, двумя двоеточиями в середине или двоеточием в конце подстроки *value*.)

> По причинам, связанным с безопасностью системы, никогда не включайте текущий каталог в переменную окружения PATH. Подробности в [Garfinkel et al. 2003].

Если функция execlp или execvp находит исполняемый файл, используя один из префиксов пути, но этот файл не является двоичным исполняемым фай-

лом, сгенерированным редактором связей, то функция предполагает, что найденный файл является сценарием командной оболочки и пытается вызывать /bin/sh с именем файла в качестве аргумента.

Следующее различие касается передачи списка аргументов (l означает список (list), v означает вектор, или массив (vector)). Функции требуют, чтобы каждый из аргументов командной строки новой программы был оформлен в виде отдельного аргумента функции. Конец списка аргументов отмечается пустым указателем. Для других трех функций (execv, execvp и execve) необходимо сформировать массив указателей на аргументы командной строки и передать адрес этого массива в качестве аргумента.

До появления прототипов ISO C было принято показывать аргументы командной строки, передаваемые функциям execl, execlp и execle, следующим образом:

```
char *arg0, char *arg1, ..., char *argn, (char *)0
```

В таком прототипе ясно видно, что заключительный аргумент функции является пустым указателем. Если этот пустой указатель задается как 0, то мы должны явно привести его к типу указателя; если этого не сделать, он будет интерпретироваться как целочисленный аргумент. Если при этом размер целочисленного типа не будет совпадать с размером типа char *, то фактически функция exec получит неверные аргументы.

И последнее различие – передача списка переменных окружения новой программе. Две функции, имена которых оканчиваются на e (execle и execve), позволяют передавать массив указателей на строки окружения. Остальные четыре функции для передачи копии среды окружения новой программе используют переменную environ вызывающего процесса. (Вспомните обсуждение строк окружения в разделе 7.9 и загляните в табл. 7.2. Там мы упоминали, что если система поддерживает функции setenv и putenv, то можно изменить текущую среду окружения и среду окружения любых дочерних процессов, но нельзя изменить среду окружения родительского процесса.) Обычно среда окружения процесса передается дочерним процессам без изменения, но в некоторых случаях возникает необходимость создать особую среду окружения для дочернего процесса. Пример такого случая – программа login, которая инициализирует новую командную оболочку. Обычно программа login создает определенную среду окружения с небольшим количеством переменных и позволяет нам через файл начального запуска командной оболочки добавить свои переменные окружения при входе в систему.

До появления прототипов ISO C, аргументы функции execle принято было показывать следующим образом:

```
char *pathname, char *arg0, ..., char *argn, (char *)0, char *envp[]
```

В таком прототипе ясно видно, что заключительный аргумент функции является адресом массива указателей на строки окружения. Прототипы стандарта ISO C не показывают этого, в них все аргументы командной строки, пустой указатель и указатель *envp* заменяются многоточием (...).

Аргументы всех шести функций семейства exec достаточно сложно запомнить. Но буквы в именах функций немного помогают в этом. Буква p означает, что функция принимает аргумент *filename* и использует переменную окружения PATH, чтобы найти исполняемый файл. Буква l означает, что функция принимает список аргументов, а буква v означает, что она принимает массив (вектор) *argv[]*. Наконец, буква e означает, что функция принимает массив *envp[]* вместо использования текущей среды окружения. В табл. 8.6 показаны различия между этими шестью функциями.

Таблица 8.6. Различия между шестью функциями семейства exec

Функция	*pathname*	*filename*	**Список аргументов**	*argv[]*	*environ*	*envp[]*
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
Буква в имени		p	l	v		e

Каждая система накладывает свои ограничения на размер списка аргументов командной строки и списка переменных окружения. Из раздела 2.5.2 и табл. 2.8 следует, что этот предел задается с помощью константы ARG_MAX. Для POSIX.1-совместимых систем его значение должно быть не менее 4096 байт. Иногда приходится сталкиваться с этим пределом при использовании масок командного интерпретатора для создания списка файлов. Например, в некоторых системах команда

```
grep getrlimit /usr/share/man/*/*
```

может выдать сообщение об ошибке

```
Argument list too long
```

то есть «список аргументов слишком велик».

> В ранних версиях System V этот предел составлял 5120 байт. Ранние версии BSD имели предел 20480 байт. В современных системах этот предел намного выше. (См. данные вывода программы из листинга 2.2, приведенные в табл. 2.12.)

Чтобы обойти ограничение на размер списка, можно воспользоваться командой xargs(1), которая способна обрабатывать списки аргументов большого размера. Например, чтобы отыскать все вхождения слова getrlimit в страницах справочного руководства вашей системы, можно использовать такую команду:

```
find /usr/share/man -type f -print | xargs grep getrlimit
```

Однако если файлы со страницами справочного руководства сжаты, то лучше попробовать так:

```
find /usr/share/man -type f -print | xargs bzgrep getrlimit
```

Мы использовали опцию `-type f` команды `find`, чтобы ограничить список обычными файлами, поскольку команды `grep` не способны производить поиск по шаблону в каталогах, и мы хотим избежать ненужных сообщений об ошибках.

Уже было отмечено, что идентификатор процесса не изменяется после вызова функции `exec`. Кроме того, новая программа наследует от вызывающего процесса ряд дополнительных характеристик:

- Идентификатор процесса и идентификатор родительского процесса
- Реальный идентификатор пользователя и реальный идентификатор группы
- Идентификаторы дополнительных групп
- Идентификатор группы процессов
- Идентификатор сессии
- Управляющий терминал
- Время, оставшееся до срабатывания таймера
- Текущий рабочий каталог
- Маску режима создания файлов
- Блокировки файлов
- Маску сигналов процесса
- Сигналы, ожидающие обработки
- Ограничения на ресурсы
- Значения `tms_utime`, `tms_stime`, `tms_cutime` и `tms_cstime`

Судьба открытых файлов зависит от значения флага close-on-exec (закрыть-при-вызове-exec) для каждого дескриптора. Вспомните рис. 3.1 и упоминание флага `FD_CLOEXEC` в разделе 3.14. Там мы говорили, что каждый открытый процессом дескриптор имеет флаг close-on-exec. Если этот флаг установлен, дескриптор закрывается функцией `exec`. В противном случае дескриптор остается открытым. По умолчанию после вызова функции `exec` дескриптор остается открытым, если флаг close-on-exec не был специально установлен с помощью функции `fcntl`.

Стандарт POSIX.1 требует, чтобы открытые каталоги (вспомните функцию `opendir` из раздела 4.21) обязательно закрывались при вызове функции `exec`. Обычно это обеспечивает функция `opendir`, которая вызывает `fcntl`, чтобы установить флаг close-on-exec для дескриптора, соответствующего открытому файлу каталога.

Обратите внимание, что реальные идентификаторы пользователя и группы не изменяются при вызове функции `exec`, но эффективные идентификаторы могут быть изменены в зависимости от состояния битов set-user-ID и set-group-ID файла запускаемой программы. Если бит set-user-ID установлен, то в качестве эффективного идентификатора пользователя процесса прини-

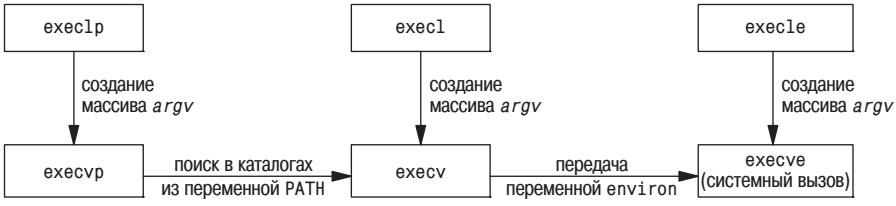


***Рис. 8.2.** Взаимоотношения между шестью функциями exec*

мается идентификатор владельца файла программы. В противном случае эффективный идентификатор пользователя не изменяется (он не устанавливается равным реальному идентификатору пользователя). Эффективный идентификатор группы устанавливается аналогичным образом.

В большинстве реализаций UNIX только одна из этих шести функций, `execve`, является системным вызовом. Остальные пять – обычные библиотечные функции, которые в конечном счете обращаются к этому системному вызову. На рис. 8.2 изображена схема взаимоотношений между шестью функциями `exec`.

В соответствии с этой схемой, библиотечные функции `execlp` и `execvp` обрабатывают переменную окружения `PATH` в поисках первого каталога, который содержит исполняемый файл с именем *`filename`*.

Пример

Программа, представленная листингом 8.8, демонстрирует работу с функциями `exec`.

Листинг 8.8. Пример использования функций exec

```
#include "apue.h"
#include <sys/wait.h>

char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t pid;

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) { /* задать полный путь к файлу и среду окружения */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                   "MY ARG2", (char *)0, env_init) < 0)
            err_sys("ошибка вызова функции execle");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("ошибка вызова функции wait");

    if ((pid = fork()) < 0) {
```

```
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) { /* задать имя файла, наследовать среду окружения */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("ошибка вызова функции execlp");
    }

    exit(0);
}
```

Сначала мы вызываем функцию `execle`, которая требует указать полный путь к файлу и среду окружения. Далее вызывается функция `execlp`, которой передается имя файла, а среда окружения наследуется новой программой от вызывающего процесса. В данном примере обращение к функции `execlp` не завершается ошибкой по той единственной причине, что каталог `/home/sar/bin` входит в переменную `PATH`. Кроме того, обратите внимание, что в качестве первого аргумента (`argv[0]`) командной строки новой программы мы передаем только имя файла. Некоторые командные оболочки передают в этом аргументе полный путь к файлу. Но это всего лишь соглашение, на самом деле в `argv[0]` можно записать любую строку. Команда `login` именно так и поступает, когда запускает командную оболочку. Перед ее запуском `login` добавляет в начало строки *argv[0]* дефис, тем самым сообщая командной оболочке, что она вызывается как оболочка входа в систему. В этом случае она производит запуск команд начальной настройки, в то время как при обычном вызове командная оболочка этого не делает.

Программа `echoall`, дважды запускаемая программой из листинга 8.8, приведена в листинге 8.9. Это простенькая программа, которая выводит все аргументы командной строки и список переменных окружения.

Листинг 8.9. Выводит все аргументы командной строки и переменные окружения

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int i;
    char **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++) /* вывести все аргументы командной строки */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++) /* и все переменные окружения */
        printf("%s\n", *ptr);

    exit(0);
}
```

После запуска программы из листинга 8.9 мы получили следующие результаты:

```
$ ./a.out
argv[0]: echoall
```

```
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
```

еще 47 строк здесь не показаны

```
HOME=/home/sar
```

Обратите внимание, что приглашение командной оболочки появилось перед выводом значения `argv[0]` для второго вызова функции `exec`. Произошло это потому, что родительский процесс не стал ждать, пока этот потомок завершит свою работу.

8.11. Изменение идентификаторов пользователя и группы

В системе UNIX предоставление привилегий (таких как возможность изменять текущую дату) и управление доступом к файлам (например, право на чтение или запись) основаны на идентификаторах пользователя и группы. Когда программе необходимы дополнительные привилегии, чтобы получить доступ к ресурсам, недоступным в настоящее время, она должна изменить свой идентификатор пользователя или группы на идентификатор, который имеет соответствующие привилегии. Точно так же, чтобы понизить свои привилегии или предотвратить доступ к некоторым ресурсам, программа должна изменить идентификатор пользователя или группы на идентификатор, не обладающий указанными привилегиями или достаточными правами для обращения к ресурсу.

Вообще при разработке приложений следует использовать принцип наименьших привилегий. Следуя этому принципу, приложения должны использовать минимальный набор привилегий, необходимый для выполнения возложенных на них задач. Это уменьшает вероятность того, что злоумышленник сможет «обмануть» систему безопасности, используя программы и их привилегии непредусмотренным способом.

Изменить реальный и эффективный идентификаторы пользователя можно с помощью функции `setuid`. Точно так же можно изменить реальный и эффективный идентификаторы группы с помощью функции `setgid`.

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

Обе возвращают 0 в случае успеха, −1 в случае ошибки

Существуют определенные правила, согласно которым изменяются идентификаторы. Рассмотрим их на примере идентификатора пользователя. (Все перечисленное ниже в равной степени относится и к идентификатору группы.)

1. Если процесс обладает привилегиями суперпользователя, функция `setuid` устанавливает реальный, эффективный и сохраненный идентификаторы пользователя в соответствии с аргументом *uid*.
2. Если процесс не обладает привилегиями суперпользователя, но аргумент *uid* совпадает с реальным или сохраненным идентификатором пользователя, то `setuid` изменяет только эффективный идентификатор. Реальный и сохраненный идентификаторы не меняются.
3. Если ни одно из этих условий не соблюдено, `setuid` возвращает значение −1 и записывает в переменную `errno` код ошибки `EPERM`.

Здесь мы предполагаем, что конфигурационный параметр `_POSIX_SAVED_IDS` имеет значение `true`. Если эта функциональная возможность не предоставляется вашей системой, то исключите из вышеприведенных правил упоминание о сохраненном идентификаторе.

> Сохраненные идентификаторы стали обязательными для реализации в версии POSIX.1 от 2001 года. В более ранних версиях POSIX эта функциональная особенность относилась к разряду необязательных. Чтобы узнать, поддерживается ли она системой, приложение может проверить константу `_POSIX_SAVED_IDS` во время компиляции или вызвать функцию `sysconf` с аргументом `_SC_SAVED_IDS` во время выполнения.

Можно сформулировать несколько правил относительно трех идентификаторов пользователя.

1. Изменить реальный идентификатор пользователя может только процесс, обладающий привилегиями суперпользователя. Как правило, реальный идентификатор пользователя устанавливается программой `login`(1) при входе в систему и никогда не изменяется. Поскольку `login` является процессом, обладающим привилегиями суперпользователя, с помощью функции `setuid` он устанавливает все три идентификатора пользователя.
2. Эффективный идентификатор пользователя устанавливается функцией `exec` только в том случае, когда файл программы имеет установленный бит set-user-ID. Если этот бит не установлен, функция `exec` не изменяет эффективный идентификатор пользователя. В любой момент времени можно вызвать функцию `setuid`, чтобы установить эффективный идентификатор равным реальному или сохраненному идентификатору. Но, как правило, нельзя установить эффективный идентификатор пользователя в произвольное значение.
3. Функция `exec` копирует эффективный идентификатор пользователя в сохраненный идентификатор. Если файл программы имеет установленный бит set-user-ID, то эта копия сохраняется после того, как функция `exec` установит эффективный идентификатор равным идентификатору владельца файла.

В табл. 8.7 обобщаются возможные варианты изменения этих трех идентификаторов.

Таблица 8.7. Варианты изменения идентификаторов пользователя

Идентификатор	exec		setuid(*uid*)	
	Бит set-user-ID выключен	**Бит set-user-ID включен**	**Суперпользователь**	**Непривилегированный пользователь**
Реальный	Не изменяется	Не изменяется	Устанавливается в соответствии с *uid*	Не изменяется
Эффективный	Не изменяется	Устанавливается в соответствии с идентификатором владельца файла программы	Устанавливается в соответствии с *uid*	Устанавливается в соответствии с *uid*
Сохраненный	Копия эффективного идентификатора	Копия эффективного идентификатора	Устанавливается в соответствии с *uid*	Не изменяется

Обратите внимание, что с помощью функций getuid и geteuid, описанных в разделе 8.2, можно получить только текущие значения реального и эффективного идентификаторов пользователя. У нас нет возможности получить текущее значение сохраненного идентификатора.

Пример

Чтобы увидеть, где может пригодиться сохраненный идентификатор пользователя, рассмотрим программу, которая его использует. В качестве примера возьмем утилиту man(1), которая выводит на экран страницы справочного руководства. Как правило, файл программы man имеет установленный бит set-user-ID или set-group-ID, и его владельцем является специальный пользователь или группа, обычно зарезервированные для этой программы. Программа man может быть настроена таким образом, что она имеет возможность читать и, возможно, перезаписывать определенные файлы, местоположение которых определяется либо в конфигурационных файлах (обычно /etc/man.config или /etc/manpath.config), либо с помощью аргументов командной строки.

Программе man иногда приходится вызывать ряд других команд для обработки файлов, содержащих страницы справочного руководства. Чтобы предотвратить возможность выполнения посторонних команд или перезаписи посторонних файлов, команда man должна переключаться между двумя наборами привилегий: привилегиями пользователя, запустившего команду, и привилегиями владельца исполняемого файла man. Это происходит следующим образом:

1. Допустим, что исполняемый файл man имеет установленный бит set-user-ID и его владельцем является пользователь man. Когда функция exec запускает эту программу, мы получаем:

   реальный идентификатор пользователя = идентификатору пользователя, запустившего программу

эффективный идентификатор пользователя = `man`

сохраненный идентификатор пользователя = `man`

2. Программе `man` необходим доступ к конфигурационным файлам и файлам справочного руководства. Владельцем этих файлов является пользователь `man`, а поскольку эффективный идентификатор пользователя процесса тоже `man`, то доступ к файлам разрешен.
3. Прежде чем запустить какую-либо команду от имени пользователя, утилита `man` вызывает функцию `setuid(getuid())`. Поскольку процесс `man` не обладает привилегиями суперпользователя, то изменяется только эффективный идентификатор пользователя. В результате мы получаем:

реальный идентификатор пользователя = идентификатору пользователя, запустившего программу (не изменился)

эффективный идентификатор пользователя = идентификатору пользователя, запустившего программу

сохраненный идентификатор пользователя = `man` (не изменился)

   Теперь процесс `man` работает с эффективным идентификатором пользователя, равным идентификатору пользователя, его запустившего. Это означает, что пользователь может обратиться только к тем файлам, к которым он имеет право доступа. Он не получает никаких дополнительных привилегий. В результате программа `man` может достаточно безопасно вызвать любую команду-фильтр от имени пользователя.
4. По окончании работы команды-фильтра `man` вызывает `setuid(`*`euid`*`)`, где *`euid`* – идентификатор пользователя с именем `man`. (Он был сохранен программой `man` при помощи функции `geteuid`.) Это вполне допустимо, поскольку аргумент функции `setuid` совпадает с сохраненным идентификатором. (Именно для этого и нужен сохраненный идентификатор.) Теперь мы имеем:

реальный идентификатор пользователя = идентификатор пользователя, запустившего программу (не изменился)

эффективный идентификатор пользователя = `man`

сохраненный идентификатор пользователя = `man` (не изменился)

5. После этого программа `man` снова может работать со своими файлами, поскольку ее эффективный идентификатор пользователя стал равным идентификатору пользователя `man`.

Благодаря сохраненному идентификатору мы можем в начале и в конце работы процесса пользоваться дополнительными привилегиями, которые дает нам установленный бит set-user-ID. Однако все остальное время процесс выполняется с привилегиями обычного пользователя. Если бы отсутствовала

возможность вернуться к сохраненному идентификатору, то, вероятно, нам пришлось бы позволить процессу постоянно работать с повышенными привилегиями, что чревато определенными проблемами.

Теперь посмотрим, что произойдет, если man запустит командную оболочку. (Командная оболочка запускается с помощью функций fork и exec.) Поскольку реальный и эффективный идентификаторы процесса man – это идентификаторы обычного пользователя (пункт 3), то запущенная командная оболочка не будет иметь дополнительных привилегий. Сохраненный идентификатор командной оболочки будет скопирован функцией exec из эффективного идентификатора. Таким образом, все три идентификатора дочернего процесса, запущенного с помощью функции exec, будут идентификаторами обычного пользователя.

Приведенное описание использования функции setuid утилитой man будет не совсем корректно, если владельцем исполняемого файла является суперпользователь, поскольку в этом случае функция setuid установит все три идентификатора пользователя. Чтобы все работало так, как мы описали, функция setuid должна изменять только эффективный идентификатор.

Функции setreuid и setregid

BSD-системы традиционно поддерживают возможность менять местами реальный и эффективный идентификаторы пользователя с помощью функции setreuid.

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);
```

Обе возвращают 0 в случае успеха, –1 в случае ошибки

Можно передать значение –1 в любом из аргументов, чтобы указать, что соответствующий идентификатор должен остаться неизменным.

Правило использования этих функций очень просто: непривилегированный пользователь всегда может поменять местами реальный и эффективный идентификаторы. Это позволяет программам с установленным битом set-user-ID переходить к привилегиям обычного пользователя и затем при необходимости возвращаться к привилегиям владельца файла программы. Когда в стандарте POSIX.1 появились сохраненные идентификаторы, это правило было расширено, чтобы позволить непривилегированному процессу устанавливать эффективный идентификатор пользователя в соответствии также с сохраненным идентификатором.

> Обе функции, setreuid и setregid, являются расширениями XSI стандарта Single UNIX Specifications. Таким образом, предполагается, что все версии UNIX должны обеспечивать поддержку этих функций.

В версии 4.3BSD отсутствовало понятие сохраненных идентификаторов, описанное выше. Вместо этого использовались функции `setreuid` и `setregid`. Это позволяло непривилегированному пользователю свободно переключаться между двумя идентификаторами. Однако когда программа, использовавшая эту возможность, запускала командную оболочку, она должна была перед вызовом функции `exec` устанавливать реальный идентификатор пользователя равным идентификатору обычного пользователя. Если этого не сделать, то реальный идентификатор может оказаться принадлежащим привилегированному пользователю (как результат вызова функции `setreuid`), и процесс, запущенный из такой командной оболочки, может с помощью `setreuid` поменять идентификаторы и получить более высокие привилегии. В качестве меры предосторожности и реальный, и эффективный идентификаторы перед вызовом функции `exec` в дочернем процессе устанавливались равными идентификатору обычного пользователя.

Функции seteuid и setegid

Стандарт POSIX.1 включает еще две функции: `seteuid` и `setegid`. Они очень похожи на функции `setuid` и `setgid`, но изменяют только эффективный идентификатор пользователя и группы.

```
#include <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);
```

Обе возвращают 0 в случае успеха, −1 в случае ошибки

Непривилегированный пользователь может установить свой эффективный идентификатор равным реальному или сохраненному идентификатору.

Рисунок 8.3 показывает все описанные нами функции, предназначенные для изменения трех идентификаторов пользователя.

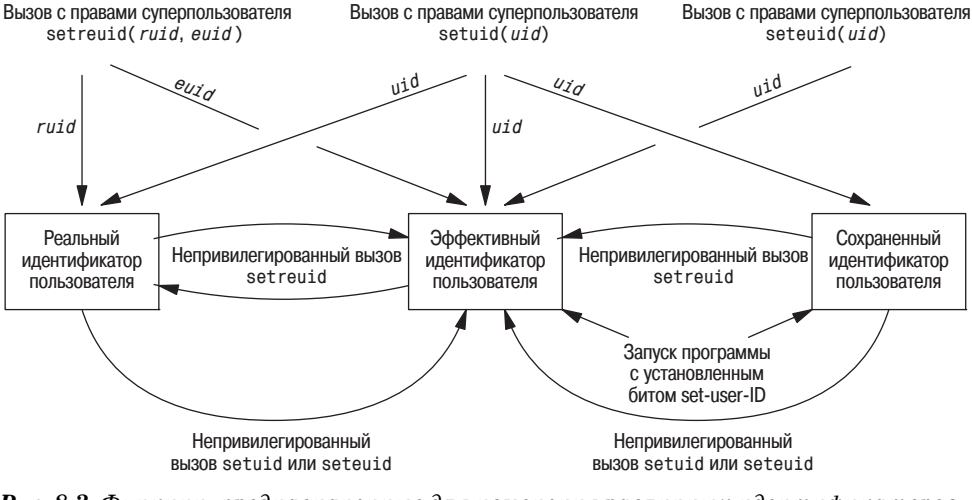


***Рис. 8.3.** Функции, предназначенные для изменения различных идентификаторов пользователя*

Идентификаторы группы

Все, о чем мы говорили в этом разделе, в равной степени относится и к идентификаторам группы. Функции `setgid`, `setregid` и `setegid` не оказывают влияния на идентификаторы дополнительных групп.

8.12. Интерпретируемые файлы

Все современные версии UNIX поддерживают интерпретируемые файлы. Это обычные текстовые файлы, которые начинаются со строки вида

```
#! pathname [optional-argument]
```

Пробел между восклицательным знаком и параметром *pathname* необязателен. Чаще всего интерпретируемые файлы начинаются со строки

```
#!/bin/sh
```

Строка *pathname* обычно представляет собой абсолютный путь к исполняемому файлу интерпретатора, поскольку никаких дополнительных операций над ней не производится (то есть переменная `PATH` не используется). Распознавание интерпретируемых файлов производится ядром в процессе выполнения системного вызова `exec`. В действительности ядро запускает на исполнение не сам интерпретируемый файл, а программу, указанную в параметре *pathname* в первой его строке. Необходимо понимать разницу между интерпретируемым файлом, который представляет собой текстовый файл, начинающийся с последовательности `#!`, и интерпретатором, то есть исполняемым файлом, путь к которому указывается в первой строке интерпретируемого файла.

Помните, что существует ограничение операционной системы на размер первой строки интерпретируемого файла. Это ограничение включает последовательность символов `#!`, параметр *pathname*, необязательные аргументы, завершающий символ перевода строки и все пробельные символы.

> В ОС FreeBSD 5.2.1 длина первой строки ограничивается 128 байтами, в Mac OS X 10.3 – 512 байтами, в Linux 2.4.22 – 127 байтами,а в Solaris 9 – 1023 байтами.

Пример

Давайте рассмотрим на следующем примере, что делает ядро с параметрами функции `exec`, если запускаемый файл является интерпретируемым файлом и в первой его строке имеется дополнительный аргумент. Программа, представленная листингом 8.10, осуществляет запуск интерпретируемого файла.

Листинг 8.10. Программа, запускающая интерпретируемый файл

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
```

```
    pid_t pid;

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) {                          /* дочерний процесс */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("ошибка вызова функции execl");
    }

    if (waitpid(pid, NULL, 0) < 0)                  /* родительский процесс */
        err_sys("waitpid error");

    exit(0);
}
```

Ниже приводится содержимое интерпретируемого файла из одной строки, запускаемого программой из листинга 8.10, и результаты работы программы.

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

Программа `echoarg` (интерпретатор) просто выводит все аргументы, переданные ей в командной строке (это программа из листинга 7.3). Обратите внимание, что когда ядро запускает интерпретатор, в качестве `argv[0]` используется полный путь к исполняемому файлу интерпретатора, `argv[1]` – необязательный аргумент, взятый из интерпретируемого файла, `argv[2]` представляет собой полный путь к файлу программы (`/home/sar/bin/testinterp`), а `argv[3]` и `argv[4]` – это второй и третий аргументы функции `execl` (`myarg1` и `MY ARG2`). Оба аргумента функции `execl`, `argv[1]` и `argv[2]`, смещаются вправо на две позиции. Обратите внимание: ядро берет аргумент *pathname* из вызова функции `execl` вместо первого аргумента (`testinterp`) исходя из предположения, что аргумент *pathname* содержит больше информации, чем первый аргумент.

Пример

Очень часто в качестве необязательного аргумента, следующего за именем интерпретатора, передается опция `-f` для тех программ, которые ее поддерживают. Так, например, программа `awk`(1) может быть запущена как

```
awk -f myfile
```

Таким образом ей сообщается, что текст программы на языке `awk` находится в файле `myfile`.

> Системы, происходящие от System V, часто содержат две версии языка `awk`. В этих системах `awk` часто называется old awk (старый awk) и соответствует оригинальной вер-

сии, распространявшейся в составе Version 7. В противоположность ему, nawk (new awk – новый awk) содержит многочисленные расширения и соответствует языку, описанному в [Aho, Kernighan, and Weinberger 1988]. Эта новая версия предоставляет доступ к аргументам командной строки. ОС Solaris 9 поддерживает обе версии.

Программа awk – это одна из утилит, включенных POSIX в стандарт 1003.2 (теперь это часть базовых спецификаций POSIX.1 в стандарте Single UNIX Specification). Она также основывается на языке, описанном в книге [Aho, Kernighan, and Weinberger 1988].

Версия awk в Mac OS X 10.3 основана на версии Bell Laboratories, которую компания Lucent сделала свободно распространяемой. В составе ОС FreeBSD 5.2.1 и Linux 2.4.22 распространяется утилита GNU awk, называемая gawk. В этих системах awk является символической ссылкой на gawk. Утилита gawk соответствует стандарту POSIX, но при этом также включает ряд дополнительных расширений. Поскольку awk от Bell Laboratories и gawk представляют собой более современные версии, которым следует отдавать предпочтение перед nawk или «старым awk». (Версия awk от Bell Labaratories доступна по адресу *http://cm.bell-labs.com/cm/cs/awkbook/index.html*.)

Опция -f позволяет оформлять интерпретируемый файл следующим образом:

```
#!/bin/awk -f
(далее следует программа на языке awk)
```

Например, листинг 8.11 показывает содержимое интерпретируемого файла /usr/local/bin/awkexample.

Листинг 8.11. Интерпретируемый файл с программой на языке awk

```
#!/bin/awk -f
BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

Если имя каталога /usr/local/bin содержится в переменной окружения PATH, то мы можем запустить программу из листинга 8.11 (при условии, что у нас есть право на исполнение) следующим образом:

```
$ awkexample file1 FILENAME2 f3
ARGV[0] = awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

При запуске программа /bin/awk получит следующие аргументы командной строки:

```
/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3
```

Интерпретатору передается полный путь к интерпретируемому файлу (/usr/local/bin/awkexample). Простого имени файла (которое мы набрали в командной строке) будет недостаточно, поскольку не предполагается, что интерпретатор (в данном случае /bin/awk) сможет определить местонахождение файла, используя переменную окружения PATH. Когда awk начинает разбирать

интерпретируемый файл, он игнорирует первую строку, так как в языке awk символ # означает начало комментария.

Мы можем проверить аргументы командной строки с помощью следующей последовательности команд:

```
$ /bin/su                          получаем привилегии суперпользователя
Password:                          вводим пароль суперпользователя
# mv /bin/awk /bin/awk.save        сохраним оригинальный файл программы
# cp /home/sar/bin/echoarg /bin/awk      и заменим его на время
# suspend          приостановим работу командной оболочки суперпользователя
[1] + Stopped          /bin/su
$ awkexample file1 FILENAME2 f3
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILENAME2
argv[5]: f3
$ fg                   возобновим работу командной оболочки суперпользователя
/bin/su
# mv /bin/awk.save /bin/awk        восстановим оригинальный файл программы
# exit                             и покинем командную оболочку
```

В этом примере опция -f совершенно необходима интерпретатору. Как мы уже говорили, она сообщает awk о том, где находится текст программы на языке awk. Если убрать эту опцию из интерпретируемого файла, то при его запуске мы получим сообщение об ошибке. Точный текст сообщения зависит от того, где находится интерпретируемый файл и представляют ли остальные аргументы существующие файлы. Это происходит потому, что аргументы командной строки приобретают вид

```
/bin/awk /usr/local/bin/awkexample file1 FILENAME2 f3
```

и в результате awk пытается интерпретировать строку /usr/local/bin/awkexample как текст программы на языке awk. Если бы отсутствовала возможность передавать хотя бы один необязательный аргумент интерпретатору (в данном случае -f), то интерпретируемые файлы были бы пригодны к использованию только с командными оболочками.

Действительно ли так необходимы интерпретируемые файлы? На самом деле нет. Но они предоставляют целый ряд удобств для пользователей, хотя и за счет некоторого увеличения нагрузки на ядро (поскольку именно ядро их распознает и запускает указанный интерпретатор). Интерпретируемые файлы удобны по следующим причинам.

1. Они скрывают тот факт, что программа фактически является сценарием на том или ином языке. Так, например, запустить программу из листинга 8.11 можно с помощью примерно такой команды:

```
awkexample необязательные-аргументы
```

Совсем необязательно помнить о том, что эта программа в действительности представляет собой сценарий на языке awk, который пришлось бы запускать командой

```
awk -f awkexample необязательные-аргументы
```

2. Интерпретируемые сценарии дают выигрыш в эффективности. Вернемся к предыдущему примеру. Мы можем скрыть, что файл является сценарием на языке awk, обернув текст программы в сценарий командной оболочки:

```
awk 'BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}' $*
```

Однако такой подход требует от системы дополнительной работы. Прежде всего командная оболочка считывает команду и пытается выполнить ее с помощью функции execlp. Поскольку сценарий командной оболочки является исполняемым файлом, но не содержит машинных инструкций, возвращается признак ошибки и execlp делает предположение, что это файл сценария (как это и есть на самом деле). Тогда запускается программа /bin/sh, которой в качестве аргумента передается имя файла сценария. Командная оболочка запускает сценарий, но чтобы запустить awk, она вызывает функции fork, exec и wait. Таким образом, «обертывание» сценариев на других языках в сценарии командной оболочки приводит к увеличению нагрузки.

3. Интерпретируемые файлы позволяют писать сценарии на языках других командных оболочек, отличных от /bin/sh. Когда функция execlp обнаруживает, что исполняемый файл не содержит машинных инструкций, она вызывает командную оболочку, и это всегда /bin/sh. Однако, используя возможность указания интерпретатора в первой строке интерпретируемого файла, мы можем просто написать

```
#!/bin/csh
(далее следует текст сценария на языке командной оболочки C shell)
```

Опять же, можно обернуть этот код в сценарий командной оболочки /bin/sh, как показано немного выше, но это повлечет за собой дополнительную нагрузку.

Ни один из указанных приемов не работал бы, если бы командные оболочки и awk не использовали бы символ # в качестве знака комментария.

8.13. Функция system

Функция system предоставляет удобный способ выполнения команд внутри программы. Например, мы хотим поместить строку с датой и временем в некоторый файл. Для этого можно было бы использовать функции, описанные в разделе 6.10: получить текущее календарное время с помощью функции time, затем преобразовать его в структуру tm с помощью функции localtime,

сформировать строку с помощью функции strftime и записать результат в файл. Однако гораздо проще сделать так:

```
system("date > file");
```

Функция system определяется стандартом ISO C, но порядок взаимодействия с ней очень сильно зависит от системы. Стандарт POSIX.1 включает интерфейс system, расширяя определение ISO C, чтобы уточнить поведение функции в среде POSIX.

```
#include <stdlib.h>

int system(const char *cmdstring);
```

Возвращает: см. ниже

Если в аргументе *cmdstring* передается пустой указатель, функция system возвращает ненулевое значение только в том случае, если командный процессор доступен. Таким способом можно проверить, поддерживается ли функция system в данной системе. В системах UNIX она поддерживается всегда.

Поскольку функция system реализована на основе функций fork, exec и waitpid, она может возвращать значения трех типов.

1. Если функция fork терпит неудачу или функция waitpid возвращает код ошибки, отличный от EINTR, функция system возвращает значение –1.
2. Если функция exec терпит неудачу, это означает, что командная оболочка не может быть запущена, и функция system возвращает такое значение, как если бы командная оболочка вызвала функцию exit(127).
3. Когда обращение ко всем трем функциям – fork, exec и waitpid – заканчивается успехом, функция system возвращает код завершения командной оболочки в формате, предназначенном для функции waitpid.

> Некоторые старые реализации функции system возвращали код ошибки EINTR, если выполнение функции waitpid было прервано поступившим сигналом. Поскольку нет достаточно ясной стратегии восстановления после такой ошибки, стандарт POSIX позднее добавил требование, чтобы функция system не возвращала в этом случае код ошибки. (Прерывание системных вызовов рассматривается в разделе 10.5.)

В листинге 8.12 приводится пример реализации функции system. Единственный ее недостаток – отсутствие возможности обработки сигналов. Эту возможность мы добавим в разделе 10.18.

Листинг 8.12. Функция system без обработки сигналов

```
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int
system(const char *cmdstring) /* версия без обработки сигналов */
{
    pid_t pid;
```

```
    int status;

    if (cmdstring == NULL)
        return(1);      /* UNIX всегда поддерживает командный процессор */

    if ((pid = fork()) < 0) {
       status = -1;      /* вероятно, превышено максимальное количество процессов */
    } else if (pid == 0) {    /* дочерний процесс */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);           /* ошибка вызова функции execl */
    } else {                  /* родительский процесс */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1;  /* waitpid вернула ошибку, отличную от EINTR */
                break;
            }
        }
    }

    return(status);
}
```

Флаг `-c` сообщает командной оболочке, что следующий за ней аргумент – это команда, которую нужно выполнить. Командная оболочка анализирует эту строку и разбивает ее на отдельные аргументы. Аргумент *cmdstring* может содержать любую допустимую команду оболочки. Например, для перенаправления ввода-вывода могу быть использованы операторы < и >.

Чтобы выполнить команду самостоятельно, не прибегая к услугам командной оболочки, потребовались бы значительные усилия. Прежде всего нам пришлось бы вызвать функцию `execlp` вместо `execl`, чтобы использовать переменную окружения `PATH` подобно командной оболочке. Нам также пришлось бы разбивать командную строку на отдельные аргументы, чтобы передать их функции `execlp`. И наконец, мы не смогли бы воспользоваться метасимволами командной оболочки.

Обратите внимание, что вместо функции `exit` мы вызвали функцию `_exit`. Сделано это для предотвращения сброса буферов ввода-вывода, которые могли быть унаследованы дочерним процессом от родительского при вызове функции `fork`.

Мы можем протестировать нашу версию функции `system` с помощью программы, представленной в листинге 8.13. (Исходный код функции `pr_exit` вы найдете в листинге 8.3.)

Листинг 8.13. Вызов функции system

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    int status;
```

```
    if ((status = system("date")) < 0)
        err_sys("ошибка вызова функции system()");
    pr_exit(status);

    if ((status = system("nosuchcommand")) < 0)
        err_sys("ошибка вызова функции system()");
    pr_exit(status);

    if ((status = system("who; exit 44")) < 0)
        err_sys("ошибка вызова функции system()");
    pr_exit(status);

    exit(0);
}
```

Запуск программы из листинга 8.13 дал следующие результаты:

```
$ ./a.out
Sun Mar 21 18:41:32 EST 2004
нормальное завершение, код выхода = 0              команда date
sh: nosuchcommand: command not found
нормальное завершение, код выхода = 127            команда nosuchcommand
sar :0 Mar 18 19:45
sar pts/0 Mar 18 19:45 (:0)
sar pts/1 Mar 18 19:45 (:0)
sar pts/2 Mar 18 19:45 (:0)
sar pts/3 Mar 18 19:45 (:0)
нормальное завершение, код выхода = 44             команда exit
```

Основное преимущество использования функции `system` вместо прямого обращения к функциям `fork` и `exec` заключается в том, что она производит все необходимые действия по обработке ошибочных ситуаций, а также (в нашей следующей версии этой функции, в разделе 10.18) по обработке сигналов.

Ранние версии UNIX, включая SVR3.2 и 4.3BSD, не имели функции `waitpid`. Вместо этого родительский процесс дожидался завершения работы потомка с помощью примерно такой инструкции:

```
while ((lastpid = wait(&status)) != pid && lastpid != -1)
    ;
```

Проблема возникает в том случае, если у процесса уже имеются дочерние процессы, запущенные до обращения к функции `system`. Поскольку показанный выше цикл `while` продолжает работу до тех пор, пока не завершится дочерний процесс, созданный функцией `system`, то если какой-либо из дочерних процессов, запущенных ранее, завершится до процесса, указанного переменной `pid`, то его идентификатор и код завершения будут утеряны в цикле `while`. Эта неспособность функции `wait` ждать завершения определенного дочернего процесса – одна из причин, приводимых в «POSIX.1 Rationale» для обоснования добавления функции `waitpid`. В разделе 15.3 мы увидим, что та же самая проблема возникает при работе с функциями `popen` и `pclose`, если система не поддерживает функцию `waitpid`.

Программы с установленным битом set-user-ID

Что произойдет, если функция `system` будет вызвана из программы с установленным битом set-user-ID? Такой вызов создает брешь в системе безопасности, и никогда нельзя допускать этого. В листинге 8.14 приводится программа, которая просто вызывает функцию `system` для обработки своих аргументов командной строки.

Листинг 8.14. Обработка аргументов командной строки с помощью функции system

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int status;

    if (argc < 2)
        err_quit("требуется хотя бы один аргумент командной строки");

    if ((status = system(argv[1])) < 0)
        err_sys("ошибка вызова функции system()");
    pr_exit(status);

    exit(0);
}
```

Скомпилируем эту программу в исполняемый файл `tsys`.

В листинге 8.15 приводится другая простая программа, которая выводит значения реального и эффективного идентификаторов пользователя.

Листинг 8.15. Вывод реального и эффективного идентификаторов пользователя

```
#include "apue.h"

int
main(void)
{
    printf("реальный uid = %d, эффективный uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

Скомпилируем эту программу в исполняемый файл `printuids`. Запуск обеих программ дал следующие результаты:

```
$ tsys printuids                обычный запуск без дополнительных привилегий
реальный uid = 205, эффективный uid = 205
нормальное завершение, код выхода = 0
$ su                            получаем права суперпользователя
Password:                       вводим пароль суперпользователя
# chown root tsys               меняем владельца файла
# chmod u+s tsys                устанавливаем бит set-user-ID
# ls -l tsys                    проверяем владельца и права доступа
-rwsrwxr-x 1 root    16361 Mar 16 16:59 tsys
# exit                          покидаем командную оболочку суперпользователя
```

```
$ tsys printuids
реальный uid = 205, эффективный uid = 0      вот она, брешь в системе безопасности
нормальное завершение, код выхода = 0
```

Привилегии суперпользователя, которые мы дали программе `tsys`, сохранились после вызовов функций `fork` и `exec`, которые производит функция `system`.

> Если в качестве `/bin/sh` используется `bash` версии 2, то предыдущий пример работать не будет, потому что `bash` записывает значение реального идентификатора в эффективный, если они не совпадают.

Если предполагается, что программа будет работать с повышенными привилегиями – с установленными битами set-user-ID или set-group-ID – и должна порождать другие процессы, то она должна делать это непосредственно с помощью функций `fork` и `exec`, выполняя переход к привилегиям обычного пользователя после вызова `fork` и перед вызовом `exec`. Функция `system` никогда не должна использоваться в программах с установленными битами set-user-ID или set-group-ID.

> Еще одна из причин заключается в том, что функция `system` вызывает командную оболочку для разбора аргументов командной строки, а сама оболочка использует значение переменной окружения `IFS` в качестве разделителя полей во входной строке. Ранние версии командной оболочки при запуске не сбрасывали эту переменную к стандартному набору символов. Это позволяло злоумышленнику изменить значение переменной окружения `IFS` до вызова функции `system` и заставить ее выполнить совсем другую команду.

8.14. Учет использования ресурсов процессами

Большинство версий UNIX предоставляют возможность вести учет использования ресурсов процессами. Когда режим учета включен, ядро создает запись учета всякий раз, когда процесс завершает работу. Такая запись обычно представляет собой небольшой блок двоичных данных, в котором хранится имя команды, количество использованного процессорного времени, идентификаторы пользователя и группы, время запуска и тому подобное. В этом разделе мы поближе рассмотрим записи учета, поскольку это дает нам возможность взглянуть на процессы с другой стороны, используя для этого функцию `fread` из раздела 5.9.

> Возможность учета использования ресурсов процессами не определяется ни одним из стандартов. Таким образом, все реализации имеют достаточно существенные различия. Например, учет ввода-вывода в ОС Solaris 9 производится в байтах, тогда как в FreeBSD 5.2.1 и Mac OS X 10.3 – в блоках, хотя при этом не делается никаких различий между размерами блоков, что делает такой учет достаточно бесполезным. С другой стороны, ОС Linux 2.4.22 вообще не поддерживает учет операций ввода-вывода.
>
> Кроме того, каждая из реализаций имеет свой собственный набор административных команд для работы с учетной информацией. Например, Solaris предоставляет для сбора, обработки и вывода учетных сведений команды `runacct`(1m) и `acctcom`(1), а FreeBSD – команду `sa`(8).

Включение и выключение режима сбора статистической информации о процессах производится с помощью функции `acct`. Эта функция используется

в единственной программе – accton(8) (она, к счастью, на всех платформах называется одинаково). Чтобы включить режим учета, суперпользователь должен запустить команду accton с аргументом командной строки, в котором передается полный путь к файлу. В этот файл будет записываться учетная информация. Обычно он имеет имя /var/account/acct в FreeBSD и Mac OS X, /var/account/pacct – в Linux и /var/adm/pacct – в Solaris. Учет выключается, когда команда accton запускается без параметров.

Структура записи с учетной информацией определена в заголовочном файле <sys/acct.h> и выглядит примерно так:

```
typedef u_short comp_t; /* 3-битная, по основанию 8, экспонента; 13 бит - мантисса */
struct acct
{
    char ac_flag;    /* флаг (табл. 8.8) */
    char ac_stat;    /* код завершения (только номер сигнала  */
                     /* и признак создания файла core) (только в Solaris) */
    uid_t ac_uid;    /* реальный идентификатор пользователя */
    gid_t ac_gid;    /* реальный идентификатор группы */
    dev_t ac_tty;    /* управляющий терминал */
    time_t ac_btime; /* календарное время запуска */
    comp_t ac_utime; /* пользовательское время (в тактах) */
    comp_t ac_stime; /* системное время (в тактах) */
    comp_t ac_etime; /* общее время работы (в тактах) */
    comp_t ac_mem;   /* средний расход памяти */
    comp_t ac_io;    /* количество переданных байт (функции read и write) */
                     /* в "блоках" для BSD-систем */
    comp_t ac_rw;    /* количество прочитанных и записанных блоков */
                     /* (отсутствует в BSD-системах) */
    char ac_comm[8]; /* имя команды: [8] в Solaris, */
                     /* [10] в Mac OS X, [16] в FreeBSD и [17] в Linux */
};
```

В поле ac_flag заносится информация о некоторых событиях, зафиксированных во время работы процесса. Эти события перечислены в табл. 8.8.

Таблица 8.8. Значения флага ac_flag структуры acct

ac_flag	Описание	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
AFORK	Процесс был порожден функцией fork, но без вызова функции exec	•	•	•	•
ASU	Процесс использовал привилегии суперпользователя		•	•	•
ACOMPAT	Процесс использовал режим совместимости				
ACORE	Был создан файл с дампом памяти процесса (core)	•	•	•	
AXSIG	Процесс был завершен по сигналу	•	•	•	
AEXPND	Расширенная запись с учетными данными				•

Все необходимые данные, такие как количество использованного процессорного времени или объем операций ввода-вывода, хранятся в таблице процессов и инициализируются при создании нового процесса после вызова функции `fork`. Каждая запись формируется и записывается в файл в момент завершения процесса. Это означает, что записи в файле следуют в порядке завершения процессов, а не в порядке запуска. Чтобы определить порядок запуска, мы должны просмотреть файл с учетной информацией и отсортировать его по календарному времени запуска процессов. Но это даст не совсем точный порядок запуска, так как календарное время измеряется в секундах (раздел 1.10), а на протяжении одной секунды может быть запущено несколько процессов. С другой стороны, общее время работы дается в тактах системных часов (обычно от 60 до 128 тактов в секунду). Но мы не знаем точное время окончания работы процесса; все, что у нас есть, – это время запуска и порядок завершения процессов. Это означает, что даже при том, что общее время работы процесса измеряется более точно, чем время запуска, мы все еще не в состоянии определить точный порядок запуска процессов по тем данным, которые имеются в файле учета.

Каждая запись с учетной информацией соответствует процессу, а не программе. Новая запись создается ядром только при создании нового дочернего процесса вызовом функции `fork`, а не в момент, когда запускается новая программа. Хотя вызов функции `exec` и не приводит к созданию новой записи, тем не менее имя команды изменяется и поэтому сбрасывается флаг `AFORK`. Это означает, что если программа A запускает B, B запускает C и после этого C завершает работу, то такой последовательности запущенных программ будет соответствовать всего одна запись с учетной информацией. Имя команды в этой записи будет соответствовать программе C, но процессорное время будет представлять собой сумму времени, потраченного всеми тремя программами.

Пример

Чтобы получить некоторый объем данных для исследования, создадим тестовую программу, которая реализует следующую схему действий.

Рис. 8.4. *Структура процесса, на примере которого будет рассматриваться учетная информация*

Исходный текст программы приводится в листинге 8.16. Эта программа вызывает функцию fork четыре раза. Каждый из дочерних процессов выполняет некоторые действия и завершает работу.

Листинг 8.16. Программа генерации учетной информации

```
#include "apue.h"

int
main(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid != 0) {      /* родительский процесс */
        sleep(2);
        exit(2);              /* завершение с кодом 2 */
    }
                              /* первый дочерний процесс */
    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid != 0) {
        sleep(4);
        abort();              /* завершение с созданием файла core */
    }
                              /* второй дочерний процесс */
    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid != 0) {
        execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
        exit(7);              /* программа не должна доходить до этой точки */
    }

    /* third child */
    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid != 0) {
        sleep(8);
        exit(0);              /* нормальный выход */
    }

                              /* четвертый дочерний процесс */
    sleep(6);
    kill(getpid(), SIGKILL);  /* завершение по сигналу без создания файла core */

    exit(6);                  /* программа не должна доходить до этой точки */
}
```

Запустим эту программу в ОС Solaris и затем выведем учетную информацию с помощью программы из листинга 8.17.

Листинг 8.17. Вывод учетной информации из системного файла учетных данных

```
#include "apue.h"
#include <sys/acct.h>
```

```
#ifdef HAS_SA_STAT
#define FMT "%-*.*s e = %6ld, chars = %7ld, stat = %3u: %c %c %c %c\n"
#else
#define FMT "%-*.*s e = %6ld, chars = %7ld, %c %c %c %c\n"
#endif
#ifndef HAS_ACORE
#define ACORE 0
#endif
#ifndef HAS_AXSIG
#define AXSIG 0
#endif

static unsigned long
compt2ulong(comp_t comptime) /* преобразовать comp_t в unsigned long */
{
    unsigned long val;
    int exp;

    val = comptime & 0x1fff;    /* 13 бит - мантисса */
    exp = (comptime >> 13) & 7; /* 3 бита - экспонента (0-7) */

    while (exp-- > 0)
        val *= 8;
    return(val);
}

int
main(int argc, char *argv[])
{
    struct acct acdata;
    FILE        *fp;

    if (argc != 2)
        err_quit("Использование: pracct имя_файла");
    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("невозможно открыть %s", argv[1]);
    while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
        printf(FMT, (int)sizeof(acdata.ac_comm),
               (int)sizeof(acdata.ac_comm), acdata.ac_comm,
        compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io),
#ifdef HAS_SA_STAT
        (unsigned char) acdata.ac_stat,
#endif
        acdata.ac_flag & ACORE ? 'D' : ' ',
        acdata.ac_flag & AXSIG ? 'X' : ' ',
        acdata.ac_flag & AFORK ? 'F' : ' ',
        acdata.ac_flag & ASU ? 'S' : ' ');
    }
    if (ferror(fp))
        err_sys("ошибка вызова функции read");
    exit(0);
}
```

На платформах, производных от BSD, поле ac_flag в структуре acct не поддерживается, поэтому мы объявляем константу HAS_SA_STAT для платформ, которые поддерживают это поле. Использование константы с именем, соответствующим функциональной особенности вместо имени платформы, дает нам более удобочитаемый исходный текст программы и позволяет легко модифицировать ее простым добавлением дополнительных определений в строку команды компиляции. В качестве альтернативы можно было бы использовать в тексте программы

```
#if defined(BSD) || defined(MACOS)
```

что по мере переноса программы на другие платформы делает ее все более громоздкой.

Мы определяем аналогичные константы, чтобы установить, поддерживает ли платформа флаги ACORE и AXSIG. Мы не можем использовать просто имена флагов, потому что в Linux они определены в виде перечисления enum, и их нельзя использовать в выражении #ifdef.

Для тестирования нам необходимо:

1. Обладая привилегиями суперпользователя, включить сбор статистической информации командой accton. Обратите внимание, что к тому моменту, когда команда accton завершится, сбор статистической информации уже должен быть включен; поэтому первая запись в учетном файле должна относиться к этой команде.
2. Выйти из командной оболочки суперпользователя и запустить программу из листинга 8.16. В результате в файле учета должно появиться шесть дополнительных записей: одна запись должна соответствовать завершившейся командной оболочке, в которой мы работали с привилегиями суперпользователя, одна – родительскому процессу тестовой программы и четыре – дочерним процессам, порожденным тестовой программой.

   Второй дочерний процесс не создает новый процесс с помощью функции execl. Поэтому ему будет соответствовать одна запись в файле учета.
3. Получить привилегии суперпользователя и отключить сбор статистической информации. Поскольку к моменту завершения команды accton сбор статистической информации уже должен быть выключен, то в файле учета не должно появиться новой записи, соответствующей этой команде.
4. Запустить программу из листинга 8.17, которая выведет информацию, собранную в учетном файле.

Ниже приводится вывод программы, полученный на шаге 4. Для последующего обсуждения в конце некоторых строк добавлено описание процесса (курсивом).

```
accton   e =      6, chars =       0, stat =    0:      S
sh       e =   2106, chars =   15632, stat =    0:      S
dd       e =      8, chars =  273344, stat =    0:        второй потомок
a.out    e =    202, chars =     921, stat =    0:        родительский процесс
a.out    e =    407, chars =       0, stat =  134:   F    первый потомок
```

```
a.out    e =    600, chars =      0, stat =    9:   F   четвертый потомок
a.out    e =    801, chars =      0, stat =    0:   F   третий потомок
```

Значения общего затраченного времени приводятся в тактах системных часов. Для данной системы (табл. 2.12) частота хода системных часов составляет 100 тактов в секунду. Например, вызов `sleep`(2) в родительском процессе соответствует 202 тактам системных часов. Первый потомок на выполнение функции `sleep`(4) затратил 407 тактов системных часов. Обратите внимание, что время, на которое процесс был приостановлен функцией `sleep`, измеряется не совсем точно. (Мы вернемся к этой функции в главе 10.) Вызовы функций `fork` и `exit` также занимают некоторое количество времени.

Обратите также внимание на то, что поле `ac_stat` соответствует не действительному коду завершения процесса, а той его части, которую мы обсуждали в разделе 8.6. Единственная информация, которая хранится в этом поле – это номер сигнала (обычно младшие семь бит) и признак создания файла core (обычно старший бит), если процесс завершился аварийно. Если процесс завершился нормальным образом, то мы не сможем получить код выхода из файла учета. Для первого потомка код завершения имеет значение 128+6, где 128 – это признак создания файла core, а 6 – номер сигнала `SIGABRT` для данной системы, который генерируется функцией `abort`. Значение 9 кода завершения четвертого потомка соответствует номеру сигнала `SIGKILL`. Данный набор учетной информации ничего не сообщает о том, что код выхода (аргумент функции `exit`) родительского процесса равен числу 2, а аргумент функции `exit` в третьем потомке равен 0.

Размер файла `/etc/termcap`, который был скопирован процессом `dd` во втором потомке, составляет 136 663 байта. Объем операций ввода-вывода в два раза превышает это значение, поскольку 136 663 байта сначала читаются и затем те же 136 663 байта записываются. Даже несмотря на то, что вывод производится в пустое устройство, эти байты все равно учитываются.

Значения поля `ac_flag` в точности соответствуют нашим ожиданиям. Флаг `F` установлен у всех дочерних процессов, за исключением второго, который вызвал функцию `execl`. Флаг `F` отсутствует у родительского процесса, поскольку командная оболочка, запустившая его, вызвала функцию `fork`, а затем `exec` для файла `a.out`. Первый дочерний процесс вызвал функцию `abort`, которая сгенерировала сигнал `SIGABRT`, что вызвало создание файла core. Обратите внимание, что в полученных результатах отсутствуют флаги `D` и `X`, поскольку они не поддерживаются ОС Solaris; информация, которую они представляют, может быть извлечена из поля `ac_stat`. Четвертый дочерний процесс также был завершен по сигналу, но сигнал `SIGKILL` не вызывает создание файла core, он просто завершает процесс.

И заключительное замечание: первый дочерний процесс имеет нулевой объем операций ввода-вывода, хотя он завершился созданием файла core. Это говорит о том, что объем операций ввода-вывода, который требуется для создания файла core, не учитывается ядром.

8.15. Идентификация пользователя

Любой процесс может узнать свои реальные и эффективные идентификаторы пользователя и группы. Однако иногда возникает необходимость узнать имя пользователя, запустившего программу. Для этой цели можно было бы вызвать `getpwuid(getuid())`, но что делать, если один и тот же пользователь имеет несколько учетных записей с разными именами, но с одним и тем же числовым идентификатором? (В файле паролей может быть несколько записей с одинаковым числовым идентификатором, чтобы пользователь мог запускать различные командные оболочки при входе в систему.) Как правило, система отслеживает имена, под которыми осуществлялся вход (раздел 6.8), и предоставляет способ получить имя пользователя с помощью функции `getlogin`.

```
#include <unistd.h>

char *getlogin(void);
```

Возвращает указатель на строку с именем пользователя в случае успеха, `NULL` в случае ошибки

Эта функция может завершиться неудачей, если процесс не присоединен к терминалу, с которого был произведен вход пользователя в систему. Такие процессы обычно называются *демонами*. Они будут обсуждаться в главе 13.

Получив имя пользователя, можно с помощью функции `getpwnam` найти соответствующую запись в файле паролей, чтобы, например, определить тип командной оболочки.

> Чтобы найти имя пользователя, операционные системы UNIX традиционно вызывали функцию `ttyname` (раздел 18.9) и затем пытались отыскать соответствующую запись в файле `utmp` (раздел 6.8). FreeBSD и Mac OS X сохраняют имя пользователя в структуре сессии, связанной с записью в таблице процессов, и предоставляют системные вызовы для получения и сохранения этого имени.
>
> ОС System V предоставляла функцию `cuserid`, с помощью которой можно было получить имя пользователя. Эта функция вызывала `getlogin` или, в случае ее неудачи, `getpwuid(getuid())`. Стандарт IEEE 1003.1-1988 определял функцию `cuserid`, но она использовала эффективный, а не реальный идентификатор пользователя для получения имени пользователя. Функция `cuserid` была исключена в версии стандарта POSIX.1 1990 года.
>
> Программа `login`(1) обычно записывает имя пользователя в переменную окружения `LOGNAME`, которая наследуется командной оболочкой после входа в систему. Однако следует помнить, что пользователь может изменить значение этой переменной окружения, поэтому на нее нельзя полагаться при проверке имени пользователя. Вместо этого должна использоваться функция `getlogin`.

8.16. Временные характеристики процесса

В разделе 1.10 мы описывали три временные характеристики, которые можем измерить: общее время выполнения, пользовательское время и систем-

ное время. Любой процесс может вызвать функцию times, чтобы получить эти три значения для себя самого и для любого из завершивших работу потомков.

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

Возвращает количество тактов общего времени выполнения процесса в случае успеха, −1 в случае ошибки

Эта функция заполняет структуру tms, адрес которой передается в аргументе *buf*:

```
struct tms {
    clock_t tms_utime;  /* пользовательское время */
    clock_t tms_stime;  /* системное время */
    clock_t tms_cutime; /* пользовательское время */
                        /* для завершившегося потомка */
    clock_t tms_cstime; /* системное время для завершившегося потомка */
};
```

Обратите внимание: структура не содержит общего времени выполнения. Мы получаем его в виде возвращаемого значения при каждом вызове функции. Это время отмеряется от произвольного момента в прошлом, так что нельзя использовать его абсолютное значение, вместо этого следует использовать относительные значения. Например, мы вызываем функцию times и сохраняем возвращаемое значение. Через какое-то время мы еще раз вызываем функцию times и вычитаем сохраненное ранее значение из нового значения. Разница будет определять время, прошедшее между двумя вызовами функции times. (Вполне возможно, хотя и маловероятно, что у долгоживущего процесса произойдет переполнение счетчика общего времени, см. упражнение 1.6.)

Два поля структуры отводятся для хранения временных характеристик дочернего процесса, но только того, завершения которого мы ожидали с помощью функции wait, waitid или waitpid.

Все значения типа clock_t, возвращаемые функцией, могут быть преобразованы в секунды путем деления на количество тактов системных часов в секунду — значение параметра _SC_CLK_TCK, возвращаемое функцией sysconf (раздел 2.5.4).

> Большинство реализаций предоставляют функцию getrusage(2). Она возвращает затраченное процессорное время и еще 14 значений, характеризующих использование различных ресурсов. Исторически эта функция происходит из OC BSD, таким образом, все производные от BSD системы, как правило, поддерживают большее количество полей, чем другие реализации.

Пример

Программа, представленная листингом 8.18, запускает команды оболочки, переданные ей в виде аргументов, засекает время выполнения и выводит значения полей структуры tms.

Листинг 8.18. Запуск команд и определение времени их работы

```
#include "apue.h"
#include <sys/times.h>

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

int
main(int argc, char *argv[])
{
    int i;

    setbuf(stdout, NULL);
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]); /* один раз для каждого аргумента командной строки */
    exit(0);
}

static void
do_cmd(char *cmd)        /* запустить и измерить время работы "cmd" */
{
    struct tms tmsstart, tmsend;
    clock_t start, end;
    int status;

    printf("\nкоманда: %s\n", cmd);
    if ((start = times(&tmsstart)) == -1) /* начальные значения */
        err_sys("ошибка вызова функции times");
    if ((status = system(cmd)) < 0)       /* запустить команду */
        err_sys("ошибка вызова функции system()");
    if ((end = times(&tmsend)) == -1)     /* конечные значения */
        err_sys("ошибка вызова функции times");
    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
    static long clktck = 0;

    if (clktck == 0)    /* прежде всего нужно получить количество тактов в сек. */
        if ((clktck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("ошибка вызова функции sysconf");
    printf(" real: %7.2f\n", real / (double) clktck);
    printf(" user: %7.2f\n",
           (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf(" sys: %7.2f\n",
           (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf(" child user: %7.2f\n",
           (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf(" child sys: %7.2f\n",
           (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}
```

Запустив эту программу, мы получаем:

```
$ ./a.out "sleep 5" "date"

команда: sleep 5
  real: 5.02
  user: 0.00
  sys: 0.00
  child user: 0.01
  child sys: 0.00
нормальное завершение, код выхода = 0

команда: date
Mon Mar 22 00:43:58 EST 2004
  real: 0.01
  user: 0.00
  sys: 0.00
  child user: 0.01
  child sys: 0.00
нормальное завершение, код выхода = 0
```

В приведенных примерах все затраченное процессорное время относится к дочернему процессу, которым является командная оболочка, запускающая команду.

8.17. Подведение итогов

Глубокое понимание управления процессами в UNIX совершенно необходимо для профессионального программирования в этой операционной системе. Для этого нужно освоить лишь несколько функций: `fork`, семейство функций `exec`, `_exit`, `wait` и `waitpid`. Эти примитивы широко используются во многих приложениях. Кроме всего прочего, функция `fork` дала нам возможность увидеть ситуацию гонки за ресурсами.

Изучение функции `system` и возможностей учета расходования ресурсов процессами показало нам функции управления процессами под другим углом. Мы также рассмотрели еще одну возможность функции `exec` – интерпретацию файлов и то, как эта интерпретация выполняется. Понимание различий между различными идентификаторами пользователя и группы – реальным, эффективным и сохраненным идентификаторами – особенно важно для обеспечения безопасности программ с установленным битом set-user-ID.

Обладая этими знаниями о функционировании отдельного процесса и его потомков, мы рассмотрим в следующей главе взаимоотношения между разными процессами – сессии и управление заданиями. Затем в главе 10 мы завершим тему процессов описанием сигналов.

Упражнения

8.1. При обсуждении программы из листинга 8.2 мы говорили, что если вызов функции `_exit` заменить на `exit`, то стандартный поток вывода может

оказаться закрытым и функция printf вернет признак ошибки – число –1. Измените программу, чтобы проверить, присуще ли такое поведение вашей реализации. Если нет, как эту ситуацию можно смоделировать?

8.2. Вспомните типичную раскладку памяти процесса, изображенную на рис. 7.3. Каждому вызову функции обычно соответствует свой фрейм стека, но поскольку после вызова функции vfork дочерний процесс продолжает работать в адресном пространстве родительского процесса, то что может произойти, если обращение к vfork производится не из функции main, а из другой функции, и при этом дочерний процесс выполняет возврат из этой функции после вызова функции vfork? Напишите тестовую программу для проверки этой ситуации и нарисуйте схему происходящего.

8.3. Когда мы запускаем программу из листинга 8.7 один раз:

```
$ ./a.out
```

мы получаем вполне корректные результаты. Но если мы запустим эту программу несколько раз подряд:

```
$ ./a.out ; ./a.out ; ./a.out
от родительского процесса
оот родительского процесса
ото дтроочернего процесса
дительского процесса
от дочернего процесса
чернего процесса
```

результаты не соответствуют нашим ожиданиям. Почему? Как это можно исправить? Останется ли эта проблема, если дочерний процесс будет производить вывод первым?

8.4. В программе из листинга 8.10 мы вызывали функцию execl, которой передавали полный путь к интерпретируемому файлу. Если вместо этого использовать функцию execlp, передав ей только имя файла testinterp, и если каталог /home/sar/bin указан в переменной окружения PATH, что выведет программа в качестве argv[2]?

8.5. Каким образом процесс может получить значение сохраненного идентификатора пользователя?

8.6. Напишите программу, которая создает процесс-зомби и затем с помощью функции system запускает команду ps(1), чтобы проверить, действительно ли процесс является зомби.

8.7. Как было отмечено в разделе 8.10, стандарт POSIX.1 требует, чтобы все открытые каталоги закрывались при вызове функции exec. Проверить это можно следующим образом: откройте корневой каталог с помощью функции opendir, уточните содержимое структуры DIR в вашей системе и выведите состояние флага close-on-exec. Затем откройте тот же самый каталог для чтения с помощью функции open и выведите состояние флага close-on-exec.

9

Взаимоотношения между процессами

9.1. Введение

В предыдущей главе мы узнали, что между процессами существуют определенные взаимосвязи. Прежде всего, каждый процесс имеет «родителя» (начальный процесс уровня ядра обычно сам является собственным родителем). Когда дочерний процесс завершает работу, родительский процесс извещается об этом и может получить код выхода своего потомка. Мы также упоминали группы процессов, когда описывали функцию `waitpid` (раздел 8.6), которая может ожидать завершения любого процесса из указанной группы.

В этой главе мы более подробно рассмотрим группы процессов, а также коснемся понятия сессий, введенного стандартом POSIX.1. Мы также будем рассматривать отношения между командной оболочкой входа, которая запускается во время входа в систему, и всеми процессами, запускаемыми из этой оболочки.

Совершенно невозможно говорить о взаимоотношениях процессов без упоминания сигналов, а чтобы обсуждать сигналы, необходимо знание многих концепций, которые рассматриваются в этой главе. Если вы совершенно не знакомы с механизмом сигналов UNIX, то, вероятно, стоит сначала вкратце ознакомиться с содержанием главы 10.

9.2. Вход с терминала

Для начала мы рассмотрим программы, которые запускаются при входе пользователя в систему UNIX. Во времена ранних версий UNIX, таких как Version 7, пользователи входили в систему через терминалы ввода-вывода, соединенные кабелем с главным компьютером. Терминалы могли быть как локальными (связанными непосредственно с компьютером), так и удаленными (связанными с компьютером через модем). И в том и в другом случае вход в систему осуществлялся через драйвер устройства терминала в ядре. Например, наиболее типичными терминальными устройствами на PDP-11 были

DH-11 и DZ-11. Машина имела фиксированное количество таких устройств, поэтому было заранее известно максимальное количество пользователей, которые могли одновременно войти в систему.

Когда появились графические терминалы, были разработаны системы с оконным графическим интерфейсом, чтобы предоставить пользователю новый, более удобный способ взаимодействия с компьютером. Для эмуляции алфавитно-цифровых терминалов стали разрабатываться приложения, которые создавали «окно терминала», что позволяло пользователю взаимодействовать с главной машиной привычным способом (то есть через командную строку).

В настоящее время некоторые системы позволяют запускать оконный интерфейс после входа, тогда как другие системы делают это автоматически. В последнем случае вам, возможно, все же придется вводить свое имя и пароль – в зависимости от конфигурации вашей оконной системы (некоторые системы могут быть настроены на выполнение автоматического входа).

Процедура, которую мы сейчас описываем, используется для входа в систему UNIX посредством терминала. Она не зависит от типа используемого терминала – это может быть алфавитно-цифровой терминал, графический терминал, эмулирующий простой алфавитно-цифровой терминал или графический терминал, на котором запущена оконная система.

Вход в систему с терминала в BSD-системах

Эта процедура практически не изменилась за последние 30 лет. Системный администратор создает файл, обычно это `/etc/ttys`, в котором каждая строка соответствует одному терминальному устройству. В каждой строке определяется имя устройства и другие параметры, которые передаются программе `getty` – например, скорость передачи данных (baud rate). Во время загрузки системы ядро создает процесс с идентификатором 1, то есть процесс `init`, который переводит систему в многопользовательский режим. Процесс `init` читает файл `/etc/ttys` и для каждого терминала запускает программу `getty` с помощью функций `fork` и `exec`. Это дает нам схему процессов, показанную на рис. 9.1.

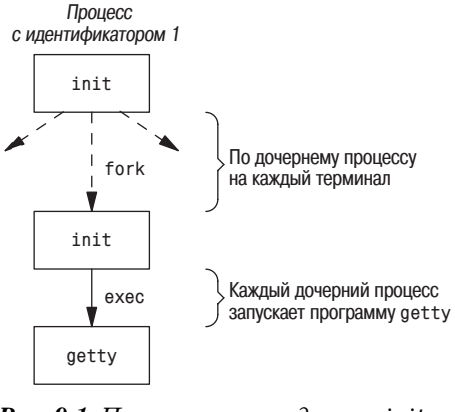


Рис. 9.1. *Процессы, порождаемые init, чтобы разрешить вход в систему с терминалов*

Все процессы, изображенные на рис. 9.1, имеют реальный и эффективный идентификатор пользователя 0 (то есть обладают привилегиями суперпользователя). Кроме того, процесс `init` запускает программу `getty` с пустой средой окружения.

Программа `getty` с помощью функции `open` открывает устройство терминала на чтение и на запись. Если устройство представляет собой модем, функция `open` может отработать с некоторой задержкой внутри драйвера устройства, пока модем набирает номер и устанавливает соединение. Когда устройство открыто, ему назначаются файловые дескрипторы с номерами 0, 1 и 2. Далее `getty` выводит некоторую строку, например `login:`, и ожидает ввода имени пользователя. Если терминал поддерживает обмен данными на разных скоростях, программа `getty` в состоянии распознать специальные управляющие последовательности, которые указывают ей изменить скорость передачи. За дополнительными сведениями о программе `getty` и файлах данных (`gettytab`), управляющих ее действиями, обращайтесь к справочному руководству по вашей операционной системе.

После ввода имени пользователя программа `getty` завершает работу и передает управление программе `login`, примерно таким образом:

```
execle("/bin/login", "login", "-p", username, (char *)0, envp);
```

(Файл `gettytab` может содержать указания по вызову других программ, но по умолчанию вызывается программа `login`.) Процесс `init` запускает программу `getty` с пустой средой окружения, а `getty` создает для программы `login` (аргумент `envp`) окружение с именем терминала (что-нибудь вроде `TERM=foo`, где `foo` – тип терминала, который берется из файла `gettytab`) и другими переменными окружения, определенными в файле `gettytab`. Флаг `-p` сообщает программе `login`, что она должна сохранить предыдущую среду окружения и добавить к ней новую среду, не заменяя существующую. На рис. 9.2 показано состояние процессов сразу после запуска программы `login`.

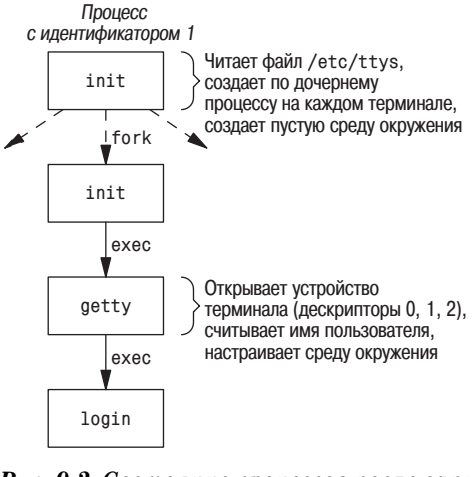


***Рис. 9.2.** Состояние процессов после запуска программы login*

Все процессы, изображенные на рис. 9.2, обладают привилегиями суперпользователя. Идентификатор трех последних процессов один и тот же, поскольку функция exec не изменяет идентификатор процесса. Кроме того, все процессы, кроме первоначального init, имеют идентификатор родительского процесса 1.

Программа login выполняет множество различных действий. Поскольку у нее уже есть имя пользователя, она может вызвать функцию getpwnam, чтобы получить строку учетной записи из файла паролей. Затем программа login вызывает функцию getpass(3), чтобы вывести приглашение к вводу Password: и прочитать пароль (при этом, разумеется, отображение вводимых символов отключено). Далее вызывается функция crypt(3), которая зашифровывает введенный пароль, и полученный результат сравнивается с полем pw_passwd из записи в теневом файле паролей. Если попытка входа в систему не удалась из-за неверно введенного пароля (после нескольких попыток), login вызывает функцию exit с аргументом 1. Такое завершение программы login будет замечено родительским процессом (init), и он с помощью функций fork и exec снова запустит программу getty для возобновления процедуры входа на данном терминале.

Это традиционная процедура аутентификации, используемая в UNIX. Современные системы UNIX поддерживают большое количество других процедур аутентификации. Например, FreeBSD, Linux, MacOS X и Solaris поддерживают более гибкую схему, известную как PAM (Pluggable Authentication Modules – сменные модули аутентификации). Эта схема позволяет системным администраторам сконфигурировать методы аутентификации для обращения к службам, которые разработаны для использования с библиотекой PAM.

Если приложение требует проверки прав пользователя на выполнение определенных задач, можно либо жестко зашить механизм аутентификации в код приложения, либо создать аналогичную функциональность, используя библиотеку PAM. Преимущество PAM заключается в том, что администратор может сконфигурировать разные способы аутентификации пользователей для выполнения различных задач, основываясь на локальной политике безопасности.

Если вход в систему выполнен корректно, то программа login:

- Изменит домашний каталог (chdir)
- Изменит владельца терминала (chown)
- Изменит права доступа к устройству терминала таким образом, что мы сможем производить операции чтения и записи
- Установит идентификатор группы вызовом функций setgid и initgroups
- Инициализирует среду окружения той информацией, которой располагает программа login: это домашний каталог пользователя (HOME), командная оболочка (SHELL), имя пользователя (USER или LOGNAME) и список каталогов для поиска исполняемых файлов (PATH)
- Изменит идентификатор пользователя (setuid) и запустит командную оболочку входа в систему

```
execl("/bin/sh", "-sh", (char *)0);
```

> Символ «-» в качестве первого символа argv[0] сообщает командной оболочке, что она запущена как оболочка входа в систему. Командная оболочка, обнаружив этот признак, может соответственно изменить перечень действий, выполняемых при запуске.

На самом деле программа login выполняет гораздо больше действий, чем мы здесь упомянули. Она, например, может выводить так называемое «сообщение дня», проверять поступление новой почты и выполнять ряд других задач. Но нас интересует только та функциональность, которую мы описали.

В разделе 8.11, при обсуждении функции setuid, мы говорили, что когда она вызывается процессом, обладающим привилегиями суперпользователя, то изменяет все три идентификатора пользователя – реальный, эффективный и сохраненный. Вызов функции setgid, который производится программой login чуть раньше, точно так же воздействует на все три идентификатора группы.

Итак, запускается командная оболочка входа. Ее родительским процессом является процесс init (с идентификатором 1), таким образом, когда командная оболочка завершит работу, процесс init получит уведомление (сигнал SIGCHLD) и сможет снова запустить процедуру входа на данном терминале. Файловые дескрипторы 0, 1 и 2 в нашей командной оболочке входа связаны с терминальным устройством. Это состояние изображено на рис. 9.3.

Далее командная оболочка входа читает файлы начальной загрузки (.profile для Bourne shell и Korn shell; .bash_profile, .bash_login или .profile для Bourne-again shell и .cshrc и .login для C shell). В них обычно изменяются значения некоторых переменных окружения и добавляется множество новых. Например, большинство пользователей создают свой список каталогов поиска (PATH) и устанавливают правильный тип терминала (TERM). Когда файлы начального запуска обработаны, мы наконец видим приглашение командной оболочки и можем вводить команды.

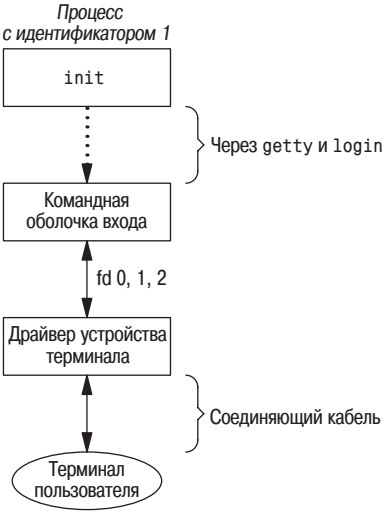


***Рис. 9.3.** Состояние процессов после входа пользователя в систему с терминала*

Вход в систему с терминала в Mac OS X

В Mac OS X вход в систему осуществляется так же, как в BSD, поскольку Mac OS X частично основана на FreeBSD, однако весь процесс выполняется через графический интерфейс.

Вход в систему с терминала в Linux

Процедура входа в систему в ОС Linux очень напоминает процедуру входа в BSD-системах. И действительно, команда `login` в Linux является производной от команды `login` в 4.3BSD. Главное отличие между процедурами входа в Linux и BSD заключается в способе, которым задается конфигурация терминала.

Конфигурационная информация, определяющая терминальные устройства, для которых должна быть вызвана программа `getty`, в Linux хранится в файле `/etc/inittab`, подобно тому, как это принято в System V. В зависимости от используемой версии `getty`, характеристики терминалов указываются либо в командной строке (команда `getty`), либо в файле `/etc/gettydefs` (команда `mgetty`).

Вход в систему с терминала в Solaris

ОС Solaris поддерживает два вида входа в систему с терминала: (а) с помощью `getty`, как это было описано выше для BSD-систем, и (б) с помощью `ttymon` – возможность, появившаяся в SVR4. Как правило, для входа с консоли используется `getty`, а для входа с других терминальных устройств – `ttymon`.

Команда `ttymon` является частью большого программного механизма, называемого SAF – Service Access Facility (механизм доступа к службам). Основная цель SAF состоит в том, чтобы обеспечить единый механизм управления службами, предоставляющими доступ к системе. (За дополнительной информацией обращайтесь к главе 6 [Rago 1993].) В нашем случае конечный результат действия этого механизма соответствует тому, что изображено на рис. 9.3, однако между процессом `init` и запуском оболочки входа выполняются несколько иные действия. Процесс `init` является родительским для процесса `sac` (service access controller – контроллер доступа к службам), который с помощью `fork` и `exec` запускает программу `ttymon`, когда система переходит в многопользовательский режим. Программа `ttymon` контролирует все терминальные порты, перечисленные в конфигурационном файле, и запускает дочерний процесс после ввода имени пользователя. Этот дочерний процесс с помощью функции `exec` запускает программу `login`, а она уже запрашивает пароль. Как только пароль будет введен, `login` запускает командную оболочку входа, и система приходит в состояние, изображенное на рис. 9.3. Единственное отличие – родительским процессом для командной оболочки становится процесс `ttymon`, а в схеме с использованием программы `getty` – процесс `init`.

9.3. Вход в систему через сетевое соединение

Главное физическое отличие между входом в систему с терминала, соединенного с главной машиной последовательным кабелем, и входом в систему через сетевое соединение состоит в том, что сетевое соединение не построено по принципу «точка-точка». В данном случае `login` – это просто служба, подобная другим сетевым службам, таким как FTP или SMTP.

В ситуациях, описанных в предыдущем разделе, процесс `init` знает, с каких устройств разрешен вход, и порождает процесс `getty` для каждого из них. Но в случае входа в систему через сетевое соединение все запросы поступают через драйвер сетевого интерфейса (например, драйвер Ethernet) и мы заранее не знаем, сколько таких запросов поступит. Вместо того, чтобы запускать отдельный процесс для каждого возможного запроса на вход в систему, мы теперь ожидаем прибытия запросов на соединение.

Чтобы одно и то же программное обеспечение могло обрабатывать вход в систему как с терминала, так и через сетевые соединения, используется программный драйвер, который называется *псевдотерминалом*. Этот драйвер эмулирует поведение обычного терминала, отображая операции с терминалом в сетевые операции и наоборот. (В главе 19 мы подробнее поговорим о псевдотерминалах.)

Вход в систему через сетевое соединение в BSD

В системах BSD большинство сетевых соединений устанавливается с помощью единственного процесса – `inetd`, который иногда называют *Internet superserver*. В этом разделе мы рассмотрим последовательность действий, которая выполняется при входе в BSD-систему через сетевое соединение. Нас не интересуют все подробности программной реализации этих процессов – их вы найдете в [Stevens, Fenner, and Rudoff 2004].

Во время запуска системы процесс `init` вызывает командный интерпретатор, который исполняет файл `/etc/rc`. Одним из демонов, запускаемых этим сценарием, является `inetd`. По окончании работы сценария родительским процессом для `inetd` становится процесс `init`. Процесс `inetd` ожидает поступления запросов на соединение по протоколу TCP/IP. Когда поступает запрос на соединение, демон `inetd` с помощью функций `fork` и `exec` запускает соответствующую программу для его обработки.

Предположим, что по адресу сервера TELNET пришел запрос на TCP-соединение. TELNET – это служба удаленного входа в систему, которая использует протокол TCP. Пользователь, находящийся за другим компьютером (который соединен с сервером сетью) или за тем же самым компьютером, инициирует вход в систему, запустив клиент TELNET:

```
telnet hostname
```

Клиент открывает TCP-соединение с узлом сети *hostname*, и на стороне сервера стартует программа, которая называется сервером TELNET. После этого клиент и сервер начинают обмен данными по протоколу TELNET. Таким образом

пользователь, запустивший клиентскую программу, выполняет вход в систему на сервере. (Разумеется, лишь в том случае, если у этого пользователя имеется учетная запись на сервере.) На рис. 9.4 показана последовательность процессов, сопутствующих запуску сервера TELNET с именем `telnetd`.

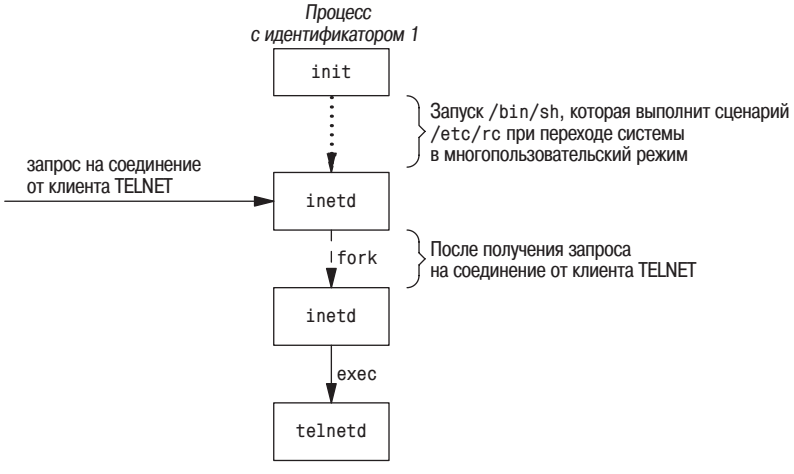


***Рис. 9.4.** Последовательность действий, приводящая к запуску сервера TELNET*

Затем процесс `telnetd` открывает устройство псевдотерминала и с помощью функции `fork` разделяется на два процесса. Родительский процесс продолжает обслуживать сетевое соединение, а дочерний процесс запускает программу `login`. Родительский процесс связан с дочерним через псевдотерминал. Перед вызовом функции `exec` дочерний процесс присоединяет файловые дескрипторы 0, 1 и 2 к псевдотерминалу. В случае удачного входа в систему программа `login` выполняет те же действия, что были описаны в разделе 9.2: она изменяет текущий каталог на домашний каталог пользователя, устанавливает идентификаторы пользователя и группы и инициализирует среду окружения. Затем программа `login` замещает себя командной оболочкой входа посредством функции `exec`. На рис. 9.5 показано состояние процессов в этот момент.

Очевидно, что между драйвером псевдотерминала и пользовательским терминалом действует еще множество процессов. Мы рассмотрим их в главе 19, когда будем говорить о псевдотерминалах более подробно.

Очень важно понимать, что независимо от того, входим ли мы в систему через терминал (рис. 9.3) или через сетевое соединение (рис. 9.5), мы получаем командную оболочку входа со стандартным вводом, стандартным выводом и стандартным выводом сообщений об ошибках, которые связаны либо с устройством терминала, либо с устройством псевдотерминала. В последующих разделах мы увидим, что оболочка входа является началом сессии POSIX.1, а терминал или псевдотерминал становится управляющим терминалом сессии.

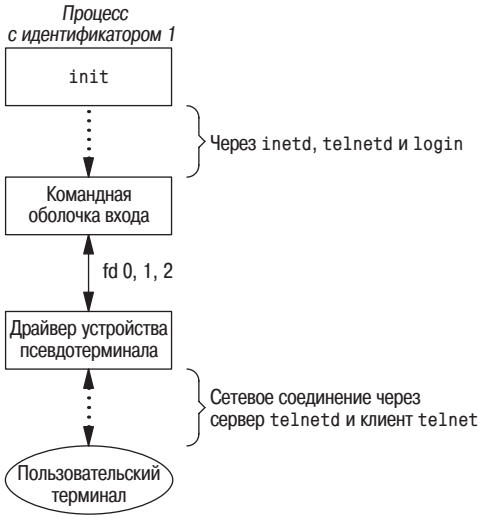


Рис. 9.5. *Состояние процессов после входа пользователя через сетевое соединение*

Вход в систему через сетевое соединение в Mac OS X

В Mac OS X процедура входа в систему через сетевое соединение совпадает с процедурой, используемой в BSD, поскольку Mac OS X частично основана на FreeBSD.

Вход в систему через сетевое соединение в Linux

Процедура входа через сетевое соединение в Linux практически такая же, как в BSD, за исключением того, что вместо процесса `inetd` используется его альтернатива – `xinetd` (extended Internet services daemon – расширенный демон сетевых служб). Демон `xinetd` предоставляет возможность более тонкого управления запуском сетевых служб по сравнению с `inetd`.

Вход в систему через сетевое соединение в Solaris

Сценарий входа через сетевое соединение в Solaris по большей части идентичен соответствующим сценариям в BSD и Linux. В Solaris, как и в BSD, используется сервер `inetd`. Его версия в Solaris имеет возможность работать под управлением механизма доступа к службам SAF, хотя такая конфигурация обычно не используется. Вместо этого сервер `inetd` запускается процессом `init`. В любом случае мы приходим к состоянию, изображенному на рис. 9.5.

9.4. Группы процессов

Каждый процесс не только имеет идентификатор процесса, но и принадлежит к определенной группе процессов. Мы еще будем встречаться с группами процессов при обсуждении сигналов в главе 10.

Группа процессов – это коллекция из одного или более процессов, обычно связанных с выполнением одного и того же задания (управление заданиями рассматривается в разделе 9.8), которые могут принимать сигналы от одного и того же терминала. Каждая группа процессов имеет уникальный идентификатор. Идентификатор группы процессов очень напоминает идентификатор процесса: это целое положительное число, которое может храниться в переменной типа `pid_t`. Функция `getpgrp` возвращает идентификатор группы процессов вызывающего процесса.

```
#include <unistd.h>

pid_t getpgrp(void);
```

Возвращает идентификатор группы процессов вызывающего процесса

В ранних версиях BSD-систем функция `getpgrp` принимала аргумент *pid* и возвращала группу процессов для заданного процесса. Стандарт Single UNIX Specification определил в качестве расширения XSI функцию `getpgid`, которая имитирует это поведение.

```
#include <unistd.h>

pid_t getpgid(pid_t pid);
```

Возвращает идентификатор группы процессов в случае успеха, −1 в случае ошибки

Если в аргументе *pid* передается значение 0, то возвращается групповой идентификатор вызывающего процесса. Таким образом, вызов

```
getpgid(0);
```

эквивалентен вызову

```
getpgrp();
```

Каждая группа процессов может иметь лидера. Идентификатор группы процессов лидера группы совпадает с его идентификатором процесса.

Вполне допустима ситуация, когда лидер группы создает группу процессов, процессы в этой группе и затем завершается. Группа процессов продолжает существовать до тех пор, пока в ней остается хотя бы один процесс, вне зависимости от того, завершил работу лидер группы или нет. Период от момента создания группы и до момента, когда последний процесс в группе покинет ее, называется временем жизни группы процессов. Последний оставшийся в группе процесс может либо завершиться, либо войти в состав другой группы процессов.

Процесс может присоединиться к группе или создать новую группу процессов с помощью функции `setpgid`. (В следующем разделе мы увидим, что функция `setsid` также создает новую группу процессов.)

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Эта функция устанавливает для процесса с идентификатором *pid* идентификатор группы процессов *pgid*. Если аргументы имеют одинаковые значения, то процесс, заданный идентификатором *pid*, становится лидером группы процессов. Если в аргументе *pid* передается значение 0, то в качестве идентификатора процесса используется идентификатор вызывающего процесса. Если же в аргументе *pgid* передается значение 0, то в качестве идентификатора группы процессов используется значение аргумента *pid*.

Процесс может установить идентификатор группы только для себя самого и для своих дочерних процессов. Но процесс не может изменить идентификатор группы процессов дочернего процесса, который вызвал одну из функций семейства `exec`.

В большинстве командных оболочек, которые поддерживают управление заданиями, функция `setpgid` вызывается после `fork`, чтобы родительский процесс мог назначить идентификатор группы процессов дочернему процессу, а дочерний процесс – установить свой собственный идентификатор группы процессов. Один из этих вызовов является излишним, но, выполняя оба, мы гарантируем, что дочерний процесс будет помещен в его собственную группу процессов в любом случае. Иначе мы столкнулись бы с ситуацией гонки за ресурсами, когда членство дочернего процесса зависело бы от того, какой из процессов первым получит управление.

При обсуждении сигналов мы увидим, как можно послать сигнал отдельному процессу (по идентификатору процесса) или группе процессов (по идентификатору группы процессов). Аналогично функция `waitpid` позволяет нам дождаться завершения конкретного процесса или одного из процессов в заданной группе.

9.5. Сессии

Сессия – это коллекция из одной или более групп процессов. Рассмотрим в качестве примера ситуацию, изображенную на рис. 9.6. Здесь мы имеем три группы процессов в одной сессии.

Процессы обычно помещаются в группу командной оболочкой при конвейерной обработке данных. Например, состояние процессов, показанное на рис. 9.6, может быть достигнуто следующей последовательностью команд:

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

Создание новой сессии производится с помощью вызова функции `setsid`.

```
#include <unistd.h>

pid_t setsid(void);
```

Возвращает идентификатор группы процессов
в случае успеха, −1 в случае ошибки

Если вызывающий процесс не является лидером группы, то функция создает новую сессию. При этом происходит следующее.

1. Процесс становится лидером новой сессии. (Лидер сессии – это процесс, который создает сессию.) Этот процесс – единственный процесс в новой сессии.
2. Процесс становится лидером новой группы процессов. Идентификатором новой группы процессов становится идентификатор вызывающего процесса.
3. Процесс теряет управляющий терминал. (Управляющие терминалы обсуждаются в следующем разделе.) Если у процесса был управляющий терминал перед вызовом функции `setsid`, связь с ним разрывается.

Эта функция возвращает признак ошибки, если вызывающий процесс уже является лидером группы. Чтобы избежать этой ошибки, обычно вызывают функцию `fork`, затем родительский процесс завершается, а дочерний процесс продолжает работу. В этом случае можно гарантировать, что дочерний процесс не будет лидером группы, поскольку его идентификатор группы процессов наследуется от родительского процесса, но сам он получит новый идентификатор процесса. Таким образом, совершенно невозможно, чтобы идентификатор дочернего процесса совпал с унаследованным идентификатором группы процессов.

Стандарт Single UNIX Specification оговаривает только определение *лидер сессии*, но в нем нет определения *идентификатор сессии*. Очевидно, что лидер сессии – это отдельный процесс, который имеет уникальный идентификатор процесса, поэтому можно утверждать, что идентификатор сессии – это

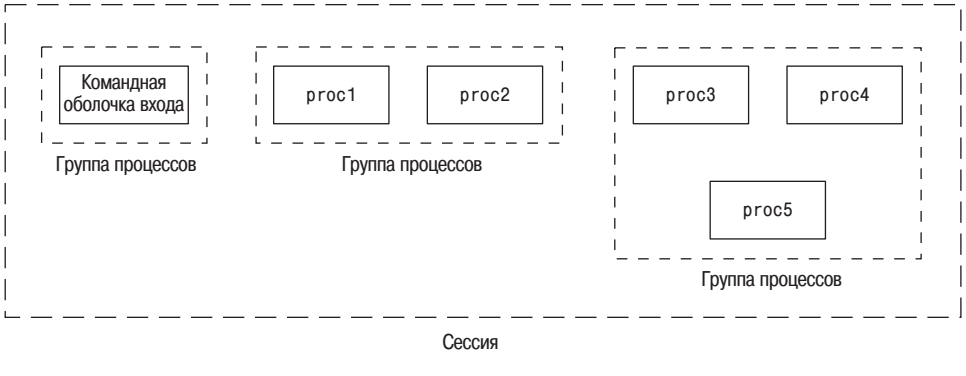


Рис. 9.6. *Распределение процессов по группам процессов и сессиям*

идентификатор процесса лидера сессии. Такое понимание идентификатора сессии было введено в SVR4. Исторически BSD-системы не поддерживали это понятие, но впоследствии положение изменилось. Функция `getsid` возвращает идентификатор группы процессов лидера сессии. Она включена в стандарт Single UNIX Specification как расширение XSI.

> Некоторые реализации, такие как Solaris, следуя Single UNIX Specification, избегают понятия «идентификатор сессии»; вместо этого они используют термин «идентификатор группы процессов лидера сессии». Эти два понятия эквивалентны, так как лидер сессии всегда является лидером группы процессов.

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

Возвращает идентификатор группы процессов лидера сессии в случае успеха, −1 в случае ошибки

Если в аргументе *pid* передается значение 0, функция `getsid` возвращает идентификатор группы процессов лидера сессии, которой принадлежит вызывающий процесс. Из соображений безопасности некоторые реализации могут ограничивать возможность получения идентификатора группы процессов лидера сессии, если в аргументе *pid* передается идентификатор процесса, не принадлежащего той же сессии, что и вызывающий процесс.

9.6. Управляющий терминал

Сессии и группы процессов обладают еще некоторыми характеристиками.

- Сессия может иметь только один *управляющий терминал*. Обычно это устройство терминала (в случае входа в систему с терминала) или устройство псевдотерминала (в случае входа в систему через сетевое соединение), с которого был произведен вход в систему.
- Лидер сессии, который устанавливает соединение с управляющим терминалом, называется *управляющим процессом*.
- Группы процессов в пределах одной сессии могут подразделяться на единственную *группу процессов переднего плана* и одну или более *групп фоновых процессов*.
- Если сессия имеет управляющий терминал, то в ней будет одна группа процессов переднего плана, а все остальные группы процессов в сессии будут группами фоновых процессов.
- Когда мы вводим с клавиатуры терминала символ прерывания (обычно DELETE или Control-C), всем процессам в группе процессов переднего плана будет послан сигнал прерывания.
- Когда мы вводим с клавиатуры терминала символ завершения (обычно Control-\), всем процессам в группе процессов переднего плана будет послан сигнал завершения.

- Если интерфейс терминала обнаружит разрыв связи с модемом или сетью, то управляющему процессу (лидеру сессии) будет послан сигнал, оповещающий о разрыве связи.

Эти характеристики показаны на рис. 9.7.

Обычно не приходится беспокоиться об управляющем терминале – он устанавливается автоматически после входа в систему.

> Стандарт POSIX.1 оставляет за конкретной реализацией выбор механизма размещения управляющего терминала. Фактические действия мы будем рассматривать в разделе 19.4.
>
> Системы, производные от System V, производят размещение управляющего терминала сессии в тот момент, когда лидер сессии открывает первое устройство терминала, еще не связанное с сессией. Это предполагает, что лидер сессии, вызывая функцию `open`, не указывает флаг `O_NOCTTY` (раздел 3.3).
>
> Системы, основанные на BSD, размещают управляющий терминал сессии, когда лидер сессии вызывает функцию `ioctl`, передавая ей в аргументе *`request`* значение `TIOCSCTTY` (третий аргумент – пустой указатель). Чтобы вызов завершился успехом, сессия не должна иметь управляющего терминала. (Обычно вызов функции `ioctl` следует за вызовом функции `setsid` – это гарантирует, что процесс является лидером сессии без управляющего терминала.) Флаг `O_NOCTTY` функции `open` не используется в BSD-системах, за исключением случаев, когда необходима поддержка совместимости с другими системами.

Иногда возникают ситуации, когда программа должна произвести обмен данными с управляющим терминалом даже в том случае, когда стандартные потоки ввода-вывода перенаправлены. Чтобы обеспечить возможность такого обмена, необходимо открыть с помощью функции `open` файл `/dev/tty`. Этот

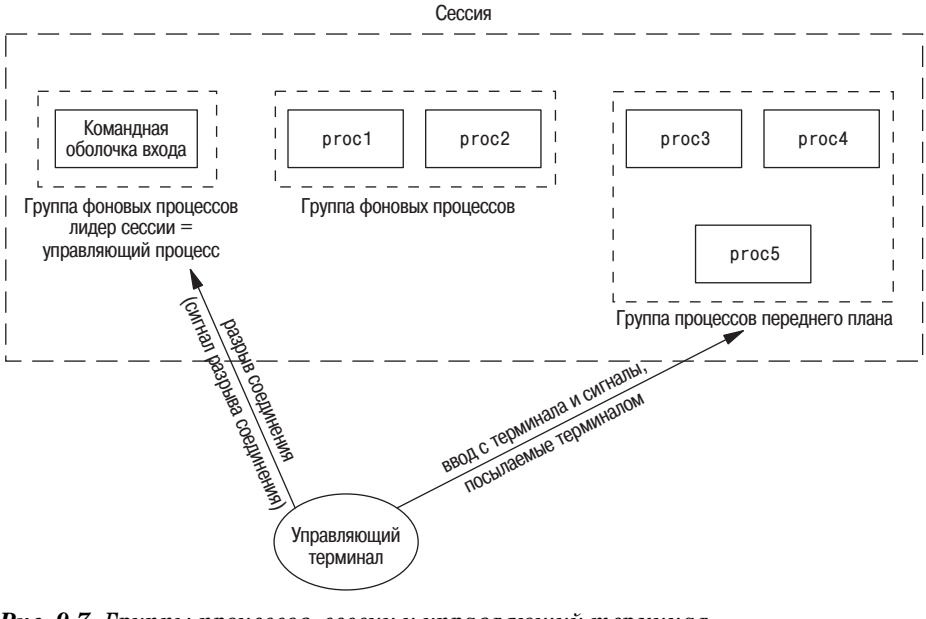


***Рис. 9.7.** Группы процессов, сессии и управляющий терминал*

специальный файл является синонимом управляющего терминала в ядре. Разумеется, если программа не имеет управляющего терминала, то попытка открыть его окончится неудачей.

Классический пример – функция `getpass`(3), которая читает пароль, вводимый с клавиатуры (естественно, при отключенном отображении вводимых символов). Эта функция вызывается программой `crypt`(1) и может быть использована в конвейере с другими командами. Например, команда

```
crypt < salaries | lpr
```

расшифрует содержимое файла `salaries` и перенаправит результат на принтер. Поскольку программа `crypt` читает входной файл со стандартного ввода, следовательно, стандартный ввод не может использоваться для ввода пароля. Кроме того, программа `crypt` спроектирована так, что при каждом вызове она заставляет нас снова вводить пароль и не дает сохранить его в файле (в противном случае это стало бы лазейкой в системе безопасности).

Существуют способы, позволяющие взломать шифр, используемый программой `crypt`. За дополнительной информацией о шифровании файлов обращайтесь к [Garfinkel et al. 2003].

9.7. Функции tcgetpgrp, tcsetpgrp и tcgetsid

Теперь нам нужен способ сообщить ядру, какая группа процессов является группой переднего плана, чтобы драйвер терминала знал, какому процессу передавать ввод с терминала и кому отправлять сигналы (рис. 9.7).

```
#include <unistd.h>

pid_t tcgetpgrp(int filedes);
```

Возвращает идентификатор группы процессов переднего плана в случае успеха, −1 в случае ошибки

```
int tcsetpgrp(int filedes, pid_t pgrpid);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Функция `tcgetpgrp` возвращает идентификатор группы процессов переднего плана, связанной с открытым файловым дескриптором терминала *filedes*.

Если процесс обладает управляющим терминалом, то он может вызвать функцию `tcsetpgrp`, чтобы назначить группу процессов с идентификатором *pgrpid* группой процессов переднего плана. Значение аргумента *pgrpid* должно быть идентификатором группы процессов в той же самой сессии, а аргумент *filedes* должен быть дескриптором управляющего терминала сессии.

Большинство приложений не вызывают эти две функции напрямую. Обычно они вызываются командными оболочками, которые поддерживают управление заданиями.

Стандарт Single UNIX Specification определяет в качестве расширения XSI функцию `tcgetsid`, которая позволяет приложению получить идентифика-

тор группы процессов лидера сессии по заданному файловому дескриптору управляющего терминала.

```
#include <termios.h>

pid_t tcgetsid(int filedes);
```

Возвращает идентификатор группы процессов лидера сессии в случае успеха, −1 в случае ошибки

Приложения, которым необходимо взаимодействовать с управляющим терминалом, могут использовать функцию `tcgetsid`, чтобы получить идентификатор сессии для лидера сессии, который владеет управляющим терминалом (что эквивалентно идентификатору группы процессов лидера сессии).

9.8. Управление заданиями

Возможность управления заданиями была добавлена в BSD около 1980 года. Она позволяет запустить несколько заданий (групп процессов) с одного терминала и определить, какие из них получат доступ к терминалу, а какие будут выполняться в фоновом режиме. Управление заданиями поддерживается, если соблюдаются следующие условия:

1. Командная оболочка должна поддерживать управление заданиями.
2. Драйвер терминального устройства в ядре должен поддерживать управление заданиями.
3. Ядро должно поддерживать ряд сигналов, с помощью которых осуществляется управление заданиями.

> В SVR3 предоставлялась возможность управления заданиями в иной форме, которая называлась *уровнями командной оболочки* (*shell layers*). Однако стандарт POSIX.1 выбрал форму управления заданиями, реализованную в BSD; именно она здесь и описывается. В ранних версиях стандарта поддержка управления заданиями была необязательной, однако теперь POSIX.1 требует, чтобы все POSIX-совместимые платформы поддерживали эту возможность.

Для нас сейчас важно, что возможность управления заданиями позволяет запустить задание на переднем плане или в фоновом режиме. Задание – это просто набор процессов, часто объединенных в конвейер. Например, команда

```
vi main.c
```

запустит задание, которое содержит один процесс переднего плана. Команды

```
pr *.c | lpr &
make all &
```

запустят два фоновых задания. Все процессы, запускаемые в рамках этих заданий, являются фоновыми.

Как мы уже говорили, чтобы пользоваться управлением заданиями, необходима командная оболочка, которая поддерживает эту возможность. Доволь-

но просто перечислить командные оболочки, которые поддерживали управление заданиями в старых системах. Так, C shell имела поддержку управления заданиями, Bourne shell – нет, а Korn shell – в зависимости от того, поддерживала ли управление заданиями сама платформа. Но позднее командная оболочка C shell была перенесена на системы, которые не поддерживали управление заданиями (например, ранние версии System V), а в SVR4 можно было включить поддержку управления заданиями в командной оболочке Bourne shell, запустив ее командой `jsh` вместо `sh`. В настоящее время возможность управления заданиями в Korn shell зависит от того, поддерживает ли эту возможность сама система. Командная оболочка Bourne-again shell поддерживает управление заданиями. Далее мы будем говорить о командной оболочке, поддерживающей управление заданиями, в противоположность командной оболочке, которая не имеет такой поддержки, если различия между конкретными оболочками для нас несущественны.

При запуске задания в фоновом режиме командная оболочка присваивает ему идентификатор задания и выводит один или более идентификаторов процессов. Ниже показано, как это делает командная оболочка Korn shell:

```
$ make all > Make.out &
[1]     1475
$ pr *.c | lpr &
[2]     1490
$                                   просто нажмите клавишу ввода
[2] +  Done          pr *.c | lpr &
[1] +  Done          make all > Make.out &
```

Задание с номером 1 представлено программой `make`, а соответствующий ей процесс имеет идентификатор 1475. Задание с номером 2 представлено конвейером, в котором первый процесс имеет идентификатор 1490. По завершении заданий, когда мы нажимаем клавишу ввода, командная оболочка сообщает, выполнение каких заданий было завершено. Командная оболочка не выводит сообщения об изменении состояния фоновых заданий по своей инициативе – только перед тем, как она выведет приглашение, которое позволяет нам вводить новые команды. В противном случае вывод сообщений командной оболочки мог бы смешиваться с вводимыми символами. Поэтому, чтобы вызвать сообщение о состоянии фоновых заданий, после появления приглашения командной оболочки нужно нажать клавишу ввода.

Управление заданием переднего плана через драйвер терминала, осуществляется с помощью ввода специальных символов, например, символа приостановки (обычно Control-Z). Ввод этого символа заставляет драйвер передать сигнал `SIGTSTP` всем процессам группы процессов переднего плана. Задания, выполняемые в фоновом режиме, при этом не затрагиваются. Драйвер терминала посылает сигналы процессам переднего плана при вводе трех специальных символов.

- Ввод символа прерывания (обычно DELETE или Control-C) порождает сигнал `SIGINT`.
- Ввод символа завершения (обычно Control-\) порождает сигнал `SIGQUIT`.

- Ввод символа приостановки (обычно Control-Z) порождает сигнал SIGTSTP.

В главе 18 мы увидим, как можно привязать эту функциональность к любым другим символам и как запретить обработку этих специальных символов драйвером терминала.

Драйверу терминала приходится обрабатывать и другие ситуации, связанные с управлением заданиями. Так как у нас может быть одно задание переднего плана и одно или более фоновых заданий, то необходимо разобраться, какие из них будут получать символы, вводимые с терминала. Ввод с терминала получает только задание переднего плана. Попытка фонового задания прочитать ввод с терминала не считается ошибкой, но оно будет обнаружено драйвером, который и пошлет специальный сигнал SIGTTIN фоновому заданию. Этот сигнал обычно приводит к остановке фонового задания, командная оболочка выводит сообщение об этом, и мы можем перевести задание на передний план, чтобы оно получило возможность прочитать ввод с терминала, например:

```
$ cat > temp.foo &      программа запущена в фоновом режиме, но пытается
                        читать со стандартного ввода
[1] 1681
$                       нажимаем клавишу ввода
[1] + Stopped (SIGTTIN)    cat > temp.foo &
$ fg %1                 перевести задание с номером 1 на передний план
cat > temp.foo          командная оболочка сообщает, какое задание находится
                        на переднем плане
hello, world            вводим одну строку
^D                      вводим символ EOF (конец файла)
$ cat temp.foo          проверяем, попала ли введенная строка в файл
hello, world
```

Командная оболочка запускает в фоновом режиме процесс cat, который пытается прочитать символы со стандартного ввода (управляющий терминал). Драйвер терминала знает, что это фоновое задание, и посылает ему сигнал SIGTTIN. Командная оболочка определяет изменение состояния своего дочернего процесса (вспомните обсуждение функций wait и waitpid в разделе 8.6) и сообщает нам о том, что задание было приостановлено. После этого мы с помощью команды fg перемещаем приостановленное задание на передний план. (За дополнительной информацией о командах управлениями заданиями, таких как fg или bg, и о различных способах идентификации заданий обращайтесь к страницам справочного руководства по командной оболочке.) В результате командная оболочка переместила задание в группу процессов переднего плана (tcsetpgrp) и передала группе сигнал продолжения работы (SIGCONT). Поскольку теперь задание принадлежит к группе процессов переднего плана, оно получает возможность читать данные с управляющего терминала.

А что произойдет, если фоновое задание попытается вывести что-нибудь на терминал? Мы можем разрешить или запретить эту возможность, обычно для этого используется команда stty(1). (В главе 18 мы покажем, как управлять этой возможностью из программы.) Например:

```
$ cat temp.foo &      запустить в фоновом режиме
```

```
[1] 1719
$ hello, world        вывод фонового задания появляется после приглашения
                      командной оболочки нажимаем клавишу ввода
[1] +  Done       cat temp.foo &
$ stty tostop         запретить фоновым заданиям вывод на терминал
$ cat temp.foo &      попробуем еще раз запустить команду в фоновом режиме
[1] 1721
$                нажимаем ввод и обнаруживаем, что задание приостановлено
[1] + Stopped(SIGTTOU)        cat temp.foo &
$ fg %1               возобновим работу задания на переднем плане
cat temp.foo          оболочка сообщила, какое из заданий выполняется
                      на переднем плане
hello, world          а это вывод задания
```

Когда мы запретили возможность вывода на терминал для фоновых заданий, утилита `cat` была заблокирована при попытке записи на стандартный вывод, так как драйвер терминала определил, что запись производится из фонового процесса, и передал ему сигнал `SIGTTOU`. Как и в предыдущем примере, мы с помощью команды `fg` перевели задание на передний план, благодаря чему оно получило возможность успешно отработать.

На рис. 9.8 изображена схема управления заданиями, описанная выше. Сплошные линии внутри драйвера терминала означают, что ввод-вывод на терминал и сигналы, посылаемые терминалом, всегда связаны с группой процессов переднего плана. Пунктирная линия, соответствующая сигналу `SIGTTOU`, означает, что возможность вывода на терминал для фоновых процессов может отсутствовать.

Является ли управление заданиями необходимым или только желательным? Изначально управление заданиями было спроектировано и реализовано еще до появления и широкого распространения терминалов, предоставляющих многооконный интерфейс. Одни берут на себя смелость утверждать, что хорошо продуманная многооконная система ликвидирует потребность в управлении заданиями. Другие выражают недовольство чрезмерно сложной реализацией управления заданиями, которое должно поддерживаться ядром, драйвером терминала, командной оболочкой и отдельными приложениями. Третьи используют и управление заданиями, и многооконную систему, утверждая, что обе возможности одинаково необходимы. Однако, независимо от вашего мнения, эта функциональность является обязательной для реализации согласно стандарту POSIX.1.

9.9. Выполнение программ командной оболочкой

Давайте рассмотрим, как командная оболочка запускает программы и как это связано с понятиями группы процессов, управляющего терминала и сессии. Для этого мы воспользуемся командой `ps`.

Для начала рассмотрим командную оболочку, которая не поддерживает управление заданиями – это классическая Bourne shell под управлением Solaris. Запустив команду

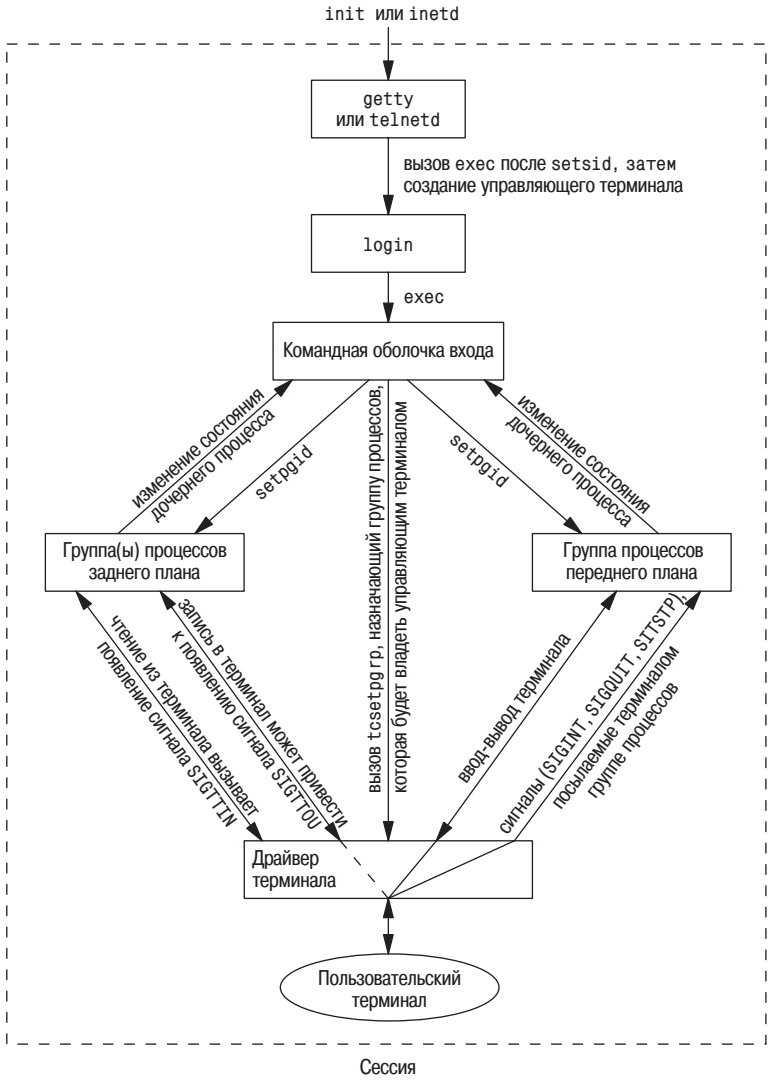


***Рис. 9.8.** Схема взаимодействия заданий переднего плана и фонового режима с драйвером терминала*

```
ps -o pid,ppid,pgid,sid,comm
```

мы получим:

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1774   949   949   949 ps
```

Как мы и ожидали, родительским процессом для команды ps является командная оболочка. И командная оболочка, и команда ps находятся в одной и той же сессии и принадлежат одной и той же группе процессов переднего плана (949). Мы говорим, что число 949 представляет группу процессов переднего плана, потому что в командных оболочках, не имеющих поддержки управления заданиями, мы получаем именно группу процессов.

> На некоторых платформах команда ps может выводить идентификатор группы процессов, связанной с управляющим терминалом сессии. Это значение отображается в столбце TPGID. К сожалению, вывод команды ps зачастую различается в разных версиях UNIX. Например, Solaris 9 не поддерживает такой возможности. В FreeBSD 5.2.1 и в Mac OS X 10.3 команда
>
> ```
> ps -o pid,ppid,pgid,sess,tpgid,command
> ```
>
> а в Linux 2.4.22 – команда
>
> ```
> ps -o pid,ppid,pgrp,session,tpgid,comm
> ```
>
> выводят то, что нам необходимо.
>
> Обратите внимание: было бы неправильно ассоциировать процесс с идентификатором группы процессов терминала (столбец TPGID – terminal process group ID). У процесса нет такого признака, как группа процессов терминала. Процесс принадлежит группе процессов, а группа процессов принадлежит сессии. Сессия может иметь управляющий терминал, а может и не иметь. Если сессия имеет управляющий терминал, то терминальное устройство знает идентификатор группы процессов переднего плана. Это значение может быть установлено в драйвере терминала с помощью функции tcsetpgrp, как это видно на рис. 9.8. Идентификатор группы процессов переднего плана – это атрибут терминала, а не процесса. Значение, выводимое командой ps в колонке TPGID, берется из драйвера терминала. Если окажется, что сессия не имеет управляющего терминала, то команда ps выведет в этой колонке значение -1.

Если мы запустим команду в фоновом режиме:

```
ps -o pid,ppid,pgid,sid,comm &
```

то единственное, что изменится – это идентификатор процесса команды:

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1812   949   949   949 ps
```

Эта командная оболочка не поддерживает управление заданиями, поэтому фоновое задание не помещается в свою собственную группу процессов и не теряет связь с управляющим терминалом.

А теперь посмотрим, как Bourne shell обслуживает конвейеры. После запуска команды

```
ps -o pid,ppid,pgid,sid,comm | cat1
```

мы получаем:

```
PID  PPID  PGID   SID COMMAND
 949   947   949   949 sh
```

```
1823   949   949   949 cat1
1824  1823   949   949 ps
```

(Программа `cat1` – это просто копия программы `cat`, сохраненная под другим именем. У нас есть еще одна копия программы `cat` под именем `cat2`, которую мы будем использовать чуть позже в этом же разделе. Запуск двух копий программы `cat` в одном конвейере дает нам возможность различать их.) Обратите внимание: последний процесс в конвейере является дочерним процессом командной оболочки, а первый процесс в конвейере является дочерним по отношению к последнему. Похоже на то, что командная оболочка создала собственную копию, которая затем в обратном порядке породила каждый из процессов в конвейере.

Если мы запустим ту же команду в фоновом режиме:

```
ps -o pid,ppid,pgid,sid,comm | cat1 &
```

то изменятся только идентификаторы процессов. Поскольку командная оболочка поддерживает управление заданиями, идентификатор группы процессов фонового режима сохраняет значение 949, равно как и идентификатор сессии.

Что произойдет, если в этой оболочке фоновый процесс попытается прочитать ввод из управляющего терминала? Например, предположим, что мы запустили такую команду:

```
cat > temp.foo &
```

При наличии поддержки управления заданиями, если фоновое задание, находящееся в группе процессов фонового режима, попытается произвести чтение из управляющего терминала, ему будет послан сигнал `SIGTTIN`. При отсутствии поддержки управления заданиями командная оболочка автоматически перенаправляет стандартный ввод фонового процесса в устройство `/dev/null`, если процесс не перенаправит его самостоятельно. При попытке чтения из устройства `/dev/null` приложение получает признак конца файла. Это означает, что фоновый процесс `cat` сразу же прочитает признак конца файла и завершит работу нормальным образом.

Предыдущий абзац описывает случай, когда фоновый процесс обращается к управляющему терминалу через стандартный ввод, но что произойдет, если фоновый процесс попытается открыть устройство `/dev/tty` и прочитать входные данные из него? Ответ: «Это зависит от реализации», но, наверное, это не то, что нам нужно. Например, команда

```
crypt < salaries | lpr &
```

является таким конвейером. Мы запускаем эту команду в фоновом режиме, но программа `crypt` открывает `/dev/tty`, изменяет характеристики терминала (запрещает отображение вводимых символов), читает из устройства и восстанавливает характеристики терминала. Когда мы запустим такой конвейер в фоновом режиме, на экране появится приглашение `Password:`, но введенный нами пароль для шифрования будет прочитан командной оболочкой, кото-

рая воспримет введенную строку как команду и попытается ее запустить. Следующая строка, введенная в командной оболочке, будет воспринята как пароль, в результате файл будет расшифрован неправильно и на принтер будет отправлен бессмысленный набор символов. Здесь присутствуют два процесса, которые пытаются читать из одного и того же устройства в одно и то же время, и конечный результат таких попыток зависит от системы. Управление заданиями, как было описано ранее, позволяет лучше организовать совместное использование одного терминала несколькими процессами.

Вернемся к нашему примеру с Bourne shell, запустим три процесса в конвейере и посмотрим, как эта оболочка осуществляет управление процессами:

```
ps -o pid,ppid,pgid,sid,comm | cat1 | cat2
```

Эта команда выведет следующее:

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1988   949   949   949 cat2
 1989  1988   949   949 ps
 1990  1988   949   949 cat1
```

> Пусть вас не тревожит, если в вашей системе вы получите неверные имена команд. Иногда можно получить примерно такой результат:
>
> ```
>   PID  PPID  PGID  SID COMMAND
>   949   947   949  949 sh
>  1831   949   949  949 sh
>  1832  1831   949  949 ps
>  1833  1831   949  949 sh
> ```
>
> Дело в том, что процесс `ps` конкурирует с командной оболочкой за обладание процессором, когда та запускает команды `cat` с помощью функций `fork` и `exec`. В ситуации, показанной выше, командная оболочка еще не успела завершить вызовы функции `exec`, а команда `ps` уже вывела список процессов.

И опять последний процесс является дочерним процессом командной оболочки, а все предыдущие процессы – дочерними процессами последнего процесса. Рис. 9.9 показывает смысл происходящего. Так как последний процесс в конвейере (`cat2`) является дочерним по отношению к командной оболочке, то она получит извещение о его завершении.

Теперь рассмотрим те же самые примеры, но уже применительно к ОС Linux и командной оболочке, которая поддерживает управление заданиями. Мы увидим, каким образом эти командные оболочки обслуживают фоновые задания. В этом примере мы использовали оболочку Bourne-again shell; результаты в других оболочках практически идентичны. Команда

```
ps -o pid,ppid,pgrp,session,tpgid,comm
```

выводит следующее:

```
  PID  PPID  PGRP  SESS  TPGID COMMAND
 2837  2818  2837  2837   5796  bash
 5796  2837  5796  2837   5796  ps
```

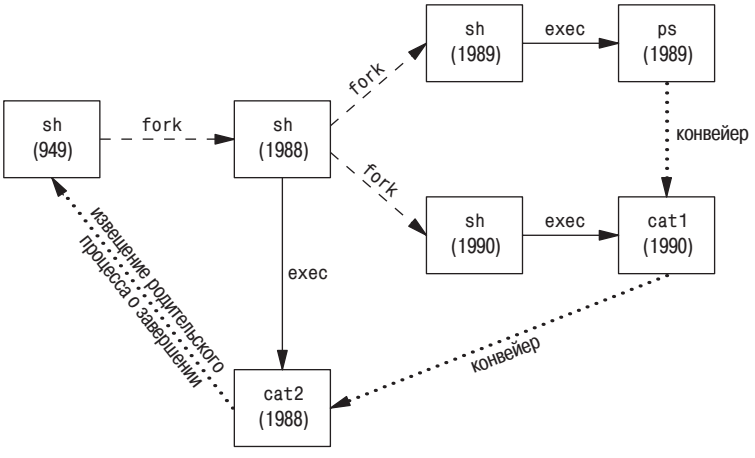


***Рис. 9.9.** Процессы в конвейере ps | cat1 | cat2, запущенном в оболочке Bourne shell*

(Начиная с этого примера мы будем отмечать группу процессов переднего плана **жирным шрифтом**.) Здесь сразу же видны отличия от Bourne shell. Оболочка Bourne-again shell помещает задание переднего плана (`ps`) в собственную группу (5796). Команда `ps` – лидер группы процессов и единственный процесс в этой группе.

Кроме того, эта группа является группой процессов переднего плана, так как она имеет управляющий терминал. На время выполнения команды `ps` командная оболочка становится группой фоновых процессов. Однако обратите внимание на тот факт, что обе группы процессов, 2837 и 5796, принадлежат одной сессии. В примерах в этом разделе мы увидим, что сессия никогда не изменяется.

Запуск этой же команды в фоновом режиме:

```
ps -o pid,ppid,pgrp,session,tpgid,comm &
```

дает нам

```
  PID  PPID  PGRP  SESS  TPGID COMMAND
 2837  2818  2837  2837   2837 bash
 5797  2837  5797  2837   2837 ps
```

И опять команда `ps` была помещена в собственную группу процессов, но на этот раз группа процессов (5797) уже не является группой процессов переднего плана. Теперь это группа фоновых процессов. Значение `TPGID` (2837) указывает на то, что группа процессов переднего плана соответствует командной оболочке.

Запуск двух команд в конвейере:

```
ps -o pid,ppid,pgrp,session,tpgid,comm | cat1
```

дает

```
  PID  PPID  PGRP  SESS  TPGID COMMAND
 2837  2818  2837  2837   5799 bash
 5799  2837  5799  2837   5799 ps
 5800  2837  5799  2837   5799 cat1
```

Оба процесса, `ps` и `cat1`, теперь помещены в отдельную группу процессов (5799), которая является группой процессов переднего плана. Здесь также имеются отличия по сравнению с аналогичным примером для Bourne shell. Командная оболочка Bourne shell первым запускала последний процесс в конвейере, и этот процесс становился родительским по отношению к первому процессу в конвейере. Здесь же родительским процессом для обеих команд становится Bourne-again shell. Если этот же конвейер запустить в фоновом режиме:

```
ps -o pid,ppid,pgrp,session,tpgid,comm | cat1 &
```

результаты будут похожими, но на этот раз `ps` и `cat1` помещаются в одну и ту же группу фоновых процессов:

```
  PID  PPID  PGRP  SESS  TPGID COMMAND
 2837  2818  2837  2837   2837 bash
 5801  2837  5801  2837   2837 ps
 5802  2837  5801  2837   2837 cat1
```

Обратите внимание, что порядок, в котором создаются процессы, может несколько варьироваться в зависимости от выбранной командной оболочки.

9.10. Осиротевшие группы процессов

Мы уже говорили, что процесс, родительский процесс которого завершился, называется осиротевшим и наследуется процессом `init`. Теперь посмотрим, что произойдет, если вся группа процессов лишится родительского процесса, и как стандарт POSIX.1 регламентирует эту ситуацию.

Пример

Рассмотрим процесс, который порождает дочерний процесс и завершает работу. Хотя в этом нет ничего необычного (такое случается постоянно), тем не менее, что произойдет, если дочерний процесс будет приостановлен (с помощью управления заданиями), а родительский процесс завершится? Каким образом можно возобновить работу дочернего процесса и узнает ли он о том, что осиротел? Эта ситуация показана на рис. 9.10: родительский процесс порождает дочерний процесс, затем потомок приостанавливается, а родительский процесс завершается.

Программа, которая создает эту ситуацию, приводится в листинге 9.1. В ней имеется ряд новых для нас особенностей. Мы подразумеваем, что она будет выполняться под управлением командной оболочки, поддерживающей управление заданиями. В предыдущем разделе мы уже говорили, что командная оболочка помещает процесс переднего плана в его собственную группу (в дан-

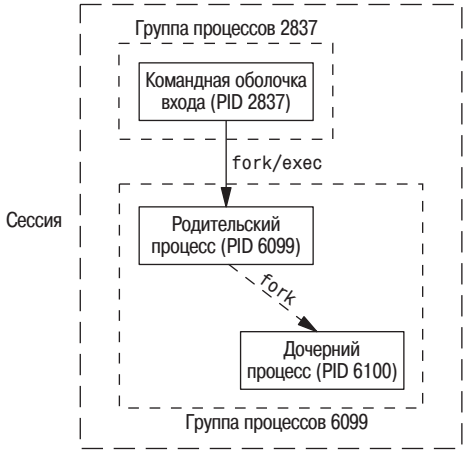


Рис. 9.10. *Пример осиротевшей группы процессов*

ном примере – 6099), а сама остается в своей группе (2837). Дочерний процесс наследует группу процессов от родительского процесса. Итак, после вызова функции `fork`

- Родительский процесс приостанавливается на 5 секунд. Это наш (пусть и несовершенный) способ позволить дочернему процессу первым продолжить работу, прежде чем родительский процесс завершится.
- Дочерний процесс устанавливает обработчик сигнала `SIGHUP`. Таким образом мы сможем узнать, посылается ли сигнал `SIGHUP` дочернему процессу. (Обработчики сигналов будут обсуждаться в главе 10.)
- Дочерний процесс с помощью функции `kill` передает самому себе сигнал `SIGTSTP`. Благодаря этому дочерний процесс приостанавливается точно так же, как приостанавливается задание переднего плана при вводе символа остановки (Control-Z).
- Когда родительский процесс завершается, дочерний процесс становится «сиротой» и обретает себе родителя в лице процесса `init` с идентификатором 1.
- С этого момента дочерний процесс становится членом *осиротевшей группы процессов*. Стандарт POSIX.1 определяет осиротевшую группу процессов как группу, в которой родительский процесс любого члена группы либо сам является членом группы, либо не является членом сессии, которой принадлежит группа. Другими словами, группа процессов не считается осиротевшей, пока в группе есть процесс, который имеет родителя в другой группе процессов, но в той же самой сессии. Если группа процессов не является осиротевшей, то есть шанс, что один из ее родительских процессов, расположенных в других группах процессов, но в той же самой сессии, перезапустит приостановленный дочерний процесс. В нашем же случае родитель каждого процесса в группе принадлежит другой сессии (так, для процесса 6100 родительским является процесс с идентификатором 1).

- Поскольку группа процессов оказывается осиротевшей, когда завершается родительский процесс, то каждому приостановленному процессу в этой группе (как наш дочерний процесс) посылается сигнал SIGHUP и вслед за ним сигнал SIGCONT, согласно требованиям стандарта POSIX.1.
- Это приводит к тому, что дочерний процесс возобновляет работу после обработки сигнала SIGHUP. Реакция на этот сигнал по умолчанию – завершение процесса, поэтому мы должны предусмотреть функцию-обработчик, чтобы перехватить его. Поэтому мы предполагаем, что вызов функции printf сначала будет произведен в функции sig_hup, а затем в функции pr_ids.

Ниже приводится результат работы программы из листинга 9.1:

```
$ ./a.out
родитель: pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099
потомок: pid = 6100, ppid = 6099, pgrp = 6099, tpgrp = 6099
$ принят сигнал SIGHUP, pid = 6100
потомок: pid = 6100, ppid = 1, pgrp = 6099, tpgrp = 2837
ошибка чтения из управляющего TTY, errno = 5
```

Обратите внимание, что приглашение командной оболочки появилось среди строк, выводимых дочерним процессом. Произошло это потому, что вывод на терминал осуществляют два процесса – командная оболочка и дочерний процесс. Как мы и ожидали, идентификатор родительского процесса стал равным 1.

Листинг 9.1. Создание осиротевшей группы процессов

```
#include "apue.h"
#include <errno.h>

static void
sig_hup(int signo)
{
    printf("принят сигнал SIGHUP, pid = %d\n", getpid());
}

static void
pr_ids(char *name)
{
    printf("%s: pid = %d, ppid = %d, pgrp = %d, tpgrp = %d\n",
           name, getpid(), getppid(), getpgrp(), tcgetpgrp(STDIN_FILENO));
    fflush(stdout);
}

int
main(void)
{
    char c;
    pid_t pid;

    pr_ids("родитель");
    if ((pid = fork()) < 0) {
```

```
            err_sys("ошибка вызова функции fork");
        } else if (pid > 0) {          /* родительский процесс */
            sleep(5);  /* приостановиться, чтобы дать потомку отработать первым */
            exit(0);   /* затем родительский процесс завершается */
        } else {                       /* дочерний процесс */
            pr_ids("потомок");
            signal(SIGHUP, sig_hup);   /* установить обработчик сигнала */
            kill(getpid(), SIGTSTP);   /* остановить самого себя */
            pr_ids("потомок");         /* вывести данные, */
                                       /* когда процесс будет возобновлен */
            if (read(STDIN_FILENO, &c, 1) != 1)
                printf("ошибка чтения из управляющего TTY, errno = %d\n",
                       errno);
            exit(0);
        }
    }
```

После вызова функции `pr_ids` в дочернем процессе производится попытка чтения со стандартного ввода. Как мы уже говорили ранее в этой главе, когда процесс из группы фоновых процессов пытается читать из управляющего терминала, то группе передается сигнал `SIGTTIN`. Но здесь мы имеем дело с осиротевшей группой процессов; если ядро остановит этим сигналом процесс из такой группы, то этот процесс, скорее всего, никогда не будет возобновлен. Стандарт POSIX.1 требует, чтобы в такой ситуации функция `read` возвращала признак ошибки с кодом ошибки `EIO` в переменной `errno` (в данной системе этот код имеет значение 5).

Наконец, обратите внимание на то, что дочерний процесс был помещен в группу фоновых процессов, когда его родительский процесс завершился, так как родительский процесс выполнялся командной оболочкой как задание переднего плана.

В разделе 19.5 мы увидим другой пример осиротевших групп процессов, когда будем рассматривать программу `pty`.

9.11. Реализация в FreeBSD

Теперь, когда мы поговорили о различных атрибутах процесса, группах процессов, сессиях и управляющих терминалах, настало время посмотреть, как все это может быть реализовано. Мы коротко рассмотрим реализацию в ОС FreeBSD. Некоторые подробности реализации в SVR4 вы сможете найти в [Williams 1989]. На рис. 9.11 показаны различные структуры данных, используемые в ОС FreeBSD.

Рассмотрим все показанные поля структур, начиная со структуры `session`. Для каждой сессии в памяти размещается отдельная структура `session` (это происходит, например, при каждом обращении к функции `setsid`).

- `s_count` – количество групп процессов в сессии. Когда этот счетчик обнуляется, память, занимаемая структурой, освобождается.
- `s_leader` – указатель на структуру `proc` лидера сессии.

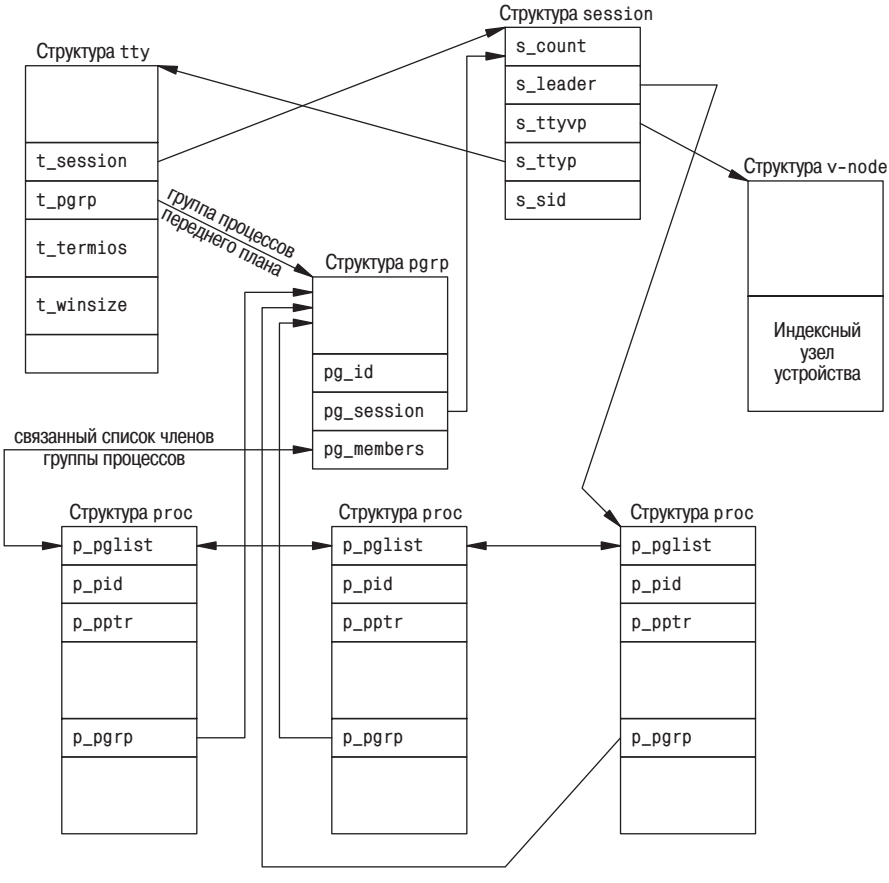


Рис. 9.11. *Реализация сессий и групп процессов в ОС FreeBSD*

- `s_ttyvp` – указатель на структуру `vnode` управляющего терминала.
- `s_ttyp` – указатель на структуру `tty` управляющего терминала.
- `s_sid` – идентификатор сессии. Помните, что понятие идентификатора сессии не определяется стандартом Single UNIX Specification.

Во время вызова функции `setsid` в памяти ядра размещается новая структура `session`. Значение поля `s_count` устанавливается равным 1, в поле `s_leader` заносится указатель на структуру `proc` вызывающего процесса, в поле `s_sid` заносится идентификатор процесса и в поля `s_ttyvp` и `s_ttyp` – пустые указатели, поскольку новая сессия не имеет управляющего терминала.

Теперь перейдем к структуре `tty`. Для каждого устройства терминала или псевдотерминала в области памяти ядра размещается одна такая структура. (Более подробно о псевдотерминалах мы поговорим в главе 19.)

- t_session – указатель на структуру session, для которой этот терминал является управляющим. (Обратите внимание, что структура tty содержит указатель на структуру session, а не наоборот.) Этот указатель используется терминалом для передачи сигнала SIGHUP лидеру сессии, когда система теряет связь с терминалом (рис. 9.7).
- t_pgrp – указатель на структуру pgrp группы процессов переднего плана. Он используется терминалом для передачи сигналов группе процессов переднего плана. Это те самые три сигнала, которые генерируются в результате ввода специальных символов (сигнал прерывания, завершения и остановки).
- t_termios – структура, которая содержит все специальные символы и дополнительную информацию о данном терминале, такую как скорость передачи данных, отображение вводимых символов (включено или выключено) и тому подобное. Мы еще вернемся к этой структуре в главе 18.
- t_winsize – структура winsize, которая содержит текущие размеры окна терминала. При изменении размеров окна терминала группе процессов переднего плана передается сигнал SIGWINCH. В разделе 18.12 мы продемонстрируем, как можно узнать и изменить размеры окна терминала.

Обратите внимание, что поиск группы процессов переднего плана для заданной сессии ядро начинает со структуры session. Следуя по указателю s_ttyp, ядро находит структуру tty управляющего терминала, а затем по указателю t_pgrp отыскивает структуру pgrp группы процессов переднего плана. Структура pgrp содержит все необходимые сведения о заданной группе процессов переднего плана.

- pg_id – идентификатор группы процессов.
- pg_session – указатель на структуру session для сессии, которой принадлежит данная группа процессов.
- pg_members – указатель на список структур proc, соответствующих процессам, которые входят в состав данной группы процессов. Структура p_pglist, входящая в состав структуры proc, содержит два поля – указатели на предыдущую и следующую структуры proc, которые служат для организации двусвязного списка процессов в группе.

Структура proc содержит всю информацию о процессе.

- p_pid – идентификатор процесса.
- p_pptr – указатель на структуру proc родительского процесса.
- p_pgrp – указатель на структуру pgrp группы, которой принадлежит процесс.
- p_pglist – структура, которая содержит указатели на предыдущий и следующий процессы в группе процессов.

И наконец, структура vnode. Эта структура размещается в памяти в момент открытия устройства управляющего терминала. Все обращения к устройству /dev/tty из процесса проходят через структуру vnode. Индексный узел (i-node) показан как часть виртуального узла (v-node).

9.12. Подведение итогов

В этой главе были описаны взаимоотношения между группами процессов – сессии, которые состоят из групп процессов. Управление заданиями в настоящее время поддерживается большинством версий UNIX, и мы показали, как оно осуществляется в командной оболочке, которая поддерживает эту функциональную возможность. Понятие управляющего терминала также связано с взаимоотношениями между процессами.

Мы много раз упоминали сигналы, которые широко используются для организации взаимодействий между процессами. В следующей главе обсуждение сигналов будет продолжено, и мы подробно рассмотрим все сигналы системы UNIX.

Упражнения

9.1. Вспомните обсуждение файлов `utmp` и `wtmp` в разделе 6.8 и ответьте на вопрос: почему запись о выходе из системы производится процессом `init`? Происходит ли то же самое в случае завершения сеанса связи через сетевое соединение?

9.2. Напишите небольшую программу, которая с помощью функции `fork` порождает дочерний процесс, создающий в свою очередь новую сессию. Проверьте, становится ли дочерний процесс лидером группы и теряет ли он управляющий терминал.

10

Сигналы

10.1. Введение

Сигналы – это программные прерывания. Большинство серьезных приложений имеют дело с сигналами. Сигналы предоставляют возможность обработки асинхронных событий – например, когда пользователь вводит символ прерывания, чтобы остановить программу, или когда одна из программ в конвейере аварийно завершается.

Сигналы появились в самых ранних версиях ОС UNIX, но модель сигналов, которую предоставляли такие системы, как Version 7, была недостаточно надежна. Сигналы могли теряться, а отключить отдельные сигналы на время выполнения критических фрагментов кода было довольно сложно. Существенные изменения в модель сигналов внесли 4.3BSD и SVR3, в этих версиях были добавлены так называемые *надежные сигналы*. Но изменения, сделанные в Беркли и в AT&T, оказались несовместимы между собой. К счастью, POSIX.1 стандартизировал функции обслуживания надежных сигналов, и именно их мы и будем обсуждать.

Эту главу мы начнем с краткого обзора сигналов и расскажем, для чего каждый из них используется. Затем мы рассмотрим проблемы, имевшие место в ранних реализациях. Чтобы разобраться во всех тонкостях, иногда очень важно понять, какие проблемы были связаны с реализацией. В этой главе приводится большое количество примеров, которые не совсем корректны, и обсуждаются имеющиеся в них недочеты.

10.2. Концепция сигналов

Прежде всего, каждый сигнал имеет собственное имя. Имена всех сигналов начинаются с последовательности `SIG`. Например, `SIGABRT` – это сигнал прерывания, который генерируется, когда процесс вызывает функцию `abort`. Сигнал `SIGALRM` генерируется, когда таймер, установленный функцией `alarm`, отмерит указанный промежуток времени. В Version 7 было 15 различных

сигналов, в SVR4 и 4.4BSD – уже 31 сигнал. ОС FreeBSD 5.2.1, Linux 2.4.22 и Mac OS X 10.3 поддерживают 31 сигнал, а Solaris 9 – 38 различных сигналов. Кроме того, Linux и Solaris поддерживают дополнительные сигналы, определяемые приложениями в виде расширений реального времени (расширения реального времени POSIX не рассматриваются в данной книге, за информацией обращайтесь к [Gallmeister 1995]).

Все имена сигналов определены как константы с положительными числовыми значениями (номера сигналов) в заголовочном файле `<signal.h>`.

> Фактически реализации определяют сигналы в отдельных заголовочных файлах, которые подключаются файлом `<signal.h>`. Вообще, считается дурным тоном в исходных текстах ядра подключать заголовочные файлы, предназначенные для приложений пользовательского уровня. Таким образом, если и приложение, и ядро нуждаются в одних и тех же определениях, то информация размещается в заголовочном файле ядра, который затем подключается в заголовочном файле пользовательского уровня. Так, FreeBSD 5.2.1 и Mac OS X 10.3 определяют сигналы в файле `<sys/signal.h>`. ОС Linux 2.4.22 определяет сигналы в файле `<bits/signum.h>`, а Solaris 9 – в файле `<sys/iso/signal_iso.h>`.

Ни один сигнал не имеет номера 0. В разделе 10.9 мы увидим, что функция `kill` использует номер сигнала 0 в особых случаях. Стандарт POSIX.1 называет такой сигнал *null signal* (пустой сигнал).

Сигналы могут порождаться различными условиями.

- Сигналы, генерируемые терминалом, возникают в случае, когда пользователь вводит определенные символы с клавиатуры терминала. Нажатие клавиши `DELETE` (или Control-C – в большинстве систем) приводит к генерации сигнала прерывания (`SIGINT`). Таким образом можно прервать выполнение программы, вышедшей из-под контроля. (В главе 18 мы увидим, что этот сигнал может быть привязан к любому символу клавиатуры терминала.)
- Аппаратные ошибки – деление на 0, ошибка доступа к памяти и прочие – также приводят к генерации сигналов. Эти ошибки обычно обнаруживаются аппаратным обеспечением, которое извещает ядро об их появлении. После этого ядро генерирует соответствующий сигнал и передает его процессу, который выполнялся в момент появления ошибки. Например, сигнал `SIGSEGV` посылается процессу в случае попытки обращения к неверному адресу в памяти.
- Функция `kill`(2) позволяет процессу передать любой сигнал другому процессу или группе процессов. Естественно, здесь существуют свои ограничения: необходимо быть владельцем процесса, которому посылается сигнал, или обладать привилегиями суперпользователя.
- Команда `kill`(1) позволяет передавать сигналы другим процессам. Эта программа является простым интерфейсом к функции `kill`. Зачастую эта команда используется для принудительного завершения вышедших из-под контроля фоновых процессов.
- Сигналы могут порождаться при условиях, определяемых программно, например, когда нужно известить приложение о наступлении некоторого события. Эти условия определяются не аппаратурой (как, например, де-

ление на 0), а программным обеспечением. Примерами таких сигналов могут служить SIGURG (посылается, когда через сетевое соединение приходят экстренные (out-of-band) данные), SIGPIPE (посылается пишущему процессу, когда он пытается записать данные в канал после завершения процесса, читающего из канала) и SIGALRM (посылается процессу по истечении установленного им таймера).

Сигналы являют собой классический пример асинхронных событий. Сигнал может быть передан процессу в любой момент времени. Чтобы выяснить причину, породившую сигнал, процесс не может просто проверить некоторую переменную (как, например, errno), вместо этого он должен обратиться к ядру с предложением: «если появится этот сигнал – сделай то-то и то-то».

В случае появления сигнала мы можем запросить ядро произвести одно из трех действий. Они называются *диспозициями* сигнала, или *действиями*, связанными с сигналом.

1. Игнорировать сигнал. Это действие возможно для большинства сигналов, но два сигнала, SIGKILL и SIGSTOP, никогда нельзя игнорировать. Причина, по которой эти два сигнала не могут быть проигнорированы, заключается в том, что ядру и суперпользователю необходима возможность завершить или остановить любой процесс. Кроме того, если проигнорировать некоторые из сигналов, возникающих в результате аппаратных ошибок (таких как деление на 0 или попытка обращения к несуществующей памяти), поведение процесса может стать непредсказуемым.
2. Перехватить сигнал. Для этого мы должны сообщить ядру адрес функции, которая будет вызываться всякий раз при обнаружении сигнала. В этой функции мы можем предусмотреть действия по обработке условия, породившего сигнал. Например, если мы создаем командный интерпретатор, то когда пользователь посылает сигнал прерывания, мы, скорее всего, захотим вернуться к главному циклу программы, прервав выполнение команды, запущенной пользователем. Если пойман сигнал SIGCHLD, который означает завершение дочернего процесса, то функция, перехватившая сигнал, может вызвать функцию waitpid, чтобы получить идентификатор дочернего процесса и код его завершения. Еще один пример: если процесс создает временные файлы, то имеет смысл написать функцию обработки сигнала SIGTERM (сигнал завершения, посылаемый командой kill по умолчанию), которая будет удалять временные файлы. Обратите внимание: сигналы SIGKILL и SIGSTOP не могут быть перехвачены.
3. Применить действие по умолчанию. Каждому сигналу поставлено в соответствие некоторое действие по умолчанию (перечислены в табл. 10.1). Заметьте, что для большинства сигналов действие по умолчанию заключается в завершении процесса.

В табл. 10.1 перечислены имена всех сигналов и указано, какими системами они поддерживаются и каково действие по умолчанию для каждого сигнала. Если в колонке SUS (Single UNIX Specification) стоит точка, это означает, что сигнал определен как часть базовой спецификации POSIX.1, а если указана аббревиатура XSI, значит сигнал определен как расширение XSI.

Таблица 10.1. Сигналы UNIX

Имя	Описание	ISO C	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Действие по умолчанию
SIGABRT	Аварийное завершение (abort)	•	•	•	•	•	•	Завершить +core
SIGALRM	Истекло время таймера (alarm)		•	•	•	•	•	Завершить
SIGBUS	Аппаратная ошибка		•	•	•	•	•	Завершить +core
SIGCANCEL	Для внутреннего использования библиотекой threads						•	Игнорировать
SIGCHLD	Изменение состояния дочернего процесса		•	•	•	•	•	Игнорировать
SIGCONT	Возобновить работу приостановленного процесса		•	•	•	•	•	Продолжить/игнорировать
SIGEMT	Аппаратная ошибка			•	•	•	•	Завершить +core
SIGFPE	Арифметическая ошибка	•	•	•	•	•	•	Завершить +core
SIGFREEZE	Закрепление контрольной точки						•	Игнорировать
SIGHUP	Обрыв связи		•	•	•	•	•	Завершить
SIGILL	Недопустимая инструкция	•	•	•	•	•	•	Завершить +core
SIGINFO	Запрос состояния с клавиатуры			•		•		Игнорировать
SIGINT	С терминала введен символ прерывания	•	•	•	•	•	•	Завершить
SIGIO	Асинхронный ввод-вывод			•	•	•	•	Завершить/игнорировать
SIGIOT	Аппаратная ошибка			•	•	•	•	Завершить +core
SIGKILL	Завершение		•	•	•	•	•	Завершить
SIGLWP	Для внутреннего использования библиотекой threads						•	Игнорировать

Таблица 10.1 (продолжение)

Имя	Описание	ISO C	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Действие по умолчанию
SIGPIPE	Запись в канал, из которого никто не читает		•	•	•	•	•	Завершить
SIGPOLL	Событие опроса (poll)		XSI		•		•	Завершить
SIGPROF	Истекло время профилирующего таймера (setitimer)		XSI	•	•	•	•	Завершить
SIGPWR	Падение напряжения питания/перезапуск				•		•	Завершить/игнорировать
SIGQUIT	С терминала введен символ завершения		•	•	•	•	•	Завершить +core
SIGSEGV	Ошибка доступа к памяти	•	•	•	•	•	•	Завершить +core
SIGSTKFLT	Ошибка, связанная со стеком сопроцессора				•			Завершить
SIGSTOP	Приостановить процесс		•	•	•	•	•	Остановить процесс
SIGSYS	Неверный системный вызов		XSI	•	•	•	•	Завершить +core
SIGTERM	Завершение	•	•	•	•	•	•	Завершить
SIGTHAW	Освобождение контрольной точки						•	Игнорировать
SIGTRAP	Аппаратная ошибка		XSI	•	•	•	•	Завершить +core
SIGTSTP	С терминала введен символ приостановки		•	•	•	•	•	Остановить процесс
SIGTTIN	Чтение из управляющего терминала фоновым процессом		•	•	•	•	•	Остановить процесс
SIGTTOU	Запись в управляющий терминал фоновым процессом		•	•	•	•	•	Остановить процесс

Имя	Описание	ISO C	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Действие по умолчанию
SIGURG	Экстренное событие (сокеты)		•	•	•	•	•	Игнорировать
SIGUSR1	Определяемый пользователем сигнал		•	•	•	•	•	Завершить
SIGUSR2	Определяемый пользователем сигнал		•	•	•	•	•	Завершить
SIGVTALRM	Истекло время виртуального таймера (setitimer)		XSI	•	•	•	•	Завершить
SIGWAITING	Для внутреннего использования библиотекой threads						•	Игнорировать
SIGWINCH	Изменение размеров окна терминала			•	•	•	•	Игнорировать
SIGXCPU	Исчерпан лимит процессорного времени (setrlimit)		XSI	•	•	•	•	Завершить +core/игнорировать
SIGXFSZ	Превышено ограничение на размер файла (setrlimit)		XSI	•	•	•	•	Завершить +core/игнорировать
SIGXRES	Превышено ограничение на использование ресурса						•	Игнорировать

Если в колонке «Действие по умолчанию» указано «завершить+core», это означает, что образ памяти процесса сохраняется в файле core в текущем рабочем каталоге процесса. (Имя файла core наглядно демонстрирует, как давно эта функциональная особенность появилась в UNIX.) Большинство отладчиков могут использовать этот файл для выяснения причин, породивших преждевременное завершение процесса.

> Возможность создания файла core – функциональная особенность, присущая большинству версий UNIX. Хотя она и не является частью POSIX.1, тем не менее она упоминается в расширении XSI стандарта Single UNIX Specification как возможное, зависящее от реализации действие.

Имя файла core варьируется в разных реализациях. В ОС FreeBSD 5.2.1, например, файл core получает имя *cmdname*.core, где *cmdname* – имя команды, соответствующей процессу, получившему сигнал. В Mac OS X 10.3 файл core получает имя core.*pid*, где *pid* – идентификатор процесса, получившего сигнал. (Эти системы позволяют настроить правила именования файлов core через параметр sysctl.)

Большинство реализаций сохраняют файл core в текущем рабочем каталоге соответствующего процесса, но Mac OS X помещает все файлы core в каталог /cores.

Файл core не создается, если (а) файл программы имеет установленный бит set-user-ID, а текущий пользователь не является его владельцем, (б) файл программы имеет установленный бит set-group-ID, а текущий пользователь не принадлежит к группе владельца файла, (в) пользователь не имеет права на запись в текущий каталог, (г) файл уже существует и пользователь не имеет права на запись в него, (д) файл слишком велик (вспомните предел RLIMIT_CORE из раздела 7.11). Файл core (если он еще не существует) обычно создается с правами на запись и на чтение для владельца, хотя в Mac OS X выдается только право на чтение для владельца.

В табл. 10.1 сигналы с описанием «аппаратная ошибка» соответствуют зависящим от реализации аппаратным ошибкам. Многие из них взяты из оригинальной реализации UNIX для PDP-11. Проверьте справочное руководство по вашей операционной системе и уточните, каким именно ошибкам соответствуют эти сигналы.

А теперь опишем каждый из сигналов более подробно.

SIGABRT Этот сигнал генерируется вызовом функции abort (раздел 10.17). Процесс завершается аварийно.

SIGALRM Этот сигнал генерируется по истечении таймера, установленного функцией alarm (раздел 10.10). Он также генерируется по истечении таймера, установленного функцией setitimer(2).

SIGBUS Соответствует аппаратной ошибке, определяемой реализацией. Обычно этот сигнал генерируется в случае некоторых ошибок, связанных с памятью, которые мы рассмотрим в разделе 14.9.

SIGCANCEL Этот сигнал используется библиотекой threads в ОС Solaris. Он не предназначен для общего использования.

SIGCHLD Когда процесс завершается или останавливается, родительскому процессу передается сигнал SIGCHLD. По умолчанию этот сигнал игнорируется, но родительский процесс может перехватить его, если желает получать извещения об изменении состояния своего дочернего процесса. Функция, перехватывающая этот сигнал, обычно вызывает одну из функций семейства wait, чтобы получить идентификатор дочернего процесса и код завершения.

В ранних версиях System V был похожий сигнал с именем SIGCLD (без H). Семантика этого сигнала отличалась от семантики других сигналов, и еще справочное руководство SVR2 рекомендовало не использовать его в новых программах. (Как ни странно, в SVR3 и SVR4 это предупреждение исчезло из справочного руководства.) Приложения должны использовать сигнал SIGCHLD, но нужно знать, что многие версии UNIX определяют сигнал SIGCLD, идентичный сигналу SIGCHLD, для сохранения обратной совместимости.

Если вам понадобится определить семантику сигнала SIGCLD в вашей системе, обратитесь к страницам справочного руководства. Эти два сигнала мы обсудим в разделе 10.7.

SIGCONT Этот сигнал передается остановленным процессам, чтобы возобновить их работу. Действие по умолчанию заключается в продолжении работы процесса, если процесс был остановлен, для работающего процесса сигнал игнорируется. Полноэкранный редактор, например, может перехватывать этот сигнал, чтобы использовать функцию-обработчик для перерисовки экрана. Дополнительная информация об этом сигнале приводится в разделе 10.20.

SIGEMT Соответствует аппаратной ошибке, определяемой реализацией.

Имя EMT происходит от инструкции PDP-11 «emulator trap» (ловушка эмулятора). Этот сигнал поддерживается не всеми платформами. В Linux, например, этот сигнал поддерживается только для некоторых аппаратных архитектур, таких как SPARC, MIPS и PA-RISC.

SIGFPE Этот сигнал свидетельствует об арифметической ошибке, такой как деление на 0 или переполнение числа с плавающей точкой.

SIGFREEZE Этот сигнал определен только в ОС Solaris. Он используется для извещения процессов, которые должны предпринять определенные действия перед фиксацией состояния системы, что обычно происходит, когда система уходит в спящий или в ждущий режим.

SIGHUP Этот сигнал передается управляющему процессу (лидеру сессии), связанному с управляющим терминалом, если обнаружен обрыв связи с терминалом. На рис. 9.11 видно, что сигнал посылается процессу, на который указывает поле s_leader в структуре session. Этот сигнал генерируется только в том случае, если сброшен флаг терминала CLOCAL. (Флаг CLOCAL устанавливается для локального терминала. Этот флаг сообщает драйверу о том, что он должен игнорировать все управляющие сигналы модема. В главе 18 мы расскажем, как устанавливается этот флаг.)

Обратите внимание, что лидер сессии, которому передается сигнал, может быть фоновым процессом (рис. 9.7). Это отличает данный сигнал от других сигналов, генерируемых терминалом (прерывание, завершение и останов), которые всегда посылаются группе процессов переднего плана.

Этот сигнал также генерируется в случае завершения лидера сессии. В такой ситуации сигнал посылается всем процессам из группы процессов переднего плана.

Нередко этот сигнал используется для извещения процессов-демонов (глава 13) о необходимости перечитать конфигурационные файлы. Причина по которой для этой цели выбирается именно сигнал SIGHUP, заключается в том, что если не послать этот сигнал явно, то демоны никогда не примут его, поскольку у них нет управляющего терминала.

SIGILL Этот сигнал указывает на то, что процесс выполнил недопустимую машинную инструкцию.

В 4.3BSD этот сигнал генерировался функцией abort. Теперь она генерирует сигнал SIGABRT.

SIGINFO Этот BSD-сигнал генерируется драйвером терминала при нажатии клавиши запроса состояния (часто Control-T). Этот сигнал передается всем процессам из группы процессов переднего плана (рис. 9.8). Обычно этот сигнал используется для вывода на терминал информации о состоянии процессов в группе процессов переднего плана.

ОС Linux не поддерживает сигнал SIGINFO, за исключением платформы Alpha, где он определен с тем же номером, что и сигнал SIFPWR.

SIGINT Этот сигнал генерируется драйвером терминала при вводе символа прерывания (часто DELETE или Control-C). Он посылается всем процессам из группы процессов переднего плана. (рис. 9.8). Этот сигнал часто используется для прерывания выполнения вышедших из-под контроля приложений, особенно когда они начинают выводить ненужную информацию на экран.

SIGIO Этот сигнал указывает на событие асинхронной операции ввода-вывода. Мы будем обсуждать это в разделе 14.6.2.

В табл. 10.1 указано, что действие по умолчанию для сигнала SIGIO – завершить процесс либо игнорировать. К сожалению, выбор действия по умолчанию зависит от реализации. В System V сигнал SIGIO идентичен сигналу SIGPOLL, поэтому действием по умолчанию является завершение процесса. В BSD этот сигнал по умолчанию игнорируется.

ОС Linux 2.4.22 и Solaris 9 определяют сигнал SIGIO с тем же номером, что и SIGPOLL, поэтому действием по умолчанию является завершение процесса. ОС FreeBSD 5.2.1 и Mac OS X 10.3 по умолчанию игнорируют этот сигнал.

SIGIOT Соответствует аппаратной ошибке, определяемой реализацией.

Имя IOT происходит от мнемоники инструкции PDP-11 – «input/output TRAP» (ловушка ввода-вывода). В ранних версиях System V этот сигнал генерировался функцией abort. Теперь она генерирует сигнал SIGABRT.

В ОС FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3 и Solaris 9 сигнал SIGIOT определен с тем же номером, что и сигнал SIGABRT.

SIGKILL Это один из двух сигналов, которые не могут быть перехвачены приложением или проигнорированы. Он дает возможность системному администратору уничтожить любой процесс.

SIGLWP Этот сигнал применяется библиотекой threads в ОС Solaris и недоступен для общего использования.

SIGPIPE Этот сигнал посылается процессу, который предпринял попытку записи в канал, когда процесс, производивший чтение из канала, уже завершился. Мы будем обсуждать каналы в разделе 15.2. Этот сигнал также генерируется при попытке выполнить запись в сокет типа SOCK_STREAM, когда соединение уже разорвано. Сокеты мы будут обсуждаться в главе 16.

SIGPOLL Этот сигнал может быть сгенерирован при наступлении определенного события в опрашиваемом устройстве. Мы рассмотрим этот сигнал при обсуждении функции poll в разделе 14.5.2. Сигнал SIGPOLL появился в SVR3 и в некоторой степени соответствует сигналам SIGIO и SIGURG в BSD.

В ОС Linux и Solaris сигнал SIGPOLL определен с тем же номером, что и сигнал SIGIO.

SIGPROF Этот сигнал генерируется по истечении интервала времени профилирующего таймера, установленного функцией setitimer(2).

SIGPWR Реализация этого сигнала зависит от системы. В основном он используется в системах, которые снабжены источником бесперебойного питания (UPS). При обнаружении сбоя в сети питания источник бесперебойного питания извещает об этом систему и принимает на себя обеспечение питания системы. Пока ничего более не предпринимается, так как система продолжает питаться от аккумуляторных батарей. Но когда напряжение в сети отсутствует продолжительное время и напряжение аккумуляторов падает ниже определенного уровня, то программное обеспечение обычно извещается об этом повторно, и с этого момента у системы остается примерно 15–30 секунд, чтобы корректно завершить работу. В этот момент посылается сигнал SIGPWR. В большинстве систем имеется процесс, который получает извещение о падении напряжения аккумуляторов и посылает сигнал SIGPWR процессу init, а init берет на себя заботу об остановке системы.

Для этих целей Linux 2.4.22 и Solaris 9 предусматривают специальные записи в файле inittab – powerfail и powerwait (или powerokwait).

В табл. 10.1 было указано, что действие по умолчанию для сигнала SIGPWR – завершить процесс или игнорировать сигнал. К сожалению, действие по умолчанию зависит от реализации. В Linux этот сигнал по умолчанию завершает процесс, в Solaris – игнорируется.

SIGQUIT Этот сигнал генерируется драйвером терминала при вводе символа завершения (часто Control-\). Сигнал посылается всем процессам из группы процессов переднего плана (рис. 9.8). При этом происходит не только завершение группы процессов переднего плана (как в случае сигнала SIGINT), но и создание файла core.

SIGSEGV Этот сигнал указывает на то, что процесс обратился к недопустимому адресу в памяти.

Имя SEGV происходит от выражения «segmentation violation» (нарушение правил сегментации).

SIGSTKFLT Этот сигнал определен только в ОС Linux. Он появился в самых ранних версиях Linux и предназначался для обнаружения ошибок, связанных со стеком арифметического сопроцессора. Этот сигнал не генерируется ядром, он сохраняется только для обратной совместимости.

SIGSTOP Этот сигнал останавливает процесс. Он похож на сигнал SIGTSTP, порождаемый драйвером терминала, но в отличие от него не может быть перехвачен или проигнорирован.

SIGSYS Этот сигнал свидетельствует о неверном системном вызове. Каким-то образом процесс выполнил машинную инструкцию, которая была воспринята ядром как системный вызов, но параметр инструкции указывал на неверный тип системного вызова. Это может произойти, если вы скомпилируете программу, которая использует недавно появившийся системный вызов, и затем попытаетесь запустить двоичный исполняемый файл на более старой версии системы, которая этот системный вызов не поддерживает.

SIGTERM Это сигнал завершения процесса, который посылается командой kill(1) по умолчанию.

SIGTHAW Этот сигнал определен только в ОС Solaris и используется для извещения процессов о том, что они должны предпринять определенные действия после выхода системы из ждущего или спящего режимов.

SIGTRAP Соответствует аппаратной ошибке, определяемой реализацией.

Имя сигнала происходит от инструкции PDP-11 TRAP (ловушка). Реализации часто используют его для передачи управления отладчикам по достижении точки останова.

SIGTSTP Этот сигнал приостановки генерируется драйвером терминала при вводе символа приостановки (часто Control-Z) с клавиатуры терминала. Сигнал посылается всем процессам из группы процессов переднего плана (рис. 9.8).

К сожалению, термин «приостановка» может иметь несколько смыслов. Когда мы обсуждали управление заданиями, мы говорили о приостановке и возобновлении работы. Однако, если речь идет о драйвере терминала, для обозначения остановки и возобновления вывода на терминал с помощью клавиш Control-S и Control-Q традиционно используется термин «останов». Таким образом, в случае драйвера терминала символ, который приводит к генерации сигнала SIGTSTP, называется символом приостановки, а не останова.

SIGTTIN Этот сигнал генерируется драйвером терминала, когда фоновый процесс пытается выполнить операцию чтения из управляющего терминала (раздел 9.8). В особых случаях – если (а) процесс, выполняющий чтение, игнорирует или блокирует этот сигнал или (б) группа процессов, в которой находится читающий процесс, является осиротевшей группой, – сигнал не генерируется; вместо этого операция чтения завершается с признаком ошибки и в переменную errno записывается код ошибки EIO.

SIGTTOU Этот сигнал генерируется драйвером терминала, когда фоновый процесс пытается выполнить запись в управляющий терминал (раздел 9.8). В отличие от только что описанного сигнала SIGTTIN, в данном случае процесс может разрешить фоновым процессам запись в управляющий терминал. Эту возможность мы рассмотрим в главе 18.

Если запись в терминал для фоновых процессов запрещена, то, как и в случае сигнала SIGTTIN, возможны два особых случая: когда (а) пишущий процесс игнорирует или блокирует этот сигнал или (б) группа пишущего процесса является осиротевшей группой. В этих случаях сигнал не генерируется; вместо этого операция записи завершается с признаком ошибки и в переменную errno записывается код ошибки EIO. Независимо от того, разрешено ли фоновому процессу выполнять запись в терминал, некоторые другие операции с терминалом также могут генерировать сигнал SIGTTOU: tcsetattr, tcsendbreak, tcdrain, tcflush, tcflow и tcsetpgrp. Эти операции рассматриваются в главе 18.

SIGURG Этот сигнал сообщает процессу о том, что произошло экстренное событие. Он может генерироваться при поступлении экстренных (out-of-band) данных через сетевое соединение.

SIGUSR1 Этот сигнал определяется пользователем и предназначен для внутреннего использования в приложениях.

SIGUSR2 Это еще один сигнал, определяемый пользователем. Подобно сигналу SIGUSR1, он также предназначен для внутреннего применения в приложениях.

SIGVTALRM Этот сигнал генерируется по истечении периода времени, назначенного виртуальному таймеру функцией setitimer(2).

SIGWAITING Этот сигнал предназначен для внутреннего использования в библиотеке threads ОС Solaris.

SIGWINCH Ядро управляет изменением размера окна, связанного с каждым терминалом или псевдотерминалом. Процесс может получить и изменить размер окна с помощью функции ioctl, которую мы рассмотрим в разделе 18.12. Если процесс изменяет размер окна с помощью команды set-window-size функции ioctl, ядро посылает сигнал SIGWINCH группе процессов переднего плана.

SIGXCPU Стандарт Single UNIX Specification поддерживает как расширение XSI концепцию ограничений ресурсов (раздел 7.11). Если процесс достигает мягкого предела на использование центрального процессора, ему посылается сигнал SIGXCPU.

В табл. 10.1 указано, что действие по умолчанию для сигнала SIGXCPU – завершить процесс или игнорировать сигнал. К сожалению, действие по умолчанию зависит от реализации. В Linux 2.4.22 и Solaris 9 действие по умолчанию состоит в завершении процесса и создании файла core, тогда как в FreeBSD 5.2.1 и Mac OS X 10.3 сигнал игнорируется. Стандарт Single UNIX Specification требует, чтобы по умолчанию происходило аварийное завершение процесса. Создавать ли при этом файл core, каждая из реализаций может определять самостоятельно.

SIGXFSZ Этот сигнал посылается процессу, который превысил мягкий предел на размер файла (раздел 7.11).

Так же как и в случае с сигналом SIGXCPU, действие по умолчанию зависит от операционной системы. В Linux 2.4.22 и Solaris 9 действие по умолчанию состоит в завершении процесса и создании файла core, тогда как в FreeBSD 5.2.1 и Mac OS X 10.3 сигнал игнорируется. Стандарт Single UNIX Specification требует, чтобы по умолчанию происходило аварийное завершение процесса. Создавать ли при этом файл core, каждая реализация может определять самостоятельно.

SIGXRES Этот сигнал определен только в ОС Solaris. Он может применяться для извещения процессов о том, что они превысили предустановленное ограничение на использование ресурса. Механизм управления ресурсами в Solaris предоставляет возможность управлять ресурсами, разделяемыми между несколькими независимыми приложениями.

10.3. Функция signal

Функция signal предоставляет простейший интерфейс к сигналам UNIX.

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

Возвращает предыдущую диспозицию сигнала (см. далее) в случае успеха, SIG_ERR – в случае ошибки

Функция signal определена стандартом ISO C, который ничего не говорит о многозадачности, группах процессов, терминальном вводе-выводе и тому подобном. Поэтому определение сигналов в этом стандарте практически бесполезно для систем UNIX.

Реализации, происходящие от System V, поддерживают функцию signal, но она предоставляет устаревшую семантику механизма ненадежных сигналов (которая будет описана в разделе 10.4). Эта функция обеспечивает обратную совместимость с приложениями, которые требуют устаревшей семантики. Новые приложения не должны использовать эти ненадежные сигналы.

Система 4.4BSD также поддерживает функцию signal, но она реализована в терминах функции sigaction (которая будет описана в разделе 10.14) – таким образом, функция signal в 4.4BSD предоставляет новую семантику надежных сигналов. ОС FreeBSD 5.2.1 и Mac OS X 10.3 следуют этой стратегии.

ОС Solaris 9 корнями уходит и в System V, и в BSD, но в случае функции signal она следует семантике, принятой в System V.

В Linux 2.4.22 семантика функции signal может соответствовать принятой в System V или в BSD, в зависимости от версии библиотеки языка C и параметров компиляции приложения.

Поскольку семантика функции signal различается в разных реализациях, вместо нее лучше использовать функцию sigaction. При обсуждении функции sigaction в разделе 10.14 мы представим реализацию функции signal на основе sigaction. Все примеры в этой книге используют функцию signal, представленную в листинге 10.12.

Аргумент *signo* – это имя сигнала из табл. 10.1. В качестве аргумента *func* может передаваться либо (а) константа SIG_IGN, либо (б) константа SIG_DFL, либо (в) адрес функции, которая будет вызвана при получении сигнала. Если указана константа SIG_IGN, то тем самым мы сообщаем системе, что сигнал должен игнорироваться. (Не забывайте, что два сигнала, SIGKILL и SIGSTOP, не могут быть проигнорированы.) Если указана константа SIG_DFL, то с сигналом связывается действие по умолчанию (последняя колонка в табл. 10.1). Если указан адрес функции, то она будет вызываться при получении сигнала, то есть будет «*перехватывать*» сигнал. Такие функции называются *обработчиками* или *перехватчиками сигналов*.

Прототип функции signal показывает, что она принимает два аргумента и возвращает указатель на функцию, которая не имеет возвращаемого значения (void). Первый аргумент функции signal, *signo*, представляет собой целое число. Второй аргумент – это указатель на функцию, которая не возвращает значения и принимает единственный целочисленный аргумент. Функция, адрес которой возвращает функция signal, также принимает один целочисленный аргумент (последний (int)). Проще говоря, функции-обработчику сигнала передается единственный аргумент (целое число – номер сигнала), и она ничего не возвращает. Когда функция signal вызывается для того, чтобы установить обработчик сигнала, второй аргумент должен быть указателем на функцию. Возвращаемое значение функции signal – это указатель на предыдущий обработчик сигнала.

Большинство систем вызывают обработчик сигнала с дополнительными, зависящими от реализации, аргументами. Этот вопрос мы рассмотрим в разделе 10.14.

Довольно сложный для восприятия прототип функции signal, приведенный в начале раздела, может быть определен гораздо проще через использование следующей инструкции typedef [Plauger 1992]:

```
typedef void Sigfunc(int);
```

Тогда прототип самой функции signal будет выглядеть так:

```
Sigfunc *signal(int, Sigfunc *);
```

Мы включили это определение в заголовочный файл apue.h (приложение B) и будем использовать его в наших примерах.

Заглянув в файл <signal.h>, мы наверняка обнаружим следующие объявления:

```
#define SIG_ERR (void (*)())-1
#define SIG_DFL (void (*)())0
#define SIG_IGN (void (*)())1
```

Эти константы могут использоваться вместо «указателя на функцию, которая принимает один целочисленный аргумент и ничего не возвращает» – это второй аргумент функции signal и одновременно ее возвращаемое значение. Значения констант не обязательно должны быть –1, 0 и 1. Но они должны быть такими, чтобы их нельзя было принять за адреса функций. В большинстве реализаций используются значения, приведенные выше.

Пример

В листинге 10.1 показан простейший обработчик сигнала, который перехватывает два сигнала, определяемые пользователем, и выводит их номера. Функцию pause мы рассмотрим в разделе 10.10, она просто приостанавливает процесс до тех пор, пока не будет получен сигнал.

Листинг 10.1. Простейшая программа, которая перехватывает сигналы SIGUSR1 и SIGUSR2

```
#include "apue.h"

static void sig_usr(int); /* один обработчик для двух сигналов */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("невозможно перехватить сигнал SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("невозможно перехватить сигнал SIGUSR2");
    for ( ; ; )
        pause();
}

static void
sig_usr(int signo)        /* аргумент - номер сигнала */
{
```

```
    if (signo == SIGUSR1)
        printf("принят сигнал SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("принят сигнал SIGUSR2\n");
    else
        err_dump("принят сигнал %d\n", signo);
}
```

Мы запускали эту программу как фоновый процесс и с помощью команды kill(1) посылали ей сигналы. Обратите внимание, что термин kill (уничтожить) в UNIX представляет собой пример не вполне корректного именования. Команда kill(1) и функция kill(2) просто посылают сигнал процессу или группе процессов. Завершится ли процесс при получении сигнала или нет, зависит от того, какой сигнал был послан и перехватывается ли этот сигнал процессом.

```
$ ./a.out &                    запустить процесс в фоновом режиме
[1]      7216                  командная оболочка вывела номер задания
                               и идентификатор процесса
$ kill -USR1 7216              передать сигнал SIGUSR1
принят сигнал SIGUSR1
$ kill -USR2 7216              передать сигнал SIGUSR2
принят сигнал SIGUSR2
$ kill 7216                    теперь передать сигнал SIGTERM
[1]+ Terminated        ./a.out
```

Когда был передан сигнал SIGTERM, процесс завершился, поскольку он не перехватывает этот сигнал, а действие по умолчанию для него заключается в завершении процесса.

Запуск программы

При запуске программы сигналам назначаются либо действия по умолчанию, либо сигналы игнорируются. Обычно для всех сигналов назначаются действия по умолчанию, если только процесс, вызвавший функцию exec, не игнорирует какие-либо сигналы. Фактически функции семейства exec изменяют диспозицию тех сигналов, которые перехватываются, на действия по умолчанию и оставляют без изменения все остальные. (Это вполне естественно, поскольку сигнал, который перехватывается процессом, вызвавшим функцию exec, не может быть перехвачен той же самой функцией в новой программе, так как адрес функции-перехватчика в вызывающей программе наверняка потеряет свой смысл в новой программе.)

Вот один характерный пример того, как интерактивная командная оболочка обращается с сигналами SIGINT и SIGQUIT фонового процесса. Если командная оболочка не поддерживает управление заданиями, то при запуске фонового процесса, например

```
cc main.c &
```

командная оболочка автоматически устанавливает диспозицию этих сигналов для фонового процесса в значение SIG_IGN. Таким образом, ввод символа

прерывания не оказывает никакого влияния на фоновый процесс. Если бы этого не было сделано, то при вводе символа прерывания завершился бы не только процесс переднего плана, но и все фоновые процессы.

Большинство интерактивных программ, перехватывающих эти два сигнала, содержат код, который выглядит примерно так:

```
void sig_int(int), sig_quit(int);

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```

Этот код устанавливает перехватчик сигнала только в том случае, если сигнал не игнорируется.

Эти два вызова наглядно демонстрируют недостаток функции `signal` – отсутствие возможности определить текущую диспозицию сигнала без ее изменения. Далее в этой главе мы увидим, что такую возможность предоставляет функция `sigaction`.

Создание процесса

Когда процесс вызывает функцию `fork`, дочерний процесс наследует диспозиции сигналов от родительского процесса. В данном случае, поскольку дочерний процесс представляет собой полную копию родительского процесса, адреса функций-обработчиков не теряют свой смысл в дочернем процессе.

10.4. Ненадежные сигналы

В ранних версиях UNIX (таких как Version 7) сигналы были ненадежными. Это означает, что сигналы могли теряться; другими словами, процесс мог не получить посланный ему сигнал. Кроме того, процесс имел весьма ограниченные возможности контроля над сигналами: он мог либо перехватить сигнал, либо игнорировать его. Иногда может возникнуть потребность заблокировать сигнал, то есть не игнорировать его, а просто отложить передачу сигнала до того момента, когда приложение будет готово принять его.

> Такое положение дел было исправлено в 4.2BSD, когда появились так называемые «надежные сигналы». Затем ряд других изменений, также обеспечивающих поддержку надежных сигналов, был внесен в SVR3. Стандарт POSIX.1 в качестве образца выбрал модель BSD.

Одна из проблем, связанных с ранними версиями, заключалась в том, что действие сигнала сбрасывалось в значение по умолчанию после передачи сигнала. (В предыдущем примере, когда мы запускали программу из листинга 10.1, эта проблема не возникала, поскольку сигнал перехватывался всего один раз.) Классический пример из книг по программированию, описывающий обработку сигнала прерывания в ранних версиях UNIX, обычно выглядит примерно так:

```
int sig_int();              /* моя функция-обработчик */

...
signal(SIGINT, sig_int);    /* установить функцию-обработчик */
...

sig_int()
{
    signal(SIGINT, sig_int); /* переустановить функцию-обработчик */
    ...                      /* обработка сигнала ... */
}
```

(Функция-обработчик объявлена как возвращающая целое число по той причине, что в ранних версиях UNIX отсутствовала поддержка типа `void` стандарта ISO C.)

Этот пример также не лишен недостатков. Проблема здесь в том, что существует некоторый промежуток времени между моментом посылки сигнала и моментом вызова функции `signal` в функции-обработчике, когда сигнал `SIGINT` может быть послан повторно. Этот повторный сигнал может вызвать выполнение действия по умолчанию, которое заключается в завершении процесса. Это один из примеров, когда все работает правильно большую часть времени, заставляя нас думать, что все в порядке, хотя на самом деле это далеко не так.

Еще одна проблема в ранних версиях UNIX состояла в том, что процесс был не в состоянии отключить сигнал на время, когда его появление было бы нежелательным. Процесс мог полностью игнорировать сигнал, но он не мог сообщить системе: «следующие сигналы не должны поступать ко мне, но система должна запомнить, что они были посланы». Классический пример, демонстрирующий этот недостаток, представлен фрагментом кода, который перехватывает сигнал и устанавливает флаг, отмечающий появление сигнала:

```
int sig_int_flag;           /* ненулевое значение, если был получен сигнал */

main()
{
    int sig_int();          /* моя функция-обработчик сигнала */
    ...
    signal(SIGINT, sig_int); /* установить функцию-обработчик */
    ...
    while (sig_int_flag == 0)
        pause();            /* приостановить работу в ожидании сигнала */
    ...
}

sig_int()
{
    signal(SIGINT, sig_int); /* переустановить функцию-обработчик */
    sig_int_flag = 1;  /* установить флаг для проверки в основной программе */
}
```

Здесь процесс вызывает функцию `pause`, ожидая, пока сигнал не будет перехвачен. При получении сигнала функция-обработчик просто устанавливает флаг `sig_int_flag` в ненулевое значение. Ядро автоматически возобновляет

работу процесса после выхода из функции-обработчика, после чего процесс обнаруживает взведенный флаг и выполняет все необходимые действия. Но здесь опять же существует промежуток времени, когда все может пойти не так, как мы ожидаем. Если сигнал будет послан в промежутке времени между проверкой флага `sig_int_flag` и вызовом функции `pause`, то процесс рискует приостановиться навсегда (при условии, что сигнал больше никогда не будет послан). В данной ситуации сигнал будет потерян. Это еще один пример того, когда ошибочный код вполне корректно работает большую часть времени. Обнаружение и отладка подобных ошибок – чрезвычайно сложная задача.

10.5. Прерванные системные вызовы

Ранние версии UNIX обладали одним свойством: если процесс, заблокированный в «медленном» системном вызове, перехватывал сигнал, то выполнение системного вызова прерывалось. В этом случае системный вызов возвращал признак ошибки с кодом `EINTR` в переменной `errno`. Так было сделано в предположении, что раз сигнал был послан и процесс перехватил его, следовательно, могло что-то произойти, что должно прервать работу заблокированного системного вызова.

> Здесь следует понимать разницу между системными вызовами и обычными функциями. Имеется в виду, что при перехвате сигнала прерывается выполнение именно системного вызова в ядре.

Чтобы обеспечить поддержку такого поведения, системные вызовы были разделены на «медленные» и все остальные. Медленные системные вызовы – это такие вызовы, которые могут заблокировать процесс навсегда. В эту категорию попали:

- Операция чтения, которая может навсегда заблокировать вызывающий процесс при отсутствии данных в файлах некоторых типов (каналы, устройства терминалов и сетевые устройства).
- Операция записи, которая может навсегда заблокировать вызывающий процесс, если записываемые данные не могут быть немедленно отправлены в файлы перечисленных выше типов.
- Операция открытия, которая может заблокировать вызывающий процесс до тех пор, пока не будут выполнены некоторые условия при открытии файлов определенных типов (таких как устройства терминалов, которые ожидают установления модемного соединения).
- Функция `pause` (которая по определению приостанавливает выполнение процесса до получения сигнала) и функция `wait`.
- Некоторые операции функции `ioctl`.
- Некоторые функции межпроцессного взаимодействия (глава 15).

Известное исключение из этих медленных системных вызовов составляют операции ввода-вывода с дисковыми устройствами. Несмотря на то, что операции чтения и записи на дисковое устройство могут временно заблокировать выполнение процесса (пока драйвер дискового устройства ставит запрос

в очередь, чтобы затем выполнить его), тем не менее при отсутствии аппаратных ошибок операции ввода-вывода всегда завершаются достаточно быстро и разблокируют вызывающий процесс.

Один из случаев, которые могут привести к прерыванию системного вызова, – когда процесс инициирует операцию чтения из терминала, а пользователь в это время уходит на неопределенно долгое время. В этом случае процесс может быть заблокирован на многие часы или даже дни и оставаться в таком состоянии, если система не будет остановлена.

> Семантика прерванных системных вызовов `read` и `write` была изменена в версии стандарта POSIX.1 от 2001 года. Предшествующие версии стандарта оставляли за реализацией выбор, как обрабатывать операции чтения и записи, которые уже частично передали некоторый объем данных. Если вызов `read` прочитал данные и поместил их в буфер приложения, но к моменту прерывания был получен не весь объем запрошенных данных, тогда решение вопроса, завершить ли системный вызов с кодом ошибки `EINTR` или позволить ему завершиться без признака ошибки и вернуть частично полученные данные, оставлялось на усмотрение операционной системы. Точно так же, если операция записи была прервана после передачи некоторого объема данных из буфера приложения, система могла завершить системный вызов с кодом ошибки `EINTR` или позволить ему завершиться без признака ошибки и вернуть информацию о количестве записанных данных. Исторически сложилось так, что реализации, происходящие от System V, завершают системный вызов с признаком ошибки, тогда как реализации, производные от BSD, возвращают управление без признака ошибки с информацией о фактически выполненной работе. Начиная с версии 2001 года стандарт POSIX.1 требует соблюдения BSD-подобной семантики.

Проблема с прерванными системными вызовами заключается в том, что приходится явно обрабатывать возможные ошибочные ситуации. Типичная последовательность инструкций (в случае операции чтения, когда необходимо прочитать полный объем данных, даже если операция чтения была прервана) может быть следующей:

```
again:
    if ((n = read(fd, buf, BUFFSIZE)) < 0) {
        if (errno == EINTR)
            goto again; /* просто прерванный системный вызов */
        /* обработать другие возможные ошибки */
    }
```

Чтобы избавить приложения от необходимости обрабатывать ситуации прерванных системных вызовов, в 4.2BSD было введено понятие автоматического перезапуска прерванных системных вызовов. К системным вызовам, которые перезапускаются автоматически, были отнесены `ioctl`, `read`, `readv`, `write`, `writev`, `wait` и `waitpid`. Как мы уже упоминали, первые пять вызовов прерываются сигналами только в том случае, если они взаимодействуют с медленными устройствами. Системные вызовы `wait` и `waitpid` всегда прерываются перехваченными сигналами. Так как это породило другую проблему для приложений, которые не желали, чтобы системный вызов автоматически перезапускался в случае его прерывания, в 4.3BSD у процессов появилась возможность изменить такое поведение для отдельных сигналов.

Стандарт POSIX.1 позволяет реализациям перезапускать системные вызовы, но это не обязательное требование. Стандарт Single UNIX Specification определяет флаг как расширение XSI `SA_RESTART` функции `sigaction`, который позволяет приложениям запрашивать перезапуск прерванных системных вызовов.

По умолчанию System V никогда не перезапускала системные вызовы. С другой стороны, BSD перезапускает их, если они были прерваны сигналами. По умолчанию FreeBSD 5.2.1, Linux 2.4.22 и Mac OS X 10.3 перезапускают системные вызовы, прерванные сигналами. Однако в ОС Solaris 9 по умолчанию должна возвращаться ошибка (`EINTR`).

Одна из причин, по которой в 4.2BSD был введен автоматический перезапуск, заключается в том, что иногда мы просто не знаем, является ли устройство ввода-вывода медленным устройством. Если мы пишем программу, которая может работать в интерактивном режиме, то ей, вероятно, придется работать с медленным устройством, так как терминалы относятся к этой категории. Если эта программа перехватывает сигналы, то в случае, когда система не поддерживает возможность перезапуска системных вызовов, пришлось бы выполнять проверку каждой операции чтения и записи на предмет появления ошибки `EINTR` и возобновлять прерванную операцию.

В табл. 10.2 приводятся функции, предназначенные для работы с сигналами, и их семантика для некоторых реализаций.

Таблица 10.2. Функции, предоставляемые различными реализациями

Функция	Система	Обработчик сигнала остается установленным	Возможность блокировать сигналы	Автоматический перезапуск прерванных системных вызовов
`signal`	ISO C, POSIX.1	Не определено	Не определено	Не определено
	V7, SVR2, SVR3, SVR4, Solaris			Никогда
	4.2BSD	•	•	Всегда
	4.3BSD, 4.4BSD, FreeBSD, Linux, Mac OS X	•	•	По умолчанию
`sigset`	XSI	•	•	Не определено
	SVR3, SVR4, Solaris, Linux	•	•	Никогда
`sigvec`	4.2BSD	•	•	Всегда
	4.3BSD, 4.4BSD, FreeBSD, Mac OS X	•	•	По умолчанию
`sigaction`	POSIX.1	•	•	Не определено
	XSI, 4.4BSD, SVR4, FreeBSD, Mac OS X, Linux, Solaris	•	•	По выбору

Мы не будем рассматривать устаревшие функции `sigset` и `sigvec`. Их задачи теперь выполняет функция `sigaction` – мы упомянули их лишь для полноты списка. Функция

signal, напротив, сохраняется в некоторых реализациях как упрощенный интерфейс к функции sigaction.

Имейте в виду, что версии UNIX от других производителей могут иметь значения, отличные от перечисленных в таблице. Например, функция sigaction в ОС SunOS 4.1.2 по умолчанию перезапускает прерванные системные вызовы, что отличает ее от платформ, представленных в таблице.

В листинге 10.12 приводится наша версия функции signal, которая автоматически пытается перезапустить прерванные системные вызовы (за исключением сигнала SIGALRM). Другая функция, signal_intr, которая приводится в листинге 10.13, никогда не пытается выполнить перезапуск.

Мы еще будем говорить о прерванных системных вызовах в разделе 14.5, при обсуждении функций select и poll.

10.6. Реентерабельные функции

Когда процесс обрабатывает перехваченный сигнал, нормальная последовательность выполнения инструкций временно нарушается обработчиком сигнала. После этого процесс продолжает работу, но выполняет инструкции уже в функции-обработчике. Если обработчик сигнала возвращает управление (а не вызывает, например, функцию exit или longjmp), то процесс продолжает выполнение нормальной последовательности инструкций, прерванной перехваченным сигналом. (Это напоминает ситуацию, когда работа приложения прерывается аппаратным прерыванием.) Но, находясь внутри обработчика сигнала, мы не можем определить, в каком месте процесса произошло прерывание. А что если процесс вызвал функцию malloc, чтобы распределить дополнительную память, и обработчик сигнала также вызвал функцию malloc? Или если процесс произвел вызов функции, такой как getpwnam (раздел 6.2), сохраняющей результат статически, и обработчик сигнала также вызвал ту же самую функцию? В случае функции malloc результаты такого вызова могут оказаться разрушительными для приложения, поскольку обычно функция malloc поддерживает связанный список всех выделенных ею областей памяти, и вызов из обработчика сигнала может произойти как раз в тот момент, когда она вносит изменения в этот список. В случае с функцией getpwnam информация, записанная по запросу процесса, может оказаться затертой информацией, запрошенной из обработчика сигнала.

Стандарт Single UNIX Specification определяет перечень функций, которые должны обеспечивать возможность повторного вхождения (реентерабельность). Эти функции перечислены в табл. 10.3.

Таблица 10.3. Реентерабельные функции, которые могут быть вызваны из обработчика сигнала

accept	fchmod	lseek	sendto	stat
access	fchown	lstat	setgid	symlink
aio_error	fcntl	mkdir	setpgid	sysconf

aio_return	fdatasync	mkfifo	setsid	tcdrain
aio_suspend	fork	open	setsockopt	tcflow
alarm	fpathconf	pathconf	setuid	tcflush
bind	fstat	pause	shutdown	tcgetattr
cfgetispeed	fsync	pipe	sigaction	tcgetpgrp
cfgetospeed	ftruncate	poll	sigaddset	tcsendbreak
cfsetispeed	getegid	posix_trace_event	sigdelset	tcsetattr
cfsetospeed	geteuid	pselect	sigemptyset	tcsetpgrp
chdir	getgid	raise	sigfillset	time
chmod	getgroups	read	sigismember	timer_getoverrun
chown	getpeername	readlink	signal	timer_gettime
clock_gettime	getpgrp	recv	sigpause	timer_settime
close	getpid	recvfrom	sigpending	times
connect	getppid	recvmsg	sigprocmask	umask
creat	getsockname	rename	sigqueue	uname
dup	getsockopt	rmdir	sigset	unlink
dup2	getuid	select	sigsuspend	utime
execle	kill	sem_post	sleep	wait
execve	link	send	socket	waitpid
_Exit и _exit	listen	sendmsg	socketpair	write

Большинство функций, отсутствующих в табл. 10.3, не были внесены в список либо потому, что (а) известно, что они используют структуры данных, размещаемые статически, либо (б) они вызывают функцию `malloc` или `free`, либо (в) они входят в стандартную библиотеку ввода-вывода. Большинство реализаций стандартной библиотеки ввода-вывода используют глобальные структуры данных способом, исключающим реентерабельность. Обратите внимание, что хотя мы вызываем функцию `printf` из обработчиков сигналов в некоторых наших примерах, нельзя гарантировать, что этот вызов даст предсказуемый результат, так как обработчик сигнала может быть вызван в процессе работы функции `printf`, вызванной в другом месте программы.

Также необходимо знать, что даже если вызов одной из функций, перечисленных в табл. 10.3, производится из обработчика сигнала, для каждого из потоков управления существует единственная переменная `errno` (вспомните раздел 1.7), и мы можем изменить ее значение. Например, обработчик сигнала может быть вызван сразу же после того, как код ошибки был записан в переменную `errno` в главной программе. Если обработчик сигнала вызывает, например, функцию `read`, то она может изменить значение этой переменной, затерев значение, только что записанное в главной программе. Таким образом, основное правило, которому вы должны следовать при вызове

функций из табл. 10.3, заключается в том, чтобы сохранять значение переменной errno в начале обработчика и восстанавливать его перед возвратом в главную программу. (Чаще всего при перехвате сигнала SIGCHLD функция-обработчик обращается к одной из функций wait, которая может изменить значение переменной errno.)

Обратите внимание: функции longjmp (раздел 7.10) и siglongjmp (раздел 10.15) отсутствуют в табл. 10.3, поскольку обработчик сигнала может быть вызван как раз в тот момент, когда эти функции выполняют обновление структур данных нереентерабельным способом. Эти структуры данных могут оказаться обновленными частично, если вместо обычного возврата из функции-обработчика вызвать функцию siglongjmp. Если необходимо сделать что-то с глобальными структурами данных в то время, когда может быть вызван обработчик сигнала, вызывающий функцию sigsetjmp, приложение должно блокировать сигналы на время обновления этих данных.

Пример

В листинге 10.2 представлена программа, которая вызывает нереентерабельную функцию getpwnam из обработчика сигнала, вызываемого один раз в секунду. Функцию alarm мы рассмотрим в разделе 10.10. Мы использовали ее для генерации сигнала SIGALRM каждую секунду.

Листинг 10.2. Вызов нереентерабельной функции из обработчика сигнала

```
#include "apue.h"
#include <pwd.h>

static void
my_alarm(int signo)
{
    struct passwd *rootptr;

    printf("внутри обработчика сигнала\n");
    if ((rootptr = getpwnam("root")) == NULL)
        err_sys("ошибка вызова функции getpwnam(root)");
    alarm(1);
}

int
main(void)
{
    struct passwd *ptr;

    signal(SIGALRM, my_alarm);
    alarm(1);
    for ( ; ; ) {
        if ((ptr = getpwnam("sar")) == NULL)
            err_sys("ошибка вызова функции getpwnam");
        if (strcmp(ptr->pw_name, "sar") != 0)
            printf("возвращаемое значение повреждено!, pw_name = %s\n",
                   ptr->pw_name);
    }
}
```

Когда мы запустили эту программу, она начала выдавать совершенно непредсказуемые результаты. Обычно программа завершалась по сигналу SIGSEGV, когда обработчик сигнала возвращал управление в программу после нескольких итераций. Исследование файла core показало, что функция main вызывала функцию getpwnam, но некоторые внутренние указатели были повреждены в результате вызова той же функции из обработчика сигнала. Иногда программе удавалось проработать несколько секунд, прежде чем она получала сигнал SIGSEGV. Когда функция main работала вполне корректно после вызова обработчика сигнала, возвращаемое значение getpwnam иногда оказывалось поврежденным, а иногда – нет. Однажды в Mac OS X было выведено сообщение от библиотечной функции malloc, предупреждающее о попытке освободить память, которая не была распределена функцией malloc.

Как показывает этот пример, вызов нереентерабельных функций из обработчиков сигналов может дать самые непредсказуемые результаты.

10.7. Семантика сигнала SIGCLD

Два сигнала, которые продолжают вносить сумятицу в умы программистов – SIGCLD и SIGCHLD. Прежде всего, SIGCLD (без H) – это имя сигнала, пришедшее из System V. Семантика этого сигнала отличается от семантики BSD-сигнала с именем SIGCHLD. Соответствующий сигнал из стандарта POSIX.1 также получил имя SIGCHLD.

Семантика сигнала SIGCHLD подобна семантике всех остальных сигналов. Когда изменяется состояние дочернего процесса, генерируется сигнал SIGCHLD, и мы должны вызвать одну из функций семейства wait, чтобы узнать, что произошло.

Однако System V традиционно обслуживает сигнал SIGCLD иначе, чем остальные сигналы. Системы, основанные на SVR4, продолжают эту сомнительную (в смысле совместимости) традицию, если диспозиция этого сигнала устанавливается функциями signal или sigset (устаревшие SVR3-совместимые функции, предназначенные для изменения диспозиции сигнала). Этот способ обслуживания сигнала SIGCLD заключается в следующем.

1. Если процесс установит его диспозицию в значение SIG_IGN, то дочерние процессы не будут порождать процессы-зомби. Обратите внимание: это действие отличается от действия по умолчанию SIG_DFL, которое в соответствии с табл. 10.1 заключается в том, чтобы игнорировать сигнал. Вместо этого по завершении дочернего процесса его код завершения просто теряется. Если затем вызвать одну из функций wait, то вызывающий процесс окажется заблокированным до тех пор, пока все его дочерние процессы не завершат работу, после чего wait вернет значение –1 с кодом ошибки ECHILD в переменной errno. (Диспозиция сигнала по умолчанию – игнорировать сигнал, но это не означает, что его семантика будет следовать семантике SIG_IGN. Поэтому мы должны явно установить диспозицию сигнала в значение SIG_IGN.)

Стандарт POSIX.1 не определяет поведение системы в том случае, когда сигнал SIGCHLD игнорируется, поэтому подобное поведение вполне допустимо. Стандарт Single UNIX Specification включает расширение XSI, определяющее, что такое поведение должно поддерживаться для сигнала SIGCHLD.

В 4.4BSD, если сигнал SIGCHLD игнорируется, это всегда приводит к созданию зомби. Чтобы избежать появления зомби, мы должны вызывать функцию wait для дочерних процессов. ОС FreeBSD 5.2.1 ведет себя аналогично 4.4BSD. Однако Mac OS X 10.3 не создает зомби, если сигнал SIGCHLD игнорируется.

Если в SVR4 вызывается функция signal или sigset, чтобы установить диспозицию сигнала SIGCHLD в значение SIG_IGN, зомби никогда не появляются. ОС Solaris 9 и Linux 2.4.22 в своем поведении следуют за SVR4.

При использовании функции sigaction можно установить флаг SA_NOCLDWAIT (табл. 10.5), чтобы избежать появления зомби. Это действие поддерживается всеми четырьмя платформами: FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3 и Solaris 9.

2. Если для сигнала SIGCLD назначена функция-обработчик, то ядро сразу же проверяет наличие дочерних процессов и вызывает обработчик сигнала SIGCLD, если такие процессы имеются.

Пункт 2 меняет алгоритм обработчика сигнала, как показано в следующем примере.

Пример

Как мы уже говорили в разделе 10.4, первое, что нужно сделать на входе в обработчик сигнала – это переустановить его с помощью вызова функции signal. (Тем самым мы минимизируем интервал времени, когда сигнал может быть потерян в результате временного сброса диспозиции в значение по умолчанию.) Этот прием демонстрируется в листинге 10.3. Данная программа не работает на некоторых платформах. Если скомпилировать и запустить ее на одной из традиционных платформ System V, таких как OpenServer 5 или UnixWare 7, то мы получим непрерывный поток сообщений SIGCLD received (принят сигнал SIGCLD), пока, наконец, процесс не завершится аварийно в результате исчерпания пространства, отведенного под стек.

В ОС FreeBSD 5.2.1 и Mac OS X 10.3 эта проблема не проявляется, потому что системы, основанные на BSD, вообще не поддерживают семантику System V для сигнала SIGCLD. ОС Linux 2.4.22 также лишена этого недостатка, потому что она не вызывает обработчик сигнала SIGCHLD сразу же после его установки – даже при том, что сигналы SIGCLD и SIGCHLD определены с одним и тем же номером. С другой стороны, в ОС Solaris 9 обработчик сигнала в такой ситуации действительно вызывается, но в ядро включен дополнительный код, который помогает избежать описанной проблемы.

Хотя все четыре рассматриваемые в этой книге платформы разрешили данную проблему, тем не менее есть такие системы (например, UnixWare), в которых она все еще существует.

Листинг 10.3. Обработчик сигнала SIGCLD из System V, который не работает

```
#include "apue.h"
#include <sys/wait.h>
```

```
static void sig_cld(int);
int
main()
{
    pid_t pid;

    if (signal(SIGCLD, sig_cld) == SIG_ERR)
        perror("ошибка вызова функции signal");
    if ((pid = fork()) < 0) {
        perror("ошибка вызова функции fork");
    } else if (pid == 0) {                /* дочерний процесс */
        sleep(2);
        _exit(0);
    }
    pause();                              /* родительский процесс */
    exit(0);
}

static void
sig_cld(int signo)                        /* прерывает функцию pause() */
{
    pid_t pid;
    int status;

    printf("принят сигнал SIGCLD\n");
    if (signal(SIGCLD, sig_cld) == SIG_ERR) /* переустановить обработчик */
        perror("ошибка вызова функции signal");
    if ((pid = wait(&status)) < 0)  /* получить состояние дочернего процесса */
        perror("ошибка вызова функции wait");
    printf("pid = %d\n", pid);
}
```

Проблема этой программы состоит в том, что она вызывает функцию `signal` в самом начале функции-обработчика, и это приводит к тому, что ядро проверяет наличие дочерних процессов (а дочерний процесс имеется, поскольку мы уже находимся в обработчике сигнала `SIGCLD`), что в свою очередь приводит к очередному вызову обработчика сигнала. Функция-обработчик опять вызывает функцию `signal`, и цикл повторяется.

Чтобы исправить эту ошибку, мы должны поместить вызов функции `signal` после вызова функции `wait`. Благодаря этому вызов функции `signal` будет производиться после получения кода завершения дочернего процесса, и в следующий раз сигнал будет сгенерирован ядром, только если завершится один из дочерних процессов.

> Формулировка стандарта POSIX.1 не определяет, должен ли генерироваться сигнал в случае, когда к моменту установки обработчика сигнала `SIGCHLD` существует завершившийся дочерний процесс, код завершения которого еще не был получен. Это допускает реализацию поведения, описанного выше. Но поскольку стандарт POSIX.1 не предполагает сброс диспозиции сигнала в значение по умолчанию после его появления (здесь мы предполагаем, что для изменения диспозиции сигнала используется функция POSIX.1 – `sigaction`), то и нет никакой потребности в том, чтобы переустанавливать обработчик сигнала `SIGCHLD` в теле функции-обработчика.

Поинтересуйтесь семантикой сигнала SIGCHLD в вашей системе. В особенности обратите внимание на определение, которое в некоторых системах выглядит как #define SIGCHLD SIGCLD, а в других – наоборот. Изменение имени сигнала может устранить проблемы при сборке программы, которая была написана для другой системы, но если эта программа зависит от конкретной семантики, то возможно, что она не будет работать.

> На четырех платформах, обсуждаемых в данной книге, сигнал SIGCLD эквивалентен сигналу SIGCHLD.

10.8. Надежные сигналы. Терминология и семантика

Нам необходимо определить некоторые термины, которые будут использоваться при обсуждении сигналов. Прежде всего, для процесса *генерируется* (или процессу посылается) сигнал, когда происходит некоторое событие. Таким событием может быть аппаратная ошибка (например, деление на 0), программное событие (например, истечение интервала времени, отмеряемого таймером), сигнал, сгенерированный терминалом, или вызов функции kill. Когда генерируется сигнал, ядро, как правило, взводит некоторый флаг в таблице процессов.

Когда выполняется действие, предусмотренное для сигнала, мы говорим, что сигнал был *доставлен* процессу. Интервал времени между генерацией сигнала и его доставкой называется периодом *ожидания обработки*.

Процесс может *заблокировать* доставку сигнала. Если процессу посылается сигнал, который был заблокирован, и при этом для сигнала установлено либо действие по умолчанию, либо перехват, то сигнал остается в состоянии ожидания обработки до тех пор, пока процесс (а) не разблокирует сигнал или (б) не установит диспозицию сигнала в значение SIG_IGN. Что делать с сигналом, определяется системой в момент доставки, но не в момент генерации. Это позволяет процессам изменять диспозицию сигнала до того, как он будет доставлен. Процесс может получить перечень ожидающих или заблокированных сигналов с помощью функции sigpending.

Что произойдет, если заблокированный сигнал будет сгенерирован несколько раз, прежде чем процесс разблокирует его? Стандарт POSIX.1 допускает доставку как единственного сигнала, так и всех сгенерированных сигналов. Если система доставляет процессу все сгенерированные сигналы, мы говорим, что сигналы ставятся в очередь. Однако большинство версий UNIX при отсутствии расширений реального времени POSIX.1 не ставят сигналы в очередь, то есть ядро доставляет единственный сигнал.

> Справочное руководство SVR2 утверждало, что сигнал SIGCLD ставится в очередь, если процесс в данный момент выполняет функцию обработки сигнала SIGCLD. Возможно, это было истинно только на концептуальном уровне, поскольку фактическая реализация была иной. На самом деле ядро повторно генерировало сигнал, как это было описано в разделе 10.7. В SVR3 текст справочного руководства претерпел некоторые из-

менения: сообщается, что в момент обработки сигнала SIGCLD последующие его доставки игнорируются. В справочном руководстве SVR4 вообще исчезло любое упоминание о том, что происходит с сигналом SIGCLD, полученным в то время, когда процесс выполняет код функции-обработчика этого сигнала.

Страница справочного руководства SVR4 к функции sigaction(2) утверждает, что существует надежный способ поставить сигнал в очередь с помощью флага SA_SIGINFO. Это не соответствует истине. Очевидно, такая возможность была частично реализована в ядре, но она не используется в SVR4. Любопытно, что руководство SVID не делает подобных заявлений.

Что произойдет, если сразу несколько сигналов одновременно будут готовы к доставке? Стандарт POSIX.1 не определяет порядок доставки сигналов. Однако «POSIX.1 Rationale» предлагает в первую очередь доставлять сигналы, которые имеют отношение к текущему состоянию процесса (один из таких сигналов – SIGSEGV).

Каждый процесс имеет *маску сигналов*, с помощью которой определяется множество блокируемых сигналов. Ее можно представлять себе как битовую маску, в которой каждый бит соответствует отдельному сигналу. Если некоторый бит включен, то доставка соответствующего ему сигнала блокируется. Процесс может проверить и изменить маску с помощью функции sigprocmask, которая будет описана в разделе 10.12.

Поскольку существует вероятность, что количество сигналов превысит количество бит в целочисленном типе, стандарт POSIX.1 предусматривает специальный тип данных sigset_t для хранения *набора сигналов*. В разделе 10.11 мы рассмотрим пять функций, предназначенных для работы с наборами сигналов.

10.9. Функции kill и raise

Функция kill посылает сигнал процессу или группе процессов. Функция raise позволяет процессу послать сигнал себе самому.

Изначально функция raise была определена стандартом ISO C. Стандарт POSIX.1 включает эту функцию, чтобы соблюсти соответствие со стандартом ISO C, однако он расширяет ее спецификацию для обеспечения работы с потоками (взаимодействие потоков с сигналами мы обсудим в разделе 12.8). Поскольку стандарт ISO C ничего не говорит о многозадачности, в нем отсутствует определение функции kill, которая требует передачи идентификатора процесса в качестве одного из аргументов.

```
#include <signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);
```

Обе возвращают 0 в случае успеха, −1 в случае ошибки

Вызов

```
raise(signo);
```

эквивалентен вызову

```
kill(getpid(), signo);
```

Интерпретация аргумента *pid* функции kill производится в соответствии со следующими правилами.

pid > 0 Сигнал посылается процессу с идентификатором *pid*.

pid == 0 Сигнал посылается всем процессам с идентификатором группы процессов, равным идентификатору группы процессов посылающего процесса, которым данный процесс имеет право посылать сигналы. Обратите внимание, что в понятие *все процессы* не входят определяемые реализацией системные процессы. В большинстве версий UNIX под системными понимаются процессы ядра и процесс init (идентификатор процесса 1).

pid < 0 Сигнал посылается всем процессам с идентификатором группы процессов, равным абсолютному значению *pid*, которым данный процесс имеет право посылать сигналы. Опять же в понятие *все процессы* не входят определяемые реализацией системные процессы.

pid == -1 Сигнал посылается всем процессам в системе, которым посылающий процесс имеет право посылать сигналы. Здесь точно так же из понятия *все процессы* исключаются некоторые системные процессы.

Как мы уже упоминали, процесс должен обладать определенными правами, чтобы посылать сигналы другим процессам. Так, суперпользователь может послать сигнал любому процессу. В остальных случаях должно соблюдаться основное правило – реальный или эффективный идентификатор пользователя процесса, посылающего сигнал, должен совпадать с реальным или эффективным идентификатором пользователя процесса, принимающего сигнал. Если реализация поддерживает возможность _POSIX_SAVED_IDS (которая ныне считается обязательной), то вместо эффективного идентификатора пользователя проверяется сохраненный идентификатор пользователя. Из этого правила существует одно исключение: сигнал SIGCONT может быть послан любому другому процессу, принадлежащему той же самой сессии.

Стандарт POSIX.1 определяет сигнал с номером 0 как пустой сигнал. Если аргумент *signo* имеет значение 0, то функция kill выполняет обычную проверку на наличие ошибок, но сам сигнал не посылается. Это часто используется для того, чтобы определить, существует ли еще некоторый процесс. Если несуществующему процессу послать пустой сигнал, функция kill вернет значение –1 и код ошибки ESRCH в переменной errno. Однако следует иметь в виду, что через некоторый промежуток времени идентификаторы процессов могут быть использованы повторно, поэтому наличие процесса с заданным идентификатором вовсе не означает, что это тот самый процесс, который вам нужен.

Кроме того, проверка существования процесса не является атомарной операцией. К моменту, когда функция kill вернет управление в вызывающую программу, проверяемый процесс уже может завершиться, что сильно ограничивает область применения такого приема.

Если в результате вызова функции kill генерируется сигнал для вызывающего процесса и при этом сигнал не заблокирован, тогда либо сигнал с номером *signo*, либо другой ожидающий обработки сигнал будет доставлен процессу еще до того, как функция kill вернет управление. (В случае потоков возникает ряд дополнительных вариантов, за информацией обращайтесь к разделу 12.8.)

10.10. Функции alarm и pause

Функция alarm позволяет установить таймер, по истечении периода времени которого будет сгенерирован сигнал SIGALRM. Если этот сигнал не игнорируется и не перехватывается приложением, он вызывает завершение процесса.

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Возвращает 0 или количество секунд до истечения периода времени, установленного ранее

Аргумент *seconds* определяет количество секунд, через которое должен быть сгенерирован сигнал. Следует помнить, что между моментом генерации сигнала и моментом доставки его приложению пройдет некоторое время.

> В ранних версиях UNIX оговаривалось, что сигнал может быть сгенерирован чуть раньше (на секунду или менее). Стандарт POSIX.1 не допускает этого.

Каждый процесс может обладать только одним таким таймером. Если функция alarm вызывается до истечения таймера, установленного ранее, то она возвращает количество оставшихся секунд, а ранее установленный интервал времени заменяется новым.

Если ранее установленный интервал времени еще не истек и в аргументе *seconds* передается значение 0, то взведенный таймер останавливается, а функция возвращает количество секунд, оставшихся до истечения периода времени таймера.

Действие сигнала SIGALRM по умолчанию заключается в завершении процесса, но большинство приложений перехватывают его. Если в результате получения этого сигнала приложение должно завершить работу, оно может выполнить все необходимые заключительные операции перед выходом. Если предполагается перехват сигнала SIGALRM, то необходимо установить обработчик сигнала до того, как будет вызвана функции alarm. Если функция alarm будет вызвана первой и при этом успеет сгенерировать сигнал до установки обработчика сигнала, то процесс завершится.

Функция pause приостанавливает вызывающий процесс до тех пор, пока не будет перехвачен сигнал.

```
#include <unistd.h>
int pause(void);
```

Возвращает значение −1 с кодом ошибки EINTR в переменной errno

Функция pause возвращает управление только тогда, когда отработает функция-обработчик сигнала. В этом случае она возвращает значение –1 с кодом ошибки EINTR в переменной errno.

Пример

С помощью функций alarm и pause можно приостановить процесс на определенный промежуток времени. На первый взгляд функция sleep1 из листинга 10.4 выполняет эту задачу, однако в ней кроется ряд ошибок, о которых мы вскоре поговорим.

Листинг 10.4. Простейшая, но не полная реализация функции sleep

```
#include <signal.h>
#include <unistd.h>

static void
sig_alrm(int signo)
{
    /* ничего не делаем, просто возвращаем управление */
}

unsigned int
sleep1(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    alarm(nsecs);       /* запустить таймер, следующий перехваченный сигнал */
                        /* возобновит работу процесса */
    return(alarm(0));   /* выключить таймер и вернуть время, */
                        /* оставшееся до его истечения */
}
```

Эта функция напоминает функцию sleep, которая будет описана в разделе 10.19, но в данной реализации кроются три проблемы.

1. Если вызывающий процесс уже установил таймер, то его значение будет затерто первым вызовом функции alarm. Мы можем исправить эту ошибку, проанализировав возвращаемое функцией значение. Если оставшееся количество секунд меньше, чем значение аргумента nsecs, то мы должны оставить прежнее значение таймера. Если значение таймера больше, чем значение аргумента nsecs, то после того, как таймер сработает через nsecs секунд, мы должны переустановить его так, чтобы он повторно сработал в указанное ранее время.
2. Наша функция изменяет диспозицию сигнала SIGALRM. Если мы предполагаем использовать функцию в других приложениях, то мы должны сохранять диспозицию сигнала при вызове функции и восстанавливать ее по завершении. Эту ошибку можно исправить, сохраняя возвращаемое значение функции signal и восстанавливая прежнюю диспозицию перед выходом.
3. Между первым вызовом функции alarm и вызовом функции pause возникает состояние гонки за ресурсами. При значительной нагрузке на систе-

му есть вероятность того, что таймер сработает и функция-обработчик будет вызвана еще до вызова функции pause. Если это произойдет, вызывающий процесс будет навсегда приостановлен в функции pause (если, конечно, он не перехватит какой-нибудь другой сигнал).

Ранние реализации функции sleep выглядели подобно нашей программе, но ошибки 1 и 2 были исправлены, как описано выше. Для исправления третьей ошибки существует два пути. Первый из них – использовать функцию setjmp, этот подход мы продемонстрируем в следующем примере. Второй – использовать функции sigprocmask и sigsuspend, которые мы рассмотрим в разделе 10.19.

Пример

В SVR2 реализация функции sleep во избежание гонки за ресурсами использовала функции setjmp и longjmp (раздел 7.10). Простейшая версия этой функции, которую мы назвали sleep2, приводится в листинге 10.5. (Чтобы сократить размер листинга, мы не включили устранение ошибок 1 и 2.)

Листинг 10.5. Другая (неполная) реализация функции sleep

```
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>

static jmp_buf env_alrm;

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}

unsigned int
sleep2(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    if (setjmp(env_alrm) == 0) {
        alarm(nsecs);   /* запустить таймер */
        pause();        /* следующий же перехваченный сигнал */
                        /* возобновит работу процесса */
    }
    return(alarm(0));   /* выключить таймер и вернуть время, */
                        /* оставшееся до его истечения */
}
```

В функции sleep2 исключается возможность попасть в состояние гонки за ресурсами. Даже если функция pause никогда не будет вызвана, sleep2 все равно вернет управление после доставки сигнала SIGALRM.

Однако в данной версии существует еще одна малозаметная ошибка, которая связана с взаимодействием с другими сигналами. Если сигнал SIGALRM

будет получен при выполнении функции-обработчика другого сигнала, то вызов функции longjmp оборвет обработку этого сигнала. Программа из листинга 10.6 демонстрирует такое развитие событий. Цикл в обработчике сигнала SIGINT построен таким образом, что в операционной системе, которую использовал автор книги, он выполняется дольше 5 секунд. Это нужно для того, чтобы время работы этого обработчика было больше, чем значение аргумента функции sleep2. Переменная k объявлена со спецификатором volatile, чтобы предотвратить нарушение цикла в результате оптимизации, которую выполняет компилятор. Запуск программы из листинга 10.6 дал следующие результаты:

```
$ ./a.out
Б?                                  был введен символ прерывания
функция sig_int начала обработку
функция sleep2 вернула значение: 0
```

Как видите, вызов longjmp из sleep2 оборвал работу другого обработчика сигнала (sig_int), не дав ему завершиться. С этим вы столкнетесь, если будете смешивать использование функции sleep с обработкой других сигналов (упражнение 10.3).

Листинг 10.6. Вызов функции sleep2 из программы, которая перехватывает другие сигналы

```
#include "apue.h"

unsigned int sleep2(unsigned int);
static void sig_int(int);

int
main(void)
{
    unsigned int unslept;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGINT)");
    unslept = sleep2(5);
    printf("функция sleep2 вернула значение: %u\n", unslept);
    exit(0);
}

static void
sig_int(int signo)
{
    int i, j;
    volatile int k;

    /*
     * Настройте параметры циклов таким образом, чтобы они выполнялись
     * дольше 5 секунд в системе, где запускается эта программа.
     */
    printf("\nфункция sig_int начала обработку\n");
    for (i = 0; i < 300000; i++)
        for (j = 0; j < 4000; j++)
```

```
            k += i * j;
    printf("функция sig_int закончила обработку\n");
}
```

Цель этих двух примеров функций `sleep1` и `sleep2` состоит в том, чтобы продемонстрировать возможные проблемы при работе с сигналами. В следующем разделе мы покажем приемы, которые помогают избежать этих проблем и надежно обрабатывают сигналы, не вступая в конфликт с другими участками кода.

Пример

Очень часто функция `alarm` используется в паре с функцией `pause` для того, чтобы установить предельное время выполнения некоторых операций, которые могут блокировать процесс. Например, если мы выполняем операцию чтения с «медленного» устройства (раздел 10.5), которая может заблокировать процесс, то у нас может появиться желание ограничить время работы функции `read` некоторым промежутком времени. Программа из листинга 10.7 читает данные со стандартного ввода и выводит их на стандартный вывод, ограничивая при этом время операции чтения.

Листинг 10.7. Вызов функции read с тайм-аутом

```
#include "apue.h"

static void sig_alrm(int);
int

main(void)
{
    int n;
    char line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGALRM)");
    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("ошибка вызова функции read");
    alarm(0);
    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    /* ничего не делать, просто прервать работу функции read */
}
```

Такая последовательность инструкций – обычное дело для приложений UNIX, но в этой программе кроются две проблемы.

1. Программа из листинга 10.7 имеет один из недостатков, присущих программе из листинга 10.4: вероятность попасть в состояние гонки за ресур-

сами между первым вызовом функции alarm и вызовом функции read. Если ядро успеет заблокировать процесс между этими двумя вызовами на больший период времени, чем период срабатывания таймера, то процесс рискует оказаться навсегда заблокированным в функции read. В большинстве подобных случаев используются длительные тайм-ауты, порядка минуты или больше, но, тем не менее, вероятность попасть в состояние гонки за ресурсами все равно сохраняется.

2. Если системные вызовы перезапускаются автоматически, то выполнение функции read не будет прервано после выхода из обработчика сигнала SIGALRM. В этом случае установка тайм-аута ничего не даст.

Пример

Давайте перепишем предыдущий пример так, чтобы он использовал функцию longjmp. Это позволит прервать работу медленного системного вызова в любом случае.

Листинг 10.8. Вызов функции read с тайм-аутом с использованием функции longjmp

```
#include "apue.h"
#include <setjmp.h>

static void sig_alrm(int);
static jmp_buf env_alrm;

int
main(void)
{
    int n;
    char line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGALRM)");
    if (setjmp(env_alrm) != 0)
        err_quit("работа функции read прервана по тайм-ауту");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("ошибка вызова функции read");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}
static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}
```

Эта версия работает так, как мы и ожидали, независимо от того, перезапускает система прерванные системные вызовы или нет. Однако не забывайте,

что эта версия все еще подвержена проблеме, связанной с обработкой других сигналов.

Если мы хотим ограничить время выполнения операций ввода-вывода, мы должны использовать функцию longjmp, как показано выше, но при этом не забывать о возможных конфликтах с другими обработчиками сигналов. Другой способ ограничения выполнения операций по времени предоставляют функции select и poll, которые будут рассматриваться в разделах 14.5.1 и 14.5.2.

10.11. Наборы сигналов

Для представления множества сигналов нам необходим специальный тип данных – *набор сигналов*. Он используется такими функциями, как sigprocmask (будет описана в следующем разделе), чтобы передать ядру набор сигналов, которые должны быть заблокированы. Как уже говорилось ранее, количество различных сигналов может превышать количество бит в целочисленном типе, поэтому в большинстве случаев нельзя использовать тип int для представления набора сигналов, в котором каждому сигналу отводится отдельный бит. Стандарт POSIX.1 определяет для этих целей специальный тип sigset_t, который может хранить набор сигналов и с которым работают следующие пять функций.

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);
```

Все четыре возвращают 0 в случае успеха, –1 в случае ошибки

```
int sigismember(const sigset_t *set, int signo);
```

Возвращает 1 (истина), 0 (ложь), –1 в случае ошибки

Функция sigemptyset инициализирует пустой набор сигналов, на который указывает аргумент *set*. Функция sigfillset инициализирует набор сигналов, в который включены все сигналы. Все приложения должны вызывать функцию sigemptyset или sigfillset для каждого набора сигналов перед его использованием, потому что нельзя предполагать, что инициализация глобальных или статических переменных, выполняемая языком C, соответствует реализации сигналов в заданной системе.

После того как набор сигналов будет инициализирован, можно добавлять или удалять из него сигналы. Добавление одного сигнала в существующий набор производится функцией sigaddset, а удаление сигнала из набора – функцией sigdelset. Все функции, которым передается набор сигналов, в виде аргумента всегда получают указатель на набор сигналов.

Реализация

Если количество сигналов в реализации меньше, чем количество бит в целочисленном типе, набор сигналов может быть реализован на основе этого типа и представлять каждый сигнал отдельным битом. Далее в этом разделе мы будем исходить из предположения, что реализация насчитывает 31 сигнал, а для представления целых чисел используется 32 бита. Таким образом, функция `sigemptyset` обнуляет целое число, а функция `sigfillset` – взводит все биты в целом числе. Эти две функции могут быть реализованы в виде макроопределений в заголовочном файле `<signal.h>`:

```
#define sigemptyset(ptr) (*(ptr) = 0)
#define sigfillset(ptr)  (*(ptr) =  (sigset_t)0, 0)
```

Обратите внимание: поскольку функция `sigfillset` должна устанавливать все биты в наборе сигналов и возвращать значение 0, то в данном определении использован оператор языка C «запятая», который возвращает в качестве значения всего выражения значение, стоящее после запятой.

В такой реализации функция `sigaddset` включает, а функция `sigdelset` – выключает один бит в наборе. Функция `sigismember` проверяет состояние указанного бита. Поскольку сигнал с номером 0 отсутствует, то при определении номера бита из номера сигнала вычитается 1. В листинге 10.9 показана реализация этих функций.

Листинг 10.9. Реализация функций sigaddset, sigdelset и sigismember

```
#include <signal.h>
#include <errno.h>

/*
 * Обычно в файле <signal.h> имеется определение константы NSIG,
 * которая учитывает сигнал с номером 0.
 */
#define SIGBAD(signo) ((signo) <= 0 || (signo) >= NSIG)

int
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set |= 1 << (signo - 1);    /* включить бит */
    return(0);
}

int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set &=  (1 << (signo - 1)); /* выключить бит */
    return(0);
}

int
```

```
sigismember(const sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    return((*set & (1 << (signo - 1))) != 0);
}
```

У нас может возникнуть желание реализовать эти функции в виде коротких макроопределений в заголовочном файле <signal.h>, но стандарт POSIX.1 требует проверки аргумента с номером сигнала, чтобы в случае недопустимого номера сигнала устанавливалась переменная errno. В функции сделать это гораздо проще, чем в макроопределении.

10.12. Функция sigprocmask

В разделе 10.8 мы говорили, что каждый процесс обладает маской сигналов, которая представляет собой набор сигналов, доставка которых должна быть заблокирована. Процесс может получить текущее значение маски, изменить маску или выполнить сразу обе операции с помощью следующей функции.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Прежде всего, если в аргументе *oset* передается пустой указатель, через него возвращается текущая маска сигналов процесса.

Далее, если в аргументе *set* передается непустой указатель, то значение аргумента *how* определяет, как должна измениться маска сигналов. В табл. 10.4 приводятся возможные значения аргумента *how*. Операция SIG_BLOCK представляет собой логическую операцию включающего ИЛИ, тогда как SIG_SETMASK – обычное присваивание. Обратите внимание, что сигналы SIGKILL и SIGSTOP не могут быть заблокированы.

Таблица 10.4. Способы изменения текущего значения маски сигналов с помощью функции sigprocmask

how	Описание
SIG_BLOCK	Новая маска сигналов представляет собой объединение текущей маски сигналов с набором, на который указывает аргумент *set*. Это означает, что аргумент *set* содержит дополнительные сигналы, которые мы желаем заблокировать.
SIG_UNBLOCK	Новая маска сигналов представляет собой пересечение текущей маски сигналов с набором, на который указывает аргумент *set*. Это означает, что аргумент *set* содержит сигналы, которые мы хотим разблокировать.
SIG_SETMASK	Новая маска сигналов, которая представлена аргументом *set*, замещает текущую маску сигналов.

Если в аргументе *set* передается пустой указатель, маска сигналов процесса не изменяется и значение аргумента *how* игнорируется.

После вызова функции sigprocmask, если имеются какие-либо разблокированные сигналы, ожидающие обработки, по меньшей мере один из них будет доставлен приложению перед возвратом из функции.

> Функция sigprocmask определена только для однопоточных процессов. Для работы с масками сигналов в многопоточных приложениях предоставляются отдельные функции. Мы обсудим их в разделе 12.8.

Пример

В листинге 10.10 приводится функция, которая выводит имена сигналов, составляющих маску сигналов вызывающего процесса. Мы будем использовать эту функцию в листингах 10.14 и 10.15.

Листинг 10.10. Вывод маски сигналов процесса

```
#include "apue.h"
#include <errno.h>

void
pr_mask(const char *str)
{
    sigset_t sigset;
    int errno_save;

    errno_save = errno;/* функция может вызываться из обработчиков сигналов */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_sys("ошибка вызова функции sigprocmask");

    printf("%s", str);
    if (sigismember(&sigset, SIGINT)) printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT)) printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1)) printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM)) printf("SIGALRM ");

    /* здесь вы можете продолжить список сигналов */

    printf("\n");
    errno = errno_save;
}
```

Чтобы не «раздувать» листинг, мы не выполняем проверку наличия в маске сигналов каждого сигнала из табл. 10.1 (см. упражнение 10.9).

10.13. Функция sigpending

Функция sigpending возвращает набор сигналов, доставка которых заблокирована и которые в данный момент ожидают обработки. Набор сигналов возвращается через аргумент *set*.

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Пример

Программа, представленная листингом 10.11, демонстрирует многие из описанных выше возможностей сигналов.

Листинг 10.11. Пример работы с наборами сигналов и с функцией sigprocmask

```
#include "apue.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("невозможно перехватить сигнал SIGQUIT");

    /*
     * Заблокировать SIGQUIT и сохранить маску сигналов.
     */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);

    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("ошибка вызова sigprocmask с аргументом SIG_BLOCK");

    sleep(5); /* здесь SIGQUIT останется в ожидании обработки */

    if (sigpending(&pendmask) < 0)
        err_sys("ошибка вызова функции sigpending");
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nсигнал SIGQUIT ожидает обработки\n");

    /*
     * Восстановить маску сигналов, которая разблокирует SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка вызова sigprocmask с аргументом SIG_SETMASK");
    printf("сигнал SIGQUIT разблокирован\n");

    sleep(5); /* здесь SIGQUIT завершит приложение с созданием файла core */
    exit(0);
}

static void
sig_quit(int signo)
{
    printf("перехвачен сигнал SIGQUIT\n");
    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
```

```
        err_sys("невозможно переустановить диспозицию сигнала SIGQUIT");
}
```

Процесс блокирует сигнал SIGQUIT, предварительно сохранив текущую маску сигналов для последующего восстановления, и затем приостанавливается на 5 секунд. Любой сигнал SIGQUIT, сгенерированный в этот промежуток времени, будет заблокирован и не будет доставлен процессу до тех пор, пока не окажется разблокированным. Перед последней 5-секундной приостановкой проверяется наличие ожидающего обработки сигнала SIGQUIT, после чего блокировка сигнала снимается.

Обратите внимание: сначала мы сохранили маску сигналов, а затем заблокировали сигнал. Чтобы разблокировать его, мы воспользовались операцией SIG_SETMASK, с помощью которой восстановили прежнее значение маски сигналов. Как альтернативный вариант, можно было бы использовать для разблокирования сигнала операцию SIG_UNBLOCK. Однако необходимо понимать, что если мы пишем функцию, которая может быть использована в других приложениях, и в этой функции мы должны заблокировать некоторый сигнал, то мы не можем использовать операцию SIG_UNBLOCK для разблокирования сигнала. В таких случаях следует использовать операцию SET_SIGMASK для восстановления первоначального значения маски, потому что возможно, что вызывающая программа перед обращением к функции уже заблокировала этот сигнал. Мы увидим это на примере функции system в разделе 10.18.

Если сигнал SIGQUIT будет сгенерирован во время этой приостановки, то ожидающий обработки сигнал окажется разблокированным и будет доставлен процессу перед возвратом из функции sigprocmask. Мы обнаружим это, так как вызов функции printf в обработчике сигнала, произойдет раньше, чем вызов функции printf, следующий за вызовом sigprocmask.

После этого процесс приостановится еще на 5 секунд. Если сигнал SIGQUIT будет сгенерирован в течение этого периода, то он завершит процесс, поскольку в момент перехвата сигнала мы переустановили его диспозицию в значение по умолчанию. В следующем ниже выводе символы ^\ показаны там, где мы нажимали комбинацию клавиш Control-\ – терминальный символ завершения процесса.

```
$ ./a.out
Б\                                  сгенерировать сигнал SIGQUIT
                                    (до завершения 5-секундной задержки)
сигнал SIGQUIT ожидает обработки    по окончании задержки
перехвачен сигнал SIGQUIT           в обработчике сигнала
сигнал SIGQUIT разблокирован        после выхода из sigprocmask
Б\Quit(coredump)                    повторная генерация сигнала SIGQUIT
$ ./a.out
Б\Б\Б\Б\Б\Б\Б\Б\Б\Б\                сгенерировать сигнал SIGQUIT 10 раз
                                    (до завершения 5-секундной задержки)
сигнал SIGQUIT ожидает обработки
перехвачен сигнал SIGQUIT           был доставлен только один сигнал
сигнал SIGQUIT разблокирован
Б\Quit(coredump)                    повторная генерация сигнала SIGQUIT
```

Сообщение Quit(coredump) выводится командной оболочкой, когда она обнаруживает аварийное завершение дочернего процесса. Обратите внимание, что, запустив программу повторно, мы десять раз сгенерировали сигнал SIGQUIT, а когда разблокировали его, процессу был доставлен только один сигнал. Это говорит о том, что в данной системе сигналы не помещаются в очередь.

10.14. Функция sigaction

Функция sigaction позволяет проверить действие, связанное с определенным сигналом, изменить его или выполнить обе эти операции. Эта функция служит заменой функции signal из ранних версий UNIX. В конце этого раздела мы продемонстрируем реализацию функции signal на основе sigaction.

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Через аргумент *signo* передается номер сигнала, диспозицию которого мы желаем получить или изменить. Если в аргументе *act* передается непустой указатель, то диспозиция сигнала изменяется. Если в аргументе *oact* передается непустой указатель, функция возвращает предыдущее значение диспозиции сигнала, помещая его по указанному в *oact* адресу. Эта функция использует следующую структуру:

```
struct sigaction {
    void     (*sa_handler)(int);      /* адрес функции-обработчика сигнала, */
                                      /* или SIG_IGN, или SIG_DFL */
    sigset_t sa_mask;                 /* дополнительные блокируемые сигналы */
    int      sa_flags;                /* флаги, табл.10.5 */

    /* альтернативный обработчик сигнала */
    void     (*sa_sigaction)(int, siginfo_t *, void *);
};
```

Если при изменении диспозиции сигнала поле sa_handler содержит адрес функции-обработчика (а не константы SIG_IGN или SIG_DEL), то поле sa_mask определяет набор сигналов, которые будут добавлены к маске сигналов процесса перед вызовом функции-обработчика. Перед возвратом из обработчика сигнала маска сигналов будет автоматически восстановлена в прежнее состояние. Таким образом мы получаем возможность блокировать определенные сигналы на время работы функции-обработчика. Перед доставкой сигнала, когда вызывается функция-обработчик, сам сигнал также включается в маску сигналов, таким образом блокируется доставка того же самого сигнала на время выполнения обработчика. В разделе 10.8 мы уже говорили, что обычно заблокированные сигналы не помещаются в очередь. Если сигнал был сгенерирован пять раз за период времени, когда он был заблокирован, то функция-обработчик обычно вызывается всего один раз.

После установки диспозиции сигнала она остается неизменной до тех пор, пока мы явно не изменим ее вызовом функции sigaction. В отличие от ранних версий UNIX с их ненадежными сигналами, стандарт POSIX.1 требует, чтобы действие сигнала оставалось неизменным до тех пор, пока явно не будет изменено программой.

Поле sa_flags структуры *act* определяет различные параметры обработки этого сигнала. В табл. 10.5 приводится подробное описание всех возможных флагов. Если в колонке SUS стоит точка, соответствующий флаг определен как часть базовых спецификаций стандарта POSIX.1. Если в этой колонке стоит аббревиатура XSI, то флаг определен как расширение XSI.

Таблица 10.5. Флаги (sa_flags) обработки каждого сигнала

Флаг	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Описание
SA_INTERRUPT			•			Системный вызов, прерываемый сигналом, не должен перезапускаться автоматически (в XSI – по умолчанию для sigaction). Дополнительная информация в разделе 10.5.
SA_NOCLDSTOP	•	•	•	•	•	Для сигнала SIGCHLD – не генерировать этот сигнал при приостановке дочернего процесса. Разумеется, при завершении дочернего процесса этот сигнал все равно будет сгенерирован (но обратите внимание на флаг SA_NOCLDWAIT ниже). Если установлен этот флаг, сигнал SIGCHLD также не будет генерироваться и при возобновлении работы дочернего процесса после приостановки.
SA_NOCLDWAIT	XSI	•	•	•	•	Для сигнала SIGCHLD – предотвращает создание процессов-зомби по завершении дочерних процессов. Если родительский процесс впоследствии вызовет функцию wait, то он окажется заблокированным до тех пор, пока последний дочерний процесс не завершится, после чего wait вернет значение −1 и код ошибки ECHLD в переменной errno (раздел 10.7).
SA_NODEFER	XSI	•	•	•	•	Не блокировать сигнал автоматически при вызове функции-обработчика (если, конечно, сигнал не включен в маску sa_mask).

Флаг	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Описание
						Заметьте: такое поведение соответствует поведению ненадежных сигналов в ранних версиях UNIX.
SA_ONSTACK	XSI	•	•	•	•	Доставлять сигнал на альтернативном стеке, если таковой был объявлен обращением к функции sigaltstack(2).
SA_RESETHAND	XSI	•	•	•	•	На входе в функцию-обработчик установить диспозицию сигнала в значение SIG_DFL и сбросить флаг SA_SIGINFO. Обратите внимание: такое поведение соответствует поведению ненадежных сигналов в ранних версиях UNIX. Однако диспозиция сигналов SIGILL и SIGTRAP не может быть переустановлена автоматически. При наличии этого флага функция sigaction ведет себя так, как если бы был установлен флаг SA_NODEFER.
SA_RESTART	XSI	•	•	•	•	Производить автоматический перезапуск системных вызовов, прерванных данным сигналом (раздел 10.5).
SA_SIGINFO	•	•	•	•	•	По этому флагу в обработчик сигнала передается дополнительная информация: указатели на структуру siginfo и контекст процесса.

Поле sa_sigaction представляет альтернативную функцию обработки сигнала при использовании флага SA_SIGINFO. Реализации могут использовать для хранения указателей из полей sa_handler и sa_sigaction одну и ту же область памяти, поэтому приложения должны заполнять только одно них.

Обычно прототип функции-обработчика сигнала выглядит так:

```
void handler(int signo);
```

Однако если установлен флаг SA_SIGINFO, то прототип функции-обработчика выглядит иначе:

```
void handler(int signo, siginfo_t *info, void *context);
```

Структура siginfo_t хранит информацию о причинах появления сигнала. Пример определения структуры приводится ниже. Все POSIX-совместимые реализации должны включать в эту структуру как минимум поля si_signo и si_code. Дополнительно XSI-совместимые реализации должны включать в состав структуры следующие поля:

```
struct siginfo {
    int    si_signo;   /* номер сигнала */
    int    si_errno;   /* если не 0, то значение errno из <errno.h> */
    int    si_code;    /* дополнительная информация (зависит от сигнала) */
    pid_t  si_pid;     /* идентификатор процесса-отправителя */
    uid_t  si_uid;  /* реальный идентификатор пользователя процесса-отправителя */
    void  *si_addr;    /* адрес, где возникла ошибка */
    int    si_status;  /* код завершения или номер сигнала */
    long   si_band;    /* направление и приоритет события SIGPOLL */
    /* далее могут быть определены дополнительные поля */
};
```

В табл. 10.6 перечислены значения поля `si_code` для различных сигналов, определяемых стандартом Single UNIX Specification. Обратите внимание, что реализации могут определять дополнительные значения.

Таблица 10.6. Значения кодов в структуре siginfo_t

Сигнал	Код	Значение
	ILL_ILLOPC	Недопустимая инструкция
	ILL_ILLOPN	Недопустимый операнд
	ILL_ILLADR	Недопустимый режим адресации
	ILL_ILLTRP	Некорректная ловушка
SIGILL	ILL_PRVOPC	Привилегированная операция
	ILL_PRVREG	Привилегированный регистр
	ILL_COPROC	Ошибка сопроцессора
	ILL_BADSTK	Внутренняя ошибка стека
	FPE_INTDIV	Деление на ноль при работе с целыми числами
	FPE_INTOVF	Переполнение при работе с целыми числами
	FPE_FLTDIV	Деление на ноль при работе с числами с плавающей точкой
	FPE_FLTOVF	Переполнение при работе с числами с плавающей точкой
SIGFPE	FPE_FLTUND	Нехватка значащих разрядов при работе с числами с плавающей точкой
	FPE_FLTRES	Потеря точности при работе с числами с плавающей точкой
	FPE_FLTINV	Неверная операция при работе с числами с плавающей точкой
	FPE_FLTSUB	Выход индекса за границы диапазона
SIGSEGV	SEGV_MAPPER	Адрес не соответствует объекту
	SEGV_ACCERR	Недостаточно прав для доступа к объекту
	BUS_ADRALN	Неправильное выравнивание адреса
SIGBUS	BUS_ADRERR	Несуществующий физический адрес
	BUS_OBJERR	Аппаратная ошибка, специфичная для объекта
SIGTRAP	TRAP_BRKPT	Точка останова процесса

Сигнал	Код	Значение
	TRAP_TRACE	Ловушка трассировки процесса
SIGCHLD	CLD_EXITED	Дочерний процесс завершился
	CLD_KILLED	Дочерний процесс завершился аварийно (без файла core)
	CLD_DUMPED	Дочерний процесс завершился с созданием файла core
	CLD_TRAPPED	Сработала ловушка в отлаживаемом дочернем процессе
	CLD_STOPPED	Дочерний процесс приостановлен
	CLD_CONTINUED	Приостановленный дочерний процесс продолжил работу
SIGPOLL	POLL_IN	Входные данные доступны для чтения
	POLL_OUT	Можно записать выходные данные
	POLL_MSG	Доступно входящее сообщение
	POLL_ERR	Ошибка ввода-вывода
	POLL_PRI	Доступно высокоприоритетное входящее сообщение
	POLL_HUP	Устройство отсоединено
Любой сигнал	SI_USER	Сигнал послан функцией kill
	SI_QUEUE	Сигнал послан функцией sigqueue (расширение реального времени)
	SI_TIMER	Истекло время таймера, установленного функцией timer_settime (расширение реального времени)
	SI_ASYNCIO	Завершено выполнение запрошенной асинхронной операции ввода-вывода (расширение реального времени)
	SI_MESGQ	В очередь сообщений поступило новое сообщение (расширение реального времени)

Для сигнала SIGCHLD устанавливаются значения полей si_pid, si_status и si_uid. Для сигналов SIGILL и SIGSEGV в поле si_addr заносится адрес, в котором была обнаружена ошибка, хотя адрес может быть неточным. Для сигнала SIGPOLL поле si_band будет содержать направление и полосу приоритета для сообщений STREAMS, которые генерируют события POLL_IN, POLL_OUT и POLL_MSG (полное описание полос приоритетов вы найдете в [Rago 1993]). Поле si_errno содержит код ошибки, который соответствует ситуации, вызвавшей появление сигнала, хотя это во многом зависит от реализации.

Аргумент *context*, передаваемый обработчику сигнала, представляет собой нетипизированный указатель, который может быть приведен к типу struct ucontext_t, идентифицирующему контекст процесса в момент доставки сигнала.

> Если реализация поддерживает расширения сигналов реального времени, установка обработчика сигнала с флагом SA_SIGINFO гарантирует, что сигналы будут ставиться в очередь. Для приложений реального времени зарезервирован отдельный диапазон сигналов. Через структуру siginfo можно передавать данные приложения при условии, что сигнал будет генерироваться функцией sigqueue. Мы не будем здесь обсуждать расширения реального времени – см. [Gallmeister 1995].

Пример – функция signal

Теперь перейдем к реализации функции signal на основе функции sigaction. Такую реализацию предусматривают многие платформы («POSIX.1 Rationale» утверждает, что это и было замыслом стандарта POSIX). С другой стороны, операционные системы с ограниченной совместимостью на уровне двоичных кодов могут предоставлять функцию signal, поддерживающую устаревшую семантику ненадежных сигналов. Если вам не требуется поддержка устаревшей семантики (например, для сохранения обратной совместимости), используйте приводимую ниже реализацию функции signal или непосредственно функцию sigaction. (Как вы уже наверняка догадались, чтобы вернуться к устаревшей семантике, нужно вызывать функцию sigaction с флагами SA_RESETHAND и SA_NODEFER.) Все примеры в этой книге, которые обращаются к функции signal, используют функцию, представленную в листинге 10.12.

Листинг 10.12. Реализация функции signal на основе функции sigaction

```
#include "apue.h"

/* Надежная версия функции signal() с использованием функции sigaction() */
/* стандарта POSIX. */

Sigfunc *
signal(int signo, Sigfunc *func)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;
#endif
    } else {
#ifdef SA_RESTART
        act.sa_flags |= SA_RESTART;
#endif
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Обратите внимание: для инициализации поля sa_mask мы должны использовать функцию sigemptyset. Нельзя гарантировать, что

```
act.sa_mask = 0;
```

сделает то же самое.

Мы преднамеренно пробуем установить флаг SA_RESTART для всех сигналов, кроме SIGALRM, что дает возможность автоматически перезапускать систем-

ные вызовы, прерванные другими сигналами. Сигнал SIGALRM исключен из этого списка по той причине, что нам понадобится задавать тайм-ауты для операций ввода-ввода (листинг 10.7).

В некоторых старых системах, таких как SunOS, определен флаг SA_INTERRUPT. В этих системах перезапуск прерванных системных вызовов производится по умолчанию, поэтому установка этого флага предотвратит автоматический перезапуск прерванных системных вызовов. Расширение XSI стандарта Single UNIX Specification оговаривает, что функция sigaction не должна перезапускать прерванные системные вызовы, если явно не указан флаг SA_RESTART.

Пример – функция signal_intr

В листинге 10.13 приводится версия функции signal, которая пытается предотвратить перезапуск любого прерванного системного вызова.

Листинг 10.13. Функция signal_intr

```
#include "apue.h"

Sigfunc *
signal_intr(int signo, Sigfunc *func)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
#ifdef SA_INTERRUPT
    act.sa_flags |= SA_INTERRUPT;
#endif
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Чтобы повысить переносимость функции, мы используем для предотвращения перезапуска прерванных системных вызовов флаг SA_INTERRUPT, если он определен в системе.

10.15. Функции sigsetjmp и siglongjmp

В разделе 7.10 мы говорили о функциях setjmp и longjmp, которые используются для выполнения дальних, или нелокальных, переходов. Функция longjmp достаточно часто используется в обработчиках сигналов, когда нужно вернуться в главный цикл программы, не выполняя возврат из обработчика. Мы продемонстрировали это в листингах 10.5 и 10.8.

Однако использование функции longjmp сопряжено с одной проблемой. Когда сигнал перехвачен, перед входом в функцию обработки производится автоматическое добавление текущего сигнала к маске сигналов процесса. Это

препятствует прерыванию обработчика этим же сигналом. Как вы думаете, что произойдет с маской сигналов, если выполнить дальний переход (longjmp) из функции-обработчика?

> В ОС FreeBSD 5.2.1 и Mac OS X 10.3 функции setjmp и longjmp сохраняют и восстанавливают маску сигналов. Однако Linux 2.4.22 и Solaris 9 этого не делают. ОС FreeBSD 5.2.1 и Mac OS X 10.3 предоставляют функции _setjmp и _longjmp, которые не сохраняют и не восстанавливают маску сигналов.

Стандарт POSIX.1 не оговаривает воздействие функций setjmp и longjmp на маску сигналов – вместо этого он определяет две новые функции, sigsetjmp и siglongjmp. Эти две функции всегда должны использоваться для выхода из обработчика сигнала.

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);

        Возвращает 0, если вызвана непосредственно, и ненулевое значение,
      если возврат произошел в результате обращения к функции siglongjmp

void siglongjmp(sigjmp_buf env, int val);
```

Единственное их отличие от функций setjmp и longjmp заключается в том, что функция sigsetjmp принимает один дополнительный аргумент. Если аргумент *savemask* содержит ненулевое значение, sigsetjmp сохраняет текущую маску сигналов процесса в буфере *env*. При вызове siglongjmp, если аргумент *env* был сохранен в результате вызова sigsetjmp с ненулевым значением *savemask*, маска сигналов восстанавливается из сохраненного значения.

Пример

Программа, представленная листингом 10.14, демонстрирует, как производится автоматическое включение сигнала в маску сигналов при вызове функции-обработчика. Здесь также показано использование функций sigsetjmp и siglongjmp.

Листинг 10.14. Пример работы с маской сигналов и функций sigsetjmp и siglongjmp

```
#include "apue.h"
#include <setjmp.h>
#include <time.h>

static void                     sig_usr1(int), sig_alrm(int);
static sigjmp_buf               jmpbuf;
static volatile sig_atomic_t    canjump;

int
main(void)
{
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGUSR1)");
```

```
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGALRM)");
    pr_mask("в начале функции main: ");     /* листинг 10.10 */

    if (sigsetjmp(jmpbuf, 1)) {
        pr_mask("в конце функции main: ");
        exit(0);
    }
    canjump = 1;                /* теперь можно выполнять переход */

    for ( ; ; )
        pause();
}

static void
sig_usr1(int signo)
{
    time_t starttime;

    if (canjump == 0)
        return;                 /* получен неожиданный сигнал, игнорировать */

    pr_mask("в начале функции sig_usr1: ");
    alarm(3);                   /* запланировать SIGALRM через 3 секунды */
    starttime = time(NULL);
    for ( ; ; )                 /* ждать 5 секунд */
        if (time(NULL) > starttime + 5)
            break;
        pr_mask("в конце функции sig_usr1: ");

    canjump = 0;
    siglongjmp(jmpbuf, 1);      /* переход в функцию main - не возврат */
}

static void
sig_alrm(int signo)
{
    pr_mask("в функции sig_alrm: ");
}
```

Эта программа демонстрирует методику, которая должна применяться всякий раз, когда планируется использовать функцию `siglongjmp` в обработчике сигнала. Здесь переменная `canjump` устанавливается в значение, отличное от нуля, только после вызова функции `sigsetjmp`. Значение этой переменной проверяется в обработчике сигнала, и функция `siglongjmp` вызывается только в том случае, когда значение переменной `canjump` отлично от нуля. Это предотвращает несвоевременный вызов обработчика сигнала, когда буфер перехода не будет подготовлен функцией `sigsetjmp`. (В этой достаточно простой программе все заканчивается практически сразу же после вызова `siglongjmp`, но в больших программах обработчик сигнала может оставаться установленным и после вызова `siglongjmp`.) Подобного рода защита обычно не требуется при использовании функции `longjmp` в обычных функциях языка C (в противоположность обработчикам сигналов). Однако, учитывая, что сигнал мо-

жет быть сгенерирован в *любой* момент времени, мы вынуждены предусматривать меры предосторожности в обработчике сигналов.

Здесь мы использовали тип данных sig_atomic_t, который определен стандартом ISO C для переменных, запись в которые не может быть прервана. Это означает, например, что переменные этого типа не должны пересекать границы страниц виртуальной памяти, и доступ к ним должен осуществляться единственной машинной инструкцией. Кроме того, мы всегда используем спецификатор volatile с этим типом данных, поскольку доступ к переменной возможен из двух различных потоков управления – из функции main и из обработчика асинхронного сигнала. На рис. 10.1 приводится временная диаграмма для этой программы.

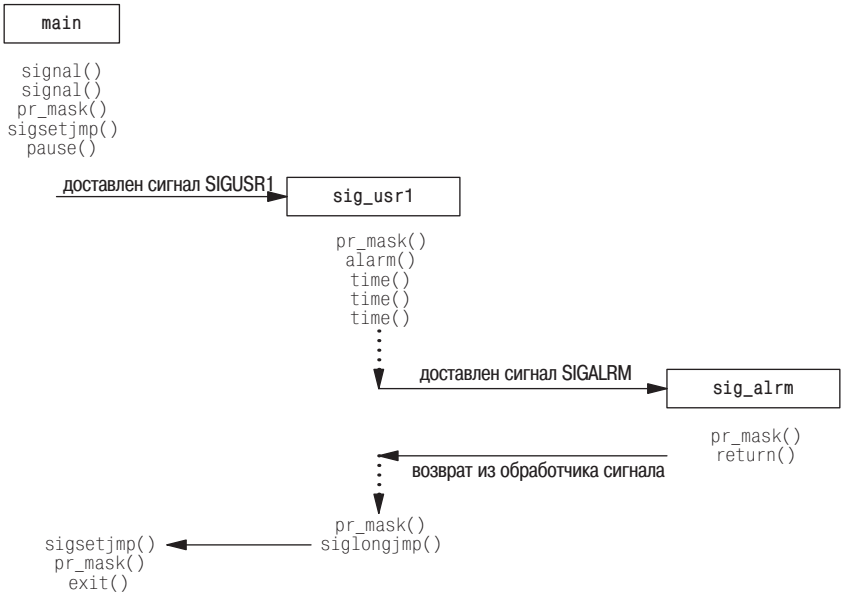


***Рис. 10.1.** Временная диаграмма программы, обрабатывающей два сигнала*

Рисунок 10.1 может быть разделен на три части: левая часть соответствует функции main, центральная часть – функции sig_usr1 и правая часть – функции sig_alrm. Пока выполнение процесса происходит в левой части, маска сигналов пуста (нет блокируемых сигналов). В центральной части в маске сигналов находится сигнал SIGUSR1. В правой части в маске сигналов находятся сигналы SIGUSR1 и SIGALRM.

А теперь посмотрим, что выведет программа из листинга 10.14 после запуска:

```
$ ./a.out &                 запуск процесса в фоновом режиме
в начале функции main:
[1]    531                  командная оболочка вывела идентификатор процесса
$ kill -USR1 531            посылаем процессу сигнал SIGUSR1
в начале функции sig_usr1: SIGUSR1
```

```
$ в функции sig_alrm: SIGUSR1 SIGALRM
в конце функции sig_usr1: SIGUSR1
в конце функции main:
                                        нажимаем ввод
[1]  +  Done                  ./a.out &
```

Как мы и ожидали, на входе в обработчик сигнала перехваченный сигнал добавляется к маске сигналов процесса. После выхода из обработчика маска сигналов восстанавливается. Кроме того, функция `siglongjmp` восстанавливает маску сигналов, которая была сохранена вызовом функции `sigsetjmp`.

Если в программе из листинга 10.14 заменить функции `sigsetjmp` и `siglongjmp` на `setjmp` и `longjmp` в Linux (или `_setjmp` и `_longjmp` в FreeBSD), то последняя строка, выведенная программой, будет

```
в конце функции main: SIGUSR1
```

Это означает, что после вызова `longjmp` функция `main` будет продолжать работу с заблокированным сигналом `SIGUSR1`, а это скорее всего не то, что нам нужно.

10.16. Функция sigsuspend

Мы рассмотрели порядок изменения маски сигналов процесса, с помощью которой можно заблокировать или разблокировать отдельные сигналы. Эту методику мы можем использовать для защиты критических участков программы, выполнение которых не должно прерываться сигналами. А если нам нужно разблокировать сигнал и затем с помощью функции `pause` дождаться его доставки? Если предположить, что ожидаемый сигнал – `SIGINT`, то неправильная реализация могла бы выглядеть так:

```
sigset_t newmask, oldmask;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

/* заблокировать SIGINT и сохранить текущую маску сигналов */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("ошибка выполнения операции SIG_BLOCK");

/* критический участок программы */

/* восстановить прежнюю маску сигналов, в которой SIGINT не заблокирован */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("ошибка выполнения операции SIG_SETMASK");

/* интервал времени, когда доставка сигнала станет проблемой */

pause(); /* дождаться доставки сигнала */

/* продолжить работу */
```

Если сигнал будет послан процессу в то время, когда он еще заблокирован, то доставка сигнала будет отложена до тех пор, пока процесс не разблокирует его. С точки зрения приложения это выглядит так, как будто сигнал был сге-

нерирован между операцией разблокирования и вызовом функции pause (в зависимости от того, как реализован механизм сигналов в ядре). Если все происходит именно так или если сигнал действительно будет доставлен процессу в промежутке времени между моментом снятия блокировки и вызовом функции pause, то возникнут определенные сложности. Сигнал, доставленный в этот промежуток времени, приведет к тому, что функция pause может заблокировать процесс «навечно», если сигнал не будет сгенерирован еще хотя бы раз. Это еще одна проблема, связанная с ранними ненадежными сигналами.

Чтобы преодолеть ее, нам необходим способ, с помощью которого можно было бы производить восстановление маски сигналов и приостанавливать процесс атомарно. Такую возможность предоставляет функция sigsuspend.

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Возвращает −1 и записывает в переменную errno код ошибки EINTR

Маска сигналов, передаваемая в аргументе *sigmask*, переносится в маску сигналов процесса. Затем процесс приостанавливается до тех пор, пока не будет перехвачен сигнал или пока какой-либо сигнал не завершит процесс. В случае, если сигнал был перехвачен и функция-обработчик вернула управление, функция sigsuspend также возвращает управление вызывающему процессу и при этом восстанавливает маску сигналов процесса в состояние, которое предшествовало ее вызову.

Обратите внимание: эта функция всегда возвращает признак ошибки с кодом EINTR в переменной errno (который говорит о том, что выполнение системного вызова было прервано сигналом).

Пример

Листинг 10.15 демонстрирует корректный способ защиты критического участка программы от конкретного сигнала.

Листинг 10.15. Защита критического участка программы от сигнала

```
#include "apue.h"

static void sig_int(int);

int
main(void)
{
    sigset_t newmask, oldmask, waitmask;

    pr_mask("в начале программы: ");
    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGINT)");
    sigemptyset(&waitmask);
    sigaddset(&waitmask, SIGUSR1);
    sigemptyset(&newmask);
```

```
    sigaddset(&newmask, SIGINT);

    /*
     * Заблокировать SIGINT и сохранить текущую маску сигналов.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("ошибка выполнения операции SIG_BLOCK");

    /*
     * Критический участок программы.
     */
    pr_mask("внутри критического участка: ");

    /*
     * Промежуток времени, когда может быть доставлен любой сигнал,
     * кроме SIGUSR1.
     */
    if (sigsuspend(&waitmask) != -1)
        err_sys("ошибка вызова функции sigsuspend");

    pr_mask("после выхода из sigsuspend: ");

    /*
     * Восстановить прежнюю маску сигналов, которая разблокирует SIGINT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка выполнения операции SIG_SETMASK");

    /*
     * И продолжить работу ...
     */
    pr_mask("в конце программы: ");

    exit(0);
}

static void
sig_int(int signo)
{
    pr_mask("\nв функции sig_int: ");
}
```

Обратите внимание: когда функция `sigsuspend` **возвращает управление, она восстанавливает маску сигналов в состояние, предшествовавшее ее вызову. В данном примере к моменту вызова этой функции сигнал** `SIGINT` **был заблокирован. Поэтому мы восстанавливаем маску сигналов, записывая туда значение, сохраненное ранее (**`oldmask`**).**

В результате запуска программы из листинга 10.15 мы получили:

```
$ ./a.out
в начале программы:
внутри критического участка: SIGINT
^?                                          ввод символа прерывания
в функции sig_int: SIGUSR1
```

```
после выхода из sigsuspend: SIGINT
в конце программы:
```

Перед обращением к функции sigsuspend к существующей маске сигналов мы добавили сигнал SIGUSR1. Затем, внутри обработчика сигнала, маска изменилась. Далее видно, что когда sigsuspend возвращала управление, она восстановила маску сигналов.

Пример

Функция sigsuspend также используется для того, чтобы приостановить работу процесса, пока обработчик сигнала не установит глобальную переменную. В программе, представленной листингом 10.16, мы перехватываем два сигнала, SIGINT и SIGQUIT, но при этом продолжение работы возможно только после перехвата сигнала SIGQIUIT.

Листинг 10.16. Функция sigsuspend приостанавливает процесс до тех пор, пока не будет установлена глобальная переменная

```
#include "apue.h"

volatile sig_atomic_t quitflag; /* обработчик сигнала записывает сюда */
                                /* ненулевое значение */
static void
sig_int(int signo)              /* единый обработчик для SIGINT и SIGQUIT */
{
    if (signo == SIGINT)
        printf("\nпрерывание\n");
    else if (signo == SIGQUIT)
        quitflag = 1;           /* установить флаг для главного цикла */
}

int
main(void)
{
    sigset_t newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGINT)");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGQUIT)");

    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);

    /*
     * Заблокировать SIGQUIT и сохранить текущую маску сигналов.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("ошибка выполнения операции SIG_BLOCK");

    while (quitflag == 0)
        sigsuspend(&zeromask);
```

```
    /*
     * Сигнал SIGQUIT был перехвачен и к настоящему моменту опять заблокирован.
     */
    quitflag = 0;

    /*
     * Восстановить маску сигналов, в которой SIGQUIT разблокирован.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка выполнения операции SIG_SETMASK");

    exit(0);
}
```

Примерный результат работы программы:

```
$ ./a.out
Б?                  ввод символа прерывания
прерывание
Б?                  ввод символа прерывания еще раз
прерывание
Б?                  и еще раз
прерывание
Б?                  и еще раз
прерывание
Б?                  и еще раз
прерывание
Б?                  и еще раз
прерывание
Б?                  и еще раз
прерывание
Б\ $                а теперь ввод символа завершения
```

Для сохранения переносимости между POSIX.1-совместимыми системами и системами, которые не совместимы со стандартом POSIX, но поддерживают стандарт ISO C, необходимо только одно действие: внутри обработчика сигнала нужно присваивать некоторое значение переменной типа `sig_atomic_t`. Стандарт POSIX.1 пошел дальше и определил список функций, которые можно безопасно вызывать из обработчика сигнала (табл. 10.3), но в этом случае программа, вероятно, не будет правильно работать в системах, которые не поддерживают стандарт POSIX.

Пример

В следующем примере мы продемонстрируем, как с помощью сигналов можно синхронизировать работу родительского и дочернего процессов. В листинге 10.17 представлена реализация пяти процедур – `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT` и `WAIT_CHILD` – из раздела 8.9.

Листинг 10.17. Процедуры для синхронизации родительского и дочернего процессов

```
#include "apue.h"

static volatile sig_atomic_t sigflag; /* устанавливается обработчиком */
```

```
                                        /* в ненулевое значение */
static sigset_t newmask, oldmask, zeromask;

static void
sig_usr(int signo) /* единый обработчик для сигналов SIGUSR1 и SIGUSR2 */
{
    sigflag = 1;
}

void
TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGUSR1)");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGUSR2)");

    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    /*
     * Заблокировать сигналы SIGUSR1 и SIGUSR2,
     * и сохранить текущую маску сигналов.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("ошибка выполнения операции SIG_BLOCK");
}

void
TELL_PARENT(pid_t pid)
{
    kill(pid, SIGUSR2); /* сообщить родительскому процессу, что мы готовы */
}

void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask); /* и дождаться ответа от родительского процесса */

    sigflag = 0;

    /*
     * Восстановить маску сигналов в начальное состояние.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка выполнения операции SIG_SETMASK");
}

void
TELL_CHILD(pid_t pid)
{
    kill(pid, SIGUSR1); /* сообщить дочернему процессу, что мы готовы */
```

```
}

void
WAIT_CHILD(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask); /* и дождаться ответа от дочернего процесса */

    sigflag = 0;

    /*
     * Восстановить маску сигналов в начальное состояние.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка выполнения операции SIG_SETMASK");
}
```

В этом примере мы использовали сигналы, определяемые пользователем: сигнал SIGUSR1 передается от родительского процесса дочернему, а SIGUSR2 – от дочернего процесса родительскому. В листинге 15.3 мы покажем другую реализацию этих пяти функций с использованием неименованных каналов.

Функция sigsuspend прекрасно подходит для случая, когда процесс должен приостановить работу до тех пор, пока ему не будет доставлен сигнал (как это было в двух предыдущих примерах), но что если нам необходимо во время ожидания сигнала обращаться к другим системным функциям? К сожалению, эта проблема не имеет достаточно надежного решения, за исключением выполнения приложения в нескольких потоках, из которых один выделяется специально для обработки сигналов, о чем мы будем говорить в разделе 12.8.

Если отказаться от многопоточной модели, то лучшее, что можно предложить – это записывать определенные значения в глобальные переменные во время обработки сигналов. Например, если мы выполняем перехват сигналов SIGINT и SIGALRM и устанавливаем обработчики сигналов с помощью функции signal_intr, то доставка сигналов будет прерывать любые медленные системные вызовы. Чаще всего эти сигналы будут доставляться во время работы системного вызова select (раздел 14.5.1), который ожидает окончания операции ввода с медленного устройства. (Особенно это относится к сигналу SIGALRM, который используется для прерывания затянувшихся операций ввода-вывода.) Код, который обслуживает подобную ситуацию, мог бы выглядеть следующим образом:

```
if (intr_flag) /* флаг устанавливается обработчиком сигнала SIGINT */
    handle_intr();
if (alrm_flag) /* флаг устанавливается обработчиком сигнала SIGALRM */
    handle_alrm();

/* сигналы, появившиеся в этот момент времени, будут утеряны */

while (select( ... ) < 0) {
    if (errno == EINTR) {
        if (alrm_flag)
            handle_alrm();
        else if (intr_flag)
```

```
            handle_intr();
        } else {
            /* обработка других ошибок */
        }
    }
```

Мы проверяем значения каждой из глобальных переменных перед вызовом функции select и всякий раз после того, как она возвращает ошибку прерывания системного вызова. Проблема возникает, когда происходит перехват сигнала между первыми двумя условными операторами и последующим вызовом функции select. Сигналы, доставленные в этом промежутке времени, будут утеряны, что отмечено в комментарии. Обработчики сигналов, разумеется, будут вызваны, и они установят соответствующие глобальные переменные, но select никогда не вернет управление (если, конечно, какие-либо данные не будут готовы для чтения).

Нам требуется выполнить следующую последовательность действий:

1. Заблокировать сигналы SIGINT и SIGALRM.
2. Проверить значения глобальных переменных, чтобы убедиться, не был ли доставлен какой-либо сигнал, и при необходимости выполнить соответствующие действия.
3. Вызвать select (или любой другой системный вызов, например read) и разблокировать оба сигнала атомарно.

Функция sigsuspend может помочь, только если на шаге 3 используется операция pause.

10.17. Функция abort

Ранее уже упоминалось, что вызов функции abort приводит к аварийному завершению процесса.

```
#include <stdlib.h>

void abort(void);
```

Эта функция никогда не возвращает управление

Эта функция передает сигнал SIGABRT вызывающему процессу. (Процессы не должны игнорировать этот сигнал.) Стандарт ISO C определяет, что функция abort должна извещать операционную систему об аварийном завершении с помощью функции raise(SIGABRT).

Стандарт ISO C требует, чтобы функция abort никогда не возвращала управление, даже в том случае, когда приложение перехватывает сигнал SIGABRT. Если сигнал перехватывается, то единственный способ для обработчика не вернуть управление в вызывающий процесс – это вызвать одну из функций exit, _exit, _Exit, longjmp или siglongjmp. (Различия между функциями longjmp и siglongjmp обсуждались в разделе 10.15.) Помимо этого, стандарт POSIX.1

указывает, что функция abort должна выполняться даже в том случае, если процесс заблокировал или игнорирует сигнал SIGABRT.

Процессу позволено перехватывать сигнал SIGABRT, чтобы он мог выполнить необходимые действия перед завершением. Если процесс не завершается из обработчика сигнала, то стандарт POSIX.1 указывает, что функция abort должна завершить процесс, когда обработчик сигнала вернет управление.

Стандарт ISO C оставляет на усмотрение реализации решение вопроса о сбросе буферов ввода-вывода и удалении временных файлов (раздел 5.13). Стандарт POSIX.1 пошел гораздо дальше и требует, чтобы функция abort, если она завершает процесс, воздействовала на открытые потоки ввода-вывода так же, как функция fclose.

> В ранних версиях System V функция abort генерировала сигнал SIGIOT. Кроме того, допускалась возможность игнорировать сигнал или перехватывать его. В последнем случае, если обработчик возвращал управление обычным образом, то и функция abort возвращала управление вызывающему процессу.
>
> В 4.3BSD генерировался сигнал SIGILL, но перед этим функция abort снимала блокировку с сигнала и сбрасывала его диспозицию в значение SIG_DFL (завершение с созданием файла core). Это не позволяло процессам игнорировать сигнал или перехватывать его.
>
> Традиционно различные реализации функции abort по-разному обслуживали потоки ввода-вывода. Если необходимо, чтобы перед аварийным завершением все потоки ввода-вывода были сброшены должным образом, это нужно сделать перед вызовом функции abort. Именно так делает наша функция err_dump (приложение B), и это повышает надежность и переносимость программ.
>
> Поскольку в большинстве реализаций UNIX функция tmpfile сразу же вызывает unlink, то проблема удаления временных файлов, о которой предупреждает стандарт ISO C, отпадает сама собой.

Пример

В листинге 10.18 приводится реализация функции abort, соответствующая требованиям стандарта POSIX.1.

Листинг 10.18. Реализация функции abort, соответствующая требованиям стандарта POSIX.1

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
abort(void) /* функция abort() в стиле POSIX */
{
    sigset_t mask;
    struct sigaction action;

    /*
     * Вызывающий процесс не может игнорировать  SIGABRT,
     * иначе - сбросить диспозицию сигнала в значение по умолчанию.
```

```
         */
        sigaction(SIGABRT, NULL, &action);
        if (action.sa_handler == SIG_IGN) {
            action.sa_handler = SIG_DFL;
            sigaction(SIGABRT, &action, NULL);
        }
        if (action.sa_handler == SIG_DFL)
            fflush(NULL); /* сбросить все буферы потоков ввода-вывода */

        /*
         * Вызывающий процесс не может заблокировать SIGABRT;
         * убедитесь, что он не заблокирован.
         */
        sigfillset(&mask);
        sigdelset(&mask, SIGABRT); /* в маске разблокирован только SIGABRT */
        sigprocmask(SIG_SETMASK, &mask, NULL);
        kill(getpid(), SIGABRT);               /* послать сигнал */

        /*
         * Если мы вернулись сюда, значит, процесс обработал SIGABRT
         * и вернул управление.
         */
        fflush(NULL);                          /* сбросить все буферы */
        action.sa_handler = SIG_DFL;
        sigaction(SIGABRT, &action, NULL);     /* установить диспозицию сигнала */
                                               /* в значение по умолчанию*/
        sigprocmask(SIG_SETMASK, &mask, NULL); /* на всякий случай ... */
        kill(getpid(), SIGABRT);               /* и еще раз */
        exit(1);                       /* этот вызов никогда не будет выполнен ... */
    }
```

Прежде всего мы проверяем, будет ли выполнено действие по умолчанию для сигнала – если это так, то мы сбрасываем все буферы стандартных потоков ввода-вывода. Это не равносильно вызову функции `fclose` (поскольку мы лишь сбрасываем буферы, а не закрываем потоки), но система сама закроет все открытые файлы, когда процесс завершится. Если процесс перехватил сигнал и вернул управление, мы опять сбрасываем все буферы ввода-вывода, поскольку процесс мог выводить некоторые данные в обработчике сигнала. Единственное, что мы не сможем обработать, – это вызов функции `_exit` или `_Exit` из обработчика. В этом случае все данные, оставшиеся в буферах ввода-вывода, будут утеряны. Но мы будем исходить из предположения, что вызывающий процесс просто не пожелал сбрасывать содержимое буферов.

В разделе 10.9 мы говорили, что если в результате вызова функции `kill` будет сгенерирован сигнал для вызывающего процесса и этот сигнал не заблокирован (что гарантирует функция из листинга 10.18), то этот сигнал (или любой другой незаблокированный сигнал, ожидающий обработки) будет доставлен процессу еще до выхода из функции `kill`. В данном случае мы блокируем доставку всех сигналов, за исключением `SIGABRT`, поэтому мы наверняка знаем, что если вызов `kill` вернул управление, значит, сигнал был перехвачен и обработан процессом.

10.18. Функция system

В разделе 8.13 мы приводили пример реализации функции system. Однако эта версия не обрабатывала сигналы. Стандарт POSIX.1 требует, чтобы функция system игнорировала сигналы SIGINT и SIGQUIT и блокировала сигнал SIGCHLD. Прежде чем продемонстрировать версию, которая обрабатывает сигналы, мы расскажем, почему это необходимо.

Пример

Программа, представленная листингом 10.19, использует версию функции system из раздела 8.13 для вызова редактора ed(1). (Этот редактор уже давно входит в состав операционных систем UNIX. Мы использовали его по той причине, что он перехватывает и обрабатывает сигналы SIGINT и SIGQUIT. Если запустить редактор ed из командной оболочки и ввести символ прерывания, то он перехватит его и выведет символ «?». Кроме того, программа ed устанавливает диспозицию сигнала SIGQUIT в значение SIG_IGN.) Программа из листинга 10.19 перехватывает сигналы SIGINT и SIGCHLD. Запустив ее, мы получим следующее:

```
$ ./a.out
a                   включить режим добавления текста в буфер редактора
Это одна строка текста
.                   точка на отдельной строке выключает режим добавления
1,$p                вывести строки из буфера с первой по последнюю,
                    чтобы увидеть его содержимое
Это одна строка текста
w temp.foo          записать буфер в файл
23                  редактор сообщает, что записано 23 байта
q                   выйти из редактора
перехвачен сигнал SIGCHLD
```

Когда редактор завершает работу, система посылает родительскому процессу (a.out) сигнал SIGCHLD. Мы перехватываем его и возвращаем управление из обработчика сигнала. Родительский процесс должен это делать, если желает знать, когда завершился дочерний процесс. Доставка этого сигнала родительскому процессу должна быть заблокирована на время работы функции system, как того и требует стандарт POSIX.1. Иначе процесс, запустивший функцию system, будет думать, что завершился один из его собственных дочерних процессов. После получения сигнала вызывающий процесс должен обратиться к одной из функций семейства wait, чтобы получить код завершения дочернего процесса.

Листинг 10.19. Вызов редактора ed с помощью функции system

```
#include "apue.h"

static void
sig_int(int signo)
{
    printf("перехвачен сигнал SIGINT\n");
```

```
}

static void
sig_chld(int signo)
{
    printf("перехвачен сигнал SIGCHLD\n");
}

int
main(void)
{
    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGINT)");
    if (signal(SIGCHLD, sig_chld) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGCHLD)");
    if (system("/bin/ed") < 0)
        err_sys("ошибка вызова функции system()");
    exit(0);
}
```

Если мы запустим программу еще раз и отправим ей сигнал SIGINT, то получим следующий результат:

```
$ ./a.out
a                          включить режим добавления текста
                           в буфер редактора
привет, мир
.                          точка на отдельной строке выключает режим
добавления
1,$p                       вывести строки из буфера с первой по последнюю,
                           чтобы увидеть его содержимое
привет, мир
w temp.foo                 записать содержимое буфера в файл
12                         редактор сообщает, что записано 12 байт
Б?                         ввод символа прерывания
?                          редактор перехватил сигнал и вывел знак вопроса
перехвачен сигнал SIGINT   и то же самое сделал родительский процесс
q                          выход из редактора
перехвачен сигнал SIGCHLD
```

В разделе 9.6 мы уже говорили, что ввод символа прерывания приводит к передаче сигнала SIGINT всем процессам из группы процессов переднего плана. На рис. 10.2 показана схема процессов после запуска редактора.

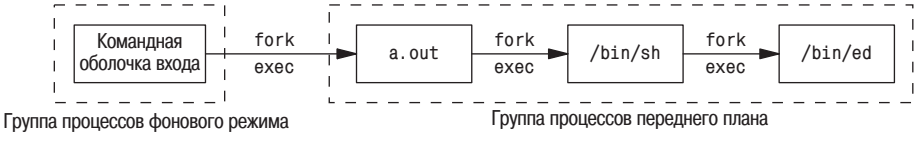


***Рис. 10.2.** Группы процессов переднего плана и фонового режима*

В данном примере сигнал SIGINT посылается трем процессам переднего плана. (Командная оболочка игнорирует его.) Как видно из вывода программы, оба процесса, a.out и редактор, перехватывают сигнал. Но когда мы запускаем программу с помощью функции system, у нас не должно получаться так, что и родительский, и дочерний процессы перехватывают сгенерированные терминалом сигналы SIGINT и SIGQUIT. В действительности эти сигналы должны посылаться только запущенной программе – дочернему процессу. Программа, запускаемая функцией system, может быть интерактивной (как программа ed в этом примере), а процесс, вызывающий функцию system, отдает управление другой программе, ожидая ее завершения, поэтому он не должен принимать эти два сигнала, генерируемые терминалом. По этой причине стандарт POSIX.1 требует, чтобы функция system игнорировала эти два сигнала, пока она ожидает завершения команды.

Пример

В листинге 10.20 приводится реализация функции system, которая предусматривает необходимую обработку сигналов.

Листинг 10.20. Корректная реализация функции system, соответствующая стандарту POSIX.1

```
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>

int
system(const char *cmdstring) /* предусматривает обработку сигналов */
{
    pid_t pid;
    int status;
    struct sigaction ignore, saveintr, savequit;
    sigset_t chldmask, savemask;

    if (cmdstring == NULL)
        return(1); /* UNIX всегда поддерживает командный процессор */

    ignore.sa_handler = SIG_IGN; /* игнорировать SIGINT и SIGQUIT */
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, &saveintr) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
        return(-1);
    sigemptyset(&chldmask);      /* заблокировать SIGCHLD */
    sigaddset(&chldmask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0)
        return(-1);

    if ((pid = fork()) < 0) {
        status = -1;            /* вероятно, превышено максимальное */
```

```
                                /* количество процессов */
    } else if (pid == 0) {      /* дочерний процесс */
        /* восстановить предыдущие действия сигналов и сбросить маску */
        sigaction(SIGINT, &saveintr, NULL);
        sigaction(SIGQUIT, &savequit, NULL);
        sigprocmask(SIG_SETMASK, &savemask, NULL);
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);             /* ошибка вызова функции exec */
    } else {                    /* родительский процесс */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1;    /* получен код ошибки, отличный от EINTR */
                break;
            }
    }

    /* восстановить предыдущие действия сигналов и сбросить маску */
    if (sigaction(SIGINT, &saveintr, NULL) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &savequit, NULL) < 0)
        return(-1);
    if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
        return(-1);
    return(status);
}
```

Если собрать программу из листинга 10.19 с этой версией функции `system`, то работа программы претерпит следующие изменения.

1. **Вызывающий процесс не будет получать сигналы `SIGINT` и `SIGQUIT`, сгенерированные терминалом.**
2. **По завершении работы редактора сигнал `SIGCHLD` не будет послан вызывающему процессу. Вместо этого он блокируется, пока не будет разблокирован последним вызовом функции `sigprocmask` уже после того, как функция `waitpid` получит код завершения дочернего процесса.**

> Стандарт POSIX.1 указывает, что если функция `wait` или `waitpid` получает код завершения дочернего процесса, когда сигнал `SIGCHLD` находится в ожидании обработки, то сигнал `SIGCHLD` не должен доставляться процессу, если не существует неполученного кода завершения другого дочернего процесса. Ни одна из четырех обсуждаемых в этой книге платформ не реализует такой семантики. Вместо этого сигнал `SIGCHLD` остается ждать обработки после того, как функция `system` вызовет `waitpid`. Когда блокировка сигнала снимается, он доставляется вызывающему процессу. Вызвав функцию `wait` в функции `sig_chld` из листинга 10.19, мы получили бы признак ошибки с кодом `ECHILD`, поскольку код завершения дочернего процесса уже был получен функцией `system`.

Во многих устаревших руководствах в качестве примера, как можно игнорировать сигналы `SIGINT` и `SIGQUIT`, приводится следующий код:

```
if ((pid = fork()) < 0) {
    err_sys("ошибка вызова функции fork");
} else if (pid == 0) {
```

```
    /* дочерний процесс */
    execl(...);
    _exit(127);
}

/* родительский процесс */
old_intr = signal(SIGINT, SIG_IGN);
old_quit = signal(SIGQUIT, SIG_IGN);
waitpid(pid, &status, 0)
signal(SIGINT, old_intr);
signal(SIGQUIT, old_quit);
```

Проблема этого кода заключается в том, что нельзя заранее точно сказать, какой из процессов первым получит управление после вызова функции `fork` – родительский или дочерний. Если первым начнет работу дочерний процесс, то может получиться так, что сигнал будет сгенерирован еще до того, как родительский процесс получит возможность изменить его диспозицию. По этой причине в листинге 10.20 мы изменяем диспозиции сигналов еще до вызова `fork`.

Обратите внимание: диспозиции этих сигналов в дочернем процессе необходимо переустановить до вызова `execl`. Это позволит функции `execl` изменить их диспозиции на значения по умолчанию на основе диспозиций сигналов вызывающего процесса, как это было описано в разделе 8.10.

Возвращаемое значение функции system

Будьте осторожны с возвращаемым значением функции `system`. Это код завершения командной оболочки, который не всегда совпадает с кодом завершения самой команды. В листинге 8.13 мы видели ряд примеров, когда результаты оказывались вполне ожидаемыми: если выполнялась простая команда, такая как `date`, код завершения был равен 0. Команда `exit 44` дала код завершения – 44. А что случится, если команда во время выполнения получит сигнал?

Запустим программу из листинга 8.14 и попробуем посылать сигналы выполняемым командам:

```
$ tsys "sleep 30"
Б?нормальное завершение, код выхода = 130    мы нажали клавишу
                                             прерывания (Control-C)
$ tsys "sleep 30"
Б\sh: 946 Quit                               мы нажали клавишу завершения
нормальное завершение, код выхода = 131
```

Когда мы прервали команду `sleep` сигналом `SIGINT`, функция `pr_exit` (листинг 8.3) восприняла это как нормальное завершение. То же самое произошло, когда мы прервали команду `sleep` сигналом `SIGQUIT`. Дело в том, что командная оболочка Bourne shell имеет плохо документированную особенность – она возвращает 128 плюс номер сигнала, если работа команды была прервана сигналом. Это можно наблюдать и в интерактивном сеансе работы с командной оболочкой.

```
$ sh                              убедимся, что запущена Bourne shell
$ sh -c "sleep 30"
Б?                                нажали клавишу прерывания
$ echo $?                         вывести код завершения последней команды
130
$ sh -c "sleep 30"
Б\sh: 962 Quit - core dumped      нажали клавишу завершения
$ echo $?                         вывести код завершения последней команды
131
$ exit                            выйти из Bourne shell
```

В системе, где сигнал SIGINT имеет номер 2, а сигнал SIGQUIT – номер 3, мы получили коды завершения 130 и 131 соответственно.

Теперь сделаем то же самое, но на этот раз пошлем сигналы самой командной оболочке и посмотрим, что возвращает функция system:

```
$ tsys "sleep 30" &               на этот раз запустим в фоновом режиме
9257
$ ps -f                           посмотрим идентификаторы процессов
     UID   PID  PPID  TTY      TIME CMD
     sar  9260   949  pts/5    0:00 ps -f
     sar  9258  9257  pts/5    0:00 sh -c sleep 60
     sar   949   947  pts/5    0:01 /bin/sh
     sar  9257   949  pts/5    0:00 tsys sleep 60
     sar  9259  9258  pts/5    0:00 sleep 60
$ kill -KILL 9258                 завершим саму командную оболочку
аварийное завершение, номер сигнала = 9
```

Мы видим, что возвращаемое значение функции system свидетельствует об аварийном завершении только тогда, когда сама командная оболочка завершается аварийно. Если вызывать непосредственно fork, exec и wait, то код завершения дочернего процесса не будет соответствовать возвращаемому значению функции system.

10.19. Функция sleep

Мы уже пользовались функцией sleep во многих примерах и даже продемонстрировали две ее реализации в листингах 10.4 и 10.5; впрочем, у них есть определенные недостатки.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Возвращает 0 или количество секунд, оставшихся до окончания приостановки

Эта функция приостанавливает выполнение вызывающего процесса до тех пор, пока

1. Не истечет установленный интервал времени.

2. Не будет получен сигнал и обработчик сигнала не вернет управление.

В первом случае функция возвращает 0. Если выход из функции происходит раньше из-за того, что процессу был доставлен сигнал (второй случай), то возвращаемое значение содержит количество секунд, оставшихся до истечения запрошенного интервала времени (заданное количество секунд минус количество секунд, прошедших с момента вызова функции).

Как и в случае с функцией `alarm`, функция `sleep` может вернуть управление чуть позже, чем было запрошено, в зависимости от загруженности системы. Функция `sleep` может быть реализована на базе функции `alarm` (раздел 10.10), но это совсем не обязательно. Однако, если в основе реализации функции `sleep` лежит функция `alarm`, то могут возникнуть взаимовлияния этих двух функций. Стандарт POSIX.1 никак не оговаривает возможность взаимного влияния. Например, что произойдет, если сначала вызвать `alarm(10)`, а затем, спустя 3 секунды, вызвать `sleep(5)`? Функция `sleep` вернет управление через 5 секунд (разумеется, если процессом не был перехвачен какой-либо сигнал), но будет ли сгенерирован сигнал `SIGALRM` через 2 секунды после этого? Решение этих вопросов остается за реализацией.

> ОС Solaris 9 реализует функцию `sleep` на основе функции `alarm`. Страница справочного руководства `sleep`(3) в Solaris утверждает, что ранее запланированный сигнал `SIGALRM` будет доставлен процессу вовремя. Например, в предыдущем сценарии функция `sleep` перед возвратом запланирует генерацию сигнала `SIGALRM` по истечении оставшегося интервала времени (то есть через 2 секунды) – в этом случае `sleep` вернет значение 0. (Очевидно, что `sleep` должна сохранить адрес обработчика сигнала `SIGALRM` и восстановить его перед возвратом в вызывающую программу.) Более того, если вызвать `alarm(6)` и через 3 секунды – `sleep(5)`, то функция `sleep` вернет управление через 3 секунды (когда будет доставлен сигнал `SIGALRM`), а не через 5. В данном случае она вернет значение 2 (количество секунд, оставшихся до окончания запрошенного интервала времени).
>
> С другой стороны, в ОС FreeBSD 5.2.1, Linux 2.4.22 и Mac OS X 10.3 применяется другой подход: приостановка осуществляется с помощью функции `nanosleep`(2). Эта функция, определяемая расширениями реального времени в Single UNIX Specification, предоставляет возможность устанавливать время задержки с высокой точностью. Она позволяет создать реализацию `sleep`, не зависящую от механизма сигналов.
>
> Для сохранения переносимости приложений не следует делать какие-либо предположения о реализации функции `sleep`, но если вам необходимо смешивать вызовы функции `sleep` с любыми другими функциями, отмеряющими интервалы времени, то вам придется побеспокоиться по поводу возможных взаимовлияний этих функций.

Пример

В листинге 10.21 показана реализация POSIX.1-совместимой функции `sleep`. Эта функция является модификацией функции из листинга 10.4, она надежно обслуживает сигналы и избегает состояния гонки за ресурсами, которое наблюдалось в предыдущей реализации. Однако она по-прежнему не учитывает то, что функция `alarm` могла предварительно установить таймер. (Как уже упоминалось ранее, стандарт POSIX.1 явно не оговаривает возможность взаимного влияния этих двух функций.)

Листинг 10.21. Надежная реализация функции sleep

```
#include "apue.h"

static void
sig_alrm(int signo)
{
    /* ничего не делаем, просто возвращаем управление,
     * чтобы выйти из функции sigsuspend()
     */
}

unsigned int
sleep(unsigned int nsecs)
{
    struct sigaction newact, oldact;
    sigset_t newmask, oldmask, suspmask;
    unsigned int unslept;

    /* установить свой обработчик, сохранив предыдущую информацию */
    newact.sa_handler = sig_alrm;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGALRM, &newact, &oldact);

    /* заблокировать сигнал SIGALRM и сохранить текущую маску сигналов */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);
    alarm(nsecs);
    suspmask = oldmask;
    sigdelset(&suspmask, SIGALRM);/* убедиться, что SIGALRM не заблокирован */
    sigsuspend(&suspmask);        /* дождаться, пока не будет перехвачен */
                                  /* какой-либо сигнал */
    /* был перехвачен некоторый сигнал, сейчас SIGALRM заблокирован */
    unslept = alarm(0);
    sigaction(SIGALRM, &oldact, NULL); /* восстановить предыдущее действие */

    /* восстановить маску сигналов, в которой сигнал SIGALRM разблокирован */
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    return(unslept);
}
```

Чтобы создать более надежную реализацию, потребовался больший объем кода, чем в листинге 10.4. Мы не используем функции дальних переходов (как это делалось в листинге 10.5, чтобы избежать гонки за ресурсами) и поэтому не оказываем влияния на другие обработчики сигналов, которые могли выполняться в момент доставки сигнала SIGALRM.

10.20. Сигналы управления заданиями

Шесть сигналов, перечисленных в табл. 10.1, стандарт POSIX.1 рассматривает как сигналы управления заданиями.

SIGCHLD	Дочерний процесс приостановлен или завершен.
SIGCONT	Возобновление работы приостановленного процесса.
SIGSTOP	Сигнал останова (не может быть проигнорирован).
SIGTSTP	Интерактивный сигнал приостановки.
SIGTTIN	Чтение из управляющего терминала процессом из группы процессов фонового режима.
SIGTTOU	Запись в управляющий терминал процессом из группы процессов фонового режима.

Большинство программ не обрабатывают эти сигналы, за исключением сигнала SIGCHLD. Обычно все необходимые действия по их обработке принимают на себя интерактивные командные оболочки. При вводе символа приостановки (обычно Control-Z), всем процессам переднего плана передается сигнал SIGTSTP. Когда мы даем команду возобновить работу фонового задания или задания переднего плана, командная оболочка посылает всем процессам в задании сигнал SIGCONT. Аналогично, когда процесс получает сигнал SIGTTIN или SIGTTOU, по умолчанию он приостанавливается, а командная оболочка, распознав эту ситуацию, уведомляет нас о ней.

Исключение составляют процессы, которые управляют терминалом, например редактор vi(1). Такие программы должны знать, когда пользователь желает приостановить их работу, чтобы восстановить состояние терминала, предшествовавшее запуску программы. Кроме того, программы, подобные редактору vi, при возобновлении работы должны надлежащим образом перенастроить терминал и перерисовать экран. Позднее мы увидим на примере, как программа, подобная vi, выполняет все необходимые действия.

Сигналы управления заданиями оказывают определенное влияние друг на друга. Когда генерируется любой из четырех сигналов, вызывающих приостановку процесса (SIGSTOP, SIGTSTP, SIGTTIN или SIGTTOU), то система отменяет ожидающий обработки сигнал SIGCONT для этого же процесса. Аналогично, когда генерируется сигнал SIGCONT, система отменяет все ожидающие обработки сигналы приостановки.

Обратите внимание: действие по умолчанию для сигнала SIGCONT заключается в возобновлении процесса, если он был приостановлен, в противном случае сигнал игнорируется. Обычно при получении этого сигнала ничего делать не нужно. Когда генерируется сигнал SIGCONT, приостановленный процесс возобновляет свою работу, даже если этот сигнал заблокирован или игнорируется.

Пример

Программа, представленная листингом 10.22, демонстрирует обычную последовательность действий, выполняемую программами при обработке сигналов управления заданиями. Данная программа просто копирует данные со стандартного ввода на стандартный вывод; в тех местах, где обычно осуществляется управление терминалом, даны соответствующие комментарии.

При запуске программа устанавливает обработчик сигнала SIGTSTP только в том случае, если его диспозиция имеет значение SIG_DFL. Причина состоит в том, что когда программа запускается из командной оболочки, не поддерживающей управление заданиями (например, /bin/sh), диспозиция сигнала должна быть установлена в значение SIG_IGN. На самом деле командная оболочка явно не устанавливает диспозиции трех сигналов (SIGTSTP, SIGTTIN и SIGTTOU) в значение SIG_IGN, изначально это делает процесс init, после чего эти диспозиции наследуются всеми оболочками входа. И только те оболочки, которые обладают возможностью управления заданиями, переустанавливают диспозиции этих трех сигналов в значение SIG_DFL.

Когда мы вводим символ приостановки, процесс получает сигнал SIGTSTP и вызывает обработчик сигнала. На этом этапе нужно выполнить все необходимые действия, связанные с терминалом: переместить курсор в нижний левый угол, восстановить режим работы терминала и тому подобное. После этого процесс отправляет самому себе этот же сигнал, предварительно разблокировав его и установив его диспозицию в значение SIG_DFL. Разблокирование сигнала должно производиться обязательно, так как в это самое время ведется обработка этого же сигнала, и система автоматически заблокировала его в момент вызова обработчика. Здесь процесс приостанавливается системой. Он возобновит работу только при получении сигнала SIGCONT (который обычно посылается в ответ на команду fg). Мы не перехватываем сигнал SIGCONT. По умолчанию он должен возобновить работу приостановленного процесса – когда это произойдет, программа продолжит выполнение, как если бы функция kill вернула управление. В этот момент восстанавливается диспозиция сигнала SIGTSTP и выполняются необходимые действия с терминалом (например, перерисовка экрана).

Листинг 10.22. Обработка сигнала SIGTSTP

```
#include "apue.h"

#define BUFFSIZE 1024

static void sig_tstp(int);
int
main(void)
{
    int n;
    char buf[BUFFSIZE];

    /*
     * Сигнал SIGTSTP следует перехватывать только в том случае,
     * если командная оболочка поддерживает управление заданиями.
     */
    if (signal(SIGTSTP, SIG_IGN) == SIG_DFL)
        signal(SIGTSTP, sig_tstp);

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("ошибка вызова функции write");

    if (n < 0)
```

```
        err_sys("ошибка вызова функции read");
    exit(0);
}

static void
sig_tstp(int signo) /* Обработчик сигнала SIGTSTP */
{
    sigset_t mask;

    /* ... переместить курсор в левый нижний угол, установить режим терминала ... */
    /*
     * Разблокировать SIGTSTP, так как он был заблокирован системой.
     */
    sigemptyset(&mask);
    sigaddset(&mask, SIGTSTP);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    signal(SIGTSTP, SIG_DFL); /* установить диспозицию в значение SIG_DFL */
    kill(getpid(), SIGTSTP);  /* и послать сигнал самому себе */

    /*
     * Функция kill не вернет управление,
     * пока работа процесса не будет возобновлена.
     */
    signal(SIGTSTP, sig_tstp); /* переустановить обработчик сигнала */

    /* ... переустановить режим терминала, перерисовать экран ... */
}
```

10.21. Дополнительные возможности

В этом разделе мы рассмотрим дополнительные возможности работы с сигналами, предоставляемые некоторыми реализациями.

Имена сигналов

В некоторых системах имеется массив

```
extern char *sys_siglist[];
```

Этот массив индексируется номерами сигналов и содержит указатели на строки с именами сигналов.

> Этот массив существует в ОС FreeBSD 5.2.1, Linux 2.4.22 и Mac OS X 10.3. В ОС Solaris 9 также имеется этот массив, но он носит имя `_sys_siglist`.

Кроме того, обычно в этих системах имеется функция `psignal`.

```
#include <signal.h>

void psignal(int signo, const char *msg);
```

Она выводит на стандартное устройство вывода сообщений об ошибках строку (обычно имя программы), за которой следуют двоеточие, пробел, описа-

ние сигнала и символ перевода строки. Эта функция очень похожа на perror (раздел 1.7).

Еще одна распространенная функция – strsignal. Она напоминает функцию strerror (раздел 1.7).

```
#include <string.h>

char *strsignal(int signo);
```

Возвращает указатель на строку с описанием сигнала

По заданному номеру сигнала возвращается строка с описанием этого сигнала. Эта строка может использоваться приложениями для формирования сообщений об ошибках при получении сигналов.

> Все обсуждаемые в этой книге платформы поддерживают функции psignal и strsignal, но они в их реализации имеются различия. В ОС Solaris 9 функция strsignal возвращает пустой указатель, если задан некорректный номер сигнала, тогда как в FreeBSD 5.2.1, Linux 2.4.22 и Mac OS X 10.3 возвращается указатель на строку с сообщением о том, что сигнал не распознан. Кроме того, чтобы использовать функцию psignal в Solaris 9, необходимо подключить к программе заголовочный файл <siginfo.h>.

Отображение сигналов

ОС Solaris предоставляет несколько функций для отображения номеров сигналов в их имена и наоборот.

```
#include <signal.h>

int sig2str(int signo, char *str);

int str2sig(const char *str, int *signop);
```

Обе возвращают 0 в случае успеха, –1 в случае ошибки

Эти функции удобны при разработке интерактивных программ, которые должны принимать и выводить номера сигналов и их имена.

Функция sig2str преобразует номер сигнала в строку и сохраняет результат в памяти по адресу, переданному в аргументе *str*. Вызывающий процесс должен предоставить буфер достаточного размера для хранения строки максимально возможной длины, с учетом завершающего нулевого символа. Для этих целей Solaris предусматривает в заголовочном файле <signal.h> константу SIG2STR_MAX, которая представляет максимальный размер строки, возвращаемой функцией sig2str. Возвращаемая строка содержит имя сигнала без префикса SIG. Например, если функции передать номер сигнала SIGKILL, она вернет строку «KILL» в буфере, на который указывает аргумент *str*.

Функция str2sig преобразует заданное имя сигнала в его номер. Номер сигнала сохраняется в целочисленной переменной, на которую указывает аргумент *signop*. В качестве имени сигнала можно передавать как имя сигнала без префикса SIG, так и строку с десятичным номером сигнала (например, «9»).

10.22. Подведение итогов

Сигналы широко используются в большинстве серьезных приложений. Понимание того, как и зачем обрабатываются сигналы, является основой профессионального подхода к программированию для системы UNIX. В этой главе представлен достаточно объемный и полный обзор сигналов UNIX. Мы начали с рассмотрения недостатков, присущих ранним реализациям сигналов, и того, как они проявляются. Затем мы перешли к обсуждению надежных сигналов POSIX.1 и связанных с ними функций. Разобравшись со всеми тонкостями, мы смогли реализовать свои версии функций `abort`, `system` и `sleep`. И наконец, мы закончили главу рассмотрением сигналов управления заданиями и способов преобразования между именами сигналов и их номерами.

Упражнения

10.1. В листинге 10.1 удалите оператор `for(;;)`. Что произойдет и почему?

10.2. Реализуйте функцию `sig2str`, которая была описана в разделе 10.21.

10.3. Нарисуйте схему, которая показывает фреймы стека программы из листинга 10.6.

10.4. В листинге 10.8 мы продемонстрировали методику использования функций `setjmp` и `longjmp`, которая достаточно часто применяется для ограничения по времени продолжительных операций ввода-вывода. Там имеется следующий код:

```
signal(SIGALRM, sig_alrm);
alarm(60);
if (setjmp(env_alrm) != 0) {
    /* обработать ситуацию выхода по тайм-ауту */
    ...
}
...
```

Скажите, что еще в нем неправильно.

10.5. Используя единственный системный таймер (либо `alarm`, либо `setitimer` – таймер с высоким разрешением), реализуйте набор функций, которые предоставляли бы в распоряжение процесса произвольное количество таймеров.

10.6. Напишите программу, с помощью которой можно было бы проверить функции синхронизации родительского и дочернего процессов из листинга 10.17. Процесс должен создавать файл и записывать в него число 0. Затем вызывается функция `fork`, после чего родительский и дочерний процессы должны по очереди увеличивать число, прочитанное из файла. При каждом увеличении счетчика процесс должен выводить информацию о том, кто произвел увеличение – родитель или потомок.

10.7. В функции из листинга 10.18 предусмотрен сброс диспозиции сигнала SIGABRT в значение по умолчанию и повторный вызов функции kill на случай, если обработчик сигнала вернет управление. Почему в этом случае нельзя просто вызвать функцию _exit?

10.8. Как вы думаете, почему структура siginfo (раздел 10.14) помещает в поле si_uid реальный, а не эффективный идентификатор пользователя?

10.9. Перепишите функцию из листинга 10.10 так, чтобы она могла обслуживать все сигналы из табл. 10.1. Функция должна выполнять одну итерацию цикла для каждого включенного в маску, а не для каждого возможного сигнала.

10.10. Напишите программу, которая вызывала бы sleep(60) в бесконечном цикле. Каждые пять проходов цикла (т. е. каждые 5 минут) программа должна получать текущее время суток и выводить содержимое поля tm_sec. Запустите программу на ночь и объясните полученные результаты. Подумайте, как может быть реализована программа, которая «просыпается» каждую минуту, как демон cron в BSD?

10.11. Измените программу из листинга 3.3 следующим образом: (а) константу BUFFSIZE установите в значение 100, (б) перехватите сигнал SIGXFSZ с помощью функции signal_intr, выведите сообщение при выходе из обработчика сигнала и (в) выведите значение, полученное от функции write, если она не смогла записать заданное количество байт. Измените «мягкий» предел RLIMIT_FSIZE (раздел 7.11), установив его в значение 1024, и с помощью измененной программы попробуйте скопировать файл, размер которого превышает 1024 байта. (Попробуйте установить новое значение предела из командной оболочки. Если это не удастся, вызовите функцию setrlimit прямо из программы.) Запустите эту программу на другой системе, которая вам доступна. Что произошло и почему?

10.12. Напишите программу, которая передает функции fwrite буфер гигантского размера (порядка нескольких сотен мегабайт). Перед обращением к fwrite вызовите alarm, чтобы запланировать генерацию сигнала через одну секунду. Ваш обработчик сигнала должен просто выводить сообщение о том, что сигнал перехвачен, и возвращать управление. Успеет ли функция fwrite завершить работу? Объясните, что произойдет?

11

Потоки

11.1. Введение

В предыдущих главах мы обсуждали процессы. Мы рассмотрели среду окружения процессов в UNIX, взаимоотношения между процессами и способы управления ими.

В этой главе мы продолжим изучение внутреннего устройства процессов и узнаем, как можно использовать несколько *потоков управления* (или просто *потоков* (*threads*)) для решения нескольких задач в рамках единственного процесса. Все потоки внутри процесса имеют доступ к одним и тем же компонентам процесса, таким как файловые дескрипторы или переменные.

Всякий раз при попытке организовать одновременный доступ нескольких пользователей к одному и тому же ресурсу приходится сталкиваться с проблемой согласования доступа. В конце этой главы мы рассмотрим механизмы синхронизации потоков, которые позволяют предотвратить доступ разных потоков к разделяемым ресурсам, находящимся в несогласованном состоянии.

11.2. Концепция потоков

Типичный процесс в UNIX можно представить как имеющий единственный поток управления – каждый процесс в один момент времени решает только одну задачу. При использовании нескольких потоков управления мы можем спроектировать приложение таким образом, что оно будет решать одновременно несколько задач в рамках единственного процесса, где каждый поток решает отдельную задачу. Такой подход имеет следующие преимущества.

- Мы можем значительно упростить код, обрабатывающий асинхронные события, привязав каждый тип события к отдельному потоку. В результате каждый поток может обслуживать свое событие, используя для этого синхронную модель программирования, которая намного проще асинхронной.

- Чтобы организовать совместный доступ нескольких процессов к одним и тем же ресурсам, таким как разделяемая память или файловые дескрипторы, необходимо использовать достаточно сложные механизмы синхронизации, предоставляемые операционной системой (об этом в главах 15 и 17). Потоки же, в отличие от процессов, автоматически получают доступ к одному и тому же адресному пространству и файловым дескрипторам.
- Решение некоторых задач можно разбить на более мелкие подзадачи, что может дать прирост производительности программы. Однопоточный процесс, выполняющий решение нескольких задач, неявно вынужден решать их последовательно, поскольку он имеет только один поток управления. При наличии нескольких потоков управления независимые друг от друга задачи могут решаться одновременно отдельными потоками. Две задачи могут решаться одновременно только при условии, что они не зависят друг от друга.
- Аналогичным образом, интерактивные программы могут сократить время отклика на действия пользователя, используя многопоточную модель для того, чтобы отделить обработку ввода-вывода пользователя от других частей программы.

У многих многопоточное программирование ассоциируется с многопроцессорными системами. Однако преимущества многопоточной модели проявляют себя, даже если программа работает в однопроцессорной системе. Независимо от количества процессоров, программа может быть упрощена благодаря многопоточной модели, поскольку количество процессоров не влияет на структуру программы. Кроме того, в то время, как однопоточный процесс вынужден периодически простаивать при последовательном решении нескольких задач, многопоточный процесс может повысить производительность и в однопроцессорной системе, так как часть потоков могут продолжать работу, когда другие приостановлены в ожидании наступления некоторых событий.

Поток содержит набор информации, необходимой для представления контекста исполнения внутри процесса. Сюда включаются *идентификатор потока*, который идентифицирует поток внутри процесса, набор значений в регистрах процессора, стек, приоритет, маска сигналов, переменная `errno` (раздел 1.7) и дополнительные данные, специфичные для потока (раздел 12.6). Все компоненты процесса, включая исполняемый код программы, глобальные переменные и динамическую память, стеки и файловые дескрипторы, могут совместно использоваться различными потоками этого процесса.

Интерфейс потоков, о котором мы будем говорить, определяется стандартом POSIX.1-2001. Этот интерфейс, известный также как «pthreads» (от «POSIX threads»), представляет собой дополнительную функциональную возможность, включенную в стандарт POSIX.1-2001. Поддержку потоков POSIX можно проверить с помощью макроопределения `_POSIX_THREADS`. Приложения могут выполнять проверку поддержки потоков во время компиляции, используя команду условной компиляции `#ifdef`, или во время выполнения, вызывая функцию `sysconf` с аргументом `_SC_THREADS`.

11.3. Идентификация потоков

Как любой процесс обладает идентификатором процесса, так же и каждый поток имеет свой идентификатор потока. В отличие от процессов, идентификаторы которых являются уникальными в пределах системы, идентификатор потока имеет смысл только в контексте процесса, которому он принадлежит.

Мы уже говорили, что идентификатор процесса представлен типом `pid_t` и является целым неотрицательным числом. Идентификатор потока представлен типом `pthread_t`. Реализациям разрешается использовать структуру для представления типа `pthread_t`, поэтому, чтобы сохранить переносимость приложений, мы не должны рассматривать этот тип как целое число. Следовательно, сравнение двух идентификаторов потоков должно выполняться с помощью функции.

```
#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Возвращает ненулевое значение, если идентификаторы равны, 0 в противном случае

Для представления типа `pthread_t` ОС Linux 2.4.22 использует тип `long int`, Solaris 9 – `unsigned int`, а FreeBSD 5.2.1 и Mac OS X 10.3 в качестве типа `pthread_t` используют указатель на структуру `pthread`.

Поскольку тип `pthread_t` может быть структурой, не существует достаточно переносимого способа вывести его значение. Иногда в процессе отладки программы бывает удобно выводить идентификаторы потоков, но, как правило, в других случаях в этом нет необходимости. В самом худшем случае это приводит к написанию непереносимого отладочного кода, поэтому данное ограничение можно считать несущественным.

Поток может получить свой собственный идентификатор, обратившись к функции `pthread_self`.

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Возвращает идентификатор вызывающего потока

Эта функция может использоваться совместно с `pthread_equal`, если внутри потока возникнет необходимость самоидентификации. Например, главный поток может размещать задания в некоторой очереди и сопровождать их идентификаторами потоков, чтобы каждый поток мог выполнять задания, предназначенные конкретно для него. Эта методика показана на рис. 11.1. Главный поток помещает новые задания в очередь, а три рабочих потока извлекают их из очереди. Вместо того, чтобы позволить произвольному потоку извлекать очередное задание из начала очереди, главный поток, используя идентификаторы потоков, назначает задания конкретным потокам. В этом

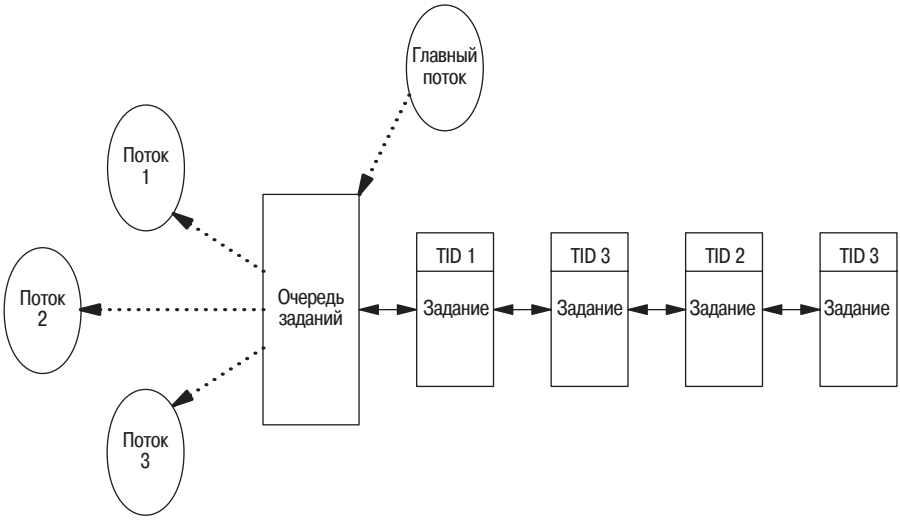


***Рис. 11.1.** Пример очереди заданий*

случае рабочий поток извлекает из очереди только те задания, которые отмечены его идентификатором.

11.4. Создание потока

Традиционная модель процессов в UNIX поддерживает только один поток управления на процесс. Концептуально это та же модель, основанная на потоках, когда каждый процесс состоит из одного потока. При наличии поддержки pthreads программа также запускается как процесс, состоящий из одного потока управления. Поведение такой программы ничем не отличается от поведения традиционного процесса, пока она не создаст дополнительные потоки управления. Создание дополнительных потоков производится с помощью функции `pthread_create`.

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void), void *restrict arg);
```

Возвращает 0 в случае успеха, код ошибки в случае неудачи

Аргумент *tid* – это указатель на область памяти, в которой будет размещен идентификатор созданного потока, если вызов функции `pthread_create` завершится успехом. Аргумент *atr* используется для настройки различных атрибутов потока. Об атрибутах потоков мы поговорим в разделе 12.3, а пока будем передавать в этом аргументе пустой указатель (`NULL`), что соответствует созданию потока со значениями атрибутов по умолчанию.

Вновь созданный поток начинает исполнение с функции start_rtn. Эта функция принимает единственный аргумент, *arg*, который представляет собой нетипизированный указатель. Если необходимо передать функции start_rtn значительный объем информации, то ее следует сохранить в виде структуры и передать указатель на структуру в аргументе *arg*.

При создании нового потока нельзя заранее предполагать, кто первым получит управление – вновь созданный поток или поток, вызвавший функцию pthread_create. Новый поток имеет доступ к адресному пространству процесса и наследует от вызывающего потока среду окружения арифметического сопроцессора и маску сигналов, однако набор сигналов, ожидающих обработки, для нового потока очищается.

Обратите внимание: функции семейства pthread, как правило, возвращают код ошибки в случае неудачи. Они не изменяют значение переменной errno подобно другим функциям POSIX. Экземпляр переменной errno для каждого потока предоставляется только для сохранения совместимости с существующими функциями, которые используют эту переменную. Вообще, при работе с потоками принято возвращать код ошибки из функций, что дает возможность локализовать ошибку, а не полагаться на некоторую глобальную переменную, которая могла быть изменена в результате побочного эффекта.

Пример

Несмотря на то, что не существует достаточно переносимого способа вывода значений идентификаторов потоков, тем не менее можно написать небольшую программу, которая делает это, и таким образом получить представление о некоторых особенностях потоков. Программа, представленная листингом 11.1, выводит идентификатор процесса и идентификаторы начального и вновь созданного потоков.

Листинг 11.1 Вывод идентификаторов потоков

```
#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t pid;
    pthread_t tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
           (unsigned int)tid, (unsigned int)tid);
}

void *
thr_fn(void *arg)
{
```

```
    printids("новый поток: ");
    return((void *)0);
}

int
main(void)
{
    int err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_quit("невозможно создать поток: %s\n", strerror(err));
    printids("главный поток:");
    sleep(1);
    exit(0);
}
```

В этом примере есть два интересных момента, связанных с возможностью гонки за ресурсами между основным и вновь созданным потоками. (Далее в этой же главе мы рассмотрим более правильные способы синхронизации потоков.) В первую очередь необходимо приостановить основной поток. Если этого не сделать, то основной поток может завершиться и тем самым завершить весь процесс еще до того, как новый поток получит возможность начать работу. Такое поведение потоков во многом зависит от реализации потоков в операционной системе и алгоритма планирования.

Второй интересный момент заключается в том, что новый поток получает свой идентификатор с помощью функции `pthread_self`, а не берет его из глобальной переменной или из аргумента запускающей функции. При описании функции `pthread_create` мы уже говорили, что она возвращает идентификатор созданного потока через аргумент *tidp*. В нашем примере основной поток сохраняет его в переменной `ntid`, но новый поток не может ее использовать. Если новый поток получит управление первым, еще до того, как функция `pthread_create` вернет управление в основной поток, то вместо идентификатора новый поток обнаружит неинициализированное значение переменной `ntid`.

Запустив программу из листинга 11.1 в ОС Solaris, мы получили следующие результаты:

```
$ ./a.out
главный поток: pid 7225 tid 1 (0x1)
новый поток:   pid 7225 tid 4 (0x4)
```

Как мы и ожидали, оба потока обладают одним и тем же идентификатором процесса, но разными идентификаторами потоков. Запуск программы из листинга 11.1 в ОС FreeBSD дал следующие результаты:

```
$ ./a.out
главный поток: pid 14954 tid 134529024 (0x804c000)
новый поток:   pid 14954 tid 134530048 (0x804c400)
```

В этом случае потоки также имеют один и тот же идентификатор процесса. Если рассматривать идентификаторы потоков как целые десятичные числа,

то они могут показаться достаточно странными, но если их рассматривать в шестнадцатеричном представлении, то они приобретают некоторый смысл. Как мы уже отмечали ранее, в качестве идентификатора потока FreeBSD использует указатель на структуру с данными потока.

В Mac OS X можно было бы ожидать похожих результатов, однако идентификаторы главного потока и потока, созданного с помощью функции `pthread_create`, принадлежат к разным диапазонам адресов.

```
$ ./a.out
главный поток: pid 779 tid 2684396012 (0xa000a1ec)
новый поток:   pid 779 tid 25166336 (0x1800200)
```

Запуск программы в ОС Linux дал несколько иные результаты:

```
$ ./a.out
новый поток:   pid 6628 tid 1026 (0x402)
главный поток: pid 6626 tid 1024 (0x400)
```

Идентификаторы потоков в Linux выглядят более или менее объяснимо, но идентификаторы процессов не совпадают. Это результат реализации потоков в Linux, где функция `pthread_create` использует системный вызов `clone`. Этот системный вызов создает дочерний процесс, который получает доступ к контексту исполнения родительского процесса – файловым дескрипторам и адресному пространству.

Кроме того, обратите внимание, что во всех операционных системах, кроме Linux, первым вывел сведения главный поток. Это еще раз подтверждает то, что невозможно заранее предполагать, какой из потоков первым получит управление.

11.5. Завершение потока

Если один из потоков вызовет функцию `exit`, `_exit` или `_Exit`, то будет завершен весь процесс. Аналогичным образом, если потоку будет послан сигнал, действие которого заключается в завершении процесса, то этот сигнал завершит весь процесс (более подробно о взаимодействиях между сигналами и потоками мы поговорим в разделе 12.8).

Завершить работу единственного потока, то есть без завершения всего процесса, можно тремя способами.

1. Поток может просто вернуть управление из запускающей процедуры. Возвращаемое значение этой процедуры – код завершения потока.
2. Поток может быть принудительно завершен другим потоком того же самого процесса.
3. Поток может вызвать функцию `pthread_exit`.

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

Аргумент *rval_ptr* представляет собой нетипизированный указатель, аналогичный аргументу, передаваемому запускающей процедуре. Этот указатель смогут получить другие потоки процесса, вызвавшие функцию pthread_join.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **rval_ptr);
```

Возвращает 0 в случае успеха, код ошибки в случае неудачи

Вызывающий поток будет заблокирован до тех пор, пока указанный поток не вызовет функцию pthread_exit, не вернет управление из запускающей процедуры или не будет принудительно завершен другим потоком. Если поток просто выйдет из запускающей процедуры, то *rval_ptr* будет содержать возвращаемое значение. Если поток был принудительно завершен, то по адресу *rval_ptr* будет записано значение PTHREAD_CANCELED.

Вызов функции pthread_join автоматически переводит поток в обособленное состояние (вскоре мы обсудим это), которое позволяет вернуть ресурсы потока обратно. Если он уже находится в обособленном состоянии, то поток, вызвавший pthread_join, получит код ошибки EINVAL.

Если нас не интересует возвращаемое значение потока, то мы можем передать пустой указатель в аргументе *rval_ptr*. В этом случае обращение к функции pthread_join позволит нам дождаться завершения указанного потока, но не вернет код его завершения.

Пример

Листинг 11.2 показывает, как можно получить код выхода завершившегося потока.

Листинг 11.2. Получение кода выхода потока

```
#include "apue.h"
#include <pthread.h>

void *
thr_fn1(void *arg)
{
    printf("поток 1: выход\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("поток 2: выход\n");
    pthread_exit((void *)2);
}

int
main(void)
{
```

```
    int err;
    pthread_t tid1, tid2;
    void *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_quit("невозможно создать поток 1: %s\n", strerror(err));
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);

    if (err != 0)
        err_quit("невозможно создать поток 2: %s\n", strerror(err));
    err = pthread_join(tid1, &tret);

    if (err != 0)
        err_quit("невозможно присоединить поток 1: %s\n", strerror(err));
    printf("код выхода потока 1: %d\n", (int)tret);
    err = pthread_join(tid2, &tret);

    if (err != 0)
        err_quit("невозможно присоединить поток 2: %s\n", strerror(err));
    printf("код выхода потока 2: %d\n", (int)tret);

    exit(0);
}
```

Запустив программу из листинга 11.2, мы получили:

```
$ ./a.out
поток 1: выход
поток 2: выход
код выхода потока 1: 1
код выхода потока 2: 2
```

Как видите, когда поток завершается вызовом функции `pthread_exit` или просто возвращая управление из запускающей процедуры, другой поток может получить код выхода через вызов функции `pthread_join`.

Нетипизированный указатель, передаваемый функциям `pthread_create` и `pthread_exit`, может использоваться для передачи более одного значения. В этом указателе можно передать адрес структуры, которая содержит большой объем информации. Помните, что этот адрес должен оставаться действительным после выхода из вызывающей функции. Если, к примеру, структура размещается на стеке вызывающей функции, то ее содержимое может оказаться измененным к моменту, когда она будет использована. Если поток размещает структуру на стеке и передает указатель на нее функции `pthread_exit`, то стек этого потока может оказаться разрушенным, а память, занимаемая им, может быть использована повторно для других целей к моменту, когда поток, вызвавший `pthread_join`, попытается обратиться к ней.

Пример

Программа, представленная листингом 11.3, демонстрирует проблему, связанную с использованием переменной с автоматическим классом размещения (на стеке) в качестве аргумента функции `pthread_exit`.

Листинг 11.3. Некорректное использование аргумента функции pthread_exit

```
#include "apue.h"
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf(s);
    printf(" структура по адресу 0x%x\n", (unsigned)fp);
    printf(" foo.a = %d\n", fp->a);
    printf(" foo.b = %d\n", fp->b);
    printf(" foo.c = %d\n", fp->c);
    printf(" foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo foo = {1, 2, 3, 4};

    printfoo("поток 1:\n", &foo);
    pthread_exit((void *)&foo);
}

void *
thr_fn2(void *arg)
{
    printf("поток 2: идентификатор - %d\n", pthread_self());
    pthread_exit((void *)0);
}

int
main(void)
{
    int err;
    pthread_t tid1, tid2;
    struct foo *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_quit("невозможно создать поток 1: %s\n", strerror(err));
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_quit("невозможно присоединить поток 1: %s\n", strerror(err));
    sleep(1);
    printf("родительский процесс создает второй поток\n");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_quit("невозможно создать поток 2: %s\n", strerror(err));
```

```
    sleep(1);
    printfoo("родительский процесс:\n", fp);
    exit(0);
}
```

Запустив эту программу в ОС Linux, мы получили:

```
$ ./a.out
поток 1:
  структура по адресу 0x409a2abc
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
родительский процесс создает второй поток
поток 2: идентификатор - 32770
родительский процесс:
  структура по адресу 0x409a2abc
  foo.a = 0
  foo.b = 32770
  foo.c = 1075430560
  foo.d = 1073937284
```

Разумеется, результаты зависят от архитектуры памяти, компилятора и реализации библиотеки функций для работы с потоками. В ОС FreeBSD были получены похожие результаты:

```
$ ./a.out
поток 1:
  структура по адресу 0xbfafefc0
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
родительский процесс создает второй поток
поток 2: идентификатор - 134534144
родительский процесс:
  структура по адресу 0xbfafefc0
  foo.a = 0
  foo.b = 134534144
  foo.c = 3
  foo.d = 67164259
```

Как видите, содержимое структуры (размещенной на стеке потоком *tid1*) изменилось к тому моменту, когда главный поток получил к ней доступ. Обратите внимание на то, как стек второго потока (*tid2*) наложился на стек первого потока. Чтобы решить эту проблему, можно либо использовать глобальную память, либо размещать структуру с помощью функции `malloc`.

Один поток может передать запрос на принудительное завершение другого потока того же самого процесса, обратившись к функции `pthread_cancel`.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

Возвращает 0 в случае успеха, код ошибки в случае неудачи

По умолчанию вызов функции pthread_cancel заставляет указанный поток вести себя так, как будто бы он вызвал функцию pthread_exit с аргументом PTHREAD_CANCELED. Однако поток может отвергнуть запрос или как-то иначе отреагировать на него. Более подробно мы обсудим эту тему в разделе 12.7. Обратите внимание, что функция pthread_cancel не ждет завершения потока. Она просто посылает запрос.

Поток может назначить некоторую функцию, которая будет вызвана в момент его завершения, примерно так же, как это делается для процессов с помощью функции atexit (раздел 7.3), которая регистрирует функции, запускаемые при завершении процесса. Эти функции называют *функциями обработки выхода из потока*. Поток может зарегистрировать несколько таких функций обработки выхода. Обработчики заносятся в стек – это означает, что они будут вызываться в порядке, обратном порядку их регистрации.

```
#include <pthread.h>

void pthread_cleanup_push(void (*rtn)(void *), void *arg);

void pthread_cleanup_pop(int execute);
```

Функция pthread_cleanup_push регистрирует функцию *rtn*, которая будет вызвана с аргументом *arg*, когда поток выполнит одно из следующих действий:

- Вызовет функцию pthread_exit
- Ответит на запрос о принудительном завершении
- Вызовет функцию pthread_cleanup_pop с ненулевым аргументом *execute*

Если аргумент *execute* имеет значение 0, то функция обработки выхода из потока вызываться не будет. В любом случае функция pthread_cleanup_pop удаляет функцию-обработчик, зарегистрированную последним обращением к функции pthread_cleanup_push.

Ограничение, связанное с этими функциями, заключается в том, что они могут быть реализованы в виде макроопределений и тогда они должны использоваться в паре, в пределах одной и той же области видимости в потоке. Макроопределение функции pthread_cleanup_push может включать в себя символ {, и тогда парная ей скобка } будет находиться в макроопределении pthread_cleanup_pop.

Пример

В листинге 11.4 показан порядок использования функций обработки выхода из потока. Хотя это достаточно искусственный пример, тем не менее он прозрачно иллюстрирует описываемую методику. Обратите внимание: хотя ненулевой аргумент и не передается в функцию pthread_cleanup_pop, тем не менее

мы по-прежнему вынуждены вызывать функции pthread_cleanup_push и pthread_cleanup_pop в паре, в противном случае программа может не скомпилироваться.

Листинг 11.4. Обработчик выхода из потока

```
#include "apue.h"
#include <pthread.h>

void
cleanup(void *arg)
{
    printf("выход: %s\n", (char *)arg);
}

void *
thr_fn1(void *arg)
{
    printf("запуск потока 1\n");
    pthread_cleanup_push(cleanup, "поток 1, первый обработчик");
    pthread_cleanup_push(cleanup, "поток 1, второй обработчик");
    printf("поток 1, регистрация обработчиков закончена\n");
    if (arg)
        return((void *)1);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("запуск потока 2\n");
    pthread_cleanup_push(cleanup, "поток 2, первый обработчик");
    pthread_cleanup_push(cleanup, "поток 2, второй обработчик");
    printf("поток 1, регистрация обработчиков закончена\n");
    if (arg)
        pthread_exit((void *)2);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_exit((void *)2);
}

int
main(void)
{
    int err;
    pthread_t tid1, tid2;
    void *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, (void *)1);
    if (err != 0)
        err_quit("невозможно создать поток 1: %s\n", strerror(err));
    err = pthread_create(&tid2, NULL, thr_fn2, (void *)1);
    if (err != 0)
```

```
        err_quit("невозможно создать поток 2: %s\n", strerror(err));
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_quit("невозможно присоединить поток 1: %s\n", strerror(err));
    printf("код выхода потока 1: %d\n", (int)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_quit("невозможно присоединить поток 2: %s\n", strerror(err));
    printf("код выхода потока 2: %d\n", (int)tret);
    exit(0);
}
```

Запуск программы из листинга 11.4 дал нам следующие результаты:

```
$ ./a.out
запуск потока 1
поток 1, регистрация обработчиков закончена
запуск потока 2
поток 2, регистрация обработчиков закончена
выход: поток 2, второй обработчик
выход: поток 2, первый обработчик
код выхода потока 1: 1
код выхода потока 2: 2
```

Из полученных результатов видно, что оба потока нормально запустились и корректно завершились, но функции обработки выхода были вызваны только для второго потока. Таким образом, можно сделать вывод, что функции обработки выхода из потока не вызываются, если поток завершается простым возвратом из его процедуры запуска. Кроме того, обратите внимание, что функции обработки выхода запускаются в порядке, обратном порядку их регистрации.

Сейчас вы уже должны обнаружить некоторые черты сходства между функциями управления процессами и функциями управления потоками. В табл. 11.1 приводится список аналогичных функций.

Таблица 11.1. Функции управления процессами и потоками

Процессы	Потоки	Описание
fork	pthread_create	Создает новый поток управления
exit	pthread_exit	Завершает существующий поток управления
waitpid	pthread_join	Возвращает код выхода из потока управления
atexit	pthread_cleanup_push	Регистрирует функцию обработки выхода из потока управления
getpid	pthread_self	Возвращает идентификатор потока управления
abort	pthread_cancel	Запрашивает аварийное завершение потока управления

По умолчанию код завершения потока сохраняется до тех пор, пока для этого потока не будет вызвана функция pthread_join. Основная память потока

может быть немедленно освобождена по его завершении, если поток был *обособлен*. Когда поток обособлен, функция pthread_join не может использоваться для получения его кода завершения, в этом случае она возвращает код ошибки EINVAL. Обособить поток можно с помощью функции pthread_detach.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Возвращает 0 в случае успеха, код ошибки в случае неудачи

Как мы увидим в следующей главе, существует возможность создания потока, который изначально находится в обособленном состоянии, через изменение атрибутов потока, передаваемых функции pthread_create.

11.6. Синхронизация потоков

При наличии нескольких потоков управления, совместно использующих одни и те же данные, необходимо гарантировать, что каждый из потоков будет видеть эти данные в непротиворечивом состоянии. Если каждый из потоков использует переменные, которые не используются в других потоках, то проблем не возникает. Аналогично, если переменная доступна одновременно нескольким потокам только для чтения, то здесь так же отсутствует проблема сохранения непротиворечивости. Однако, если один поток изменяет значение переменной, читать или изменять которое могут также другие потоки, то необходимо синхронизировать доступ к переменной, чтобы гарантировать, что потоки не будут получать неверное значение переменной при одновременном доступе к ней.

Когда поток изменяет значение переменной, существует потенциальная опасность, что другой поток может прочитать еще не до конца записанное значение. На аппаратных платформах, где запись в память осуществляется более чем за один цикл, может произойти так, что между двумя циклами записи вклинится цикл чтения. Разумеется, такое поведение во многом зависит от аппаратной архитектуры, но при написании переносимых программ мы не можем полагаться на то, что они будут выполняться только на определенной платформе.

На рис. 11.2 приводится пример гипотетической ситуации, когда два потока одновременно выполняют запись и чтение значения одной и той же переменной. В данном примере поток A считывает значение переменной и затем записывает в нее новое значение, но операция записи производится за два цикла. Если поток B прочитает значение этой же переменной между двумя циклами записи, он обнаружит переменную в противоречивом состоянии.

Для решения этой проблемы потоки должны использовать блокировки, которые позволят только одному потоку работать с переменной в один момент времени. На рис. 11.3 показана подобная синхронизация. Если поток B должен прочитать значение переменной, он устанавливает блокировку. Аналогичным образом, когда поток A изменяет значение переменной, он также

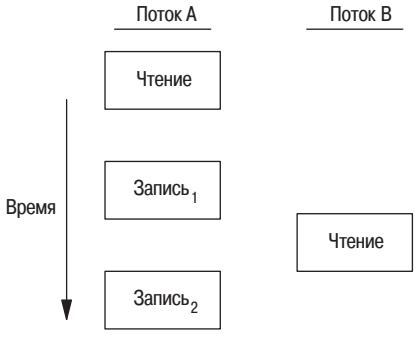


Рис. 11.2. *Перемежение циклов доступа к памяти из двух потоков*

устанавливает блокировку. Таким образом, поток B не сможет прочитать значение переменной, пока поток A не снимет блокировку.

Точно так же следует синхронизировать два или более потоков, которые могут попытаться одновременно изменить значение переменной. Рассмотрим случай, когда выполняется увеличение значения переменной на 1 (рис. 11.4). Операцию увеличения (инкремента) обычно можно разбить на три шага.

1. Прочитать значение переменной из памяти в регистр процессора.
2. Увеличить значение в регистре.
3. Записать новое значение из регистра процессора в память.

Если два потока попытаются одновременно увеличить значение одной и той же переменной, не согласуя свои действия между собой, то результаты могут быть получены самые разные. В конечном итоге полученное значение может оказаться на 1 или на 2 больше предыдущего в зависимости от того,

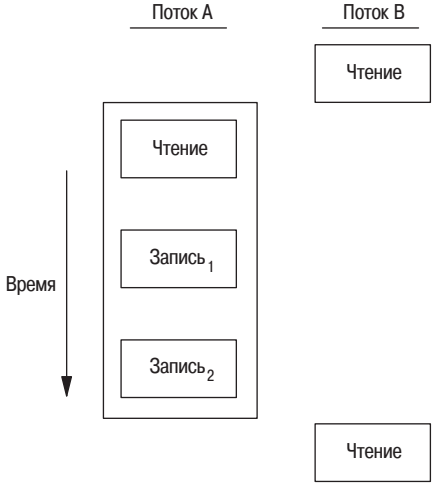


Рис. 11.3. *Синхронизированный доступ к памяти из двух потоков*

какое значение получил второй поток перед началом операции. Если второй поток выполнил шаг 1 до того, как первый выполнил шаг 3, то второй поток прочитает то же самое значение, что и первый поток, увеличит его на 1 и запишет обратно в память, фактически не оказав никакого влияния на значение переменной.

Если изменение переменной производится атомарно, то подобная гонка между потоками отсутствует. В предыдущем примере, если увеличение производится за одно обращение к памяти, состояние гонки между потоками не возникает. Если данные постоянно находятся в *непротиворечивом состоянии*, то нет необходимости предусматривать дополнительную синхронизацию. Операции являются последовательно непротиворечивыми, если различные потоки не могут получить доступ к данным, когда они находятся в противоречивом состоянии. В современных компьютерных системах доступ к памяти выполняется за несколько тактов шины, а в многопроцессорных системах доступ к шине вообще чередуется между несколькими процессорами, поэтому невозможно гарантировать непротиворечивое состояние данных в любой произвольный момент времени.

В непротиворечивой среде можно описать изменения данных как последовательность операций, выполняемых потоками. Мы можем сказать: «Поток А увеличил значение переменной, затем поток В увеличил значение переменной, в результате значение переменной было увеличено на 2» или: «Поток В увеличил значение переменной, затем поток А увеличил значение перемен-

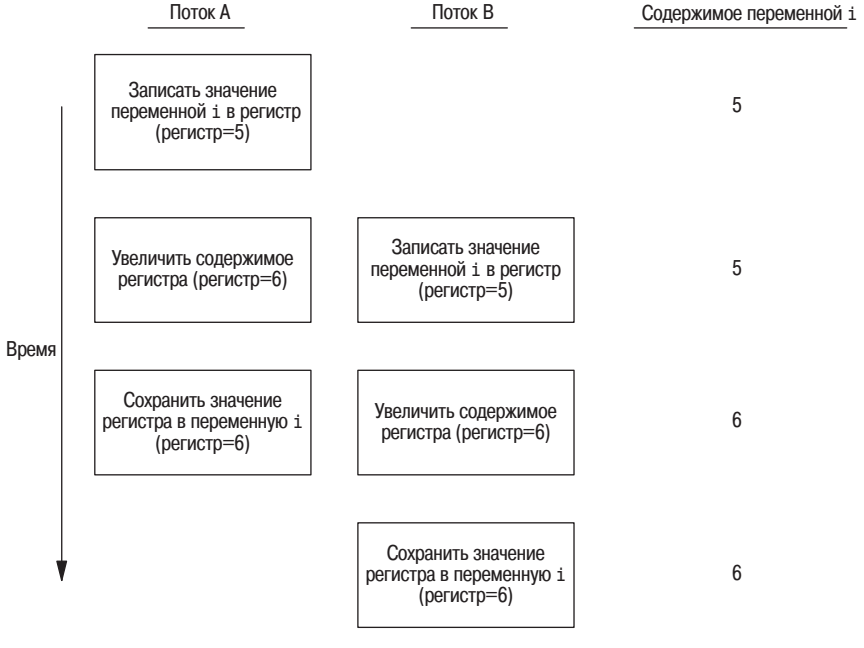


Рис. 11.4. *Два несинхронизированных потока пытаются увеличить значение одной и той же переменной*

ной, в результате значение переменной было увеличено на 2». Конечный результат не зависит от того или иного порядка выполнения потоков.

Помимо особенностей аппаратной архитектуры, состояние гонки может быть вызвано алгоритмом использования переменных в программах. Например, мы можем увеличить значение переменной и затем, основываясь на полученном значении, принять решение о дальнейшем порядке выполнения операций. Комбинация операций, состоящая из увеличения переменной и проверки полученного значения, не является атомарной, и таким образом появляется вероятность принятия неверного решения.

Мьютексы

Мы можем защитить данные и ограничить доступ к ним одним потоком в один момент времени с помощью интерфейса взаимоисключений (mutual-exclusion) pthreads. *Мьютекс* (*mutex*) – это фактически блокировка, которая устанавливается (запирается) перед обращением к разделяемому ресурсу и снимается (отпирается) после выполнения требуемой последовательности операций. Если мьютекс заперт, то любой другой поток, который попытается запереть его, будет заблокирован до тех пор, пока мьютекс не будет отперт. Если в момент, когда отпирается мьютекс, заблокированными окажутся несколько потоков, все они будут запущены и первый из них, который успеет запереть мьютекс, продолжит работу. Все остальные потоки обнаружат, что мьютекс по-прежнему заперт, и опять перейдут в режим ожидания. Таким образом, доступ к ресурсу сможет получить одновременно только один поток.

Такой механизм взаимоисключений будет корректно работать только при условии, что все потоки приложения будут соблюдать одни и те же правила доступа к данным. Операционная система никак не упорядочивает доступ к данным. Если мы позволим одному потоку производить действия с разделяемыми данными, предварительно не ограничив доступ к ним, то остальные потоки могут обнаружить эти данные в противоречивом состоянии, даже если перед обращением к ним будут устанавливать блокировку.

Переменные-мьютексы определяются с типом `pthread_mutex_t`. Прежде чем использовать переменную-мьютекс, мы должны сначала инициализировать ее, записав в нее значение константы `PTHREAD_MUTEX_INITIALIZER` (только для статически размещаемых мьютексов) или вызвав функцию `pthread_mutex_init`. Если мьютекс размещается в динамической памяти (например, с помощью функции `malloc`), то прежде чем освободить занимаемую память, необходимо вызвать функцию `pthread_mutex_destroy`.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Возвращает 0 в случае успеха, код ошибки в случае неудачи

Чтобы инициализировать мьютекс со значениями атрибутов по умолчанию, мы должны передать значение NULL в аргументе *attr*. Конкретные значения атрибутов мьютексов мы рассмотрим в разделе 12.4.

Запирается мьютекс вызовом функции pthread_mutex_lock. Если мьютекс уже заперт, вызывающий поток будет заблокирован до тех пор, пока мьютекс не будет отперт. Мьютекс отпирается вызовом функции pthread_mutex_unlock.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Все три возвращают 0 в случае успеха, код ошибки в случае неудачи

Если поток не должен блокироваться при попытке запереть мьютекс, он может воспользоваться функцией pthread_mutex_trylock. Если к моменту вызова этой функции мьютекс будет отперт, функция запрет мьютекс и вернет значение 0. В противном случае pthread_mutex_trylock вернет код ошибки EBUSY.

Пример

Листинг 11.5 иллюстрирует использование мьютексов для защиты структуры данных. Если более чем один поток работает с данными, размещаемыми динамически, мы можем предусмотреть в структуре данных счетчик ссылок на объект, чтобы освобождать память только в том случае, когда все потоки завершат работу с объектом.

Листинг 11.5. Использование мьютексов для защиты структур данных

```
#include <stdlib.h>
#include <pthread.h>

struct foo {
    int f_count;
    pthread_mutex_t f_lock;
    /* ... другие поля структуры ... */
};

struct foo *
foo_alloc(void) /* размещает объект в динамической памяти */
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... продолжение инициализации ... */
```

```
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* наращивает счетчик ссылок на объект */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

void
foo_rele(struct foo *fp) /* освобождает ссылку на объект */
{
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* последняя ссылка */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Мьютекс запирается перед увеличением или уменьшением счетчика ссылок и перед его проверкой на равенство нулю. При инициализации счетчика ссылок значением 1 в функции `foo_alloc` запирать мьютекс нет необходимости, поскольку пока только поток, который размещает структуру, имеет к ней доступ. Если бы в этой точке структура включалась в некий список, она могла бы быть обнаружена другими потоками, и тогда пришлось бы сначала запереть мьютекс.

Прежде чем приступить к работе с объектом, поток должен увеличить счетчик ссылок на него. По окончании работы с объектом поток должен удалить ссылку. Когда удаляется последняя ссылка, память, занимаемая объектом, освобождается.

Предотвращение тупиковых ситуаций

Поток может попасть в тупиковую ситуацию (deadlock), если попытается дважды захватить один и тот же мьютекс, но есть и менее очевидные способы. Например, тупиковая ситуация может возникнуть в случае, когда в программе используется более одного мьютекса и мы позволим одному потоку удерживать первый мьютекс и пытаться запереть второй мьютекс, в то время как некий другой поток аналогичным образом может удерживать второй мьютекс и пытается запереть первый. В результате ни один из потоков не сможет продолжить работу, поскольку каждый из них будет ждать освобождения ресурса, захваченного другим потоком, и возникает тупиковая ситуация.

Тупиковых ситуаций можно избежать, жестко определив порядок, в котором производится захват ресурсов. Приведем пример. Предположим, что есть два

мьютекса, A и B, которые необходимо запереть одновременно. Если все потоки сначала будут запирать мьютекс A, а потом B, то тупиковой ситуации с этими мьютексами никогда не возникнет. Аналогичным образом, если все потоки сначала будут запирать мьютекс B, а потом A, то тупиковой ситуации с этими мьютексами также никогда не возникнет. Опасность попадания в тупиковую ситуацию возникает только тогда, когда разные потоки могут попытаться запереть мьютексы в разном порядке.

Иногда архитектура приложения не позволяет заранее предопределить порядок захвата мьютексов. Если программа использует достаточно много мьютексов и структур данных, а доступные функции, которые работают с ними, не укладываются в достаточно простую иерархию, то придется попробовать иной подход. Например, при невозможности запереть мьютекс можно отпереть захваченные мьютексы и повторить попытку немного позже. В этом случае во избежание блокировки потока можно использовать функцию `pthread_mutex_trylock`. Если мьютекс удалось запереть с помощью `pthread_mutex_trylock`, то можно продолжить работу. Однако, если мьютекс запереть не удалось, можно отпереть уже захваченные мьютексы, освободить занятые ресурсы и повторить попытку немного позже.

Пример

В этом примере приводится версия листинга 10.10, дополненная с целью продемонстрировать работу с двумя мьютексами. Во избежание тупиковой ситуации, которая может возникнуть при попытке одновременного захвата обоих ресурсов, во всех потоках используется один и тот порядок запирания мьютексов. Второй мьютекс защищает хеш-список структур `foo`. Таким образом, мьютекс `hashlock` защищает хеш-таблицу `fh` и поле связи `f_next` в структуре `foo`. Доступ к остальным полям структуры `foo` производится под защитой мьютекса `f_lock`.

Листинг 11.6. Использование двух мьютексов

```
#include <stdlib.h>
#include <pthread.h>

#define NHASH 29
#define HASH(fp) (((unsigned long)fp)%NHASH)

struct foo *fh[NHASH];

pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int f_count;
    pthread_mutex_t f_lock;
    struct foo *f_next; /* защищается мьютексом hashlock */
    int f_id;
    /* ... другие поля структуры ... */
};

struct foo *
```

```
foo_alloc(void) /* размещает объект в динамической памяти */
{
    struct foo *fp;
    int idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp->f_next;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... продолжение инициализации ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    return(fp);
}

void
foo_hold(struct foo *fp)  /* добавить ссылку на объект */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

struct foo *
foo_find(int id)          /* найти существующий объект */
{
    struct foo *fp;
    int idx;

    idx = HASH(fp);
    pthread_mutex_lock(&hashlock);
    for (fp = fh[idx]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            foo_hold(fp);
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp)  /* освободить ссылку на объект */
{
    struct foo *tfp;
```

```
        int idx;

        pthread_mutex_lock(&fp->f_lock);
        if (fp->f_count == 1) {           /* последняя ссылка */
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_lock(&hashlock);
            pthread_mutex_lock(&fp->f_lock);
            /* необходима повторная проверка условия */
            if (fp->f_count != 1) {
                fp->f_count--;
                pthread_mutex_unlock(&fp->f_lock);
                pthread_mutex_unlock(&hashlock);
                return;
            }
            /* удалить из списка */
            idx = HASH(fp);
            tfp = fh[idx];
            if (tfp == fp) {
                fh[idx] = fp->f_next;
            } else {
                while (tfp->f_next != fp)
                    tfp = tfp->f_next;
                tfp->f_next = fp->f_next;
            }
            pthread_mutex_unlock(&hashlock);
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_destroy(&fp->f_lock);
            free(fp);
        } else {
            fp->f_count--;
            pthread_mutex_unlock(&fp->f_lock);
        }
    }
```

Сравнив листинги 11.6 и 11.5, мы без труда заметим, что теперь функция размещения объекта в динамической памяти блокирует доступ к хеш-таблице, добавляет в нее новую структуру, а перед снятием блокировки с хеш-таблицы запирает новую структуру. Поскольку новая структура размещается в глобальном списке, ее может обнаружить любой другой поток, и поэтому мы вынуждены запирать ее до тех пор, пока не будет закончена инициализация структуры.

Функция `foo_find` запирает хеш-таблицу и производит поиск запрошенной структуры. Если таковая будет найдена, мы увеличиваем в ней счетчик ссылок и возвращаем указатель на структуру. Обратите внимание, что здесь мы соблюдаем порядок захвата мьютексов, запирая мьютекс `hashlock` до того, как функция `foo_hold` запрет мьютекс `f_lock`.

Теперь перейдем к функции `foo_rele`, алгоритм работы которой несколько сложнее. Если освобождается последняя ссылка на объект, то необходимо отпереть мьютекс `f_lock`, чтобы запереть `hashlock`, поскольку нам необходимо удалить структуру из списка. После этого необходимо запереть мьютекс

f_lock. Учитывая, что поток мог быть заблокирован во время повторной попытки захватить мьютексы, мы вынуждены повторить проверку необходимости удаления структуры. Если какой-либо другой поток нашел структуру и нарастил счетчик ссылок в ней, в то время как данный поток был заблокирован в ожидании освобождения мьютекса, мы просто уменьшаем счетчик ссылок, отпираем оба мьютекса и возвращаем управление.

Такой алгоритм работы с мьютексами достаточно сложен, поэтому нужно пересмотреть его. Алгоритм заметно упростится, если мьютекс hashlock будет защищать еще и счетчик ссылок. Мьютекс f_lock будет защищать все остальные поля структуры foo. Эти изменения отражены в листинге 11.7.

Листинг 11.7. Упрощенный вариант использования мьютексов

```
#include <stdlib.h>
#include <pthread.h>

#define NHASH 29
#define HASH(fp) (((unsigned long)fp)%NHASH)

struct foo *fh[NHASH];
pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int f_count;          /* защищается мьютексом hashlock */
    pthread_mutex_t f_lock;
    struct foo *f_next;   /* защищается мьютексом hashlock */
    int f_id;
    /* ... другие поля структуры ... */
};
struct foo *
foo_alloc(void)           /* размещает объект в динамической памяти */
{
    struct foo *fp;
    int idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp->f_next;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... продолжение инициализации ... */
    }
    return(fp);
}

void
```

```
foo_hold(struct foo *fp) /* добавляет ссылку на объект */
{
    pthread_mutex_lock(&hashlock);
    fp->f_count++;
    pthread_mutex_unlock(&hashlock);
}

struct foo *
foo_find(int id)         /* найти существующий объект */
{
    struct foo *fp;
    int idx;

    idx = HASH(fp);
    pthread_mutex_lock(&hashlock);
    for (fp = fh[idx]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            fp->f_count++;
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* освобождает ссылку на объект */
{
    struct foo *tfp;
    int idx;

    pthread_mutex_lock(&hashlock);
    if (--fp->f_count == 0) { /* последняя ссылка, удалить из списка */
        idx = HASH(fp);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&hashlock);
    }
}
```

Обратите внимание, насколько проще стала программа по сравнению с листингом 11.6. Когда мы стали использовать один и тот же мьютекс для защиты хеш-списка и счетчика ссылок, отпала проблема соблюдения порядка за-

хвата мьютексов. При разработке многопоточных приложений достаточно часто приходится идти на подобные компромиссы. Слишком грубая детализация блокировок в конечном итоге приведет к тому, что большинство потоков будут простаивать при попытках запереть один и тот же мьютекс, а преимущества многопоточной архитектуры приложения будут сведены к минимуму. Если детализация блокировок будет слишком мелкой, это существенно усложнит код, а производительность приложения снизится из-за избыточного количества мьютексов. Программист должен отыскать правильный баланс между производительностью и сложностью алгоритма и при этом выполнить все требования, связанные с захватом ресурсов.

Блокировки чтения-записи

Блокировки чтения-записи похожи на мьютексы, за исключением того, что они допускают более высокую степень параллелизма. Мьютексы могут иметь всего два состояния, закрытое и открытое, и только один поток может владеть мьютексом в каждый момент времени. Блокировки чтения-записи могут иметь три состояния: режим блокировки для чтения, режим блокировки для записи и отсутствие блокировки. Режим блокировки для записи может установить только один поток, но установка режима блокировки для чтения доступна нескольким потокам одновременно.

Если блокировка чтения-записи установлена в режиме блокировки для записи, все потоки, которые будут пытаться захватить эту блокировку, будут приостановлены до тех пор, пока блокировка не будет снята. Если блокировка чтения-записи установлена в режиме блокировки для чтения, все потоки, которые будут пытаться захватить эту блокировку для чтения, получат доступ к ресурсу, но если какой-либо поток попытается установить режим блокировки для записи, он будет приостановлен до тех пор, пока не будет снята последняя блокировка для чтения. Различные реализации блокировок чтения-записи могут значительно различаться, но обычно, если блокировка для чтения уже установлена и имеется поток, который пытается установить блокировку для записи, то остальные потоки, которые пытаются получить блокировку для чтения, будут приостановлены. Это предотвращает возможность блокирования пишущих потоков непрекращающимися запросами на получение блокировки для чтения.

Блокировки чтения-записи прекрасно подходят для ситуаций, когда чтение данных производится намного чаще, чем запись. Когда блокировка чтения-записи установлена в режиме для записи, можно безопасно выполнять модификацию защищаемых ею данных, поскольку только один поток может владеть блокировкой для записи. Когда блокировка чтения-записи установлена в режиме для чтения, защищаемые ею данные могут быть безопасно прочитаны несколькими потоками, если эти потоки смогли получить блокировку для чтения.

Блокировки чтения-записи еще называют совместно-исключающими блокировками. Когда блокировка чтения-записи установлена в режиме для чтения, то говорят, что блокировка находится в режиме совместного использования.

Когда блокировка чтения-записи установлена в режиме для записи, то говорят, что блокировка находится в режиме исключительного использования.

Как и в случае с мьютексами, блокировки чтения-записи должны быть инициализированы перед их использованием и разрушены перед освобождением занимаемой ими памяти.

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Функция `pthread_rwlock_init` инициализирует блокировку чтения-записи. Если в аргументе *attr* передается пустой указатель, блокировка инициализируется с атрибутами по умолчанию. Атрибуты блокировок чтения-записи мы рассмотрим в разделе 12.4.

Перед освобождением памяти, занимаемой блокировкой чтения-записи, нужно вызвать функцию `pthread_rwlock_destroy`, чтобы освободить все занимаемые блокировкой ресурсы. Функция `pthread_rwlock_init` размещает все необходимые для блокировки ресурсы, а `pthread_rwlock_destroy` освобождает их. Если освободить память, занимаемую блокировкой чтения-записи, без предварительного обращения к функции `pthread_rwlock_destroy`, то все ресурсы, занимаемые блокировкой, будут потеряны для системы.

Чтобы установить блокировку в режиме для чтения, необходимо вызвать функцию `pthread_rwlock_rdlock`. Чтобы установить блокировку в режиме для записи, необходимо вызвать функцию `pthread_rwlock_wrlock`. Независимо от того, в каком режиме установлена блокировка чтения-записи, снятие блокировки выполняется функцией `pthread_rwlock_unlock`.

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Все три возвращают 0 в случае успеха, код ошибки в случае неудачи

Реализации могут ограничивать количество блокировок, установленных в режиме совместного использования, поэтому обязательно нужно проверять значение, возвращаемое функцией `pthread_rwlock_rdlock`. Даже когда функции `pthread_rwlock_wrlock` и `pthread_rwlock_unlock` возвращают код ошибки, нет необходимости проверять возвращаемые значения этих функций, если схема наложения блокировок разработана надлежащим образом. Эти функции могут вернуть код ошибки только в том случае, когда блокировка не бы-

ла инициализирована или когда может возникнуть тупиковая ситуация при попытке повторно установить уже установленную блокировку.

Стандарт Single UNIX Specification определяет дополнительные версии примитивов для работы с блокировками, которые могут использоваться для проверки состояния блокировки.

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Если блокировка была успешно установлена, эти функции возвращают значение 0. В противном случае они возвращают код ошибки EBUSY. Эти функции могут использоваться в тех случаях, когда невозможно заранее предопределить порядок установки блокировок, чтобы избежать тупиковых ситуаций, которые мы обсуждали ранее.

Пример

Программа, представленная листингом 11.8, иллюстрирует применение блокировок чтения-записи. Очередь запросов на выполнение заданий защищается единственной блокировкой чтения-записи. Этот пример является одной из возможных реализаций приложения, представленного на рис. 11.1, где множество потоков получают задания, назначаемые им главным потоком.

Листинг 11.8. Использование блокировки чтения-записи

```
#include <stdlib.h>
#include <pthread.h>

struct job {
    struct job *j_next;
    struct job *j_prev;
    pthread_t j_id; /* сообщает, какой поток выполняет это задание */
    /* ... другие поля структуры ... */
};

struct queue {
    struct job *q_head;
    struct job *q_tail;
    pthread_rwlock_t q_lock;
};

/*
 * Инициализация очереди.
 */
int
queue_init(struct queue *qp)
{
    int err;
```

```
    qp->q_head = NULL;
    qp->q_tail = NULL;
    err = pthread_rwlock_init(&qp->q_lock, NULL);
    if (err != 0)
        return(err);

    /* ... продолжение инициализации ... */

    return(0);
}

/*
 * Добавить задание в начало очереди.
 */
void
job_insert(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = qp->q_head;
    jp->j_prev = NULL;
    if (qp->q_head != NULL)
        qp->q_head->j_prev = jp;
    else
        qp->q_tail = jp; /* список был пуст */
    qp->q_head = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Добавить задание в конец очереди.
 */
void
job_append(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = NULL;
    jp->j_prev = qp->q_tail;
    if (qp->q_tail != NULL)
        qp->q_tail->j_next = jp;
    else
        qp->q_head = jp; /* список был пуст */
    qp->q_tail = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Удалить задание из очереди.
 */
void
job_remove(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    if (jp == qp->q_head) {
        qp->q_head = jp->j_next;
```

```
        if (qp->q_tail == jp)
            qp->q_tail = NULL;
    } else if (jp == qp->q_tail) {
        qp->q_tail = jp->j_prev;
        if (qp->q_head == jp)
            qp->q_head = NULL;
    } else {
        jp->j_prev->j_next = jp->j_next;
        jp->j_next->j_prev = jp->j_prev;
    }
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Найти задание для потока с заданным идентификатором.
 */
struct job *
job_find(struct queue *qp, pthread_t id)
{
    struct job *jp;

    if (pthread_rwlock_rdlock(&qp->q_lock) != 0)
        return(NULL);
    for (jp = qp->q_head; jp != NULL; jp = jp->j_next)
        if (pthread_equal(jp->j_id, id))
            break;
    pthread_rwlock_unlock(&qp->q_lock);
    return(jp);
}
```

В этом примере блокировка чтения-записи очереди устанавливается в режиме для записи только тогда, когда необходимо добавить новое задание в очередь или удалить задание из очереди. Когда нужно выполнить поиск задания в очереди, мы устанавливаем блокировку в режиме для чтения, допуская возможность поиска заданий несколькими рабочими потоками одновременно. В данном случае использование блокировки чтения-записи дает прирост производительности.

Рабочие потоки извлекают из очереди только те задания, которые соответствуют их идентификаторам. Поскольку сама структура с заданием используется только одним потоком, для организации доступа к ней не требуется дополнительных блокировок.

Переменные состояния

Переменные состояния – это еще один механизм синхронизации потоков. Переменные состояния предоставляют потокам своеобразное место встречи. При использовании вместе с мьютексами переменные состояния позволяют потокам ожидать наступления некоторого события, избегая состояния гонки.

Переменные состояния сами по себе защищаются мьютексами. Прежде чем изменить значение такой переменной, поток должен захватить мьютекс.

Другие потоки не будут замечать изменений переменной, пока они не попытаются захватить этот мьютекс, потому что для оценки переменной состояния необходимо запереть мьютекс.

Переменная состояния, представленная типом `pthread_cond_t`, должна быть инициализирована перед использованием. При статическом размещении переменной можно присвоить ей значение константы `PTHREAD_COND_INITIALIZER`, но если переменная состояния размещается динамически, нужно инициализировать ее вызовом функции `pthread_cond_init`.

Для уничтожения переменной состояния перед освобождением занимаемой ею памяти используется функция `pthread_cond_destroy`.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
                      pthread_condattr_t *restrict attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Если в аргументе *attr* передается пустой указатель, переменная состояния будет инициализирована со значениями атрибутов по умолчанию. Атрибуту переменных состояния мы рассмотрим в разделе 12.4.

Функция `pthread_cond_wait` ожидает, пока переменная не перейдет в истинное состояние. Если нужно ограничить время ожидания заданным интервалом, используется функция `pthread_cond_timedwait`.

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict timeout);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Мьютекс, передаваемый функции `pthread_cond_wait`, защищает доступ к переменной состояния. Вызывающий поток передает его функции в запертом состоянии, а функция атомарно помещает вызывающий поток в список потоков, ожидающих изменения состояния переменной, и отпирает мьютекс. Это исключает вероятность того, что переменная изменит состояние между моментом ее проверки и моментом приостановки потока, благодаря чему поток не пропустит наступление ожидаемого события. Когда функция `pthread_cond_wait` возвращает управление, мьютекс снова запирается.

Функция `pthread_cond_timedwait` работает аналогичным образом, но дополнительно предоставляет возможность ограничить время ожидания. Значение аргумента *timeout* определяет, как долго поток будет ожидать наступления

события. Время тайм-аута задается структурой timespec, в которой время представлено в секундах и долях секунды. Доли секунды исчисляются в наносекундах:

```
struct timespec {
    time_t tv_sec; /* секунды */
    long tv_nsec;  /* наносекунды */
};
```

При использовании этой структуры следует указывать абсолютное время, а не относительное. Например, если нам нужно ограничить время ожидания периодом в 3 минуты, то мы должны преобразовать в эту структуру не 3 минуты, а текущее время в минутах + 3.

Для этого можно воспользоваться функцией gettimeofday (раздел 6.10), которая возвращает текущее время в виде структуры timeval, и затем преобразовать полученное значение в структуру timespec. Чтобы получить абсолютное время для аргумента *timeout*, можно использовать следующую функцию:

```
void
maketimeout(struct timespec *tsp, long minutes)
{
    struct timeval now;

    /* получить текущее время */
    gettimeofday(&now);
    tsp->tv_sec = now.tv_sec;
    tsp->tv_nsec = now.tv_usec * 1000; /* usec to nsec */

    /* добавить величину тайм-аута */
    tsp->tv_sec += minutes * 60;
}
```

Если тайм-аут истечет до появления ожидаемого события, функция pthread_cont_timedwait запрет мьютекс и вернет код ошибки ETIMEDOUT. Когда функция pthread_cont_wait или pthread_cont_timedwait завершится успехом, поток должен оценить значение переменной, поскольку к этому моменту другой поток мог изменить его.

Для передачи сообщения о наступлении события существуют две функции. Функция pthread_cond_signal возобновит работу одного потока, ожидающего наступления события, а pthread_cond_broadcast – всех потоков, ожидающих наступления события.

> Для упрощения реализации стандарт POSIX допускает, чтобы функция pthread_cond_signal возобновляла работу нескольких потоков.

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Когда вызывается функция pthread_cond_signal, говорят, что посылается сигнал о наступлении события. Мы должны сделать все возможное, чтобы сигнал о наступлении события посылался только после изменения состояния переменной.

Пример

В листинге 11.9 приводится пример синхронизации потоков с помощью переменных состояния и мьютексов.

Листинг 11.9. Пример использования переменных состояния

```
#include <pthread.h>

struct msg {
    struct msg *m_next;
    /* ... другие поля структуры ... */
};

struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void
process_msg(void)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);

        /* обработка сообщения mp */
    }
}

void
enqueue_msg(struct msg *mp)
{
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
    pthread_cond_signal(&qready);
}
```

В данном случае отслеживается состояние очереди сообщений. Переменная состояния защищена мьютексом, а определение изменения состояния производится в цикле while. Чтобы поместить очередное сообщение в очередь, необходимо запереть мьютекс, но для того чтобы послать сигнал ожидаю-

щим потокам, запирать мьютекс не нужно. Такой вариант, когда сигнал посылается после отпирания мьютекса, будет прекрасно работать, даже если какой-либо поток успеет возобновить работу до передачи сигнала. Поскольку наступление события проверяется в цикле, это не представляет проблемы: поток просто возобновит работу, убедится, что очередь пуста, и опять перейдет в режим ожидания. Если логика программы не допускает подобной гонки, то тогда необходимо сначала вызвать `pthread_cond_signal`, а затем отпереть мьютекс.

11.7. Подведение итогов

В этой главе мы обсуждали концепцию потоков и примитивы POSIX.1 для работы с ними. Мы также коснулись проблемы синхронизации потоков. Были рассмотрены три фундаментальных механизма синхронизации – мьютексы, блокировки чтения-записи и переменные состояния – и их применение для организации доступа к совместно используемым ресурсам.

Упражнения

11.1. Измените программу из листинга 11.3 таким образом, чтобы она корректно передавала структуру данных между потоками.

11.2. Изучите листинг 11.8 и скажите, какая дополнительная синхронизация должна быть предусмотрена (если она необходима), чтобы позволить главному потоку изменять идентификатор потока в задании? Как это повлияет на функцию `job_remove`?

11.3. Примените технику, показанную в листинге 11.9, к программе (рис. 11.1 и листинг 11.3) для реализации функции рабочего потока. Не забудьте дополнить функцию `queue_init` инициализацией переменной состояния и измените функции `job_insert` и `job_append` так, чтобы они посылали сигналы рабочим потокам. Какие сложности при этом возникнут?

11.4. Какую последовательность действий можно считать правильной?

1. Запереть мьютекс (`pthread_mutex_lock`).
2. Изменить переменную состояния, защищаемую мьютексом.
3. Послать сигнал ожидающим потокам (`pthread_cond_broadcast`).
4. Отпереть мьютекс (`pthread_mutex_unlock`).

или

1. Запереть мьютекс (`pthread_mutex_lock`).
2. Изменить переменную состояния, защищаемую мьютексом.
3. Отпереть мьютекс (`pthread_mutex_unlock`).
4. Послать сигнал ожидающим потокам (`pthread_cond_broadcast`).

12

Управление потоками

12.1. Введение

В главе 11 мы рассмотрели основные понятия, связанные с потоками, и вопросы их синхронизации. В этой главе мы обсудим вопросы управления поведением потока. Мы расскажем об атрибутах потока и объектов синхронизации потоков, которые мы игнорировали в предыдущей главе, работая со значениями по умолчанию.

Мы также поговорим о том, как скрыть данные потока от других потоков того же процесса. И закончим главу описанием взаимодействий между некоторыми системными вызовами и потоками.

12.2. Пределы для потоков

В разделе 2.5.4 мы обсуждали функцию `sysconf`. Стандарт Single UNIX Specification определяет ряд пределов, связанных с потоками, которые не были приведены в табл. 2.10. Как и в случае системных пределов, значения пределов потоков могут быть получены с помощью функции `sysconf`. Эти пределы перечисляются в табл. 12.1.

Как и другие пределы, значения которых сообщает функция `sysconf`, данные пределы предназначены для повышения переносимости приложений между различными реализациями операционных систем. Например, если приложение требует, чтобы для обработки каждого файла создавалось четыре потока, то, вероятно, придется ограничить количество обрабатываемых одновременно файлов, чтобы не превысить ограничение системы на количество одновременно работающих потоков.

В таблице 12.2 приводятся значения этих пределов для четырех обсуждаемых в этой книге платформ. Если реализация не определяет константу для передачи функции `sysconf` (имя которой начинается с последовательности символов `_SC_`), в соответствующей колонке указано «не определено». Если реализация не определяет значение предела, в колонке указывается «нет ог-

раничения», однако это вовсе не говорит о том, что предел не имеет ограничений. Если реализация определяет константу, но функция sysconf не распознает ее, в соответствующей колонке указано «не поддерживается».

Таблица 12.1. Пределы для потоков и соответствующие значения аргумента name функции sysconf

Имя предела	Описание	Значение аргумента name
PTHREAD_DESTRUCTOR_ITERATIONS	Максимальное количество попыток системы уничтожить данные потока после его завершения (раздел 12.6)	_SC_THREAD_DESTRUCTOR_ITERATIONS
PTHREAD_KEYS_MAX	Максимальное количество ключей, которые могут быть созданы процессом (раздел 12.6)	_SC_THREAD_KEYS_MAX
PTHREAD_STACK_MIN	Минимальное количество байт, которое может быть использовано под стек потока (раздел 12.3)	_SC_THREAD_STACK_MIN
PTHREAD_STACK_MAX	Максимальное количество байт, которое может быть использовано под стек потока (раздел 12.3)	_SC_THREAD_STACK_MAX

Обратите внимание, что хотя реализация может и не предоставлять значения этих пределов, тем не менее это не означает, что их вообще не существует. Это означает лишь то, что реализация не предоставляет нам возможность получить значение предела с помощью функции sysconf.

Таблица 12.2. Примеры значений пределов для потоков

Предел	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
PTHREAD_DESTRUCTOR_ITERATIONS	Не определено	Не поддерживается	Не определено	Нет ограничения
PTHREAD_KEYS_MAX	Не определено	Не поддерживается	Не определено	Нет ограничения
PTHREAD_STACK_MIN	Не определено	Не поддерживается	Не определено	4096
PTHREAD_STACK_MAX	Не определено	Не поддерживается	Не определено	Нет ограничения

12.3. Атрибуты потока

Во всех примерах главы 11, где мы вызывали функцию pthread_create, мы передавали ей значение NULL вместо указателя на структуру pthread_attr_t. Структура pthread_attr_t используется для того, чтобы изменить значения атрибутов по умолчанию и связать эти атрибуты с создаваемым потоком. Для инициализации структуры pthread_attr_t можно обратиться к функции pthread_init_attr. После вызова этой функции структура pthread_attr_t будет заполнена значениями атрибутов по умолчанию, которые поддерживает данная реализация. Для изменения отдельных атрибутов необходимо обращаться к помощи других функций, которые описываются в этом разделе.

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

Для разрушения структуры `pthread_attr_t` используется функция `pthread_attr_destroy`. Если функция `pthread_attr_init` реализована таким образом, что она размещает какие-либо области в динамической памяти, то функция `pthread_attr_destroy` освободит их. Кроме того, `pthread_attr_destroy` заполнит структуру ошибочными значениями, чтобы функция `pthread_create` возвращала ошибку при случайном использовании такой структуры.

Структура `pthread_attr_t` непрозрачна для приложения. Это означает, что приложение ничего не должно знать о внутреннем устройстве структуры, что способствует повышению переносимости приложений. Следуя этой модели, стандарт POSIX.1 определяет отдельные функции для получения и изменения значений каждого атрибута.

Атрибуты потока, определяемые стандартом POSIX.1, приводятся в табл. 12.3. Кроме того, стандарт POSIX.1 определяет ряд дополнительных атрибутов для потоков реального времени, но мы не будем обсуждать их здесь. В табл. 12.3 также показано, какие атрибуты поддерживаются нашими четырьмя платформами. Если атрибут доступен через устаревшие функции, он отмечен как «уст.»

Таблица 12.3. Атрибуты потоков POSIX.1

Атрибут	Описание	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
`detachstate`	Атрибут обособленности потока	•	•	•	•
`guardsize`	Размер резервного буфера в конце стека потока		•	•	•
`stackaddr`	Самый нижний адрес стека потока	уст.	•	•	уст.
`stacksize`	Размер стека потока в байтах	•	•	•	•

В разделе 11.5 мы упомянули понятие обособленных потоков. Если нас больше не интересует код завершения существующего потока, мы можем обратиться к функции `pthread_detach`, чтобы позволить операционной системе утилизировать ресурсы, занимаемые потоком, после его завершения.

Если заранее известно, что код завершения потока не потребуется, то можно сразу же создать и запустить поток в обособленном состоянии, изменив значение атрибута *detachstate* в структуре `pthread_attr_t`. Для этого используется функция `pthread_attr_setdetachstate`, которой передается одно из двух

возможных значений – PTHREAD_CREATE_DETACHED, чтобы запустить поток в обособленном состоянии, или PTHREAD_CREATE_JOINABLE, чтобы запустить поток в нормальном состоянии, в котором приложение сможет получить код завершения потока.

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                int *detachstate);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

Чтобы получить текущее состояние атрибута *detachstate*, можно воспользоваться функцией pthread_attr_getdetachstate. По адресу, который передается во втором аргументе, функция запишет одно из двух возможных значений: PTHREAD_CREATE_DETACHED или PTHREAD_CREATE_JOINABLE, в зависимости от значения атрибута в структуре pthread_attr_t.

Пример

В листинге 12.1 приводится функция, которая может использоваться для создания потока в обособленном состоянии.

Листинг 12.1. Создание потока в обособленном состоянии

```
#include "apue.h"
#include <pthread.h>

int
makethread(void *(*fn)(void *), void *arg)
{
    int err;
    pthread_t tid;
    pthread_attr_t attr;

    err = pthread_attr_init(&attr);
    if (err != 0)
        return(err);
    err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (err == 0)
        err = pthread_create(&tid, &attr, fn, arg);
    pthread_attr_destroy(&attr);
    return(err);
}
```

Обратите внимание, что мы игнорируем значение, возвращаемое функцией pthread_attr_destroy. В данном случае мы корректно инициализировали атрибуты потока, поэтому pthread_attr_destroy не должна завершаться с ошибкой. Тем не менее, если бы эта функция завершилась неудачей, то восстановление после такой ошибки было бы достаточно сложным: мы должны были бы разрушить только что созданный поток, который возможно уже ра-

ботает асинхронно по отношению к этой функции. Самое худшее, что может случиться в случае игнорирования возвращаемого значения функции pthread_attr_destroy, – это утечка небольшого объема памяти, который, возможно, был распределен функцией pthread_attr_init. Но в любом случае, если pthread_attr_init завершилась успехом, а pthread_attr_destroy – с ошибкой, у нас все равно нет никакой стратегии восстановления после такой ошибки, потому что структура с атрибутами непрозрачна для приложения. Для утилизации структуры определен один-единственный интерфейс pthread_attr_destroy, и он потерпел неудачу.

Поддержка атрибутов потоков, связанных со стеком, является необязательной для POSIX-совместимых операционных систем, но обязательна для систем, отвечающих требованиям XSI. Проверить наличие поддержки атрибутов стека для каждого потока можно на этапе компиляции, используя для этого макроопределения _POSIX_THREAD_ATTR_STACKADDR и _POSIX_THREAD_ATTR_STACKSIZE. Если определен какой-либо из этих символов, то поддерживается и соответствующий ему атрибут. Выполнить аналогичную проверку во время выполнения можно с помощью функции sysconf, передав ей символические имена _SC_THREAD_ATTR_STACKADDR и _SC_THREAD_ATTR_STACKSIZE.

Для работы с атрибутами стека потока стандарт POSIX.1 определяет несколько функций. Две более старые функции, pthread_attr_getstackaddr и pthread_attr_setstackaddr, отмечены как устаревшие в третьей версии Single UNIX Specification, хотя они по-прежнему присутствуют в большинстве реализаций pthreads. Получать и изменять атрибуты стека потока предпочтительнее с помощью более современных функций pthread_attr_getstack и pthread_attr_setstack. Эти функции устраняют неоднозначности, имевшиеся в определениях старых интерфейсов.

```
#include <pthread.h>
int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);

int pthread_attr_setstack(const pthread_attr_t *attr,
                          void *stackaddr, size_t *stacksize);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Эти функции используются для работы с обоими атрибутами стека потока – *stackaddr* и *stacksize*.

Виртуальное адресное пространство процесса имеет фиксированный объем. Поскольку однопоточные процессы имеют только один стек, то его размер обычно не вызывает проблем. В случае многопоточных приложений одно и то же виртуальное адресное пространство отведено под стеки всех потоков. Если приложение запускает достаточно большое количество потоков, иногда приходится уменьшать размер стека, установленный по умолчанию, чтобы суммарный объем стеков не превысил доступный объем виртуального адресного пространства. С другой стороны, если из потоков вызываются

функции, которые размещают на стеке большое число локальных переменных, или если глубина вызовов функций очень велика, то, возможно, придется увеличить размер стека.

В случае нехватки виртуального адресного пространства место под стек потока можно выделить с помощью функции malloc или mmap (раздел 14.9) и затем, посредством функции pthread_attr_setstack, изменить местоположение стека создаваемого потока. Адрес стека определяется аргументом *stackaddr*, который представляет собой наименьший адрес в диапазоне памяти, используемой под стек потока, выравненный по границе в соответствии с архитектурой процессора.

Атрибут потока *stackaddr* определяет наименьший адрес участка памяти, отведенной под стек. Однако это не обязательно дно (начало) стека. Если для определенной аппаратной архитектуры стек растет от старших адресов к младшим, атрибут *stackaddr* будет определять вершину (конец) стека, а не его дно (начало).

> Недостаток функций pthread_attr_getstackaddr и pthread_attr_setstackaddr заключается в неоднозначности толкования аргумента *stackaddr*. Он может интерпретироваться и как начало стека, и как наименьший адрес пространства памяти, отведенной под стек. В архитектурах, где стек растет вниз, от старших адресов к младшим, для того чтобы определить, где находится начало стека, необходимо было дополнительно узнать размер стека. Функции pthread_attr_getstack и pthread_attr_setstack ликвидируют этот недостаток.

Приложения также могут получать и изменять значение атрибута потока *stacksize* с помощью функций pthread_attr_getstacksize и pthread_attr_setstacksize.

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                              size_t *restrict stacksize);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Функция pthread_attr_setstacksize удобна в случае, когда необходимо изменить размер стека по умолчанию, но при этом нет желания заниматься распределением памяти для стека.

Атрибут *guardsize* управляет размером памяти, расположенной за концом стека, которая служит для предохранения стека от переполнения. По умолчанию этот атрибут имеет значение PAGESIZE. Можно установить значение атрибута *guardsize* равным 0, запретив тем самым использование предохранительного буфера. Кроме того, если мы изменим значение атрибута *stackaddr*, система будет предполагать, что мы берем на себя ответственность за распределение памяти под стек и запретит использование защитного буфера, просто записав значение 0 в атрибут *guardsize*.

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                              size_t *restrict guardsize);

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Если атрибут *guardsize* был изменен, система, возможно, округлит его значение до ближайшего целого, кратного размеру страницы. Если указатель стека потока войдет в пределы предохранительного буфера, приложение получит сообщение об ошибке – вероятно, в виде сигнала.

Стандарт Single UNIX Specification определяет еще целый ряд необязательных атрибутов потоков в виде расширений потоков реального времени, но мы не будем обсуждать их здесь.

Дополнительные атрибуты потоков

Потоки имеют еще несколько атрибутов, не представленных в структуре `pthread_attr_t`:

- Возможность принудительного завершения (обсуждается в разделе 12.7)
- Тип принудительного завершения (также обсуждается в разделе 12.7)
- Степень совмещения

Степень совмещения (concurrency level) определяет количество потоков или процессов ядра, на которые отображаются потоки пользователя. Если реализация поддерживает отображение между пользовательскими потоками и потоками ядра один в один, то изменение этого атрибута не будет оказывать никакого эффекта, поскольку каждый из потоков пользователя может обслуживаться планировщиком. Однако, если реализация предусматривает мультиплексирование потоков пользователя поверх потоков или процессов ядра, то увеличение количества одновременно выполняемых потоков пользователя может дать прирост производительности. Сообщить системе желаемую степень совмещения можно с помощью функции `pthread_setconcurrency`.

```
#include <pthread.h>

int pthread_getconcurrency(void);
```

Возвращает текущее значение степени совмещения

```
int pthread_setconcurrency(int level);
```

Возвращает 0 в случае успеха, код ошибки в случае неудачи

Функция `pthread_getconcurrency` возвращает текущее значение степени совмещения. Если этим параметром управляет операционная система (то есть предварительно не вызывалась функция `pthread_setconcurrency`), функция `pthread_getconcurrency` будет возвращать значение 0.

Изменение степени совмещения производится с помощью функции pthread_setconcurrency. Нет никакой гарантии, что запрошенное значение степени совмещения будет учитываться системой. Можно сообщить системе, что она сама должна выбрать степень совмещения, передав в аргументе *level* значение 0. Таким способом приложение может отменить действие предыдущего вызова функции pthread_setconcurrency с ненулевым значением в аргументе *level*.

12.4. Атрибуты синхронизации

Как и потоки, объекты синхронизации потоков также имеют атрибуты. В этом разделе мы рассмотрим атрибуты мьютексов, блокировок чтения-записи и переменных состояния.

Атрибуты мьютексов

Для инициализации структуры pthread_mutexattr_t используется функция pthread_mutexattr_init, а для ее разрушения – pthread_mutexattr_destroy.

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Функция pthread_mutexattr_init инициализирует структуру pthread_mutexattr_t значениями атрибутов мьютексов по умолчанию. Для нас представляют интерес два атрибута: *process-shared* и *type*. Согласно стандарту POSIX.1 атрибут *process-shared* является необязательным – если этот атрибут поддерживается на заданной платформе, то будет определен и символ _POSIX_THREAD_PROCESS_SHARED. Проверку во время выполнения можно произвести с помощью функции sysconf, передав ей параметр _SC_THREAD_PROCESS_SHARED. Хотя POSIX-совместимые системы не обязаны поддерживать этот атрибут, стандарт Single UNIX Specification требует обязательной поддержки этого атрибута в операционных системах, отвечающих требованиям XSI.

Внутри процесса множество потоков могут иметь доступ к одному и тому же объекту синхронизации. Такое поведение определено по умолчанию, о чем мы уже говорили в главе 11. В этом случае атрибут мьютекса *process-shared* имеет значение PTHREAD_PROCESS_PRIVATE.

Как мы увидим в главах 14 и 15, существуют механизмы, позволяющие независимым друг от друга процессам отображать одну и ту же область памяти в свои собственные адресные пространства. Доступ к данным, совместно используемым несколькими процессами, обычно требует синхронизации, так же как и доступ к совместно используемым данным из нескольких потоков. Если атрибут *process-shared* установлен в значение PTHREAD_PROCESS_SHARED, следовательно, мьютекс размещается в области памяти, разделяемой между несколькими процессами, и может использоваться для синхронизации этих процессов.

Получить значение атрибута *process-shared* структуры pthread_mutexattr_t можно с помощью функции pthread_mutexattr_getpshared. Чтобы изменить значение этого атрибута, следует использовать функцию pthread_mutexattr_setpshared.

```
#include <pthread.h>
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr,
                                 int *restrict pshared);

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Атрибут *process-shared* позволяет библиотеке pthread выбрать более оптимальную реализацию мьютекса в случае, когда этот атрибут имеет значение PTHREAD_PROCESS_PRIVATE, которое принимается по умолчанию для многопоточных приложений. Таким образом можно ограничить использование более ресурсоемкой реализации случаем, когда мьютексы совместно используются несколькими процессами.

Атрибут *type* позволяет указать тип мьютекса. Стандарт POSIX.1 определяет четыре типа. Тип PTHREAD_MUTEX_NORMAL – это стандартный мьютекс, который не производит дополнительных проверок на наличие ошибок или тупиковых ситуаций. Мьютексы типа PTHREAD_MUTEX_ERRORCHECK производят проверку наличия ошибок.

Мьютексы типа PTHREAD_MUTEX_RECURSIVE позволяют одному и тому же потоку многократно запирать мьютекс, не отпирая его. Рекурсивные мьютексы содержат счетчик, в котором хранится количество запираний мьютекса. Мьютекс будет освобожден только тогда, когда количество отпираний совпадет с количеством запираний. Так, если имеется рекурсивный мьютекс, который был заперт дважды, и вы отперли его один раз, то мьютекс все равно останется заблокированным до тех пор, пока вы не отопрете его второй раз.

И, наконец, мьютексы типа PTHREAD_MUTEX_DEFAULT могут использоваться для назначения семантики мьютекса по умолчанию. Реализации могут самостоятельно определять, какому из трех предыдущих типов соответствует данный тип мьютекса. Так, например, в ОС Linux этот тип мьютекса соответствует типу PTHREAD_MUTEX_NORMAL.

Поведение мьютексов этих четырех типов показано в табл. 12.4. Колонка «Попытка отпирания другим потоком» соответствует ситуации, когда производится попытка отпирания мьютекса, запертого другим потоком. Колонка «Попытка отпирания незапертого мьютекса» соответствует ситуации, когда поток пытается отпереть незапертый мьютекс, что обычно объясняется ошибкой в алгоритме.

Чтобы получить значение атрибута *type*, можно использовать функцию pthread_mutexattr_gettype, а чтобы изменить его – функцию pthread_mutexattr_settype.

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                              int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Таблица 12.4. Поведение мьютексов различного типа

Тип	Повторное запирание без отпирания	Попытка отпирания другим потоком	Попытка отпирания незапертого мьютекса
PTHREAD_MUTEX_NORMAL	Тупиковая ситуация	Не определено	Не определено
PTHREAD_MUTEX_ERRORCHECK	Возвращает код ошибки	Возвращает код ошибки	Возвращает код ошибки
PTHREAD_MUTEX_RECURSIVE	Допускается	Возвращает код ошибки	Возвращает код ошибки
PTHREAD_MUTEX_DEFAULT	Не определено	Не определено	Не определено

В разделе 11.6 мы уже говорили, что переменные состояния используются в паре с мьютексами. Прежде чем заблокировать поток, функции pthread_cond_wait и pthread_cond_timedwait отпирают мьютекс, ассоциированный с переменной состояния. Это позволяет другим потокам запирать мьютекс, изменять состояние, отпирать мьютекс и подавать сигнал об изменении состояния. Поскольку перед изменением состояния мьютекс должен быть захвачен, было бы неправильно использовать для этой цели рекурсивные мьютексы. Если рекурсивный мьютекс был заперт несколько раз, а затем передан функции pthread_cond_wait, то изменение состояния не будет замечено, потому что, отпирая мьютекс, функция pthread_cond_wait не освобождает его.

Рекурсивные мьютексы удобны, когда необходимо адаптировать существующие функции для работы в многопоточной среде, но без изменения прототипов функций, чтобы сохранить совместимость с существующим программным обеспечением. Однако использование рекурсивных блокировок имеет свои особенности, поэтому их следует применять только в том случае, когда нет иного выхода.

Пример

На рис. 12.1 показана ситуация, когда использование рекурсивного мьютекса, казалось бы, является решением проблемы адаптации функций для работы в многопоточной среде. Предположим, что func1 и func2 — это существующие библиотечные функции, и их прототипы не могут быть изменены из-за необходимости сохранения совместимости с программным обеспечением, которое не может подвергаться изменениям.

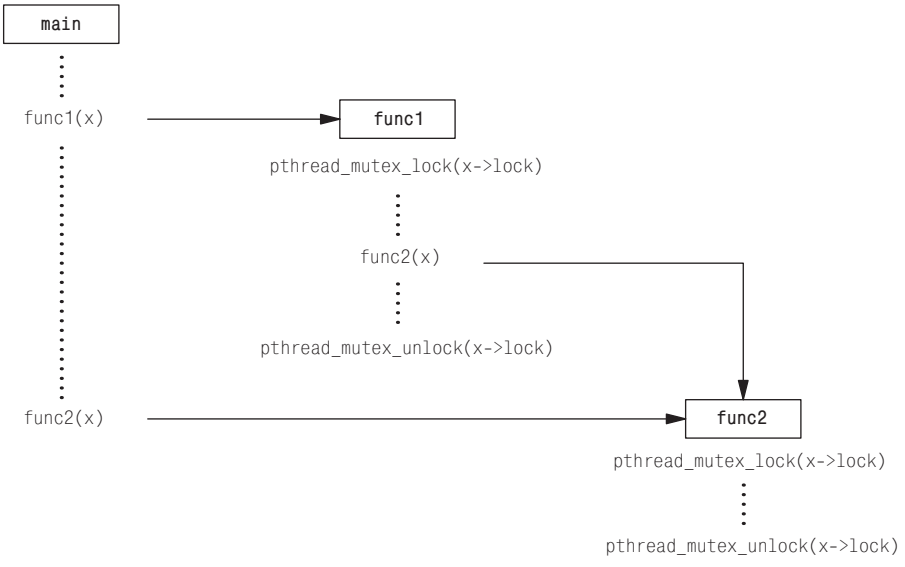


***Рис. 12.1.** Использование рекурсивной блокировки*

Чтобы сохранить прототипы функций без изменения, мы внедряем мьютекс в структуру данных, адрес которой (`x`) передается функциям в виде аргумента. Это возможно только в том случае, если размещением структуры в памяти занимается отдельная библиотечная функция, благодаря которой приложение ничего не знает о размере структуры (логично предположить, что в результате внедрения мьютекса в структуру ее размер увеличится).

> Возможно, что структура изначально была определена с дополнительным зарезервированным объемом, и это позволит без труда добавить в нее поле с мьютексом. К сожалению, большинство программистов лишены дара предсказывать будущее, так что такая практика распространена не очень широко.

Если обе функции должны выполнять некоторые действия со структурой и существует вероятность, что они будут вызываться одновременно из нескольких потоков, то `func1` и `func2` должны запирать мьютекс перед выполнением действий с данными. Если функция `func1` должна вызывать `func2`, то при использовании нерекурсивного мьютекса приложение легко может попасть в тупиковую ситуацию. Нерекурсивный мьютекс можно было бы использовать, освобождая его перед вызовом `func2` из `func1` и запирая вновь после возврата из `func2`, но при таком подходе появляется некоторый интервал времени, в течение которого другой поток может захватить мьютекс и внести нежелательные изменения в данные. Такое решение может быть неприемлемо в зависимости от того, какие данные защищены мьютексом.

На рис. 12.2 показан альтернативный вариант решения той же проблемы – без использования рекурсивного мьютекса. Мы можем оставить прототипы функций `func1` и `func2` неизменными и отказаться от использования рекур-

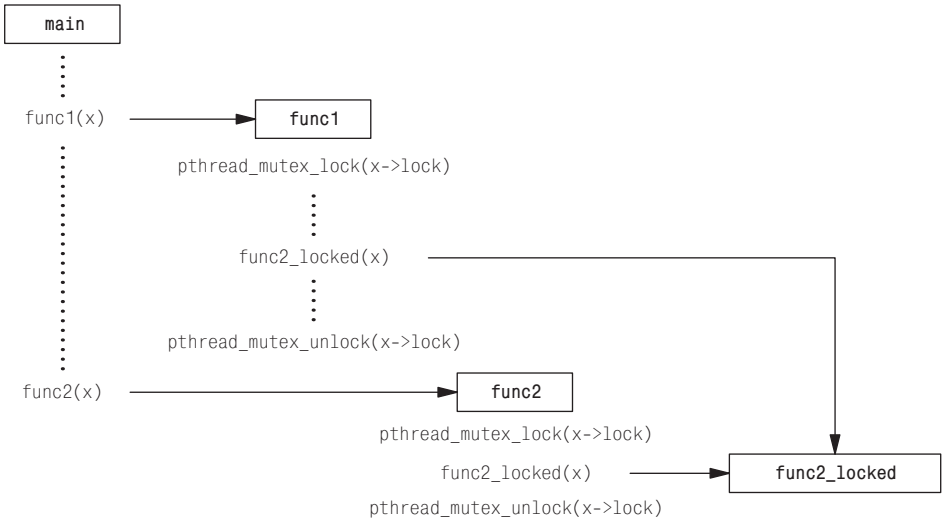


Рис. 12.2. Отказ от использования рекурсивной блокировки

сивного мьютекса за счет реализации скрытой от приложений версии функции `func2` – `func2_locked`. Перед вызовом функции `func2_locked` мьютекс, внедренный в структуру данных, которая передается в качестве аргумента, должен быть заперт. Тело `func2_locked` представляет собой копию прежней `func2`, а сама `func2` теперь просто запирает мьютекс, вызывает `func2_locked` и затем отпирает мьютекс.

Если бы не было требования неизменности прототипов библиотечных функций, то можно было бы добавить в каждую из функций дополнительный аргумент, указывающий, был ли заблокирован доступ к структуре в вызывающей функции. Однако, лучше все-таки оставлять прототипы библиотечных функций без изменения, если это возможно, чем засорять код особенностями реализации.

Стратегия использования заблокированных и незаблокированных версий функций обычно применима только в достаточно простых ситуациях. В более сложных случаях (например, когда библиотечная функция вызывает функцию, расположенную за пределами библиотеки, которая в свою очередь может обращаться к библиотечным функциям) не остается ничего другого, как полагаться на рекурсивные блокировки.

Пример

Программа, представленная листингом 12.2, иллюстрирует еще один случай, когда необходимо использование рекурсивного мьютекса. Здесь у нас имеется функция `timeout`, которая позволяет запланировать запуск другой функции на определенное время. Допустим, что поток не является дорогостоящим ресурсом, тогда для каждого из запланированных тайм-аутов мы

можем запускать отдельный поток. Он будет ожидать указанного момента времени и затем вызывать запрошенную функцию.

Проблема возникает, когда приложение не может создать новый поток или когда запрошенный момент запуска функции уже прошел. Поскольку функция все время пытается установить одну и ту же блокировку, при использовании нерекурсивной блокировки может возникнуть тупиковая ситуация.

Листинг 12.2. Использование рекурсивного мьютекса

```
#include "apue.h"
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

extern int makethread(void *(*)(void *), void *);

struct to_info {
    void (*to_fn)(void *);     /* функция */
    void *to_arg;              /* аргумент */
    struct timespec to_wait;   /* время запуска */
};
#define SECTONSEC 1000000000   /* наносекунд в секунде */
#define USECTONSEC 1000        /* наносекунд в микросекунде */

void *
timeout_helper(void *arg)
{
    struct to_info *tip;

    tip = (struct to_info *)arg;
    nanosleep(&tip->to_wait, NULL);
    (*tip->to_fn)(tip->to_arg);
    return(0);
}

void
timeout(const struct timespec *when, void (*func)(void *), void *arg)
{
    struct timespec now;
    struct timeval tv;
    struct to_info *tip;
    int err;

    gettimeofday(&tv, NULL);
    now.tv_sec = tv.tv_sec;
    now.tv_nsec = tv.tv_usec * USECTONSEC;
    if ((when->tv_sec > now.tv_sec) ||
      (when->tv_sec == now.tv_sec && when->tv_nsec > now.tv_nsec)) {
        tip = malloc(sizeof(struct to_info));
        if (tip != NULL) {
            tip->to_fn = func;
            tip->to_arg = arg;
            tip->to_wait.tv_sec = when->tv_sec - now.tv_sec;
            if (when->tv_nsec >= now.tv_nsec) {
                tip->to_wait.tv_nsec = when->tv_nsec - now.tv_nsec;
```

```
            } else {
                tip->to_wait.tv_sec--;
                tip->to_wait.tv_nsec = SECTONSEC - now.tv_nsec +
                  when->tv_nsec;
            }
            err = makethread(timeout_helper, (void *)tip);
            if (err == 0)
                return;
        }
    }

    /*
     * В эту точку управление переходит, если (а) when <= now
     * или (б) вызов функции malloc терпит неудачу
     * или (в) невозможно создать новый поток,
     * поэтому мы просто вызываем требуемую функцию.
     */
     (*func)(arg);
}

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void
retry(void *arg)
{
    pthread_mutex_lock(&mutex);
    /* выполнить действия, предусмотренные функцией ... */
    pthread_mutex_unlock(&mutex);
}

int
main(void)
{
    int err, condition, arg;
    struct timespec when;

    if ((err = pthread_mutexattr_init(&attr)) != 0)
        err_exit(err, "ошибка вызова функции pthread_mutexattr_init");
    if ((err = pthread_mutexattr_settype(&attr,
      PTHREAD_MUTEX_RECURSIVE)) != 0)
        err_exit(err, "невозможно установить рекурсивный тип мьютекса");
    if ((err = pthread_mutex_init(&mutex, &attr)) != 0)
        err_exit(err, "невозможно создать рекурсивный мьютекс");
    /* ... */
    pthread_mutex_lock(&mutex);
    /* ... */
    if (condition) {
        /* рассчитать время запуска функции "when" */
        timeout(&when, retry, (void *)arg);
    }
    /* ... */
    pthread_mutex_unlock(&mutex);
    /* ... */
```

```
        exit(0);
    }
```

Для создания потоков в обособленном состоянии мы воспользовались функцией makethread из листинга 12.1. Нам необходимо запланировать запуск функции на будущее, но мы не желаем ждать завершения потока.

Для организации задержки можно было бы воспользоваться функцией sleep, но она может отмерять интервалы времени с точностью лишь до секунды. Если нам нужна задержка на промежуток времени, отличный от целого числа секунд, следует использовать функцию nanosleep(2), которая обладает аналогичной функциональностью.

> Несмотря на то, что функция nanosleep является обязательной только в расширениях реального времени, определяемых стандартом Single UNIX Specification, она поддерживается на всех четырех платформах, обсуждаемых в данной книге.

Функция, вызывающая timeout, должна удерживать мьютекс на время проверки условия и планирования функции retry, чтобы обеспечить атомарность этих двух операций. Функция retry пытается запереть тот же самый мьютекс. Если бы в программе использовался нерекурсивный мьютекс, то прямой вызов retry из функции timeout приводил бы к тупиковой ситуации.

Атрибуты блокировок чтения–записи

Блокировки чтения-записи, подобно мьютексам, также имеют атрибуты. Для инициализации структуры pthread_rwlockattr_t используется функция pthread_rwlockattr_init, а для ее разрушения – функция pthread_rwlockattr_destroy.

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Единственный атрибут, поддерживаемый блокировками чтения-записи, – это атрибут *process-shared*, полностью идентичный аналогичному атрибуту мьютексов. Как и в случае с мьютексами, для обслуживания атрибута *process-shared* блокировок чтения-записи используется пара функций: pthread_rwlockattr_getpshared и pthread_rwlockattr_setpshared.

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
                                  int *restrict pshared);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Хотя стандарт POSIX определяет всего один атрибут для блокировок чтения-записи, тем не менее реализации могут свободно добавлять собственные нестандартные атрибуты.

Атрибуты переменных состояния

У переменных состояния также имеются атрибуты. Для их инициализации и разрушения существует пара функций, подобных функциям для мьютексов и блокировок чтения-записи.

```
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

Как и другие примитивы синхронизации, переменные состояния поддерживают атрибут *process-shared*.

```
#include <pthread.h>

int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
                                int *restrict pshared);

int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);
```

Обе возвращают 0 в случае успеха, код ошибки в случае неудачи

12.5. Реентерабельность

В разделе 10.6 обсуждались обработчики сигналов и реентерабельные функции. Потоки в чем-то похожи на обработчики сигналов, когда дело касается реентерабельности. Как и в случае с обработчиками сигналов, в многопоточных приложениях вполне вероятна ситуация, когда одну и ту же функцию одновременно вызывают несколько потоков.

Функции, которые могут безопасно вызываться одновременно из нескольких потоков, называются *безопасными в многопоточной среде* (*thread-safe*). Все функции, определяемые стандартом Single UNIX Specification, являются безопасными в многопоточной среде, за исключением перечисленных в табл. 12.5. Кроме того, функции `ctermid` и `tmpnam` не гарантируют безопасность в многопоточной среде, если им в качестве аргумента передается пустой указатель. Аналогичным образом, функции `wcrtomb` и `wcsrtombs` не гарантируют безопасность в многопоточной среде, если им в качестве аргумента *mbstate_t* передается пустой указатель.

Реализации, которые поддерживают безопасные в многопоточной среде функции, определяют в заголовочном файле `<unistd.h>` символ `_POSIX_THREAD_SAFE_FUNCTIONS`. Кроме того, для проверки поддержки безопасных функций во время выполнения приложения могут вызывать функцию `sysconf` с аргумен-

том _SC_THREAD_SAFE_FUNCTIONS. Все реализации, отвечающие требованиям XSI, обязаны обеспечить поддержку безопасных функций.

Таблица 12.5. Функции, которые не гарантируют безопасность в многопоточной среде

asctime	ecvt	gethostent	getutxline	putc_unlocked
basename	encrypt	getlogin	gmtime	putchar_unlocked
catgets	endgrent	getnetbyaddr	hcreate	putenv
crypt	endpwent	getnetbyname	hdestroy	pututxline
ctime	endutxent	getnetent	hsearch	rand
dbm_clearerr	fcvt	getopt	inet_ntoa	readdir
dbm_close	ftw	getprotobyname	l64a	setenv
dbm_delete	gcvt	getprotobynumber	lgamma	setgrent
dbm_error	getc_unlocked	getprotoent	lgammaf	setkey
dbm_fetch	getchar_unlocked	getpwent	lgammal	setpwent
dbm_firstkey	getdate	getpwnam	localeconv	setutxent
dbm_nextkey	getenv	getpwuid	localtime	strerror
dbm_open	getgrent	getservbyname	lrand48	strtok
dbm_store	getgrgid	getservbyport	mrand48	ttyname
dirname	getgrnam	getservent	nftw	unsetenv
dlerror	gethostbyaddr	getutxent	nl_langinfo	wcstombs
drand48	gethostbyname	getutxid	ptsname	wctomb

При наличии поддержки функций, безопасных в многопоточной среде, реализации предоставляют альтернативные, безопасные версии некоторых небезопасных функций POSIX.1. Эти безопасные функции перечислены в табл. 12.6. Многие из функций не являются безопасными, потому что они возвращают результаты в буфере, размещенном статически. Они делаются безопасными за счет изменения интерфейса – для этого нужно, чтобы вызывающая программа предоставила свой буфер для результатов.

Функции, перечисленные в табл. 12.6, называются подобно их аналогам, небезопасным в многопоточной среде, но с добавлением последовательности _r в конце имени, что указывает на реентерабельность этих функций.

Если функция является реентерабельной по отношению к потокам, то такая функция называется безопасной в многопоточной среде. Однако это не говорит о том, что функция реентерабельна по отношению к обработчикам сигналов. Если функция может безопасно вызываться из обработчиков асинхронных сигналов, то такая функция называется безопасной в контексте обработки асинхронных сигналов. Функции, безопасные по отношению к обработчикам сигналов, перечислялись в табл. 10.3 при обсуждении реентерабельных функций (раздел 10.6).

Таблица 12.6. Альтернативные версии функций, безопасных в многопоточной среде

acstime_r	gmtime_r
ctime_r	localtime_r
getgrgid_r	rand_r
getgrnam_r	readdir_r
getlogin_r	strerror_r
getpwnam_r	strtok_r
getpwuid_r	ttyname_r

В дополнение к функциям, перечисленным в табл. 12.6, стандарт POSIX.1 определяет еще несколько функций, которые предоставляют безопасный способ управления объектами FILE в многопоточной среде. Чтобы заблокировать доступ к определенному объекту FILE, можно использовать функции flockfile и ftrylockfile. Эта блокировка является рекурсивной: вы можете повторно установить ее, не опасаясь попасть в тупиковую ситуацию. Стандарт не оговаривает точную реализацию таких блокировок, но он требует, чтобы все функции стандартной библиотеки ввода-вывода, которые работают с объектом FILE, вели себя так, как будто они обращаются к функциям flockfile и funlockfile.

```
#include <stdio.h>

int ftrylockfile(FILE *fp);
```

Возвращает 0 в случае успеха, ненулевое значение – при невозможности установки блокировки

```
void flockfile(FILE *fp);

void funlockfile(FILE *fp);
```

Хотя функции стандартной библиотеки ввода-вывода могут быть реализованы таким образом, что они будут безопасными в многопоточной среде (в смысле безопасности их собственных внутренних структур данных), тем не менее, все равно лучше выполнять блокировку доступа в самом приложении. Это позволит приложениям производить серии вызовов функций в виде атомарных последовательностей. Разумеется, при обслуживании многочисленных объектов FILE следует остерегаться потенциальных тупиковых ситуаций и очень тщательно продумывать порядок захвата блокировок.

Если функции стандартной библиотеки ввода-вывода устанавливают свои собственные блокировки, то можно столкнуться с серьезным снижением производительности при выполнении посимвольного ввода-вывода. В этой ситуации блокировка устанавливается и снимается для каждого прочитанного или записанного символа. Чтобы избежать этой проблемы, библиотека предоставляет версии функций посимвольного ввода-вывода, которые не устанавливают блокировку.

```
#include <stdio.h>

int getchar_unlocked(void);

int getc_unlocked(FILE *fp);
```

Обе возвращают следующий символ в случае успеха, EOF – в случае ошибки

```
int putchar_unlocked(int c);

int putc_unlocked(int c, FILE *fp);
```

Обе возвращают значение аргумента c в случае успеха, EOF – в случае ошибки

Эти четыре функции должны быть окружены вызовами функций `flockfile` (или `ftrylockfile`) и `funlockfile`. В противном случае можно получить непредсказуемые результаты (так же, как и в случае несинхронизированного доступа к данным из нескольких потоков).

После установки блокировки на объект FILE и до ее снятия можно производить вызовы функций ввода-вывода. Накладные расходы на установку и снятие блокировки могут в значительной степени компенсироваться объемом прочитанных или записанных данных.

Пример

В листинге 12.3 приводится пример возможной реализации функции `getenv` (раздел 7.9). Эта версия не является реентерабельной. Если произойдет одновременное обращение к функции из двух потоков, то потоки получат неверные данные, потому что возвращаемая строка сохраняется в статическом буфере, совместно используемом всеми потоками, вызывающими функцию `getenv`.

Листинг 12.3. Нереентерабельная версия функции getenv

```
#include <limits.h>
#include <string.h>

static char envbuf[ARG_MAX];
extern char **environ;

char *
getenv(const char *name)
{
    int i, len;

    len = strlen(name);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
          (environ[i][len] == '=')) {
            strcpy(envbuf, &environ[i][len+1]);
            return(envbuf);
        }
```

```
        }
        return(NULL);
    }
```

В листинге 12.4 показана реентерабельная версия функции getenv, которая называется getenv_r. Она использует функцию pthread_once (которая будет описана в разделе 12.6), чтобы гарантировать, что в ходе выполнения процесса функция thread_init будет вызвана единственный раз.

Листинг 12.4. Реентерабельная (безопасная в многопоточной среде) версия функции getenv

```
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>

extern char **environ;

pthread_mutex_t env_mutex;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;

static void
thread_init(void)
{
    pthread_mutexattr_t attr;

    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&env_mutex, &attr);
    pthread_mutexattr_destroy(&attr);
}

int
getenv_r(const char *name, char *buf, int buflen)
{
    int i, len, olen;

    pthread_once(&init_done, thread_init);
    len = strlen(name);
    pthread_mutex_lock(&env_mutex);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
          (environ[i][len] == '=')) {
            olen = strlen(&environ[i][len+1]);
            if (olen >= buflen) {
                pthread_mutex_unlock(&env_mutex);
                return(ENOSPC);
            }
            strcpy(buf, &environ[i][len+1]);
            pthread_mutex_unlock(&env_mutex);
            return(0);
        }
    }
    pthread_mutex_unlock(&env_mutex);
```

```
        return(ENOENT);
    }
```

Чтобы `getenv_r` стала реентерабельной, мы изменили ее интерфейс таким образом, что теперь вызывающая программа должна предоставить свой собственный буфер. В результате каждый поток будет использовать отдельный буфер, что исключит возможность наложения одних данных на другие. Однако этого недостаточно для того, чтобы сделать функцию `getenv_r` безопасной в многопоточной среде. Чтобы сделать ее безопасной, нужно запретить возможность изменения среды окружения на время, пока выполняется поиск запрошенной строки. Для организации доступа к списку переменных окружения из функций `getenv_r` и `putenv` можно использовать мьютекс.

В принципе вполне возможно использовать блокировки чтения-записи, чтобы разрешить одновременный вызов функции `getenv_r` из нескольких потоков, но, скорее всего, это не принесет существенной выгоды по двум причинам. Во-первых, объем среды окружения обычно не очень велик, и поэтому мьютекс во время поиска будет запираться на достаточно короткий промежуток времени. Во-вторых, вызовы функций `putenv` и `getenv` производятся очень редко, поэтому, повышая производительность этих двух функций, мы не увеличиваем производительность всего приложения.

Если мы сделаем функцию `getenv_r` безопасной в многопоточной среде, это вовсе не означает, что она станет безопасной в контексте обработчиков сигналов. При использовании нерекурсивного мьютекса есть риск возникновения тупиковой ситуации, если произойдет вызов `getenv_r` из обработчика сигнала. Если сигнал был доставлен в тот момент, когда поток находился внутри `getenv_r` и мьютекс `env_mutex` уже был заперт, то повторная попытка запереть мьютекс будет заблокирована, что приведет поток к тупиковой ситуации. Таким образом, чтобы воспрепятствовать изменению данных из других потоков и предотвратить возникновение тупиковых ситуаций в обработчиках сигналов, необходимо использовать рекурсивные мьютексы. Однако тут есть еще одна проблема, которая состоит в том, что функции библиотеки pthreads не гарантируют безопасность в контексте обработки асинхронных сигналов – таким образом, мы не можем использовать небезопасные функции для того, чтобы сделать безопасными другие функции.

12.6. Локальные данные потоков

Локальные данные потока – это механизм хранения и поиска данных, связанных только с конкретным потоком. Локальные данные потока нужны для того, чтобы каждый поток мог обладать некоторым набором данных, принадлежащих ему одному, и не беспокоиться по поводу синхронизации при работе с этими данными.

Люди приложили огромные усилия для разработки модели совместного использования ресурсов и атрибутов в многопоточных приложениях. Итак, зачем же нам нужны интерфейсы, которые препятствуют использованию этой модели? На то существуют две причины.

Первая причина состоит в том, что иногда возникает необходимость сохранять некоторые данные, специфичные для конкретного потока. Поскольку нет никакой гарантии, что идентификатор потока будет представлять собой достаточно маленькое целое число, мы не можем просто завести массив с данными для каждого потока, который индексируется идентификатором потока. Но даже если бы это было возможно, все равно не было бы никаких гарантий, что данные одного потока не будут изменены другим потоком.

Вторая причина заключается в том, что механизм организации локальных данных потока предоставляет возможность адаптации интерфейсов процессов к многопоточной среде. Типичный пример такой адаптации – переменная `errno` (раздел 1.7). Старые интерфейсы (которые были определены еще до появления концепции потоков) рассматривают `errno` как целочисленную переменную с глобальной областью видимости в пределах процесса. Системные вызовы и библиотечные функции в случае неудачи записывают в эту переменную код ошибки. Чтобы позволить потокам использовать те же самые системные вызовы и библиотечные функции, переменная `errno` была переопределена как локальная переменная потока. Таким образом, теперь, когда поток вызывает функцию, которая изменяет значение переменной `errno`, он уже не оказывает влияния на другие потоки в процессе.

Не забывайте, что все потоки в процессе имеют доступ ко всему адресному пространству процесса. И нет никакого способа предотвратить доступ к данным одного потока из другого, за исключением использования регистров процессора. Это утверждение истинно даже для локальных данных потока. Несмотря на то, что реализация в принципе не может воспрепятствовать доступу к данным, все же существуют функции для работы с локальными данными потока, которые содействуют продвижению модели с раздельными данными потоков.

Перед размещением локальных данных потока мы должны создать *ключ*, который будет идентифицировать данные. Этот ключ будет использоваться для получения доступа к локальным данным потока. Создается такой ключ вызовом функции `pthread_key_create`.

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *));
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

Созданный ключ сохраняется по адресу *keyp*. Один и тот же ключ может использоваться различными потоками в процессе, но каждый поток будет ассоциировать с ключом отдельный набор локальных данных. После создания ключа адрес локальных данных для каждого потока устанавливается равным `NULL`.

Кроме того, функция `pthread_key_create` может связать с созданным ключом функцию-деструктор. Если адрес локальных данных при завершении потока имеет ненулевое значение, то вызывается функция-деструктор, которой

в качестве аргумента передается адрес области с локальными данными потока. Если в аргументе *destructor* передается пустой указатель, это означает, что для данного ключа не предусматривается вызов деструктора. Когда поток завершает работу вызовом функции pthread_exit или возвращает управление из запускающей процедуры, вызывается деструктор. Но если поток вызывает функцию exit, _exit, _Exit, abort или завершает работу аварийно, деструктор не вызывается.

Как правило, для выделения памяти под свои локальные данные потоки используют функцию malloc. Функция-деструктор обычно освобождает эту память. Если поток завершит работу без освобождения памяти, то эта область памяти будет потеряна для процесса.

Поток может создать несколько ключей для своих данных. Каждый ключ может быть ассоциирован с деструктором. Это могут быть отдельные деструкторы для каждого из ключей или, наоборот, все ключи могут быть ассоциированы с одной и той же функцией-деструктором. Каждая реализация операционной системы может накладывать свои ограничения на количество ключей, создаваемых процессом (PTHREAD_KEYS_MAX в табл. 12.1).

Порядок вызова деструктора при завершении потока зависит от реализации. В деструкторе допускается вызов функций, которые могут создавать новые локальные данные потока и ассоциировать их с ключом. После вызова всех деструкторов система проверяет, не сохранились ли какие-либо непустые указатели на локальные данные потока, и если таковые будут обнаружены, деструкторы будут вызваны снова. Этот процесс будет повторяться снова и снова, пока не будут обнулены все указатели на локальные данные или не будет достигнуто максимально возможное количество итераций PTHREAD_DESTRUCTOR_ITERATIONS (табл. 12.1).

Мы можем разорвать связь ключа с локальными данными для всех потоков, вызвав функцию pthread_key_delete.

```
#include <pthread.h>

int pthread_key_delete(pthread_key_t *key);
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

Обратите внимание, что вызов функции pthread_key_delete не приводит к вызову деструктора, ассоциированного с ключом. Чтобы освободить память, занимаемую локальными данными потока, мы должны предусмотреть все необходимые действия в самом приложении.

Размещая новый ключ, следует побеспокоиться о том, чтобы он не был изменен в процессе инициализации из другого потока. Код, подобный приведенному ниже, может привести к тому, что функция pthread_key_create будет вызвана одновременно из нескольких потоков:

```
void destructor(void *);

pthread_key_t key;
```

```
int init_done = 0;

int
threadfunc(void *arg)
{
    if (!init_done) {
        init_done = 1;
        err = pthread_key_create(&key, destructor);
    }
    ...
}
```

В зависимости от того, как система планирует выполнение потоков, одни потоки могут увидеть одно значение ключа, другие – другое. Решение проблемы заключается в использовании функции `pthread_once`.

```
#include <pthread.h>

pthread_once_t initflag = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *initflag, void (*initfn)(void));
```

Возвращает 0 в случае успеха, код ошибки в случае неудачи

Параметр *initflag* должен быть глобальной или статической переменной, инициализированной значением `PTHREAD_ONCE_INIT`.

Система гарантирует, что функция инициализации *initfn* будет вызвана всего один раз при самом первом обращении к `pthread_once`, независимо от того, сколько раз вызывается функция `pthread_once`. Таким образом, правильный способ создания ключа выглядит следующим образом:

```
void destructor(void *);

pthread_key_t key;
pthread_once_t init_done = PTHREAD_ONCE_INIT;

void
thread_init(void)
{
    err = pthread_key_create(&key, destructor);
}

int
threadfunc(void *arg)
{
    pthread_once(&init_done, thread_init);
    ...
}
```

После того как ключ будет создан, он может быть ассоциирован с локальными данными потока с помощью функции `pthread_setspecific`. Чтобы по заданному ключу получить адрес области памяти с локальными данными потока, следует обратиться к функции `pthread_getspecific`.

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
```

Возвращает указатель на область памяти с локальными данными или NULL, если ключ не ассоциирован с локальными данными

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

Если с ключом не были ассоциированны локальные данные потока, то функция `pthread_getspecific` будет возвращать значение `NULL`. Мы можем использовать это обстоятельство, чтобы определить, следует ли вызывать функцию `pthread_setspecific`.

Пример

В листинге 12.3 мы приводили пример возможной реализации функции `getenv`. Затем, в листинге 12.4, мы продемонстрировали вариант этой же функции, который можно безопасно использовать в многопоточной среде. Но что делать, если мы не можем изменить прикладную программу так, чтобы она пользовалась новой версией функции? В подобной ситуации можно использовать локальные данные потока, в которых будет храниться буфер для возвращаемой строки. Такой подход продемонстрирован в листинге 12.5.

Листинг 12.5. Версия функции getenv, безопасная в многопоточной среде

```
#include <limits.h>
#include <string.h>
#include <pthread.h>
#include <stdlib.h>

static pthread_key_t key;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;
pthread_mutex_t env_mutex = PTHREAD_MUTEX_INITIALIZER;

extern char **environ;

static void
thread_init(void)
{
    pthread_key_create(&key, free);
}
char *
getenv(const char *name)
{
    int i, len;
    char *envbuf;

    pthread_once(&init_done, thread_init);
    pthread_mutex_lock(&env_mutex);
    envbuf = (char *)pthread_getspecific(key);
    if (envbuf == NULL) {
```

```
            envbuf = malloc(ARG_MAX);
            if (envbuf == NULL) {
                pthread_mutex_unlock(&env_mutex);
                return(NULL);
            }
            pthread_setspecific(key, envbuf);
        }
        len = strlen(name);
        for (i = 0; environ[i] != NULL; i++) {
            if ((strncmp(name, environ[i], len) == 0) &&
              (environ[i][len] == '=')) {
                strcpy(envbuf, &environ[i][len+1]);
                pthread_mutex_unlock(&env_mutex);
                return(envbuf);
            }
        }
        pthread_mutex_unlock(&env_mutex);
        return(NULL);
    }
```

Здесь мы использовали функцию `pthread_once`, чтобы гарантировать единственность ключа, который будет ассоциироваться с локальными данными потоков. Если функция `pthread_getspecific` возвращает пустой указатель, нужно разместить в памяти буфер и связать его с полученным ключом. В противном случае используется буфер, возвращаемый функцией `pthread_getspecific`. В деструкторе мы вызываем функцию `free`, которая освобождает память, занимаемую буфером. Деструктор будет вызван только в том случае, если поток связал указатель на локальные данные с ключом и этот указатель не является пустым.

Обратите внимание: хотя эта версия функции `getenv` и может безопасно использоваться в многопоточной среде, она не безопасна в контексте обработки асинхронных сигналов. Даже если сделать мьютекс рекурсивным, это не гарантирует ее безопасное использование в обработчиках сигналов, потому что она вызывает функцию `malloc`, которая сама не является безопасной в контексте обработки сигналов.

12.7. Принудительное завершение потоков

Два атрибута потоков, которые не входят в состав структуры `pthread_attr_t`, — это атрибут возможности принудительного завершения потока (*cancelability state*) и атрибут типа принудительного завершения (*cancelability type*). Эти атрибуты определяют поведение потока в ответ на вызов функции `pthread_cancel` (раздел 11.5).

Атрибут *cancelability state* может иметь значение `PTHREAD_CANCEL_ENABLE` или `PTHREAD_CANCEL_DISABLE`. Поток может изменить значение этого атрибута вызовом функции `pthread_setcancelstate`.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

В одной атомарной операции функция `pthread_setcancelstate` изменяет значение атрибута *cancelability state* в соответствии со значением аргумента *state* и сохраняет прежнее значение атрибута по адресу, который передается в аргументе *oldstate*.

В разделе 11.5 мы уже говорили, что функция `pthread_cancel` не ждет, пока поток завершит работу. По умолчанию поток продолжает работу после вызова этой функции, пока не достигнет *точки выхода*. Точка выхода – это место, где поток может обнаружить запрос на принудительное завершение и откликнуться на него. Стандарт POSIX.1 назначает точками выхода функции, перечисленные в табл. 12.7.

Таблица 12.7. Точки выхода, определяемые стандартом POSIX.1

accept	mq_timedsend	putpmsg	sigsuspend
aio_suspend	msgrcv	pwrite	sigtimedwait
clock_nanosleep	msgsnd	read	sigwait
close	msync	readv	sigwaitinfo
connect	nanosleep	recv	sleep
creat	open	recvfrom	system
fcntl2	pause	recvmsg	tcdrain
fsync	poll	select	usleep
getmsg	pread	sem_timedwait	wait
getpmsg	pthread_cond_timedwait	sem_wait	waitid
lockf	pthread_cond_wait	send	waitpid
mq_receive	pthread_join	sendmsg	write
mq_send	pthread_testcancel	sendto	writev
mq_timedreceive	putmsg	sigpause	

В момент запуска потока значение его атрибута *cancelability state* устанавливается равным `PTHREAD_CANCEL_ENABLE`. Если поток установит значение этого атрибута равным `PTHREAD_CANCEL_DISABLE`, то вызов функции `pthread_cancel` не будет приводить к завершению потока. Вместо этого запрос на принудительное завершение становится в режим ожидания. Когда поток опять разрешит возможность принудительного завершения, он откликнется на ожидающий запрос в ближайшей точке выхода.

В дополнение к функциям, перечисленным в табл. 12.7, стандарт POSIX.1 определяет еще ряд функций (табл. 12.8), которые могут служить точками выхода.

> Обратите внимание: некоторые из функций, перечисленных в табл. 12.8, не обсуждаются в данной книге. Стандарт Single UNIX Specification определяет многие из них как необязательные для реализации.

Таблица 12.8. Дополнительные точки выхода, определяемые стандартом POSIX.1

catclose	ftell	getwc	printf
catgets	ftello	getwchar	putc
catopen	ftw	getwd	putc_unlocked
closedir	fwprintf	glob	putchar
closelog	fwrite	iconv_close	putchar_unlocked
ctermid	fwscanf	iconv_open	puts
dbm_close	getc	ioctl	pututxline
dbm_delete	getc_unlocked	lseek	putwc
dbm_fetch	getchar	mkstemp	putwchar
dbm_nextkey	getchar_unlocked	nftw	readdir
dbm_open	getcwd	opendir	readdir_r
dbm_store	getdate	openlog	remove
dlclose	getgrent	pclose	rename
dlopen	getgrgid	perror	rewind
endgrent	getgrgid_r	popen	rewinddir
endhostent	getgrnam	posix_fadvise	scanf
endnetent	getgrnam_r	posix_fallocate	seekdir
endprotoent	gethostbyaddr	posix_madvise	semop
endpwent	gethostbyname	posix_spawn	setgrent
endservent	gethostent	posix_spawnp	sethostent
endutxent	gethostname	posix_trace_clear	setnetent
fclose	getlogin	posix_trace_close	setprotoent
fcntl	getlogin_r	posix_trace_create	setpwent
fflush	getnetbyaddr	posix_trace_create_withlog	setservent
fgetc	getnetbyname	posix_trace_eventtypelist_getnext_id	setutxent
fgetpos	getnetent	posix_trace_eventtypelist_rewind	strerror
fgets	getprotobyname	posix_trace_flush	syslog
fgetwc	getprotobynumber	posix_trace_get_attr	tmpfile

fgetws	getprotoent	posix_trace_get_filter	tmpnam
fopen	getpwent	posix_trace_get_status	ttyname
fprintf	getpwnam	posix_trace_getnext_event	ttyname_r
fputc	getpwnam_r	posix_trace_open	ungetc
fputs	getpwuid	posix_trace_rewind	ungetwc
fputwc	getpwuid_r	posix_trace_set_filter	unlink
fputws	gets	posix_trace_shutdown	vfprintf
fread	getservbyname	posix_trace_timedgetnext_event	vfwprintf
freopen	getservbyport	posix_typed_mem_open	vprintf
fscanf	getservent	pthread_rwlock_rdlock	vwprintf
fseek	getutxent	pthread_rwlock_timedrdlock	wprintf
fseeko	getutxid	pthread_rwlock_timedwrlock	wscanf
fsetpos	getutxline	pthread_rwlock_wrlock	

Если поток не обращается к функциям, перечисленным в табл. 12.7 и 12.8, достаточно продолжительное время (например, при выполнении объемных вычислений), то вы можете определить свою собственную точку выхода с помощью функции `pthread_testcancel`.

```
#include <pthread.h>
void pthread_testcancel(void);
```

Функция `pthread_testcancel` проверяет наличие ожидающего запроса на принудительное завершение, и если таковой имеется и при этом атрибут *cancelability state* разрешает принудительное завершение, то поток завершит свою работу. Но если возможность принудительного завершения потока запрещена, вызов функции `pthread_testcancel` не оказывает никакого влияния.

По умолчанию для потока устанавливается тип принудительного завершения, известный как *отложенный выход*. После вызова функции `pthread_cancel` поток не завершается немедленно, он продолжает работу до тех пор, пока не достигнет ближайшей точки выхода. Изменить тип принудительного завершения можно с помощью функции `pthread_setcanceltype`.

```
#include <pthread.h>
int pthread_setcanceltype(int type, int *oldtype);
```

Возвращает 0 в случае успеха, код ошибки в случае неудачи

Аргумент *type* может содержать либо значение `PTHREAD_CANCEL_DEFERRED`, либо `PTHREAD_CANCEL_ASYNCHRONOUS`. Функция `pthread_setcanceltype` устанавливает значение атрибута в соответствии с аргументом *type* и возвращает предыдущее значение атрибута в переменной, на которую указывает аргумент *oldtype*.

Асинхронное завершение потока отличается от отложенного тем, что поток может быть принудительно завершен в любой момент времени. В этом случае поток будет завершен вне зависимости от того, достиг он точки выхода или нет.

12.8. Потоки и сигналы

Взаимодействие с сигналами может быть весьма сложным даже в однопоточных приложениях. Наличие нескольких потоков еще более запутывает дело.

Каждый поток имеет свою собственную маску сигналов, но диспозиция сигналов одна для всех потоков в процессе. Это означает, что каждый отдельно взятый поток может заблокировать доставку сигнала, но когда поток назначает определенное действие для сигнала, оно становится общим для всех потоков. Таким образом, если один поток установил диспозицию сигнала так, чтобы он игнорировался, то другой поток может отменить это действие, установив диспозицию сигнала в значение по умолчанию или назначив обработчик сигнала.

Сигналы доставляются только одному потоку в процессе. Если сигнал связан с аппаратной ошибкой или истечением таймера, то он доставляется потоку, который стал причиной этого сигнала. Однако остальные сигналы доставляются любому произвольному потоку.

В разделе 10.12 мы рассказывали, как можно использовать функцию `sigprocmask` для блокировки сигналов. Поведение функции `sigprocmask` в многопоточной среде не определено. Вместо нее потоки должны использовать функцию `pthread_sigmask`.

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

Функция `pthread_sigmask` идентична функции `sigprocmask`, за исключением того, что она предназначена для работы в многопоточной среде и в случае ошибки возвращает не значение –1 с записью кода ошибки в `errno`, а сам код ошибки.

Поток может приостановиться в ожидании доставки сигнала, вызвав функцию `sigwait`.

```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict signop);
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

Аргумент *set* определяет набор сигналов, доставка которых ожидается. По возвращении из функции по адресу *signop* будет записан номер доставленного сигнала.

Если какой-либо сигнал, который входит в набор *set*, к моменту вызова sigwait ожидает обработки, то функция вернет управление немедленно. Перед возвратом управления sigwait удалит сигнал из набора сигналов, ожидающих обработки. Во избежание ошибочной реакции на сигнал поток должен заблокировать ожидаемые сигналы перед вызовом sigwait. Функция sigwait атомарно разблокирует сигналы и перейдет в режим ожидания, пока заданные сигналы не будут доставлены. Перед возвратом управления sigwait восстановит маску сигналов потока. Если сигнал не будет заблокирован к моменту вызова функции, то возникнет промежуток времени, когда сигнал может быть доставлен потоку еще до того, как он выполнит вызов sigwait.

Преимущество использования функции sigwait заключается в том, что она позволяет упростить обработку сигналов и обрабатывать асинхронные сигналы на синхронный манер. Чтобы воспрепятствовать прерыванию выполнения потока по сигналу, можно добавить требуемые сигналы к маске сигналов каждого потока. Благодаря этому мы можем выделить отдельные потоки, которые будут заниматься только обработкой сигналов. Эти специально выделенные потоки могут обращаться к любым функциям, которые нельзя использовать в обработчиках сигналов, потому что в этой ситуации функции будут вызываться в контексте обычного потока, а не из традиционного обработчика сигнала, который прерывает работу потока.

Если сразу несколько потоков окажутся заблокированными в функции sigwait в ожидании одного и того же сигнала, то только в одном из них функция sigwait вернет управление, когда сигнал будет доставлен процессу. Если сигнал перехватывается процессом (например, когда процесс установил обработчик сигнала с помощью функции sigaction), и при этом поток, обратившийся к функции sigwait, ожидает доставки того же самого сигнала, то принятие решения о способе доставки сигнала оставляется на усмотрение реализации. Операционная система в этом случае может либо вызвать установленный обработчик сигнала, либо позволить функции sigwait вернуть управление в поток, но никак не то и другое вместе.

Для посылки сигнала процессу используется функция kill (раздел 10.9). Для посылки сигнала потоку используется функция pthread_kill.

```
#include <signal.h>

int pthread_kill(pthread_t thread, int signo);
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

Можно проверить существование потока, передав в аргументе *signo* значение 0. Если действие по умолчанию для сигнала заключается в завершении процесса, то передача такого сигнала потоку приведет к завершению всего процесса.

Обратите внимание: таймеры являются ресурсами процесса, и все потоки совместно используют один и тот же набор таймеров. Следовательно, в случае многопоточного приложения невозможно использовать таймеры в одном потоке, не оказывая влияние на другие (это тема упражнения 12.6).

Пример

В программе, представленной листингом 10.16, мы приостанавливали работу процесса до тех пор, пока обработчик сигнала не установит флаг, который указывает, что процесс должен завершить работу. Единственными потоками управления в этой программе были главный поток программы и обработчик сигнала, таким образом, блокировка сигнала была надежным средством, которое не позволяло пропустить получение сигнала и изменение флага. В случае многопоточного приложения мы вынуждены защищать доступ к флагу с помощью мьютекса, что показано в листинге 12.6.

Листинг 12.6. Синхронная обработка сигнала

```
#include "apue.h"
#include <pthread.h>

int quitflag;                   /* поток записывает сюда ненулевое значение */
sigset_t mask;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t wait = PTHREAD_COND_INITIALIZER;

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0)
            err_exit(err, "ошибка вызова функции sigwait");
        switch (signo) {
        case SIGINT:
            printf("\nпрерывание\n");
            break;

        case SIGQUIT:
            pthread_mutex_lock(&lock);
            quitflag = 1;
            pthread_mutex_unlock(&lock);
            pthread_cond_signal(&wait);
            return(0);

        default:
            printf("получен непредвиденный сигнал %d\n", signo);
            exit(1);
        }
    }
}
```

```
int
main(void)
{
    int err;
    sigset_t oldmask;
    pthread_t tid;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, &oldmask)) != 0)
        err_exit(err, "ошибка выполнения операции SIG_BLOCK");
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "невозможно создать поток");
    pthread_mutex_lock(&lock);
    while (quitflag == 0)
        pthread_cond_wait(&wait, &lock);
    pthread_mutex_unlock(&lock);

    /*
     * Сигнал SIGQUIT был перехвачен и к настоящему моменту
     * опять заблокирован.
     */
    quitflag = 0;

     /* Восстановить маску сигналов, в которой SIGQUIT разблокирован. */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка выполнения операции SIG_SETMASK");
    exit(0);
}
```

Вместо того чтобы обрабатывать сигнал в функции-обработчике, прерывающей выполнение главного потока, мы создали для этого отдельный поток. Изменение флага `quitflag` производится под защитой мьютекса, чтобы главный поток не смог пропустить изменение значения флага, когда поток-обработчик вызывает функцию `pthread_cond_signal`. Тот же самый мьютекс используется в главном потоке для контроля состояния флага, мы атомарно освобождаем его и ожидаем наступления события.

Обратите внимание: сигналы `SIGINT` и `SIGQUIT` блокируются в самом начале главного потока программы. Когда создается поток, который будет обрабатывать доставку сигналов, он наследует текущую маску сигналов. Поскольку функция `sigwait` разблокирует сигналы, только один поток сможет получить их. Это позволяет при написании программы не беспокоиться о том, что главный поток может быть прерван этими сигналами.

Запустив эту программу, мы получим результаты, похожие на те, что дала нам программа из листинга 10.16:

```
$ ./a.out
Ъ?              введем символ прерывания
прерывание
```

```
Б?                  еще раз введем символ прерывания
прерывание
Б?                  и еще раз
прерывание
Б\ $                а теперь введем символ завершения
```

ОС Linux реализует потоки в виде отдельных процессов, совместно использующих одни и те же ресурсы, с помощью системного вызова `clone`(2). По этой причине в Linux поведение потоков по отношению к сигналам отличается от того, что мы наблюдаем в других операционных системах. В соответствии со стандартом POSIX.1 асинхронные сигналы посылаются процессу, после чего внутри процесса выбирается конкретный поток, которому будет доставлен сигнал, исходя из того, в каких потоках к заданному моменту сигнал оказался незаблокированным. В ОС Linux сигналы посылаются конкретному потоку, а так как каждый поток оформлен в виде отдельного процесса, операционная система не в состоянии выбрать поток, для которого сигнал не заблокирован. В результате поток может не заметить сигнал. Таким образом, программа из листинга 12.6 будет корректно работать, если сигнал поступает от драйвера терминала, потому что в этом случае сигнал посылается группе процессов. Но если вы попытаетесь послать процессу сигнал с помощью функции `kill`, то в Linux эта программа будет работать не так, как вы можете того ожидать.

12.9. Потоки и fork

Когда поток вызывает функцию `fork`, создается копия всего адресного пространства процесса. В разделе 8.3 мы уже говорили о технике копирования при записи. Дочерний процесс – это совершенно иной процесс, отличный от родительского, и пока ни один из них не изменяет никаких данных, они совместно используют одно и то же адресное пространство. Наследуя адресное пространство, дочерний процесс наследует и состояние каждого из мьютексов, блокировок чтения-записи и переменных состояния. Если родительский процесс состоит более чем из одного потока, то дочерний процесс должен сбросить состояние блокировки, если он не собирается немедленно вызвать функцию `exec`.

Внутри дочернего процесса существует только один поток. Он представляет собой копию потока в родительском процессе, вызвавшего функцию `fork`. Если в родительском процессе поток владел какими-либо блокировками, то этими же блокировками будет владеть и дочерний процесс. Проблема состоит в том, что дочерний процесс не имеет копий других потоков, которые также могут удерживать блокировки, поэтому у дочернего процесса нет возможности узнать, какие блокировки установлены и какие из них следует освободить.

Этой проблемы можно избежать, если дочерний процесс сразу же после выхода из функции `fork` вызывает функцию `exec`. В этом случае старое адресное пространство исчезает, и потому состояние имеющихся блокировок не имеет никакого значения. Однако это не всегда возможно: бывает так, что дочерний процесс должен продолжить обработку данных, поэтому приходится использовать иную стратегию.

Чтобы сбросить состояние блокировок, доставшихся в наследство от родительского процесса, можно установить *обработчик операции ветвления* вызовом функции pthread_atfork.

```
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
                   void (*child)(void));
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

С помощью pthread_atfork можно установить до трех функций, которые служат для сброса блокировок. Функция *prepare* вызывается в родительском процессе перед созданием дочернего процесса вызовом fork. Этот обработчик должен установить все блокировки, которые имеются в родительском процессе. Функция *parent* вызывается в контексте родительского процесса после создания дочернего процесса, но до того, как fork вернет управление. Назначение этой функции состоит в снятии всех блокировок, установленных в функции *prepare*. Функция *child* вызывается в контексте дочернего процесса до того, как fork вернет управление. Подобно функции *parent*, функция *child* также должна освободить все блокировки, установленные в функции *prepare*.

Обратите внимание: здесь не происходит двойного снятия блокировок, установленных один раз, как может показаться на первый взгляд. Когда создается адресное пространство дочернего процесса, в нем находятся копии всех блокировок, определенных в родительском процессе. Поскольку обработчик *prepare* установил все блокировки, содержимое памяти в родительском и в дочернем процессах будет полностью идентично. Когда родитель и потомок разблокируют свои копии блокировок, для дочернего процесса выделяется новая область памяти, и содержимое памяти родительского процесса копируется в адресное пространство потомка (копирование при записи). Таким образом, ситуация выглядит так, как будто родительский процесс заблокировал свои копии блокировок, а дочерний процесс – свои. И родительский, и дочерний процессы снимают блокировки, расположенные в различных адресных пространствах, как если бы выполнялась следующая последовательность действий:

1. Родительский процесс установил все блокировки.
2. Дочерний процесс установил все блокировки.
3. Родительский процесс освободил все блокировки.
4. Дочерний процесс освободил все блокировки.

Мы можем вызвать функцию pthread_atfork много раз, чтобы установить несколько наборов обработчиков процедуры ветвления. Если потребность в каком-либо из трех обработчиков отсутствует, в соответствующем аргументе можно передать пустой указатель. Когда назначается несколько наборов обработчиков, порядок их вызова изменяется. Функции *parent* и *child* вызываются в том порядке, в каком они были зарегистрированы, тогда как функ-

ции *prepare* вызываются в противоположном порядке. Это позволяет различным модулям устанавливать свои обработчики процедуры ветвления и сохранять при этом иерархию блокировок.

Например, предположим, что модуль A вызывает функции из модуля B и при этом каждый модуль имеет свой собственный набор блокировок. Если в соответствии с алгоритмом модуль A должен установить блокировки раньше модуля B, то модуль B должен первым установить обработчик процедуры ветвления. После того, как родительский процесс вызвал функцию `fork`, действия будут развиваться в следующей последовательности, если предположить, что дочерний процесс первым получит управление:

1. Будет вызвана функция *prepare* модуля A, которая установит все блокировки модуля A.
2. Будет вызвана функция *prepare* модуля B, которая установит все блокировки модуля B.
3. Будет создан дочерний процесс.
4. Функция *child* модуля B освободит все блокировки модуля B в дочернем процессе.
5. Функция *child* модуля A освободит все блокировки модуля A в дочернем процессе.
6. Функция `fork` вернет управление в дочернем процессе.
7. Функция *parent* модуля B освободит все блокировки модуля B в родительском процессе.
8. Функция *parent* модуля A освободит все блокировки модуля A в родительском процессе.
9. Функция `fork` вернет управление в родительском процессе.

Если обработчики процедуры ветвления предназначены для сброса блокировок, то как можно сбросить переменные состояния? В некоторых реализациях не требуется сбрасывать переменные состояния. Однако в реализациях, которые используют блокировки как составную часть переменных состояния, необходимо сбрасывать эти блокировки. Проблема в том, что не существует интерфейсов, которые позволяют сделать это. Если в структуре переменной состояния присутствует блокировка, то мы не сможем воспользоваться этой переменной после вызова функции `fork`, потому что не существует достаточно переносимого способа сбросить эту блокировку. С другой стороны, если для защиты переменных состояния реализация использует глобальные блокировки в процессе, то операционная система сама сбросит их в функции `fork`. Однако прикладные программы не должны полагаться на подобное поведение реализации.

Пример

Программа, представленная листингом 12.7, демонстрирует использование функции `pthread_atfork` и обработчиков процедуры ветвления.

Листинг 12.7. Пример использования функции pthread_atfork

```
#include "apue.h"
#include <pthread.h>

pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

void
prepare(void)
{
    printf("подготовка блокировок...\n");
    pthread_mutex_lock(&lock1);
    pthread_mutex_lock(&lock2);
}

void
parent(void)
{
    printf("родитель снимает блокировки...\n");
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
}

void
child(void)
{
    printf("потомок снимает блокировки...\n");
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
}

void *
thr_fn(void *arg)
{
    printf("поток запущен...\n");
    pause();
    return(0);
}

int
main(void)
{
    int err;
    pid_t pid;
    pthread_t tid;

#if defined(BSD) || defined(MACOS)
    printf("функция pthread_atfork не поддерживается\n");
#else
    if ((err = pthread_atfork(prepare, parent, child)) != 0)
       err_exit(err, "невозможно установить обработчики процедуры ветвления");
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
```

```
        err_exit(err, "невозможно создать поток");
    sleep(2);
    printf("родительский процесс вызывает fork...\n");
    if ((pid = fork()) < 0)
        err_quit("ошибка вызова функции fork");
    else if (pid == 0)                          /* дочерний процесс */
        printf("функция fork вернула управление в дочерний процесс\n");
    else                                        /* родительский процесс */
        printf("функция fork вернула управление в родительский процесс \n");
#endif
    exit(0);
}
```

Здесь мы определили два мьютекса, `lock1` и `lock2`. Функция *prepare* захватывает их оба, функция *child* освобождает их в контексте дочернего процесса, а *parent* – в контексте родительского процесса.

Запуск программы дал следующие результаты:

```
$ ./a.out
поток запущен...
родительский процесс вызывает fork...
подготовка блокировок...
потомок снимает блокировки...
функция fork вернула управление в дочерний процесс
родитель снимает блокировки...
функция fork вернула управление в родительский процесс
```

Как видите, функция *prepare* вызывается после вызова функции `fork`, функция *child* запускается перед выходом из функции `fork` в дочерний процесс, а функция *parent* – перед выходом из функции `fork` в родительский процесс.

12.10. Потоки и операции ввода-вывода

В разделе 3.11 мы говорили о функциях `pread` и `pwrite`. Эти функции удобно использовать в многопоточной среде, потому что все потоки в процессе совместно используют одни и те же файловые дескрипторы.

Рассмотрим два потока, которые одновременно работают с одним и тем же файловым дескриптором.

```
        Поток A                         Поток B
    lseek(fd, 300, SEEK_SET);       lseek(fd, 700, SEEK_SET);
    read(fd, buf1, 100);            read(fd, buf2, 100);
```

Если поток A выполнит вызов функции `lseek`, а затем поток B также вызовет функцию `lseek` до того, как поток A успеет вызвать функцию `read`, то оба потока прочитают одну и ту же запись. Понятно, что это совсем не то, что нам нужно.

Чтобы решить эту проблему, можно использовать функцию `pread`, которая устанавливает текущую позицию файла и производит чтение данных атомарно.

```
Поток A                          Поток B
pread(fd, buf1, 100, 300);    pread(fd, buf2, 100, 700);
```

Используя функцию `pread`, мы можем быть уверены, что поток A прочитает запись, начиная со смещения 300, а поток B – со смещения 700. Для решения аналогичной проблемы, связанной с записью в один и тот же файл, можно использовать функцию `pwrite`.

12.11. Подведение итогов

Потоки в системе UNIX предоставляют альтернативную модель разбиения крупных задач на подзадачи. Потоки следуют модели совместного использования одних и тех же данных, что в свою очередь порождает специфические проблемы синхронизации. В этой главе мы рассмотрели вопросы, связанные с настройкой поведения потоков, и примитивы синхронизации. Мы обсудили вопрос реентерабельности относительно потоков. И кроме того, увидели, как потоки взаимодействуют с некоторыми системными вызовами.

Упражнения

12.1. Запустите программу из листинга 12.7 в ОС Linux, перенаправив вывод в файл. Объясните полученные результаты.

12.2. Реализуйте функцию `putenv_r` – реентерабельную версию функции `putenv`. Убедитесь, что ваша версия функции безопасна как в контексте обработки асинхронных сигналов, так и в многопоточной среде.

12.3. Возможно ли функцию из листинга 12.5 сделать безопасной в контексте обработки сигналов, блокируя доставку сигнала в начале функции и восстанавливая предыдущую маску сигналов перед возвратом из нее? Объясните почему.

12.4. Напишите программу для проверки версии функции `getenv` из листинга 12.5. Скомпилируйте и запустите программу в ОС FreeBSD. Объясните, что произошло.

12.5. Если существует возможность создавать многочисленные потоки для решения разнообразных задач в рамках одного процесса, объясните, зачем в этих условиях может понадобиться функция `fork`.

12.6. Измените реализацию программы из листинга 10.21 так, чтобы она стала безопасной в многопоточной среде, не используя функцию `nanosleep`.

12.7. Возможно ли в дочернем процессе после возврата из функции `fork` безопасно переинициализировать переменные состояния путем их разрушения функцией `pthread_cond_destroy` и последующей инициализации функцией `pthread_cond_init`?

13

Процессы-демоны

13.1. Введение

Демоны – это долгоживущие процессы. Зачастую они запускаются во время загрузки системы и завершают работу вместе с ней. Так как они не имеют управляющего терминала, говорят, что они работают в фоновом режиме. В системе UNIX демоны решают множество повседневных задач.

В этой главе мы рассмотрим структуру процессов-демонов и покажем, как они создаются. Так как демоны не имеют управляющего терминала, нам необходимо будет выяснить, как демон может вывести сообщение об ошибке, если что-то идет не так, как надо.

> Обсуждение истории термина *демон* применительно к компьютерным системам вы найдете в [Raymond 1996].

13.2. Характеристики демонов

Давайте рассмотрим некоторые наиболее распространенные системные демоны и их отношения с группами процессов, управляющими терминалами и сессиями. Команда `ps`(1) выводит информацию о процессах в системе. Эта команда имеет множество опций, дополнительную информацию о них вы найдете в справочном руководстве. Мы запустим команду

```
ps -axj
```

под управлением BSD-системы и будем использовать полученную от нее информацию при дальнейшем обсуждении. Ключ `-a` используется для вывода процессов, которыми владеют другие пользователи, ключ `-x` – для вывода процессов, не имеющих управляющего терминала, и ключ `-j` – для вывода дополнительных сведений, имеющих отношение к заданиям: идентификатора сессии, идентификатора группы процессов, управляющего терминала и идентификатора группы процессов терминала.

Для систем, основанных на System V, аналогичная команда выглядит как `ps -efjc`. (В целях безопасности некоторые версии UNIX не допускают просмотр процессов, принадлежащих другим пользователям, с помощью команды `ps`.) Вывод команды `ps` выглядит примерно следующим образом:

```
 PPID   PID  PGID   SID TTY TPGID UID   COMMAND
    0     1     0     0 ?      -1   0   init
    1     2     1     1 ?      -1   0   [keventd]
    1     3     1     1 ?      -1   0   [kapmd]
    0     5     1     1 ?      -1   0   [kswapd]
    0     6     1     1 ?      -1   0   [bdflush]
    0     7     1     1 ?      -1   0   [kupdated]
    1  1009  1009  1009 ?      -1  32   portmap
    1  1048  1048  1048 ?      -1   0   syslogd -m 0
    1  1335  1335  1335 ?      -1   0   xinetd -pidfile /var/run/xinetd.pid
    1  1403     1     1 ?      -1   0   [nfsd]
    1  1405     1     1 ?      -1   0   [lockd]
 1405  1406     1     1 ?      -1   0   [rpciod]
    1  1853  1853  1853 ?      -1   0   crond
    1  2182  2182  2182 ?      -1   0   /usr/sbin/cupsd
```

Из данного примера мы убрали несколько колонок, которые не представляют для нас особого интереса. Здесь показаны следующие колонки, слева направо: идентификатор родительского процесса, идентификатор процесса, идентификатор группы процессов, идентификатор сессии, имя терминала, идентификатор группы процессов терминала (группы процессов переднего плана, связанной с управляющим терминалом), идентификатор пользователя и строка команды.

> Система, на которой была запущена эта команда (Linux), поддерживает понятие идентификатора сессии, который мы упоминали при обсуждении функции `setsid` в разделе 9.5. Идентификатор сессии – это просто идентификатор процесса лидера сессии. Однако в системах, основанных на BSD, будет выведен адрес структуры `session`, соответствующей группе процессов, которой принадлежит данный процесс (раздел 9.11).

Перечень системных процессов, который вы увидите, во многом зависит от реализации операционной системы. Обычно это будут процессы с идентификатором родительского процесса 0, запускаемые ядром в процессе загрузки системы. (Исключением является процесс `init`, так как это команда уровня пользователя, которая запускается ядром во время загрузки.) Процессы ядра – это особые процессы, они существуют все время, пока работает система. Эти процессы обладают привилегиями суперпользователя и не имеют ни управляющего терминала, ни строки команды.

Процесс с идентификатором 1 – это обычно процесс `init`, о чем уже говорилось в разделе 8.2. Это системный демон, который, кроме всего прочего, отвечает за запуск различных системных служб на различных уровнях загрузки. Как правило, эти службы также реализованы в виде демонов.

В ОС Linux демон `keventd` предоставляет контекст процесса для запуска задач из очереди планировщика. Демон `kapmd` обеспечивает поддержку расши-

ренного управления питанием, которое доступно в некоторых компьютерных системах. Демон kswapd известен также как демон выгрузки страниц. Этот демон поддерживает подсистему виртуальной памяти, в фоновом режиме записывая на диск страницы, измененные со времени их чтения с диска (dirty pages), благодаря чему они могут быть использованы снова.

Сбрасывание кэшированных данных на диск в ОС Linux производится с помощью двух дополнительных демонов – bdflush и kupdated. Демон bdflush начинает сбрасывать измененные буферы из кэша на диск, когда объем свободной памяти уменьшается до определенного уровня. Демон kupdated сбрасывает измененные буферы из кэша на диск через регулярные интервалы времени, снижая тем самым риск потери данных в случае краха системы.

Демон portmap осуществляет преобразование номеров программ RPC (Remote Procedure Call – удаленный вызов процедур) в номера сетевых портов. Демон syslogd может использоваться программами для вывода системных сообщений в журнал для просмотра оператором. Сообщения могут выводиться на консоль, а также записываться в файл. (Более подробно syslogd рассматривается в разделе 13.4.)

В разделе 9.3 мы уже говорили о демоне inetd (xinetd). Этот демон ожидает поступления из сети запросов к различным сетевым серверам. Демоны nfsd, lockd и rpciod обеспечивают поддержку сетевой файловой системы (NFS – Network File System).

Демон cron (crond) производит запуск команд в определенное время. Множество административных задач выполняется благодаря регулярному запуску программ с помощью демона cron. Демон cupsd – это сервер печати, он обслуживает запросы к принтеру.

Обратите внимание: большинство демонов обладают привилегиями суперпользователя (имеют идентификатор пользователя 0). Ни один из демонов не имеет управляющего терминала – вместо имени терминала стоит знак вопроса, а идентификатор группы переднего плана равен –1. Демоны ядра запускаются без управляющего терминала. Отсутствие управляющего терминала у демонов пользовательского уровня – вероятно, результат вызова функции setsid. Все демоны пользовательского уровня являются лидерами групп и лидерами сессий, и они являются единственными процессами в своих группах процессов и сессиях. И, наконец, обратите внимание на то, что родительским для большинства демонов является процесс init.

13.3. Правила программирования демонов

При программировании демонов во избежание нежелательных взаимодействий следует придерживаться определенных правил. Сначала мы перечислим эти правила, а затем продемонстрируем функцию daemonize, которая их реализует.

1. Прежде всего нужно вызвать функцию umask, чтобы сбросить маску режима создания файлов в значение 0. Маска режима создания файлов, насле-

дуемая от запускающего процесса, может маскировать некоторые биты прав доступа. Если предполагается, что процесс-демон будет создавать файлы, может потребоваться установка определенных битов прав доступа. Например, если демон создает файлы с правом на чтение и на запись для группы, маска режима создания файла, которая выключает любой из этих битов, воспрепятствовала бы этому.

2. Вызвать функцию `fork` и завершить родительский процесс. Для чего это делается? Во-первых, если демон был запущен как обычная команда оболочки, то завершив родительский процесс, мы заставим командную оболочку думать, что команда была выполнена. Во-вторых, дочерний процесс наследует идентификатор группы процессов от родителя, но получает свой идентификатор процесса; таким образом мы гарантируем, что дочерний процесс не будет являться лидером группы, а это необходимое условие для вызова функции `setsid`, который будет произведен далее.

3. Создать новую сессию, обратившись к функции `setsid`. При этом (вспомните раздел 9.5) процесс становится (а) лидером новой сессии, (б) лидером новой группы процессов и (в) лишается управляющего терминала.

   Для систем, основанных на System V, некоторые специалисты рекомендуют в этой точке повторно вызвать функцию `fork` и завершить родительский процесс, чтобы второй потомок продолжал работу в качестве демона. Такой прием гарантирует, что демон не будет являться лидером сессии, и это препятствует получению управляющего терминала в System V (раздел 9.6). Как вариант, чтобы избежать обретения управляющего терминала, при любом открытии терминального устройства следует указывать флаг `O_NOCTTY`.

4. Сделать корневой каталог текущим рабочим каталогом. Текущий рабочий каталог, унаследованный от родительского процесса, может находиться на смонтированной файловой системе. Поскольку демон, как правило, существует все время, пока система не будет перезагружена, то в подобной ситуации, когда рабочий каталог демона находится в смонтированной файловой системе, ее невозможно будет отмонтировать.

   Как вариант, некоторые демоны могут устанавливать собственный текущий рабочий каталог, в котором они производят все необходимые действия. Например, демоны печати в качестве текущего рабочего каталога часто выбирают буферный каталог, куда помещаются задания для печати.

5. Закрыть все ненужные файловые дескрипторы. Это предотвращает удержание в открытом состоянии некоторых дескрипторов, унаследованных от родительского процесса (командной оболочки или другого процесса). С помощью нашей функции `open_max` (листинг 2.4) или с помощью функции `getrlimit` (раздел 7.11) можно определить максимально возможный номер дескриптора и закрыть все дескрипторы вплоть до этого номера.

6. Некоторые демоны открывают файловые дескрипторы с номерами 0, 1 и 2 на устройстве `/dev/null` – таким образом, любые библиотечные функции, которые пытаются читать со стандартного устройства ввода или писать на стандартное устройство вывода или сообщений об ошибках, не будут ока-

зывать никакого влияния. Поскольку демон не связан ни с одним терминальным устройством, он не сможет взаимодействовать с пользователем в интерактивном режиме. Даже если демон изначально был запущен в рамках интерактивной сессии, он все равно переходит в фоновый режим, и начальная сессия может завершиться без воздействия на процесс-демон. С этого же терминала в систему могут входить другие пользователи, и демон не должен выводить какую-либо информацию на терминал, да и пользователи не ждут того, что их ввод с терминала будет прочитан демоном.

Пример

В листинге 13.1 приводится функция, которую может вызывать приложение, желающее стать демоном.

Листинг 13.1. Инициализация процесса-демона

```
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>

void
daemonize(const char *cmd)
{
    int i, fd0, fd1, fd2;
    pid_t pid;
    struct rlimit rl;
    struct sigaction sa;

    /*
     * Сбросить маску режима создания файла.
     */
    umask(0);

    /*
     * Получить максимально возможный номер дескриптора файла.
     */
    if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
        err_quit("%s: невозможно получить максимальный номер дескриптора ",
                 cmd);

    /*
     * Стать лидером новой сессии, чтобы утратить управляющий терминал.
     */
    if ((pid = fork()) < 0)
        err_quit("%s: ошибка вызова функции fork", cmd);
    else if (pid != 0) /* родительский процесс */
        exit(0);
    setsid();

    /*
     * Обеспечить невозможность обретения управляющего терминала в будущем.
     */
```

```
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: невозможно игнорировать сигнал SIGHUP");
    if ((pid = fork()) < 0)
        err_quit("%s: ошибка вызова функции fork", cmd);
    else if (pid != 0) /* родительский процесс */
        exit(0);

    /*
     * Назначить корневой каталог текущим рабочим каталогом,
     * чтобы впоследствии можно было отмонтировать файловую систему.
     */
    if (chdir("/") < 0)
        err_quit("%s: невозможно сделать текущим рабочим каталогом /");

    /*
     * Закрыть все открытые файловые дескрипторы.
     */
    if (rl.rlim_max == RLIM_INFINITY)
        rl.rlim_max = 1024;
    for (i = 0; i < rl.rlim_max; i++)
        close(i);

    /*
     * Присоединить файловые дескрипторы 0, 1 и 2 к /dev/null.
     */
    fd0 = open("/dev/null", O_RDWR);
    fd1 = dup(0);
    fd2 = dup(0);

    /*
     * Инициализировать файл журнала.
     */
    openlog(cmd, LOG_CONS, LOG_DAEMON);
    if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
        syslog(LOG_ERR, "ошибочные файловые дескрипторы %d %d %d",
               fd0, fd1, fd2);
        exit(1);
    }
}
```

Если функция `daemonize` будет вызвана из программы, которая затем приостанавливает работу, мы сможем проверить состояние демона с помощью команды `ps`:

```
$ ./a.out
$ ps -axj
 PPID    PID  PGID   SID TTY TPGID UID   COMMAND
    1   3346  3345  3345 ?      -1 501   ./a.out
$ ps -axj | grep 3345
    1   3346  3345  3345 ?      -1 501   ./a.out
```

С помощью команды ps можно убедиться в том, что в системе нет активного процесса с идентификатором 3345. Это означает, что наш демон относится к осиротевшей группе процессов (раздел 9.10) и не является лидером сессии, поэтому он не имеет возможности обрести управляющий терминал. Это результат второго вызова функции fork в функции daemonize. Как видите, наш демон был инициализирован вполне корректно.

13.4. Журналирование ошибок

Одна из проблем, присущих демонам, связана с обслуживанием сообщений об ошибках. Демон не может просто выводить сообщения на стандартное устройство вывода сообщений об ошибках, поскольку он не имеет управляющего терминала. Мы не можем требовать от демона, чтобы он выводил сообщения на консоль, поскольку на большинстве рабочих станций в консоли запускается многооконная система. Мы также не можем требовать, чтобы демон хранил свои сообщения в отдельном файле. Это стало бы источником постоянной головной боли для системного администратора, который будет вынужден запоминать, какой демон в какой файл пишет свои сообщения. Необходим некий централизованный механизм регистрации сообщений об ошибках.

> Механизм syslog для BSD-систем был разработан в Беркли и получил широкое распространение, начиная с 4.2BSD. Большинство систем, производных от BSD, поддерживают syslog.
>
> До появления SVR4 ОС System V не имела централизованного механизма регистрации сообщений об ошибках.
>
> Функция syslog была включена в стандарт Single UNIX Specification как расширение XSI.

Механизм syslog для BSD-систем широко используется, начиная с 4.2BSD. Большинство демонов используют именно этот механизм. На рис. 13.1 показана его структура.

Существует три способа регистрации сообщений:

1. Процедуры ядра могут обращаться к функции log. Эти сообщения могут быть прочитаны любым пользовательским процессом, который может открыть и прочитать устройство /dev/klog. Мы не будем рассматривать эту функцию, поскольку написание процедур ядра не представляет для нас интереса.
2. Большинство пользовательских процессов (демонов) для регистрации сообщений вызывают функцию syslog(3). Порядок работы с ней мы рассмотрим позже. Эта функция отправляет сообщения через сокет домена UNIX – /dev/log.
3. Пользовательский процесс, который выполняется на данном или каком-либо другом компьютере, соединенном с данным компьютером сетью TCP/IP, может отправлять сообщения по протоколу UDP на порт 514. Обратите внимание, что функция syslog никогда не генерирует дейтаграммы UDP – данная функциональность требует, чтобы сама программа поддерживала сетевые взаимодействия.

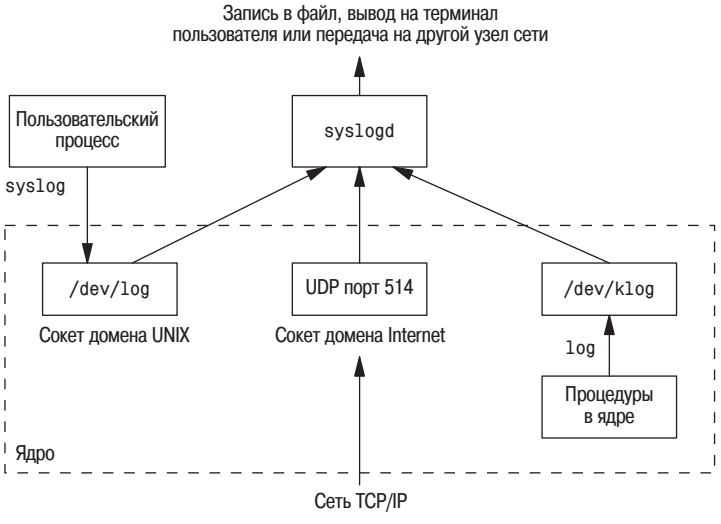


***Рис. 13.1.** Механизм syslog для BSD-систем*

За дополнительной информацией о сокетах домена UNIX обращайтесь к [Stevens, Fenner, and Rudoff 2004].

Обычно демон `syslogd` понимает все три способа регистрации сообщений. На запуске этот демон считывает конфигурационный файл (как правило, это `/etc/syslog.conf`), в котором определяется, куда должны передаваться различные классы сообщений. Например, срочные сообщения могут выводиться на консоль системного администратора (если он находится в системе), тогда как сообщения класса предупреждений могут записываться в файл.

В нашем случае взаимодействие с этим механизмом осуществляется посредством функции `syslog`.

```
#include <syslog.h>

void openlog(const char *ident, int option, int facility);

void syslog(int priority, const char *format, ...);

void closelog(void);

int setlogmask(int maskpri);
```

Возвращает предыдущее значение маски приоритета журналируемых сообщений

Можно и не вызывать функцию `openlog`. Если перед первым обращением к функции `syslog` функция `openlog` не вызывалась, то она будет вызвана автоматически. Обращаться к функции `closelog` также необязательно – она просто закрывает файловый дескриптор, который использовался для взаимодействия с демоном `syslogd`.

Функция openlog позволяет определить в аргументе *ident* строку идентификации, которая обычно содержит имя программы (например, cron или inetd). Аргумент *option* представляет собой битовую маску, которая определяет различные способы вывода сообщений. В табл. 13.1 приводятся значения, которые могут быть включены в маску. В столбце XSI отмечены те из них, которые стандарт Single UNIX Specification включает в определение функции openlog.

Таблица 13.1. Возможные значения, которые могут быть включены в аргумент option функции openlog

option	XSI	Описание
LOG_CONS	•	Если сообщение не может быть передано через сокет домена UNIX, оно будет выведено на консоль.
LOG_NDELAY	•	Сразу открыть сокет домена UNIX для взаимодействия с демоном syslogd, не дожидаясь, пока будет отправлено первое сообщение. Как правило, сокет открывается только тогда, когда отправлено первое сообщение.
LOG_NOWAIT	•	Не ждать завершения дочерних процессов, которые могли быть созданы в процессе регистрации сообщения. Это предотвращает возникновение конфликтов для тех приложений, которые перехватывают сигнал SIGCHLD, так как приложение уже могло получить код завершения дочернего процесса к моменту, когда syslog вызвал функцию wait.
LOG_ODELAY	•	Отложить установление соединения с демоном syslogd до появления первого сообщения.
LOG_PERROR		Вывести сообщение на стандартное устройство вывода сообщений об ошибках и дополнительно передать его демону syslogd (эта опция недоступна в ОС Solaris).
LOG_PID	•	Записывать идентификатор процесса вместе с текстом сообщения. Эта опция предназначена для демонов, которые порождают дочерние процессы для обработки различных запросов (в противоположность демонам, таким как syslogd, которые никогда не вызывают функцию fork).

Возможные значения аргумента *facility* приводятся в табл. 13.2. Обратите внимание, что стандарт Single UNIX Specification определяет лишь часть значений, обычно доступных на конкретной системе. Аргумент *facility* позволяет определить, как должны обрабатываться сообщения из разных источников. Если программа не вызывает функцию openlog или передает ей в аргументе *facility* значение 0, то указать источник сообщения можно с помощью функции syslog, определив его как часть аргумента *priority*.

Функция syslog вызывается для передачи сообщения. Аргумент *priority* представляет собой комбинацию значения для аргумента *facility* (табл. 13.2) и уровня важности сообщения (табл. 13.3). Уровни важности приведены в табл. 13.3 в порядке убывания, от высшего к низшему.

Таблица 13.2. Возможные значения аргумента facility функции openlog

facility	XSI	Описание
LOG_AUTH		Программы авторизации: login, su, getty, …
LOG_AUTHPRIV		То же самое, что и LOG_AUTH, но журналирование идет в файл с ограниченными правами доступа
LOG_CRON		cron и at
LOG_DAEMON		Системные демоны: inetd, routed, …
LOG_FTP		Демон FTP (ftpd)
LOG_KERN		Сообщения, сгенерированные ядром
LOG_LOCAL0	•	Определяется пользователем
LOG_LOCAL1	•	Определяется пользователем
LOG_LOCAL2	•	Определяется пользователем
LOG_LOCAL3	•	Определяется пользователем
LOG_LOCAL4	•	Определяется пользователем
LOG_LOCAL5	•	Определяется пользователем
LOG_LOCAL6	•	Определяется пользователем
LOG_LOCAL7	•	Определяется пользователем
LOG_LPR		Система печати: lpd, lpc, …
LOG_MAIL		Система электронной почты
LOG_NEWS		Система новостей Usenet
LOG_SYSLOG		Демон syslogd
LOG_USER	•	Сообщения от других пользовательских процессов (по умолчанию)
LOG_UUCP		Система UUCP

Таблица 13.3. Уровни важности сообщений (в порядке убывания)

Уровень	Описание
LOG_EMERG	Аварийная ситуация (система остановлена) (наивысший приоритет)
LOG_ALERT	Требуется немедленное вмешательство
LOG_CRIT	Критическая ситуация (например, ошибка жесткого диска)
LOG_ERR	Ошибка
LOG_WARNING	Предупреждение
LOG_NOTICE	Обычная ситуация, которая не является ошибочной, но, возможно, требует специальных действий
LOG_INFO	Информационное сообщение
LOG_DEBUG	Отладочное сообщение (низший приоритет)

Аргумент *format* и все последующие аргументы передаются функции vsprintf для создания строки сообщения. Символы %m в строке формата заменяются сообщением об ошибке (strerror), которое соответствует значению переменной errno.

Функция setlogmask может использоваться для установки маски приоритетов сообщений процесса. Эта функция возвращает предыдущее значение маски. Если маска приоритетов установлена, сообщения, уровень приоритета которых не содержится в маске, не будут журналироваться. Обратите внимание: из вышесказанного следует, что если маска имеет значение 0, то журналироваться будут все сообщения.

Во многих системах имеется программа logger(1), которая может передавать сообщения механизму syslog. Некоторые реализации позволяют передавать программе необязательные аргументы, в которых указывается источник сообщения (*facility*), уровень важности и строка идентификации (*ident*), хотя стандарт System UNIX Specification не определяет дополнительные аргументы. Команда logger предназначена для использования в сценариях на языке командной оболочки, которые исполняются в неинтерактивном режиме и нуждаются в механизме журналирования сообщений.

Пример

В (гипотетическом) демоне печати вы можете встретить следующие строки:

```
openlog("lpd", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

Обращение к функции openlog устанавливает строку идентификации, в которую записывается имя программы, указывает, что идентификатор процесса обязательно должен добавляться к сообщению, и оговаривает, что источником сообщений будет демон системы печати. В вызове функции syslog указан уровень важности сообщения и само сообщение. Если опустить вызов функции openlog, то вызов функции syslog мог бы выглядеть так:

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

Здесь в аргументе *priority* мы скомбинировали указание на источник сообщения и уровень важности сообщения.

Кроме функции syslog, большинство платформ предоставляют ее разновидность, которая принимает дополнительные аргументы в виде списка переменной длины.

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog(int priority, const char *format, va_list arg);
```

Все четыре платформы, обсуждаемые в данной книге, предоставляют функцию vsyslog, но она не входит в состав стандарта Single UNIX Specification.

Большинство реализаций `syslogd` для сокращения времени обработки запросов от приложений помещают поступившие сообщения в очередь. Если в это время демону поступит два одинаковых сообщения, то в журнал будет записано только одно. Но в конец такого сообщения демоном будет добавлена строка примерно такого содержания: «last message repeated *N* times» (последнее сообщение было повторено *N* раз).

13.5. Демоны в единственном экземпляре

Некоторые демоны реализованы таким образом, что допускают одновременную работу только одной своей копии. Причиной такого поведения может служить, например, требование монопольного владения каким-либо ресурсом. Так, если бы демон `cron` допускал одновременную работу нескольких своих копий, то каждая из них пыталась бы по достижении запланированного времени запустить одну и ту же операцию, что наверняка привело бы к ошибке.

Если демон требует наличия доступа к устройству, то некоторые действия по предотвращению открытия устройства несколькими программами может выполнить драйвер устройства. Это ограничит количество одновременно работающих экземпляров демона до одного. Однако, если не предполагается обращения демона к подобным устройствам, то нам самим придется выполнить всю необходимую работу по наложению ограничений.

Одним из основных механизмов, обеспечивающих ограничение количества одновременно работающих копий демона, являются блокировки файлов и записей. (Блокировки файлов и записей в файлах мы рассмотрим в разделе 14.3.) Если каждый из демонов создаст файл и попытается установить для этого файла блокировку для записи, то система разрешит установить только одну такую блокировку. Все последующие попытки установить блокировку для записи будут терпеть неудачу, сообщая тем самым остальным копиям демона о том, что демон уже запущен.

Блокировки файлов и записей представляют собой удобный механизм взаимного исключения. Если демон установит для целого файла блокировку для записи, она будет автоматически снята по завершении демона. Это упрощает процедуру восстановления после ошибок, поскольку снимает необходимость удаления блокировки, оставшейся от предыдущей копии демона.

Пример

Функция, представленная в листинге 13.2, демонстрирует использование блокировок файлов и записей для того, чтобы обеспечить запуск единственного экземпляра демона.

Листинг 13.2. Функция, которая гарантирует запуск только одной копии демона

```
#include <unistd.h>
#include <stdlib.h>
```

```
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

extern int lockfile(int);

int
already_running(void)
{
    int fd;
    char buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
        syslog(LOG_ERR, "не возможно открыть %s: %s",
               LOCKFILE, strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "невозможно установить блокировку на %s: %s",
               LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}
```

Каждая копия демона будет пытаться создать файл и записать в него свой идентификатор процесса. Это поможет системному администратору идентифицировать процесс. Если файл уже заблокирован, функция `lockfile` завершается неудачей с кодом ошибки `EACCESS` или `EAGAIN` в переменной `errno`, и в вызывающую программу возвращается значение 1, которое указывает, что демон уже запущен. В противном случае функция усекает размер файла до нуля, записывает в него идентификатор процесса и возвращает значение 0.

Усечение размера файла необходимо по той причине, что идентификатор процесса предыдущей копии демона, представленный в виде строки, мог иметь большую длину. Предположим, например, что ранее запускавшаяся копия демона имела идентификатор процесса 12345, а текущая копия имеет идентификатор процесса 9999. Таким образом, когда этот демон запишет

в файл свой идентификатор, то в файле окажется строка 99995. Операция усечения файла удаляет информацию, которая относится к предыдущей копии демона.

13.6. Соглашения для демонов

В системе UNIX демоны придерживаются следующих соглашений.

- Если демон использует файл блокировки, то этот файл помещается в каталог `/var/run`. Однако, чтобы создать файл в этом каталоге, демон должен обладать привилегиями суперпользователя. Имя файла обычно имеет вид *name*`.pid`, где *name* – имя демона или службы. Например, демон `cron` создает файл блокировки с именем `/var/run/crond.pid`.
- Если демон поддерживает определение дополнительных настроек, то они обычно сохраняются в каталоге `/etc`. Имя конфигурационного файла, как правило, имеет вид *name*`.conf`, где *name* – имя демона или службы. Например, конфигурационный файл демона `syslogd` называется `/etc/syslog.conf`.
- Демоны могут запускаться из командной строки, но все-таки чаще всего запуск демонов производится из сценариев инициализации системы (`/etc/rc*` или `/etc/init.d/*`). Если после завершения демон должен автоматически перезапускаться, мы можем указать на это процессу `init`, добавив запись `respawn` в файл `/etc/inittab`.
- Если демон имеет конфигурационный файл, то настройки из него считываются демоном во время запуска, и затем он обычно не обращается к этому файлу. Если в конфигурационный файл были внесены изменения, то демон пришлось бы останавливать и перезапускать снова, чтобы новые настройки вступили в силу. Во избежание этого некоторые демоны устанавливают обработчики сигнала `SIGHUP`, в которых производится считывание конфигурационного файла и перенастройка демона. Поскольку демоны не имеют управляющего терминала и являются либо лидерами сессий без управляющего терминала, либо членами осиротевших групп процессов, у них нет причин ожидать сигнала `SIGHUP`. Таким образом, он может использоваться для других целей.

Пример

Программа, представленная листингом 13.3, демонстрирует один из способов заставить демон перечитать файл конфигурации. Программа использует функцию `sigwait` и отдельный поток для обработки сигналов, как описано в разделе 12.8.

Листинг 13.3. Пример демона, который перечитывает конфигурационный файл по сигналу

```
#include "apue.h"
#include <pthread.h>
#include <syslog.h>
```

```
sigset_t mask;

extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0) {
            syslog(LOG_ERR, "ошибка вызова функции sigwait");
            exit(1);
        }
        switch (signo) {
        case SIGHUP:
            syslog(LOG_INFO, "Чтение конфигурационного файла");
            reread();
            break;
        case SIGTERM:
            syslog(LOG_INFO, "получен сигнал SIGTERM; выход");
            exit(0);
        default:
            syslog(LOG_INFO, "получен непредвиденный сигнал %d\n", signo);
        }
    }
    return(0);
}

int
main(int argc, char *argv[])
{
    int err;
    pthread_t tid;
    char *cmd;
    struct sigaction sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Перейти в режим демона.
     */
    daemonize(cmd);
```

```
    /*
     * Убедиться в том, что ранее не была запущена другая копия демона.
     */
    if (already_running()) {
        syslog(LOG_ERR, "демон уже запущен");
        exit(1);
    }

    /*
     * Восстановить действие по умолчанию для сигнала SIGHUP
     * и заблокировать все сигналы.
     */
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: невозможно восстановить действие SIG_DFL для SIGHUP");
    sigfillset(&mask);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
        err_exit(err, "ошибка выполнения операции SIG_BLOCK");

    /*
     * Создать поток, который будет заниматься обработкой SIGHUP и SIGTERM.
     */
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "невозможно создать поток");

    /*
     * Остальная часть программы-демона.
     */
    /* ... */
    exit(0);
}
```

Для перехода в режим демона программа использует функцию `daemonize` из листинга 13.1. После возврата из нее вызывается функция `already_running` из листинга 13.2, которая проверяет наличие других запущенных копий демона. В этой точке сигнал `SIGHUP` все еще игнорируется, поэтому мы должны переустановить его диспозицию в значение по умолчанию, в противном случае функция `sigwait` никогда не сможет получить его.

Далее выполняется блокировка всех сигналов, поскольку это рекомендуется для многопоточных программ, и создается поток, который будет заниматься обработкой сигналов. Поток обслуживает только сигналы `SIGHUP` и `SIGTERM`. При получении сигнала `SIGHUP` функция `reread` перечитывает файл конфигурации, а при получении сигнала `SIGTERM` поток записывает сообщение в журнал и завершает работу процесса.

В табл. 10.1 указано, что действие по умолчанию для сигналов `SIGHUP` и `SIGTERM` состоит в завершении процесса. Поскольку эти сигналы заблокированы, демон не будет завершаться, если получит один из них. Вместо этого от-

дельный поток будет получать номера доставленных сигналов с помощью функции sigwait.

Пример

Как уже отмечалось в разделе 12.8, в ОС Linux потоки ведут себя по отношению к сигналам несколько иначе. Это осложняет идентификацию процесса, которому должен быть передан сигнал. Кроме того, нет никаких гарантий, что демон будет реагировать на сигнал так, как мы этого ожидаем, из-за различий в реализации.

Программа, представленная листингом 13.4, показывает, как демон может перехватить сигнал SIGHUP и выполнить повторное чтение конфигурационного файла, не используя для этого отдельного потока.

Листинг 13.4. Альтернативная реализация демона, который перечитывает конфигурационный файл по сигналу

```
#include "apue.h"
#include <syslog.h>
#include <errno.h>

extern int lockfile(int);
extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void
sigterm(int signo)
{
    syslog(LOG_INFO, "получен сигнал SIGTERM; выход");
    exit(0);
}

void
sighup(int signo)
{
    syslog(LOG_INFO, "Чтение конфигурационного файла");
    reread();
}

int
main(int argc, char *argv[])
{
    char *cmd;
    struct sigaction sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
```

```
        cmd++;

    /*
     * Перейти в режим демона.
     */
    daemonize(cmd);

    /*
     * Убедиться в том, что ранее не была запущена другая копия демона.
     */
    if (already_running()) {
        syslog(LOG_ERR, "демон уже запущен");
        exit(1);
    }

    /*
     * Установить обработчики сигналов.
     */
    sa.sa_handler = sigterm;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGHUP);
    sa.sa_flags = 0;
    if (sigaction(SIGTERM, &sa, NULL) < 0) {
        syslog(LOG_ERR, "невозможно перехватить сигнал SIGTERM: %s",
               strerror(errno));
        exit(1);
    }

    sa.sa_handler = sighup;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGTERM);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0) {
        syslog(LOG_ERR, "невозможно перехватить сигнал SIGHUP: %s",
               strerror(errno));
        exit(1);
    }

    /*
     * Остальная часть программы-демона.
     */
    /* ... */
    exit(0);
}
```

После инициализации демона устанавливаются обработчики сигналов `SIGHUP` **и** `SIGTERM`**. У нас есть два варианта обработки сигнала** `SIGHUP`**: либо читать конфигурационный файл в функции-обработчике, либо просто установить в обработчике специальный флаг, а все необходимые действия выполнять в основном потоке программы.**

13.7. Модель клиент–сервер

Наиболее часто процессы-демоны используются в качестве серверных процессов. На рис. 13.1 показан пример взаимодействия с сервером `syslogd`, который получает сообщения от приложений (клиентов) посредством сокета домена UNIX.

Вообще, под *сервером* подразумевается некий процесс, который ожидает запросов на предоставление определенных услуг *клиентам*. Так, на рис. 13.1 сервер `syslogd` предоставляет услуги журналирования сообщений об ошибках.

Показанное на рис. 13.1 взаимодействие между сервером и клиентом носит односторонний характер. Клиент отсылает сообщения серверу, но ничего от него не получает. В последующих главах мы увидим множество примеров двустороннего взаимодействия сервера и клиента, когда клиент посылает запрос серверу, а сервер возвращает клиенту ответ.

13.8. Подведение итогов

Время работы процессов-демонов в большинстве случаев совпадает со временем работы самой системы. При разработке программ, которые будут работать как демоны, необходимо понимать и учитывать взаимоотношения между процессами, которые были описаны в главе 9. В этой главе мы разработали функцию, которую можно вызывать из процесса для корректного перехода в режим демона.

Мы также обсудили способы журналирования сообщений об ошибках демонов, поскольку они, как правило, не имеют управляющего терминала. Мы рассмотрели ряд соглашений, которым должны следовать демоны в большинстве версий UNIX, и показали примеры реализации этих соглашений.

Упражнения

13.1. Исходя из рис. 13.1, можно предположить, что при инициализации механизма `syslog` (либо прямым обращением к функции `openlog`, либо при первом обращении к функции `syslog`) он открывает специальный файл устройства `/dev/log`. Что произойдет, если пользовательский процесс (демон) вызовет функцию `chroot` перед обращением к функции `openlog`?

13.2. Перечислите все демоны в вашей системе и укажите их функциональное назначение.

13.3. Напишите программу, которая вызывает функцию `daemonize` из листинга 13.1. После вызова этой функции определите имя пользователя с помощью `getlogin` (раздел 8.15), чтобы узнать, не изменился ли пользователь процесса после перехода в режим демона. Выведите полученные результаты в файл.

14

Расширенные операции ввода-вывода

14.1. Введение

В этой главе мы обсудим большое количество тем и функций, которые объединяются под общим термином *расширенные операции ввода-вывода*: неблокирующий ввод-вывод, блокировка записей, механизм STREAMS, мультиплексирование операций ввода-вывода (функции `select` и `poll`), функции `readv` и `writev` и ввод-вывод для файлов, отображаемых в память (`mmap`). Нам необходимо рассмотреть эти темы, прежде чем перейти к обсуждению межпроцессного взаимодействия в главах 15 и 17 и многочисленных примеров в последующих главах.

14.2. Неблокирующий ввод-вывод

В разделе 10.5 мы говорили, что системные вызовы подразделяются на две категории: «медленные» и все остальные. Медленными называют такие системные вызовы, которые могут заблокировать процесс «навечно». В эту категорию входят:

- Операция чтения, которая может «навечно» заблокировать вызывающий процесс, если в файлах определенных типов (каналы, терминальные устройства, сетевые устройства) отсутствуют данные, доступные для чтения.
- Операция записи также может «навечно» заблокировать вызывающий процесс, если данные не могут быть немедленно записаны в файлы тех же типов (отсутствие места в канале, переполнено сетевое соединение и т. п.).
- Операция открытия может заблокировать вызывающий процесс до тех пор, пока не будут соблюдены некоторые условия для файлов определенных типов (например, открытие терминального устройства не может быть произведено, пока не будет установлено соединение между модемами, или открытие именованного канала FIFO только на запись будет заблокировано, пока не появится другой процесс, который откроет этот канал на чтение).

- Операции чтения и записи над файлами, для которых установлена принудительная блокировка записей.
- Некоторые операции `ioctl`.
- Некоторые функции, относящиеся к механизму межпроцессного взаимодействия (глава 15).

Мы также говорили, что системные вызовы, связанные с дисковыми операциями ввода-вывода, не относятся к категории медленных, хотя операция чтения с диска или записи на диск может на какое-то время заблокировать вызывающий процесс.

Неблокирующий режим ввода-вывода позволяет запускать такие операции, как `open`, `read` или `write`, не опасаясь, что они заблокируют процесс. Если запрошенная операция не может быть выполнена немедленно, системный вызов тут же возвращает управление вызывающему процессу с признаком ошибки, сообщающим о том, что операция может быть заблокирована.

Существует два способа указать, что для заданного дескриптора файла должны использоваться неблокирующие операции ввода-вывода.

1. Если для получения дескриптора вызывается функция `open`, можно указать флаг `O_NONBLOCK` (раздел 3.3).
2. Чтобы включить флаг `O_NONBLOCK` для уже открытого дескриптора, следует воспользоваться функцией `fcntl` (раздел 3.14). В листинге 3.5 приводится функция, с помощью которой можно установить любой флаг дескриптора файла.

> В ранних версиях System V для выбора неблокирующего режима операций ввода-вывода использовался флаг `O_NDELAY`. В этих версиях при отсутствии доступных для чтения данных функция `read` возвращала значение 0. Поскольку это противоречит принятому в UNIX соглашению, что функция `read` возвращает 0 по достижении конца файла, стандарт POSIX.1 определил флаг неблокирующего режима с другим именем и с другой семантикой. В старых версиях System V, когда функция `read` возвращала значение 0, нельзя было определить, то ли это системный вызов вернул управление, потому что мог быть заблокирован, то ли действительно был достигнут конец файла. Стандарт POSIX.1 требует, чтобы в неблокирующем режиме при отсутствии доступных для чтения данных функция read возвращала признак ошибки –1 и код ошибки `EAGAIN` в переменной `errno`. Некоторые версии UNIX, происходящие от System V, поддерживают оба флага – и устаревший `O_NDELAY`, и определяемый стандартом POSIX.1 `O_NONBLOCK`, но в данной книге мы будем использовать только ту функциональность, которая определяется стандартом POSIX.1. Флаг `O_NDELAY` поддерживается лишь для сохранения обратной совместимости и не должен использоваться в новых приложениях.
>
> В 4.3BSD появился флаг функции `fcntl FNDELAY` с несколько иной семантикой. Он воздействовал не только на флаги состояния файла в его дескрипторе – изменялись также флаги терминального устройства или сокета, что оказывало влияние на всех пользователей терминала или сокета, а не только пользователей, совместно использующих одну и ту же запись в таблице файлов (в 4.3BSD неблокирующий режим ввода-вывода мог назначаться только терминальным устройствам или сокетам). Кроме того, в 4.3BSD в вызывающую программу возвращалось значение `EWOULDBLOCK`, если операция над дескриптором в неблокирующем режиме не могла быть завершена. Современные BSD-

системы поддерживают флаг O_NONBLOCK и определяют константу EWOULDBLOCK с тем же значением, что и EAGAIN. Эти системы предоставляют семантику неблокирующего режима, совместимую со стандартом POSIX.1: изменения флагов состояния файла оказывают влияние на всех пользователей той же самой записи в таблице файлов, но не затрагивают режимы работы с одним и тем же устройством, если для доступа к нему используются различные записи в таблице файлов (рис. 3.1 и 3.3).

Пример

Рассмотрим пример ввода-вывода в неблокирующем режиме. Программа, представленная листингом 14.1, считывает 500 000 байт со стандартного ввода и пытается вывести их на стандартный вывод. Стандартное устройство вывода предварительно переводится в неблокирующий режим. Вывод результатов каждой операции записи производится на стандартное устройство вывода сообщений об ошибках. Функция clr_fl очень похожа на функцию set_fl из листинга 3.5. Эта функция просто сбрасывает один или более флагов.

Листинг 14.1. Вывод большого количества данных в неблокирующем режиме

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>

char buf[500000];

int
main(void)
{
    int ntowrite, nwrite;
    char *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "прочитано %d байт\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK);   /* установить неблокирующий режим */

    ptr = buf;
    while (ntowrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK);   /* выход из неблокирующего режима */

    exit(0);
}
```

Мы предполагаем, что функция write отработает всего один раз, если стандартный вывод перенаправить в обычный файл:

```
$ ls -l /etc/termcap                    проверим размер файла
-rw-r--r--  1 root      702559 Feb 23 2002 /etc/termcap
$ ./a.out < /etc/termcap > temp.file    для начала попробуем с обычным файлом
прочитано 500000 байт
nwrite = 500000, errno = 0              единственный вызов write
$ ls -l temp.file                       проверим размер получившегося файла
-rw-rw-r--  1 sar       500000 Jul   8 04:19 temp.file
```

Но если в качестве устройства вывода будет использоваться терминал, то мы предполагаем, что функция `write` будет иногда возвращать значение, не равное 500 000, а иногда признак ошибки. Вот что мы получили в этом случае:

```
$ ./a.out < /etc/termcap 2>stderr.out   вывод на терминал
                                        огромный объем выводимых данных...
$ cat stderr.out
прочитано 500000 байт
nwrite = 216041, errno = 0
nwrite = -1, errno = 11                 1497 таких ошибок
. . .
nwrite = 16015, errno = 0
nwrite = -1, errno = 11                 1856 таких ошибок
. . .
nwrite = 32081, errno = 0
nwrite = -1, errno = 11                 1654 таких ошибок
. . .
nwrite = 48002, errno = 0
nwrite = -1, errno = 11                 1460 таких ошибок
. . .
                                        и так далее...
nwrite = 7949, errno = 0
```

В данной системе число 11 соответствует коду ошибки `EAGAIN`. Объем данных, принимаемых терминалом за одно обращение, варьируется от системы к системе. Результаты также зависят от того, как был произведен вход в систему – с консоли, с удаленного терминала или через сетевое соединение, которое использует псевдотерминал. Если на вашем терминале работает многооконная система, значит, вы также работаете через устройство псевдотерминала.

В этом примере программа произвела несколько тысяч вызовов функции `write`, хотя фактически вся работа была выполнена 10–20 вызовами. Остальные только возвращали признак ошибки. Такой тип цикла называется *опросом* (*polling*), и в многопользовательских системах он понапрасну расходует процессорное время. В разделе 14.5 мы рассмотрим более эффективный подход к работе с дескрипторами в неблокирующем режиме.

Иногда удается избежать применения неблокирующих операций ввода-вывода за счет использования потоков (глава 11). В этом случае можно позволить заблокировать один поток, если другие потоки смогут продолжать работу. Иногда такой подход упрощает архитектуру приложения, что мы увидим в главе 21; однако в некоторых случаях проблемы, связанные с необходимостью синхронизации потоков, могут свести на нет все преимущества многопоточной модели.

14.3. Блокировка записей

Что произойдет, если два пользователя попытаются редактировать один файл в одно и то же время? В большинстве версий UNIX окончательное содержимое файла будет определяться последней операцией записи. Однако в некоторых приложениях, таких как системы управления базами данных, процесс должен убедиться, что только он пишет в файл. С этой целью коммерческие версии UNIX предоставляют возможность блокировки отдельных записей в файле. (В главе 20 мы будем разрабатывать библиотеку функций для работы с базой данных, которая использует блокировки записей.)

Термин *блокировка записи* обычно используется для описания функциональной возможности, которая позволяет одному процессу предотвратить изменение участка файла другим процессом, пока первый процесс читает или изменяет эту часть файла. Использование понятия «запись» для системы UNIX не совсем корректно, поскольку ядро UNIX не имеет представления ни о структуре файла, ни о записях в файле. Более правильный термин – *блокировка диапазона байтов*, поскольку на самом деле подразумевается некий диапазон байтов в файле (возможно, даже весь файл), доступ к которому заблокирован.

Предыстория

Ранние версии UNIX часто подвергались критике за то, что они не могли использоваться для построения систем управления базами данных из-за отсутствия механизма, позволяющего блокировать доступ к отдельным участкам файла. Позднее в различных семействах UNIX появилась поддержка механизма блокировки записей (в различных видах, разумеется).

Ранние версии из Беркли поддерживали только функцию `flock`, с помощью которой можно заблокировать файл целиком, но не отдельный его участок.

Механизм блокировки записей впервые появился в функции `fcntl` в System V Release 3. Функция `lockf` была реализована поверх `fcntl` и представляла собой упрощенный интерфейс к последней. Эти функции позволяли вызывающей программе блокировать доступ к произвольному диапазону байтов, начиная от единственного байта и заканчивая всем файлом.

Стандарт POSIX.1 выбрал для стандартизации подход на основе `fcntl`. В табл. 14.1 перечислены различные формы блокировок записи, предоставляемые разными системами. Обратите внимание: стандарт Single UNIX Specification включает функцию `lockf` как расширение XSI.

Различия между рекомендательными и принудительными блокировками рассматриваются далее в этом же разделе. В этой книге обсуждаются только блокировки на основе функции `fcntl`.

> Механизм блокировки записей изначально был добавлен к Version 7 Джоном Бассом (John Bass) в 1980 году. Системный вызов в ядре получил название `locking`. Эта функция предоставляла механизм принудительных блокировок, который прошел через множество версий System III. ОС Xenix включала эту функцию, и некоторые производ-

ные от System V системы для архитектуры Intel, такие как OpenServer 5, продолжают поддерживать ее в библиотеке совместимости с ОС Xenix.

Таблица 14.1. Способы блокировки записей, поддерживаемые различными версиями UNIX

Система	Рекомендательные	Принудительные	fcntl	lockf	flock
SUS	•		•	XSI	
FreeBSD 5.2.1	•		•	•	•
Linux 2.4.22	•	•	•	•	•
Mac OS X 10.3	•		•	•	•
Solaris 9	•	•	•	•	•

Блокировка записей на основе функции fcntl

Приведем еще раз прототип функции `fcntl` из раздела 3.14.

```
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* struct flock *flockptr */ );
```

Возвращаемое значение зависит от аргумента *cmd* (см. ниже) в случае успеха, −1 в случае ошибки

При использовании этой функции для блокировки записей в аргументе *cmd* передаются значения `F_GETLK`, `F_SETLK` или `F_SETLKW`. Третий аргумент (который обозначен как *flockptr*) представляет собой указатель на структуру `flock`.

```
struct flock {
    short l_type;    /* F_RDLCK, F_WRLCK или F_UNLCK */
    off_t l_start;   /* смещение в байтах относительно l_whence */
    short l_whence;  /* SEEK_SET, SEEK_CUR или SEEK_END */
    off_t l_len;     /* длина области в байтах; 0 означает - до конца файла */
    pid_t l_pid;     /* возвращается при использовании команды F_GETLK */
};
```

Эта структура определяет

- Тип блокировки: `F_RDLCK` (блокировка для совместного чтения), `F_WRLCK` (исключительная блокировка для записи) или `F_UNLCK` (снятие блокировки)
- Начало блокируемой или разблокируемой области (`l_start` и `l_whence`)
- Размер области в байтах (`l_len`)
- Идентификатор процесса, удерживающего блокировку, которая может заблокировать текущий процесс (возвращается только для операции `F_GETLK`)

Существует целый ряд правил, используемых для определения участка файла, который должен быть заблокирован или разблокирован.

- Два элемента структуры, которые определяют начало участка, похожи на последние два аргумента функции `lseek` (раздел 3.6). Поле `l_whence` может принимать значения `SEEK_SET`, `SEEK_CUR` или `SEEK_END`.
- Блокируемый участок может начинаться и заканчиваться за текущим концом файла, но не может начинаться или заканчиваться перед началом файла.
- Если в поле `l_len` указано значение 0, это означает, что блокировка распространяется до конца файла. Это дает возможность заблокировать область, которая может начинаться в любом месте файла и продолжаться до конца файла, включая в себя все данные, которые могут быть дописаны позже. (Мы не будем строить предположения по поводу того, сколько именно байт может быть добавлено в конец файла.)
- Чтобы заблокировать весь файл, нужно установить в поля `l_start` и `l_whence` значения, соответствующие началу файла, а в поле `l_len` – значение 0. (Существует несколько способов указать начало файла, но в большинстве приложений для этого в поле `l_start` записывается значение 0, а в поле `l_whence` – значение `SEEK_SET`.)

Мы уже упоминали два типа блокировок – совместно используемые блокировки для чтения (в поле `l_type` записывается значение `F_RDLCK`) и исключительные блокировки для записи (`F_WRLCK`). Основное различие между этими двумя типами заключается в том, что любое количество процессов могут установить совместно используемую блокировку для чтения заданного байта, но только один процесс может установить исключительную блокировку для записи заданного байта. Кроме того, если для байта установлена хотя бы одна блокировка для чтения, никакая блокировка для записи на него уже установлена быть не может, и наоборот, если на заданный байт установлена исключительная блокировка для записи, то на него уже не может быть установлена блокировка для чтения. Правила совместимости типов блокировок приводятся в табл. 14.2.

Таблица 14.2. Совместимость различных типов блокировок

		Запрос на	
		блокировку для чтения	**блокировку для записи**
К настоящему моменту область	Не заблокирована	OK	OK
	Уже имеет одну или более блокировок для чтения	OK	Отклоняется
	Уже имеет одну блокировку для записи	Отклоняется	Отклоняется

Правило совместимости блокировок применяется к запросам, поступающим от различных процессов, а не к множеству запросов, производимых одним процессом. Если процесс уже обладает одной блокировкой на некоторый участок файла и пытается установить блокировку на тот же самый участок,

то существующая блокировка будет замещена новой блокировкой. Таким образом, если процесс установил блокировку для записи на диапазон с 16-го по 32-й байты и затем попытается установить блокировку для чтения на тот же диапазон, то его запрос будет удовлетворен (при условии, что никакой другой процесс не пытается заблокировать тот же самый участок файла), и блокировка для записи будет замещена блокировкой для чтения.

Чтобы установить блокировку для чтения, дескриптор файла должен быть открыт для чтения. Чтобы установить блокировку для записи, дескриптор файла должен быть открыт для записи.

Теперь мы можем описать три команды функции `fcntl`, имеющие отношение к блокировкам.

`F_GETLK` Определяет, будет ли попытка установить блокировку заблокирована некоторой другой блокировкой. Если к моменту выполнения команды уже существует блокировка, которая может помешать установить новую блокировку, то по адресу `flockptr` записывается информация о существующей блокировке. Если таких блокировок не существует, то содержимое структуры `flockptr` не изменяется, за исключением поля `l_type`, в которое записывается значение `F_UNLCK`.

`F_SETLK` Установить блокировку, определение которой находится по адресу `flockptr`. При попытке установить блокировку, несовместимую с существующей (в соответствии с правилами, приведенными в табл. 14.2), функция `fcntl` сразу же вернет управление с кодом ошибки `EAGAIN` или `EACCES` в переменной `errno`.

> Хотя стандарт POSIX.1 допускает возврат любого кода ошибки, тем не менее все четыре реализации, обсуждаемые в данной книге, возвращают код ошибки `EAGAIN`, если запрос на установку блокировки не может быть удовлетворен.

Эта команда также используется для снятия блокировки. В этом случае в поле `l_type` указывается значение `F_UNLCK`.

`F_SETLKW` Эта команда представляет собой блокирующую версию команды `F_SETLK`. (В данном случае `W` означает «wait» – «ждать».) Если запрос на установку блокировки не может быть немедленно удовлетворен из-за того, что некоторый другой процесс уже установил несовместимую с данной блокировку на диапазон, часть которого попадает в запрошенный диапазон, то работа вызывающего процесса приостанавливается. Процесс возобновит работу, когда появится возможность установить блокировку или когда ожидание будет прервано сигналом.

Вы должны понимать, что проверка возможности установки блокировки (`F_GETLK`) и последующая попытка установки блокировки (`F_SETLK` или `F_SETLKW`) не являются атомарной операцией. Нельзя гарантировать, что между двумя обращениями к функции `fcntl` управление не будет передано некоторому другому процессу, который пожелает установить ту же самую блокировку.

Если необходимо предотвратить блокирование процесса в ожидании возможности получения блокировки, следует использовать команду F_SETLK и должным образом обрабатывать возвращаемый функцией результат.

> Обратите внимание: стандарт POSIX.1 не определяет, что может произойти, когда один процесс уже установил блокировку для чтения на некоторый диапазон в файле, второй процесс пробует установить блокировку для записи на тот же самый диапазон, а затем третий процесс пытается установить на тот же самый диапазон блокировку для чтения. Если третьему процессу будет позволено установить блокировку для чтения на диапазон, который уже заблокирован блокировкой для чтения, то реализация может «подвесить» процесс, который ожидает получения блокировки для записи. Это означает, что по мере поступления новых запросов на получение блокировки для чтения ожидание получения блокировки для записи может растянуться на неопределенное время. Если запросы на получение блокировки для чтения поступают беспрерывно и достаточно часто, то процесс, ожидающий получения блокировки для записи, может оставаться в состоянии ожидания достаточно длительное время.

Во время установки или снятия блокировки система соединяет вместе или разбивает смежные области в соответствии с характеристиками выполняемой операции. Если, например, заблокировать байты 100–199, а затем разблокировать байт 150, то ядро будет обслуживать два заблокированных диапазона: байты 100–149 и байты 151–199. Эта ситуация показана на рис. 14.1.

***Рис. 14.1.** Схема блокировки диапазона байтов*

Если затем установить блокировку на 150-й байт, система объединит смежные области в одну – с 100-го по 199-й байты. В этом случае конечный результат будет соответствовать первой диаграмме, приведенной на рис. 14.1, т. е. той, с которой мы начали.

Пример – запрос на установку и снятие блокировки

Чтобы избавить себя от необходимости всякий раз размещать и заполнять структуру flock, мы написали функцию lock_reg, которая выполняет все необходимые действия (листинг 14.2).

Листинг 14.2. Функция наложения и снятия блокировки на участок файла

```
#include "apue.h"
#include <fcntl.h>

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;

    lock.l_type = type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;  /* смещение в байтах относительно l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* количество байт (0 - до конца файла) */

    return(fcntl(fd, cmd, &lock));
}
```

Поскольку в большинстве случаев функция будет вызываться для наложения или снятия блокировки (команда `F_GETLK` используется достаточно редко), то мы обычно используем один из следующих пяти макросов, определенных в заголовочном файле `apue.h` (приложение B).

```
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))
```

Мы преднамеренно определили порядок первых трех аргументов так, чтобы он соответствовал порядку аргументов функции `lseek`.

Пример – проверка возможности наложения блокировки

В листинге 14.3 приводится исходный код функции `lock_test`, которую мы будем использовать при проверке возможности наложения блокировки.

Листинг 14.3. Функция проверки возможности наложения блокировки

```
#include "apue.h"
#include <fcntl.h>

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;

    lock.l_type = type;     /* F_RDLCK или F_WRLCK */
    lock.l_start = offset;  /* смещение в байтах относительно l_whence */
```

```
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* количество байт (0 -до конца файла) */

    if (fcntl(fd, F_GETLK, &lock) < 0)
        err_sys("fcntl error");

    if (lock.l_type == F_UNLCK)
        return(0);          /* ложь, заданная область не заблокирована */
                            /* другим процессом */
    return(lock.l_pid);     /* истина, вернуть pid владельца блокировки */
}
```

Если уже существует блокировка, которая может заблокировать выполнение запроса с заданными параметрами, эта функция возвращает идентификатор процесса, владеющего блокировкой. В противном случае функция возвращает 0 (ложь). Мы обычно будем вызывать эту функцию из следующих двух макросов (определенных в файле `apue.h`).

```
#define is_read_lockable(fd, offset, whence, len) \
          (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
           (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)
```

Обратите внимание: функция `lock_test` не может использоваться процессом для определения того, является ли он в настоящий момент владельцем блокировки заданного участка файла. Определение команды `F_GETLK` гласит, что возвращаемая информация относится к существующей блокировке, которая может помешать наложению новой блокировки. Так как команды `F_SETLK` и `F_SETLKW` всегда заменяют существующую блокировку, если ее владельцем является вызывающий процесс, то мы никогда не сможем заблокировать процесс на своей собственной блокировке. Таким образом, команда `F_GETLK` никогда не будет сообщать о наличии блокировки, если эта блокировка принадлежит вызывающему процессу.

Пример – тупиковая ситуация

Тупиковая ситуация (или ситуация взаимоблокировки) возникает, когда каждый из двух процессов ожидает освобождения ресурса, заблокированного другим процессом. Опасность появления тупиковой ситуации возникает, если процесс владеет блокировкой на некоторый участок файла и, пытаясь наложить блокировку на другой участок файла, приостанавливается в ожидании снятия блокировки с этого участка, установленной другим процессом.

В листинге 14.4 приводится пример такой тупиковой ситуации. Дочерний процесс блокирует доступ к байту 0, а родительский процесс – к байту 1. После этого каждый из процессов пытается заблокировать байт, уже заблокированный другим процессом. Для синхронизации родительского и дочернего процессов мы использовали процедуры из раздела 8.9 (`TELL_xxx` и `WAIT_xxx`), которые дают возможность каждому процессу дождаться другого процесса, чтобы наложить нужную блокировку. Запустив программу из листинга 14.4, получим следующее:

```
$ ./a.out
родитель: установлена блокировка на байт 1
потомок: установлена блокировка на байт 0
потомок: ошибка вызова функции writew_lock: Resource deadlock avoided
родитель: установлена блокировка на байт 0
```

Листинг 14.4. Пример выявления тупиковой ситуации

```
#include "apue.h"
#include <fcntl.h>

static void
lockabyte(const char *name, int fd, off_t offset)
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: ошибка вызова функции writew_lock", name);
    printf("%s: установлена блокировка на байт %ld\n", name, offset);
}

int
main(void)
{
    int fd;
    pid_t pid;

    /*
     * Создать файл и записать в него два байта.
     */
    if ((fd = creat("templock", FILE_MODE)) < 0)
        err_sys("ошибка вызова функции creat");
    if (write(fd, "ab", 2) != 2)
        err_sys("ошибка вызова функции write");

    TELL_WAIT();
    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) {                  /* потомок */
        lockabyte ("потомок", fd, 0);
        TELL_PARENT(getppid());
        WAIT_PARENT();
        lockabyte("потомок", fd, 1);
    } else {                                /* родитель */
        lockabyte("родитель", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("родитель", fd, 0);
    }
    exit(0);
}
```

Когда ядро обнаруживает наличие тупиковой ситуации, оно возвращает одному из процессов признак ошибки. В данном случае признак ошибки был возвращен дочернему процессу. В одних системах признак ошибки всегда получает дочерний процесс, в других – родительский, в третьих признак ошибки возвращается обоим процессам.

Правила наследования блокировок

Наследование и снятие блокировок записей в файле производится в соответствии со следующими тремя правилами.

1. Блокировки ассоциируются с процессом и с файлом. Это проявляется в следующем. Во-первых, по завершении процесса все его блокировки освобождаются. Во-вторых, когда закрывается дескриптор, освобождаются все блокировки, связанные с файлом, на который ссылается заданный дескриптор. Это означает, что если программа выполняет код

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = dup(fd1);
   close(fd2);
   ```

   то после закрытия дескриптора `fd2` блокировка, установленная для дескриптора `fd1`, освобождается. То же самое произойдет, если заменить вызов функции `dup` вызовом функции `open`, как в следующем фрагменте кода:

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = open(pathname, ...)
   close(fd2);
   ```

   где открывается тот же самый файл, но с другим дескриптором.

2. Блокировки никогда не наследуются дочерним процессом, созданным функцией `fork`. Это означает, что если процесс установил блокировку, а затем вызвал функцию `fork`, то относительно блокировок, установленных родительским процессом, дочерний процесс будет рассматриваться как совершенно другой процесс. Потомок должен будет вызывать функцию `fcntl`, чтобы установить свои собственные блокировки для любых дескрипторов, унаследованных от родителя. Такое положение вещей имеет определенный смысл, предотвращая возможность записи в один и тот же участок файла из нескольких процессов. Если бы дочерний процесс наследовал родительские блокировки, то они оба смогли бы одновременно писать в один и тот же файл.

3. Блокировки наследуются новыми программами при вызове функции `exec`. Однако если для дескриптора установлен флаг close-on-exec (закрыть при вызове `exec`), то все блокировки, связанные с данным файлом, освобождаются при закрытии дескриптора в функции `exec`.

Реализация в FreeBSD

Давайте поближе познакомимся со структурами данных, которые используются в ОС FreeBSD. Это поможет вам лучше понять правило 1, которое утверждает, что блокировки связаны с процессом и с файлом.

Рассмотрим следующий фрагмент программы (не принимая во внимание случаи, когда обращения к функциям завершаются неудачей):

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1);      /* родитель устанавливает блокировку */
                                      /* для записи на байт с номером 0 */
if ((pid = fork()) > 0) {             /* родительский процесс */
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
} else if (pid == 0) {
    read_lock(fd1, 1, SEEK_SET, 1);   /* потомок устанавливает блокировку */
                                      /* для чтения на байт с номером 1 */
}
pause();
```

На рис. 14.2 показано состояние структур данных после того, как оба процесса вызовут функцию `pause`.

Ранее мы уже показывали состояние структур данных после вызова функций `open`, `fork` и `dup` (рис. 3.3 и 8.1). Единственное, что изменилось на данном рисунке, – появились структуры `lockf`, которые связаны со структурой индексного узла. Обратите внимание: каждая структура `lockf` описывает отдельную область в файле (которая определяется началом и длиной), заблокированную конкретным процессом. Здесь показаны две такие структуры:

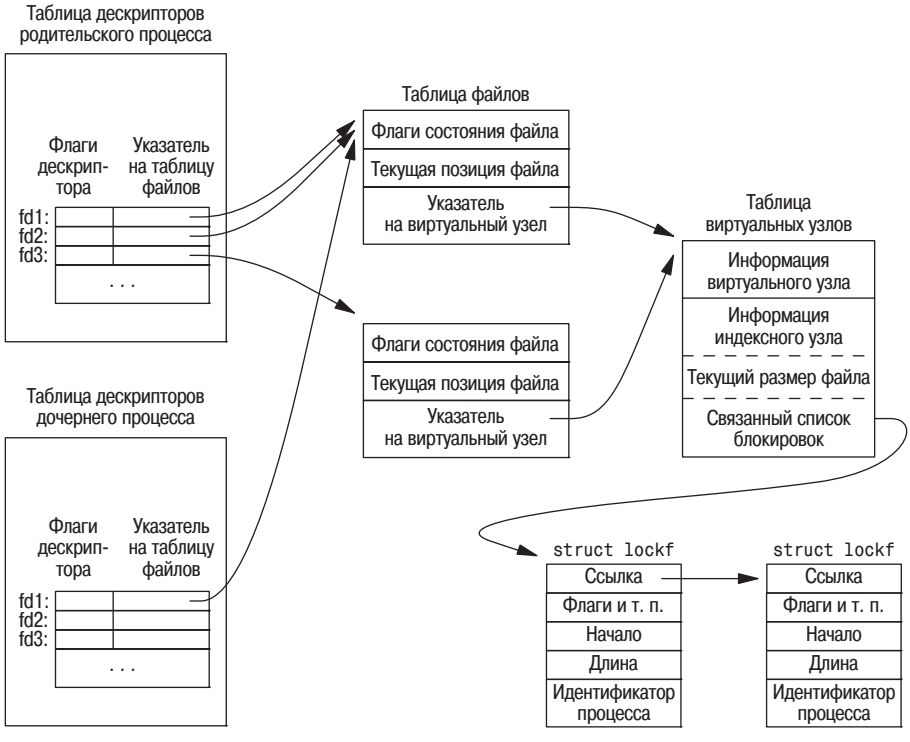


***Рис. 14.2.** Структуры данных, связанные с блокировками записей в файле, в ОС FreeBSD*

одна была создана вызовом write_lock из родительского процесса, а вторая – вызовом read_lock из дочернего процесса. Каждая структура содержит соответствующий идентификатор процесса.

При закрытии любого из трех дескрипторов в родительском процессе – fd1, fd2 или fd3 – блокировка для записи снимается. Когда закрывается какой-либо из этих дескрипторов, ядро обходит связанный список блокировок для соответствующего индексного узла и освобождает все блокировки, установленные вызывающим процессом. Ядро не имеет возможности определить, какой дескриптор был использован родительским процессом для установки блокировки.

Пример

В программе из листинга 13.2 мы видели, как демон может использовать блокировку файла, чтобы обеспечить запуск единственного экземпляра программы. В листинге 14.5 приводится реализация функции lockfile, которая использовалась демоном, чтобы установить блокировку для записи.

Листинг 14.5. Установка блокировки для записи на весь файл

```
#include <unistd.h>
#include <fcntl.h>

int
lockfile(int fd)
{
    struct flock fl;

    fl.l_type = F_WRLCK;
    fl.l_start = 0;
    fl.l_whence = SEEK_SET;
    fl.l_len = 0;
    return(fcntl(fd, F_SETLK, &fl));
}
```

Как вариант, мы могли бы определить функцию lockfile в терминах функции write_lock:

```
#define lockfile(fd) write_lock((fd), 0, SEEK_SET, 0)
```

Блокировки в конце файла

С особой осторожностью следует подходить к установке блокировок, когда начало области задается относительно конца файла. В большинстве реализаций значения поля l_whence (SEEK_CUR и SEEK_END) преобразуются в абсолютное смещение с использованием значений поля l_start и текущей позиции или текущей длины файла. Однако зачастую возникает необходимость указывать начало области относительно текущей позиции или текущей длины файла, потому что мы не можем вызывать функцию lseek для получения значения текущей позиции в файле, так как не владеем блокировкой. (В этот момент у других процессов появляется шанс вклиниться между вызовами

функции lseek и функции, которая устанавливает блокировку, и изменить длину файла.)

Рассмотрим следующую последовательность действий:

```
writew_lock(fd, 0, SEEK_END, 0);
write(fd, buf, 1);
un_lock(fd, 0, SEEK_END);
write(fd, buf, 1);
```

Этот код может делать совсем не то, что вы от него ожидаете. Здесь устанавливается блокировка для записи, начиная от текущего конца файла и дальше, включая данные, которые могут быть добавлены в конец файла позже. Предположим, что текущая позиция находится в конце файла, тогда первый вызов write добавит один байт в конец файла, и этот байт будет заблокирован. Следующая затем операция снятия блокировки разблокирует все данные, которые могут быть добавлены в конец файла позже, но оставит текущий последний байт заблокированным. Когда будет выполнена вторая операция записи, размер файла увеличится еще на один байт, и этот байт не будет заблокирован. Состояние блокировок для данной последовательности действий показано на рис. 14.3.

Когда на участок файла устанавливается блокировка, ядро преобразует указанное смещение в абсолютное смещение относительно начала файла. Кроме смещения относительно начала файла (SEEK_SET), функция fcntl позволяет указать смещение относительно текущей позиции в файле (SEEK_CUR) или относительно конца файла (SEEK_END). Ядро вынуждено запоминать положение блокировок в представлении, не зависящем от текущей позиции или конца файла, потому что текущая позиция или размер файла могут измениться, но эти изменения не должны влиять на положение блокировки.

Чтобы удалить блокировку байта, добавленного первой операцией записи, мы должны были бы указать значение –1 в качестве длины участка. Отрицательное значение длины соответствует участку, расположенному перед заданным смещением.

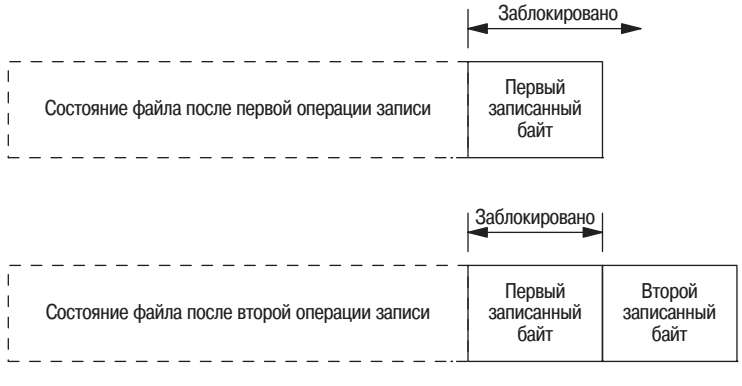


***Рис. 14.3.** Схема заблокированных участков файла*

Рекомендательные и принудительные блокировки

Рассмотрим библиотеку процедур, обеспечивающих доступ к базе данных. Если все функции в библиотеке используют возможность блокировки записей в файле непротиворечивым способом, то мы можем сказать, что любое множество процессов, использующих для доступа к базе данных эти функции, являются кооперативными (сотрудничающими друг с другом) процессами. Для данных функций вполне подходит рекомендательный тип блокировок, при условии, что только эти функции используются для доступа к базе данных. Но рекомендательные блокировки не могут предотвратить возможность записи в файлы базы данных из других процессов, которые имеют право на запись в эти файлы. Такой «жульничающий» процесс можно назвать некооперативным (не сотрудничающим), так как он не использует общепринятые методы (библиотека функций) для доступа к базе данных.

Принудительный тип блокировок вынуждает ядро проверять каждую операцию `open`, `read` и `write` на предмет, не противоречит ли она блокировкам, связанным с файлом. Принудительные блокировки иногда называют *блокировками форсированного режима*.

> В табл. 14.1 показано, что ОС Linux 2.4.22 и Solaris 9 поддерживают принудительные блокировки, а FreeBSD 5.2.1 и Mac OS X 10.3 – нет. Механизм принудительных блокировок не является частью стандарта Single UNIX Specification. При желании использовать принудительные блокировки в Linux вам придется сделать это на уровне файловой системы, для чего необходимо использовать опцию `-o mand` команды `mount`.

Применение принудительных блокировок к отдельным файлам разрешается включением бита set-group-ID и выключением бита group-execute (листинг 4.4). Поскольку установка бита set-group-ID теряет смысл при сброшенном бите group-execute, разработчики SVR3 выбрали именно такой способ для указания, что файл должен подвергаться принудительной, а не рекомендательной блокировке.

Что произойдет, если процесс попытается выполнить операцию чтения или записи в файл, для которого разрешена принудительная блокировка и указанная часть файла как раз находится под защитой блокировки для чтения или для записи, установленной другим процессом? Ответ на этот вопрос зависит от типа операции (чтение или запись), типа блокировки, установленной другим процессом (для чтения или для записи) и от того, был ли открыт дескриптор файла в неблокирующем режиме. В табл. 14.3 приводится восемь различных вариантов ответа на этот вопрос.

Таблица 14.3. Воздействие принудительных блокировок на операции чтения-записи из других процессов

Тип блокировки, установленной другим процессом	Дескриптор в блокирующем режиме		Дескриптор в неблокирующем режиме	
	`read`	`write`	`read`	`write`
Для чтения	OK	Блокируется	OK	`EAGAIN`
Для записи	Блокируется	Блокируется	`EAGAIN`	`EAGAIN`

Кроме операций read и write, указанных в табл. 14.3, принудительные блокировки могут также оказывать влияние на операцию открытия файла другим процессом. Обычно вызов функции open завершается успехом, даже если открываемый файл находится под защитой принудительной блокировки. В этом случае последующие операции чтения и записи будут выполняться в соответствии с правилами из табл. 14.3. Но если открываемый файл находится под защитой принудительной блокировки (неважно, для чтения или для записи) и функции open передается флаг O_TRUNC или O_CREAT, то в этом случае операция открытия файла будет завершаться неудачей с кодом ошибки EAGAIN и управление из функции open будет немедленно возвращено вызывающему процессу независимо от наличия флага O_NONBLOCK.

> Только ОС Solaris трактует использование флага O_CREAT в данной ситуации как ошибку. Linux допускает указание этого флага при открытии файла, на который установлена принудительная блокировка. То, что функция open возвращает признак ошибки при использовании флага O_TRUNC, вполне оправданно, потому что файл не может быть усечен, если он находится под защитой блокировки для чтения или для записи, установленной другим процессом. Генерация ошибки для флага O_CREAT не имеет большого смысла, поскольку этот флаг говорит о том, что файл должен быть создан, только если он еще не существует. Однако файл должен существовать, если другой процесс смог установить на него блокировку.

Изучение конфликтов между функцией open и блокировками может привести к неожиданным результатам. При разработке упражнений для этого раздела мы запускали тестовую программу, которая открывала файл (с разрешенным режимом принудительной блокировки), устанавливала блокировку для чтения на весь файл и затем приостанавливалась на некоторое время. (В табл. 14.3 показано, что блокировка для чтения должна предотвратить возможность записи в этот файл.) Пока программа находилась в режиме ожидания, было отмечено следующее поведение стандартных программ UNIX.

- Тот же самый файл мог быть отредактирован с помощью программы ed, и результаты записывались на диск! Получалось так, что принудительная блокировка вообще не оказывала никакого эффекта. С помощью системного вызова trace, который поддерживается некоторыми версиями UNIX, удалось выяснить, что редактор ed записывает обновленное содержимое во временный файл, удаляет оригинальный файл и затем переименовывает временный файл, называя его именем оригинального файла. Обязательная блокировка не оказывает воздействия на функцию unlink, в результате чего подобное оказалось возможным.

  > В ОС Solaris системный вызов trace используется командой truss(1). В FreeBSD и Mac OS X используются команды ktrace(1) и kdump(1). ОС Linux предоставляет команду strace(1) для трассировки системных вызовов, производимых процессом.

- Редактор vi не способен редактировать такой файл. Он мог прочитать содержимое файла, но при попытке сохранить его получал код ошибки EAGAIN. При попытках добавить в файл новые данные функция write блокировалась. Впрочем, мы предвидели такое поведение редактора vi.

- При использовании операторов перенаправления > и >> командной оболочки Korn shell для записи или добавления данных в файл мы получили ошибку «cannot create» («невозможно создать»).
- При использовании тех же самых операторов перенаправления в Bourne shell мы получали ошибку только в случае оператора >, выполнение же оператора >> просто блокировалось до момента снятия блокировки. (Различия в действиях оператора перенаправления >> для Korn shell и Bourne shell объясняется тем, что в Korn shell этот оператор вызывает функцию `open` с флагами `O_CREAT` и `O_APPEND`, а мы уже упоминали, что использование флага `O_CREAT` в подобной ситуации расценивается как ошибка. В командной оболочке Bourne shell функция `open` вызывается без флага `O_CREAT`, если запрошенный файл уже существует, поэтому обращение к функции `open` завершается успехом, а последующее обращение к функции `write` блокируется системой.)

Результаты могут различаться в зависимости от используемой версии операционной системы. Этот пример показывает, насколько осторожно следует подходить к использованию принудительных блокировок. Кроме того, пример с редактором `ed` показывает, что обойти принудительные блокировки не составляет особого труда.

Принудительные блокировки могут использоваться злонамеренным пользователем, чтобы ограничить доступ к некоторому общедоступному файлу только режимом чтения, установив на него принудительную блокировку для чтения. Такой прием не позволит никому изменить содержимое файла. (Разумеется, файл при этом должен быть доступен для установки принудительной блокировки, для чего пользователь должен иметь права на изменение прав доступа к файлу.) Представьте себе файл базы данных, который доступен на чтение всем и для которого установлена принудительная блокировка. Если злоумышленник сможет установить принудительную блокировку для чтения на весь файл, то никакой процесс не сможет записать в файл новые данные.

Пример

Программа, представленная листингом 14.6, определяет, поддерживает ли система принудительные блокировки.

Листинг 14.6. Определяет, поддерживает ли система принудительные блокировки

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int fd;
```

```
    pid_t pid;
    char buf[5];
    struct stat statbuf;

    if (argc != 2) {
        fprintf(stderr, "Использование: %s filename\n", argv[0]);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE)) < 0)
        err_sys("ошибка вызова функции open");
    if (write(fd, "abcdef", 6) != 6)
        err_sys("ошибка вызова функции write");

    /* включить бит set-group-ID и выключить бит group-execute */
    if (fstat(fd, &statbuf) < 0)
        err_sys("ошибка вызова функции fstat");
    if (fchmod(fd, (statbuf.st_mode &  S_IXGRP) | S_ISGID) < 0)
        err_sys("ошибка вызова функции fchmod");

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) {    /* родительский процесс */
        /* установить блокировку для записи на весь файл */
        if (write_lock(fd, 0, SEEK_SET, 0) < 0)
            err_sys("ошибка вызова функции write_lock");

        TELL_CHILD(pid);

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("ошибка вызова функции waitpid");
    } else {                 /* дочерний процесс */
        WAIT_PARENT();       /* дождаться, пока предок установит блокировку */

        set_fl(fd, O_NONBLOCK);

        /*
         * Прежде всего, посмотрим, возможно ли установить
         * другую блокировку на уже заблокированную область.
         */
        if (read_lock(fd, 0, SEEK_SET, 0) != -1) /* не ждать */
            err_sys("потомок: вызов read_lock завершился успехом");
        printf("вызов read_lock для заблокированного региона вернул код %d\n",
               errno);

        /* теперь попробуем читать из файла под принудительной блокировкой */
        if (lseek(fd, 0, SEEK_SET) == -1)
            err_sys("ошибка вызова функции lseek");
        if (read(fd, buf, 2) < 0)
            err_ret("ошибка чтения (принуд. блокировка сработала)");
        else
            printf("данные прочитаны (принуд. блокировка не сработала),
                   buf = %2.2s\n", buf);
    }
```

```
        exit(0);
    }
```

Эта программа создает файл и разрешает установку на него принудительных блокировок. После этого программа делится на два процесса. Родительский процесс устанавливает блокировку для записи на весь файл. Дочерний процесс устанавливает для дескриптора неблокирующий режим и затем пытается установить на файл блокировку для чтения, ожидая получить ошибку. Это позволит нам увидеть, возвращает ли система код ошибки EACCES или EAGAIN. После этого дочерний процесс переходит в начало файла и предпринимает попытку чтения из него. Если система поддерживает принудительные блокировки, то функция read должна вернуть признак ошибки с кодом EACCES или EAGAIN (поскольку дескриптор находится в неблокирующем режиме). В противном случае функция read вернет данные, которые удалось прочитать. Запуск этой программы в ОС Solaris 9 (которая поддерживает принудительные блокировки) дал следующие результаты:

```
$ ./a.out temp.lock
вызов read_lock для заблокированного региона вернул код 11
ошибка чтения (принуд. блокировка сработала): Resource temporarily unavailable
```

Если заглянуть в заголовочные файлы системы или в страницу справочного руководства intro(2), мы увидим, что коду 11 соответствует ошибка EAGAIN. В ОС FreeBSD 5.2.1 были получены следующие результаты:

```
$ ./a.out temp.lock
вызов read_lock для заблокированного региона вернул код 35
данные прочитаны (принуд. блокировка не сработала), buf = ab
```

Коду 35 соответствует ошибка EAGAIN. Принудительные блокировки не поддерживаются.

Пример

А теперь вернемся к главному вопросу этого раздела: что произойдет, если два пользователя попытаются редактировать один и тот же файл в одно и то же время? Обычные текстовые редакторы в UNIX не используют механизм блокировки записей – таким образом, ответ на этот вопрос остается прежним: результат будет соответствовать тому, что запишет в файл последний процесс.

Некоторые версии редактора vi используют рекомендательные блокировки записей в файле. Даже если мы будем пользоваться одной из таких версий vi, это все равно не сможет предотвратить использование других редакторов, которые ничего не знают о рекомендательных блокировках.

Если система поддерживает механизм принудительных блокировок, мы можем изменить свой любимый редактор таким образом, чтобы он пользовался ими (при наличии исходных текстов). Если исходные тексты редактора недоступны, мы могли бы попробовать написать программу, которая будет представлять собой интерфейс к редактору vi. Предполагается, что программа сразу же должна вызывать функцию fork, после которой родительский процесс просто становится в ожидание завершения потомка. Дочерний про-

цесс должен открыть указанный файл, разрешить для него установку принудительных блокировок, установить блокировку для записи на весь файл и затем запустить редактор `vi`. Пока работает редактор, файл будет находиться под защитой принудительной блокировки, вследствие чего никто из пользователей не сможет изменить его. По завершении работы редактора родительский процесс получит управление от функции `wait` и завершится сам.

Подобную программу можно написать достаточно быстро, но она не будет работать. Проблема заключается в том, что большинство известных редакторов считывают содержимое входного файла и закрывают его. Когда дескриптор, связанный с файлом, закрывается, освобождается и блокировка. Это означает, что когда редактор закрывает файл после считывания его содержимого, блокировка снимается. И нет никакой возможности предотвратить снятие блокировки.

В главе 20 мы будем использовать механизм блокировки записей в библиотеке для работы с базой данных, чтобы обеспечить параллельный доступ к ней из нескольких процессов. Мы также проведем ряд тестов на производительность, чтобы увидеть, какой эффект оказывают блокировки записей на производительность процесса.

14.4. STREAMS

Механизм STREAMS был реализован в System V как универсальный интерфейс взаимодействия с драйверами в ядре. Мы должны рассмотреть механизм STREAMS, чтобы понять интерфейс терминалов в System V, порядок использования функции `poll` для мультиплексирования ввода-вывода (раздел 14.5.2) и реализацию именованных и неименованных каналов, основанных на STREAMS (разделы 17.2 и 17.2.1).

> Не следует путать название STREAMS с термином *stream* (поток), который мы использовали при описании стандартной библиотеки ввода-вывода (раздел 5.2). Механизм streams был разработан Деннисом Ритчи [Ritchie 1984] для наведения порядка в традиционной символьной системе ввода-вывода и согласования сетевых протоколов. Позднее, после некоторых дополнений, механизм streams был добавлен в SVR3 и получил имя, которое записывается заглавными буквами. Полная поддержка механизма STREAMS (то есть система ввода-вывода для терминалов на основе STREAMS) была реализована в SVR4. Реализация SVR4 описана в [AT&T 1990d]. В [Rago 1993] обсуждаются вопросы программирования на основе STREAMS как на уровне пользователя, так и на уровне ядра.
>
> Механизм STREAMS определяется стандартом Single UNIX Specification как необязательная для реализации функциональная возможность (включена в стандарт как XSI STREAMS Option Group). Из всех четырех платформ, обсуждаемых в данной книге, только Solaris предоставляет полноценную поддержку механизма STREAMS. В Linux также существует подсистема STREAMS, но вы должны добавить ее самостоятельно, так как по умолчанию она обычно отключена.

Поток (stream) механизма STREAMS представляет собой дуплексный канал связи между пользовательским процессом и драйвером устройства. Для потока STREAMS не обязательно наличие драйвера аппаратного устройства –

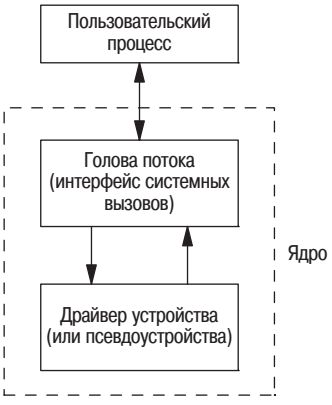


Рис. 14.4. *Простейший поток*

потоки STREAMS могут также использоваться для взаимодействия с драйверами псевдоустройств. На рис. 14.4 показано то, что называется простейшим потоком.

После головы потока мы можем поместить в поток промежуточные модули обработки. На рис. 14.5 показан поток с единственным промежуточным модулем. Мы отобразили взаимодействие между отдельными составляющими потока в виде двух стрелок в разных направлениях, чтобы подчеркнуть дуплексную природу потоков и показать, что обработка данных, движущихся в одном направлении, отличается от обработки данных, движущихся в другом направлении.

В поток может быть помещено любое количество промежуточных модулей. При добавлении нового модуля в поток он размещается сразу же за головой потока, проталкивая остальные модули на уровень ниже. (Очень похоже на стек

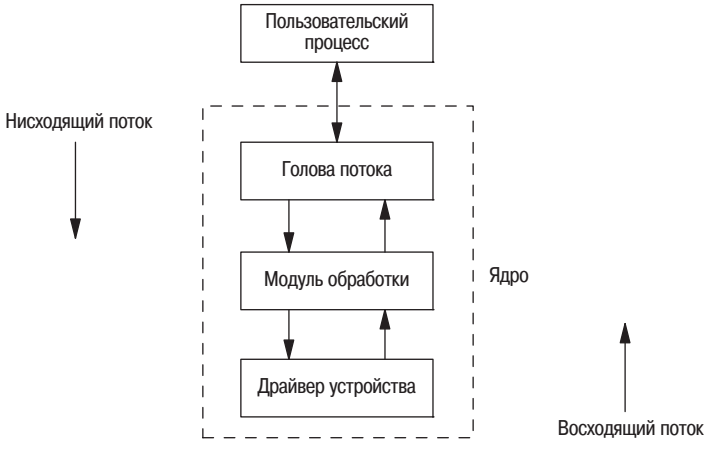


Рис. 14.5. *Поток с модулем промежуточной обработки данных*

магазинного типа, который построен по принципу «последний пришел – первый ушел».) На рис. 14.5 мы особо выделили нисходящую и восходящую стороны потока. Данные, которые записываются в поток, посылаются в нисходящем потоке. Данные, которые драйвер считывает с устройства, посылаются в восходящем потоке.

Модули STREAMS напоминают драйверы устройств в том смысле, что они работают в составе ядра и обычно встраиваются в ядро при его сборке. Если система поддерживает динамически загружаемые модули ядра (как, например, Linux или Solaris), то можно подгружать модули STREAMS динамически и пытаться поместить их в поток, однако нет никакой гарантии, что произвольные комбинации модулей и драйверов будут совместно работать должным образом.

Доступ к потокам STREAMS осуществляется посредством функций, которые обсуждались в главе 3: `open`, `close`, `read`, `write` и `ioctl`. Кроме того, для поддержки STREAMS в SVR3 были добавлены три новые функции (`getmsg`, `putmsg` и `poll`), а в SVR4 – еще две функции (`getpmsg` и `putpmsg`), предназначенные для обработки сообщений с различными приоритетами. Эти пять функций будут описаны далее в этом же разделе.

Путь, передаваемый функции `open` в аргументе *`pathname`*, в случае потоков STREAMS обычно начинается с каталога `/dev`. Просто просматривая имена устройств с помощью команды `ls -l`, нельзя сказать, является ли то или иное устройство устройством STREAMS. Все устройства STREAMS относятся к классу символьных устройств.

Хотя документация к STREAMS подразумевает, что мы можем разрабатывать модули промежуточной обработки и помещать их в поток, тем не менее для написания таких модулей требуются те же знания и умения, что и для написания драйверов устройств. Вообще, размещением модулей STREAMS в потоке занимаются только специализированные приложения или функции.

> До появления STREAMS терминальные устройства обслуживались с помощью механизма c-списков (c-list – символьный список). (C-списки для SVR2 и BSD4.4 описываются в [Bach 1986], раздел 10.3.1, и в [McKusuck et al. 1996], раздел 10.6, соответственно.) Добавление нового символьного устройства обычно сопровождалось написанием нового драйвера, в котором выполнялись все необходимые действия. Доступ к новым устройствам обычно производился через устройство, функционирующее в непосредственном режиме – это означает, что обращения к функциям `read` или `write` заканчивались прямо в драйвере устройства. Механизм STREAMS упорядочил этот вид взаимодействий, организовав обмен данными в виде сообщений STREAMS между головой потока и драйвером и допуская размещение любого количества модулей обработки в потоке.

Сообщения STREAMS

Весь ввод и вывод в механизме STREAMS основан на передаче сообщений. Обмен сообщениями между пользовательским процессом и головой потока производится с помощью функций `read`, `write`, `ioctl`, `getmsg`, `getpmsg`, `putmsg` и `putpmsg`. Аналогичным образом происходит обмен сообщениями между головой потока, модулями обработки и драйвером устройства.

Сообщения, циркулирующие между пользовательским процессом и головой потока, состоят из типа сообщения, необязательной управляющей информации и необязательных данных. В табл. 14.4 показано, как определяется тип посылаемого сообщения исходя из значений аргументов функций `write`, `putmsg` и `putpmsg`. Управляющая информация и данные передаются в виде структуры `strbuf`:

```
struct strbuf
    int maxlen; /* размер буфера */
    int len;    /* текущее количество байт в буфере */
    char *buf;  /* указатель на буфер */
};
```

При передаче сообщения с помощью функций `putmsg` или `putpmsg` в поле `len` заносится количество байт данных в буфере. При приеме сообщения с помощью функций `getmsg` или `getpmsg` поле `maxlen` задает размер буфера (чтобы ядро не смогло переполнить его), а в поле `len` ядро возвращает фактический объем данных в буфере. Позднее мы увидим, что сообщения с нулевой длиной вполне допустимы и что значение -1 в поле `len` указывает на отсутствие управляющей информации или данных.

Для чего необходима одновременная передача управляющей информации и данных? Она позволяет упростить реализацию служебных интерфейсов между пользовательским процессом и потоком. В главе 5 [Olander, McGrath, and Israel 1986] описывается оригинальная реализация служебных интерфейсов в System V. В главе 5 [AT&T 1990d] приводится подробное описание служебных интерфейсов с простым примером. Вероятно, самым известным служебным интерфейсом System V является описанный в главе 4 [Rago 1993] интерфейс транспортного уровня (TLI – Transport Layer Interface), который предоставляет интерфейс к сетевой подсистеме.

Другой пример, когда необходима передача управляющей информации, – посылка сообщений через сеть без установления соединения (дейтаграммы). Чтобы послать сообщение, мы должны определить содержимое сообщения (данные) и адрес назначения (управляющая информация). Если мы были бы лишены возможности отсылать управляющую информацию и данные вместе, то нам потребовалась бы некоторая специальная схема. Например, пришлось бы, возможно, указывать адрес с помощью функции `ioctl` и затем вызывать `write` для передачи самих данных. Как вариант, можно было бы отвести под адрес первые N байт данных, записываемых функцией `write`. Отделение управляющей информации от данных и предоставление функций для работы с ними (`putmsg` и `getmsg`) – более понятный способ взаимодействий с потоками.

Всего существует около 25 различных типов сообщений, но лишь немногие из них используются при взаимодействии между пользовательским процессом и головой потока. Остальные типы сообщений передаются вниз и вверх внутри ядра. (Эти типы сообщений будут интересны тем, кто занимается разработкой модулей промежуточной обработки STREAMS, прикладные же программисты могут просто игнорировать их.) При работе с функциями `read`, `write`, `getmsg`, `getpmsg`, `putmsg` и `putpmsg` мы столкнемся всего с тремя типами сообщений:

- M_DATA (ввод-вывод пользовательских данных)
- M_PROTO (управляющая информация протокола)
- M_PCPROTO (высокоприоритетная управляющая информация протокола)

Каждое сообщение в потоке помещается в очередь с определенным приоритетом:

- Высокоприоритетные сообщения (наивысший приоритет)
- Приоритетные сообщения
- Обычные сообщения (низший приоритет)

Обычное сообщение – это приоритетное сообщение с уровнем приоритета, равным 0. Приоритетные сообщения могут иметь уровень приоритета от 1 до 255, где большее число соответствует более высокому приоритету. Высокоприоритетные сообщения – это специальный вид сообщений в том смысле, что очередь допускает наличие только одного такого сообщения в каждый конкретный момент времени. Дополнительные высокоприоритетные сообщения будут утеряны при попытке поместить их в очередь, если в ней уже имеется высокоприоритетное сообщение.

Каждый модуль STREAMS имеет две входные очереди. Одна принимает сообщения от вышестоящего модуля (нисходящий поток данных от головы потока к драйверу устройства), а другая – от нижестоящего модуля (восходящий поток данных от драйвера к голове потока). Сообщения во входной очереди упорядочиваются в соответствии с их приоритетами. В табл. 14.4 показано, как аргументы функций write, putmsg и putpmsg влияют на уровень приоритета отправляемого сообщения.

Существуют и другие типы сообщений, которые мы не будем рассматривать в этой книге. Например, если голова потока принимает в восходящем потоке сообщение типа M_SIG, то будет сгенерирован сигнал. Именно таким образом модуль дисциплины обслуживания терминала отправляет сигнал, генерируемый терминалом, группе процессов переднего плана, ассоциированной с управляющим терминалом.

Функции putmsg и putpmsg

Сообщение STREAMS (управляющая информация, данные или и то и другое) записывается в поток с помощью функции putmsg или putpmsg. Разница между этими функциями состоит в том, что последняя из них позволяет определить приоритет сообщения.

```
#include <stropts.h>

int putmsg(int filedes, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flag);

int putpmsg(int filedes, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band, int flag);
```

Обе возвращают 0 в случае успеха, –1 в случае ошибки

Для записи сообщения в поток может также использоваться функция write, которая эквивалентна функции putmsg, вызываемой без управляющей информации и со значением 0 в аргументе *flag*.

Эти две функции могут посылать сообщения с различными уровнями приоритета: обычные сообщения, приоритетные сообщения и высокоприоритетные сообщения. В табл. 14.4 приводятся различные комбинации входных аргументов этих функций и соответствующие им типы сообщений.

Таблица 14.4. Типы сообщений STREAMS, создаваемые функциями write, putmsg и putpmsg

Функция	Управляющая информация?	Данные?	*band*	*flag*	Тип сообщения
write	–	Да	–	–	M_DATA (обычное)
putmsg	Нет	Нет	–	0	Сообщение не передается, возвращается 0
putmsg	Нет	Да	–	0	M_DATA (обычное)
putmsg	Да	Да или нет	–	0	M_PROTO (обычное)
putmsg	Да	Да или нет	–	RS_HIPRI	M_PCPROTO (высокоприоритетное)
putmsg	Нет	Да или нет	–	RS_HIPRI	Ошибка EINVAL
putpmsg	Да или нет	Да или нет	0–255	0	Ошибка EINVAL
putpmsg	Нет	Нет	0–255	MSG_BAND	Сообщение не передается, возвращается 0
putpmsg	Нет	Да	0	MSG_BAND	M_DATA (обычное)
putpmsg	Нет	Да	1–255	MSG_BAND	M_DATA (приоритетное)
putpmsg	Да	Да или нет	0	MSG_BAND	M_DATA (обычное)
putpmsg	Да	Да или нет	1–255	MSG_BAND	M_PROTO (приоритетное)
putpmsg	Да	Да или нет	0	MSG_HIPRI	M_PCPROTO (высокоприоритетное)
putpmsg	Нет	Да или нет	0	MSG_HIPRI	Ошибка EINVAL
putpmsg	Да или нет	Да или нет	Ненулевое значение	MSG_HIPRI	Ошибка EINVAL

Прочерк в ячейках означает, что данная функция не поддерживает данный аргумент. Значение «нет» в колонке «Управляющая информация?» подразумевает либо передачу пустого указателя в аргументе *ctlptr*, либо значения –1 в поле *ctlptr->len*. Значение «да» в колонке «Управляющая информация?» подразумевает, что аргумент *ctlptr* не является пустым указателем и в поле *ctlptr->len* записано значение, которое больше или равно нулю. То же самое относится и к колонке «Данные?» (только вместо *ctlptr* подразумевается аргумент *dataptr*).

Операции ioctl в STREAMS

В разделе 3.15 мы уже говорили, что функция ioctl может сделать все, чего нельзя сделать с помощью других функций ввода-вывода. Механизм STREAMS продолжает эту традицию.

В Linux и Solaris определено почти 40 различных операций, которые могут быть выполнены над потоком с помощью функции ioctl. Большая часть этих операций описана на странице справочного руководства streamio(7). Чтобы иметь возможность выполнять эти операции, программа должна подключать заголовочный файл <stropts.h>. Второй аргумент функции ioctl, *request* определяет выполняемую операцию. Все имена операций начинаются с последовательности I_. Значение и тип третьего аргумента зависит от выполняемой операции – это может быть указатель и на целое число, и на структуру.

Пример – функция isastream

Иногда необходимо определить, ссылается ли дескриптор на поток STREAMS или нет. Для этих целей ОС Linux и Solaris предоставляют функцию isastream. Она очень похожа на функцию isatty, которая определяет, ссылается ли дескриптор на терминальное устройство (раздел 18.9).

```
#include <stropts.h>

int isastream(int filedes);
```

Возвращает 1 (истина), если это поток STREAMS,
0 (ложь) в противном случае

Подобно isatty, это очень простая функция – она пытается выполнить операцию ioctl, которая считается допустимой только для устройств STREAMS. В листинге 14.7 приводится одна из возможных реализаций этой функции. Здесь мы используем операцию I_CANPUT, которая проверяет, возможно ли поместить в очередь сообщение с заданным приоритетом (в данном примере – 0). Эта операция не вызывает изменений в потоке.

Листинг 14.7. Проверка, ссылается ли дескриптор на устройство STREAMS

```
#include <stropts.h>
#include <unistd.h>

int
isastream(int fd)
{
    return(ioctl(fd, I_CANPUT, 0) != -1);
}
```

Для проверки этой функции мы можем использовать программу из листинга 14.8.

Листинг 14.8. Проверка функции isastream

```
#include "apue.h"
#include <fcntl.h>
```

```
int
main(int argc, char *argv[])
{
    int i, fd;

    for (i = 1; i < argc; i++) {
        if ((fd = open(argv[i], O_RDONLY)) < 0) {
            err_ret("невозможно открыть %s", argv[i]);
            continue;
        }
        if (isastream(fd) == 0)
            err_ret("%s: не является устройством STREAMS", argv[i]);
        else
            err_msg("%s: устройство STREAMS", argv[i]);
    }
    exit(0);
}
```

Если запустить эту программу в ОС Solaris 9, мы увидим различные виды ошибок, возвращаемых функцией `ioctl`:

```
$ ./a.out /dev/tty /dev/fb /dev/null /etc/motd
/dev/tty: устройство STREAMS
/dev/fb: не является устройством STREAMS: Invalid argument
/dev/null: не является устройством STREAMS: No such device or address
/etc/motd: не является устройством STREAMS: Inappropriate ioctl for device
```

Обратите внимание, что устройство `/dev/tty` в Solaris является устройством STREAMS. Специальный файл символьного устройства `/dev/fb` не является устройством STREAMS, но поддерживает управление с помощью функции `ioctl`. Подобные устройства возвращают код ошибки `EINVAL`, если запрошена неподдерживаемая операция. Специальный файл символьного устройства `/dev/null` не поддерживает выполнение операций с помощью функции `ioctl`, в результате для этого устройства была получена ошибка `ENODEV`. И, наконец, файл `/etc/motd` является обычным файлом, поэтому для него была получена классическая ошибка `ENOTTY`. Нам ни разу не удалось получить ошибку, которую можно было бы ожидать: `ENOSTR` («Device is not a stream» – «Устройство не является потоком»).

> Коду ошибки `ENOTTY` ранее соответствовало сообщение «Not a typewriter» («Не является печатающим устройством»); это архаизм, оставшийся с тех времен, когда ядро UNIX возвращало ошибку `ENOTTY` при попытке выполнения операций `ioctl` над дескрипторами, соответствующими файлам, которые не являются специальными файлами символьных устройств. В Solaris это сообщение было заменено на «Inappropriate ioctl for device» («Устройство не поддерживает операции `ioctl`»).

Пример

Если в аргументе *request* функции `ioctl` передается код `I_LIST`, система возвращает имена всех модулей, размещенных в потоке, включая самый верхний драйвер. (Мы упомянули самый верхний драйвер, потому что в случае муль-

типлексирования может использоваться более одного драйвера. Вопрос мультиплексирования драйверов подробно обсуждается в главе 12 [Rago 1993].) Третий аргумент функции должен быть указателем на структуру `str_list`:

```
struct str_list {
    int sl_nmods;                /* количество элементов массива */
    struct str_mlist *sl_modlist; /* указатель на первый элемент массива */
};
```

Мы должны записать в поле `sl_modlist` указатель на первый элемент массива структур `str_mlist`, а в поле `sl_nmods` определить количество элементов в массиве:

```
struct str_mlist {
    char l_name[FMNAMESZ+1];  /* имя модуля, завершающееся нулевым символом */
};
```

Константа `FMNAMESZ` определена в заголовочном файле `<sys/conf.h>` и часто имеет значение 8. В поле `l_name` отводится дополнительный байт для завершающего нулевого символа.

Если в третьем аргументе функции `ioctl` передается 0, то вместо имен модулей возвращается их количество в потоке. Мы можем использовать это обстоятельство для определения количества модулей, чтобы затем разместить требуемое количество структур `str_mlist`.

Листинг 14.9 демонстрирует использование операции `I_LIST`. Имена модулей обработки ничем не отличаются от имен драйверов, но мы точно знаем, что последний элемент в списке соответствует драйверу устройства, расположенному на дне потока.

Листинг 14.9. Вывод списка имен модулей в потоке

```
#include "apue.h"
#include <fcntl.h>
#include <stropts.h>
#include <sys/conf.h>

int
main(int argc, char *argv[])
{
    int fd,i, nmods;
    struct str_list list;

    if (argc != 2)
        err_quit("Использование: %s <полный_путь_к_файлу>", argv[0]);
    if ((fd = open(argv[1], O_RDONLY)) < 0)
        err_sys("невозможно открыть %s", argv[1]);
    if (isastream(fd) == 0)
        err_quit("%s не является потоком STREAMS", argv[1]);

    /*
     * Получить количество модулей.
     */
    if ((nmods = ioctl(fd, I_LIST, (void *) 0)) < 0)
```

```
        err_sys("ошибка операции I_LIST для получения количества модулей");
    printf("количество модулей = %d\n", nmods);

    /*
     * Разместить массив структуры требуемого размера.
     */
    list.sl_modlist = calloc(nmods, sizeof(struct str_mlist));
    if (list.sl_modlist == NULL)
        err_sys("ошибка вызова функции calloc");
    list.sl_nmods = nmods;

    /*
     * Получить имена модулей.
     */
    if (ioctl(fd, I_LIST, &list) < 0)
        err_sys("ошибка операции I_LIST для получения списка");

    /*
     * Вывести полученные имена.
     */
    for (i = 1; i <= nmods; i++)
        printf(" %s: %s\n", (i == nmods) ? "драйвер" : "модуль",
               list.sl_modlist++->l_name);
    exit(0);
}
```

С помощью этой программы мы попытались получить список модулей STREAMS, обслуживающих консоль и псевдотерминал, открытый через сетевое соединение. В результате мы получили следующее:

```
$ who
sar console May 1 18:27
sar pts/7 Jul 12 06:53
$ ./a.out /dev/console
количество модулей = 5
  модуль: redirmod
  модуль: ttcompat
  модуль: ldterm
  модуль: ptem
  драйвер: pts
$ ./a.out /dev/pts/7
количество модулей = 4
  модуль: ttcompat
  модуль: ldterm
  модуль: ptem
  драйвер: pts
```

В обоих случаях мы получили одни и те же имена модулей, за исключением того, что консоль на самом верхнем уровне имеет дополнительный модуль, который помогает выполнять перенаправление виртуальной консоли. В данном случае терминал работает под управлением многооконной системы, поэтому устройство /dev/console фактически является псевдотерминалом. К псевдотерминалам мы еще вернемся в главе 19.

Функция write и устройства STREAMS

В табл. 14.4 указано, что функция write при работе с устройствами STREAMS генерирует сообщения типа M_DATA. В общем и целом это действительно так, но существуют некоторые особенности, которые мы должны рассмотреть. Во-первых, головной модуль потока определяет минимальный и максимальный размеры пакетов, которые можно передать вниз. (У нас нет возможности запросить у модуля эти значения.) При попытке записать количество байт, превышающее максимально возможное, голова потока обычно разбивает данные на пакеты максимально возможного размера (последний пакет может иметь размер меньше максимально возможного).

Далее следует рассмотреть, что произойдет, если функция write запишет количество байт, равное нулю. Если поток не является именованным или неименованным каналом, сообщения с нулевой длиной отправляются далее вниз. В противном случае операция записи сообщения нулевой длины игнорируется для сохранения совместимости с предыдущими версиями. Такое поведение по умолчанию для каналов можно изменить с помощью функции ioctl, установив для потока режим записи.

Режим записи

Функция ioctl имеет две команды, которые могут использоваться для определения и назначения режима записи потока. Установка значения I_GWROPT в аргументе *request* требует, чтобы в третьем аргументе функции был передан указатель на целое число, в котором функция вернет текущее значение режима записи. Если в аргументе *request* передается значение I_SWROPT, то в третьем аргументе должно передаваться целое число, которое определяет режим записи потока. Как и в случае с флагами дескриптора и флагами состояния файла (раздел 3.14), мы всегда должны сначала получить текущее значение режима записи, изменить его и установить повторным вызовом ioctl. Не следует использовать для установки режима записи абсолютные значения, поскольку это может привести к сбросу некоторых других битов.

В настоящее время определено только два режима записи:

SNDZERO Запись сообщения нулевой длины в канал вызовет передачу этого сообщения дальше вниз. По умолчанию функция write не посылает вниз сообщения с нулевой длиной.

SNDPIPE Если после появления ошибки в потоке вызывается функция write или putmsg, вызывающему процессу посылается сигнал SIGPIPE.

Кроме того, потоку также может быть назначен режим чтения, но об этом мы поговорим после того, как рассмотрим функции getmsg и getpmsg.

Функции getmsg и getpmsg

Сообщения STREAMS могут быть прочитаны из головы потока с помощью функций read, getmsg и getpmsg.

```
#include <stropts.h>

int getmsg(int filedes, struct strbuf *restrict ctlptr,
           struct strbuf *restrict dataptr, int *restrict flagptr);

int getpmsg(int filedes, struct strbuf *restrict ctlptr,
            struct strbuf *restrict dataptr, int *restrict bandptr,
            int *restrict flagptr);
```

Обе возвращают неотрицательное значение в случае успеха, –1 в случае ошибки

Обратите внимание: целые числа, на которые ссылаются аргументы *flagptr* и *bandptr*, должны быть определены перед вызовом функций; они определяют желаемый тип запрашиваемого сообщения. Эти числа изменяются по возвращении из функций – в них указывается тип прочитанного сообщения.

Если указатель flagptr ссылается на число 0, функция getmsg возвращает следующее сообщение из очереди чтения головы потока. Если следующее сообщение является высокоприоритетным сообщением, по указанному адресу будет записано значение RS_HIPRI. Чтобы принимать только высокоприоритетные сообщения, необходимо перед обращением к функции getmsg записать по адресу *flagptr* значение RS_HIPRI.

Функция getpmsg использует иной набор констант. Чтобы функция getpmsg принимала только высокоприоритетные сообщения, следует записать по адресу *flagptr* значение MSG_HIPRI. Чтобы принимать с помощью этой функции только сообщения с приоритетом равным или выше заданного (включая высокоприоритетные), необходимо записать по адресу *flagptr* значение MSG_BAND, а по адресу *bandptr* – требуемое (ненулевое) значение уровня приоритета. Если нам нужно получить первое доступное для чтения сообщение, то по адресу *flagptr* мы должны записать значение MSG_ANY; когда функция вернет управление, по этому адресу будет записано значение MSG_HIPRI или MSG_BAND в зависимости от типа принятого сообщения. Если принятое сообщение не является высокоприоритетным, по адресу *bandptr* будет записано значение приоритета для этого сообщения.

Если в аргументе *ctlptr* передается пустой указатель или в поле *ctlptr->maxlen* записано число –1, то управляющая информация останется в голове потока и не будет передана вызывающему процессу. Аналогично, если в аргументе *dataptr* передается пустой указатель или в поле *dataptr->maxlen* записано число –1, данные из сообщения останутся в голове потока и не будут переданы вызывающему процессу. В ином случае мы получим такой объем управляющей информации и данных, который указанный буфер сможет вместить, а оставшаяся часть информации останется в очереди головы потока и сможет быть получена следующим обращением к функции.

Если в результате обращения к функции getmsg или getpmsg будет получено сообщение, они вернут значение 0. Если в голове потока останется часть управляющей информации, которая не поместилась в предоставленный буфер, функции вернут значение MORECTL. Аналогично, если в голове потока ос-

танется часть данных, которые не поместились в предоставленный буфер, функции вернут значение MOREDATA. Если же в голове потока останется как управляющая информация, так и данные, которые не поместились в предоставленные буферы, будет возвращено значение MORECTL|MOREDATA.

Режим чтения

Мы также должны рассмотреть чтение сообщений из потока STREAMS с помощью функции read. При использовании этой функции возможны две проблемы.

1. Что произойдет, если размер сообщения в потоке превысит размер приемного буфера?
2. Что произойдет, если вызвать функцию read для чтения сообщения, которое содержит управляющую информацию?

По умолчанию ситуация 1 обслуживается в так называемом режиме потока байтов. В этом режиме функция read получает данные из потока до тех пор, пока не будет получено запрошенное количество байт или пока не будут прочитаны все данные из потока. Границы сообщений в этом режиме игнорируются. Во втором случае, если в начале очереди находится сообщение, содержащее управляющую информацию, функция read по умолчанию завершается с признаком ошибки. Мы можем изменить поведение функции read, принятое по умолчанию.

Если в аргументе *request* функции ioctl передается значение I_GRDOPT, то третий аргумент должен представлять собой указатель на целое число, и по указанному адресу функция возвратит текущий режим чтения. Когда в аргументе *request* передается значение I_SRDOPT, в качестве третьего аргумента функции должно передаваться целое число, которое определяет устанавливаемый режим чтения. Режим чтения определяется следующими тремя константами:

RNORM — Обычный режим потока байтов (по умолчанию), который работает так, как описано выше.

RMSGN — Режим выемки сообщений без потерь с учетом границ между ними. Функция read будет изымать данные из очереди до тех пор, пока не будет получено запрошенное количество байт или пока не встретится граница между сообщениями. Если функция прочитала только часть сообщения, остальная часть данных остается в потоке и может быть прочитана последующими обращениями к функции read.

RMSGD — Режим выемки сообщений с потерями. Напоминает предыдущий режим, но если была прочитана только часть сообщения, то оставшаяся его часть будет утрачена.

При обслуживании сообщений, содержащих управляющую информацию протокола, для установки режима чтения могут использоваться три дополнительные константы:

RPROTNORM Нормальный режим: функция read возвращает код ошибки EBADMSG. Этот режим используется по умолчанию.

RPROTDAT Режим чтения информации протокола как данных: функция read будет возвращать управляющую информацию так же, как данные.

RPROTDIS Режим чтения с потерей информации протокола: в результате обращения к функции read управляющая информация теряется, но при этом возвращаются данные сообщения.

При установке режима чтения допускается указывать только одну константу, определяющую режим чтения, и одну константу, определяющую режим чтения управляющей информации протокола. Значение режима чтения по умолчанию – RNORM|RPROTNORM.

Пример

Программа, представленная листингом 14.10, по сути выполняет те же действия, что и программа из листинга 3.3, но вместо функции read она использует функцию getmsg.

Листинг 14.10. Копирование данных со стандартного ввода на стандартный вывод с помощью функции getmsg

```
#include "apue.h"
#include <stropts.h>

#define BUFFSIZE 4096

int
main(void)
{
    int n, flag;
    char ctlbuf[BUFFSIZE], datbuf[BUFFSIZE];
    struct strbuf ctl, dat;

    ctl.buf = ctlbuf;
    ctl.maxlen = BUFFSIZE;
    dat.buf = datbuf;
    dat.maxlen = BUFFSIZE;
    for ( ; ; ) {
        flag = 0;                       /* запросить любое сообщение */
        if ((n = getmsg(STDIN_FILENO, &ctl, &dat, &flag)) < 0)
            err_sys("ошибка вызова функции getmsg");
        fprintf(stderr, "flag = %d, ctl.len = %d, dat.len = %d\n",
        flag, ctl.len, dat.len);
        if (dat.len == 0)
            exit(0);
        else if (dat.len > 0)
            if (write(STDOUT_FILENO, dat.buf, dat.len) != dat.len)
                err_sys("ошибка вызова функции write");
    }
}
```

Запустив эту программу в ОС Solaris, где и каналы, и терминалы реализованы с использованием STREAMS, мы получили следующие результаты:

```
$ echo привет, МИР | ./a.out          для этого необходимы каналы
                                      на базе STREAMS
flag = 0, ctl.len = -1, dat.len = 11
привет, МИР
flag = 0, ctl.len = 0, dat.len = 0    признак разрыва связи с STREAMS
$ ./a.out                             для этого необходимы терминалы
                                      на базе STREAMS
это первая строка
flag = 0, ctl.len = -1, dat.len = 17
это первая строка
и вторая строка
flag = 0, ctl.len = -1, dat.len = 15
и вторая строка
^D                                    ввод символа EOF
flag = 0, ctl.len = -1, dat.len = 0   конец файла для терминала
                                      не означает разрыв соединения
$ ./a.out < /etc/motd
ошибка вызова функции getmsg: Not a stream device
```

Закрытие канала (когда команда `echo` завершает работу) для программы из листинга 14.10 выглядит как разрыв соединения с STREAMS: и объем управляющей информации, и объем данных равны нулю. (Каналы будут обсуждаться в разделе 15.2.) Однако в случае с терминалом при вводе символа конца файла только объем данных равен нулю. Для данного терминала признак конца файла не означает разрыв соединения с STREAMS. Как мы и ожидали, перенаправление стандартного ввода в файл, не являющийся устройством STREAMS, вызывает ошибку при обращении к функции `getmsg`.

14.5. Мультиплексирование ввода-вывода

При чтении из одного дескриптора и записи в другой можно в цикле использовать блокирующие операции ввода-вывода – например, так:

```
while ((n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("ошибка вызова функции write");
```

Мы уже много раз встречали такую форму блокирующего ввода-вывода. А что делать, если нужно читать из двух дескрипторов? В этом случае нельзя использовать блокирующую операцию чтения для любого из дескрипторов, так как данные могут появиться в одном дескрипторе, в то время как процесс заблокирован в ожидании появления данных в другом. Для решения этой проблемы существуют различные приемы.

Давайте рассмотрим структуру программы `telnet`(1). Эта программа читает данные с терминала (стандартный ввод) и записывает их в сетевое соединение, и в обратном порядке – читает из сетевого соединения и записывает на терминал (стандартный вывод). На другом конце сетевого соединения де-

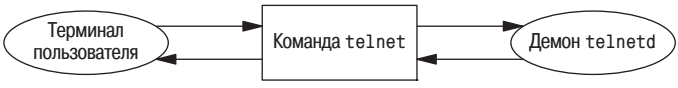


***Рис. 14.6.** Схема работы команды telnet*

мон telnetd считывает то, что мы ввели с терминала, и передает это командной оболочке. Вывод, полученный в результате запуска команд, отправляется обратно через команду telnet и отображается на нашем терминале. Схема этих действий изображена на рис. 14.6.

Процесс telnet имеет два дескриптора для ввода и два для вывода. Эта программа не может использовать блокирующие операции чтения для какого-либо из дескрипторов ввода, так как заранее не известно, в каком из них имеются готовые для чтения данные.

Один из вариантов решения этой проблемы – разделить процесс на две части (с помощью функции fork), каждая из которых будет обслуживать одно направление передачи данных. Схема такого решения показана на рис. 14.7. (Команда cu(1) из пакета uucp в System V была реализована примерно таким образом.)

Используя схему с двумя процессами, мы можем позволить каждому из них производить блокирующую операцию чтения. Но здесь появляется другая проблема, связанная с завершением работы. Если признак конца файла будет принят дочерним процессом (сетевое соединение будет закрыто со стороны демона telnetd), дочерний процесс завершится, а родительский процесс будет извещен об этом сигналом SIGCHLD. Если первым завершится родительский процесс (пользователь введет с терминала признак конца файла), он может сообщить потомку о своем завершении – например, с помощью сигнала SIGUSR1, – но это несколько усложнит программу.

Вместо схемы с двумя процессами можно использовать схему с двумя потоками. Это поможет избежать сложностей, связанных с завершением, но потребует введения синхронизации между потоками, в результате сложность программы может не только не уменьшиться, но увеличиться еще больше.

Мы могли бы использовать неблокирующие операции ввода-вывода, установив для обоих дескрипторов неблокирующий режим, и попытаться прочитать данные из первого дескриптора функцией read. Если данные присутствуют, мы сможем получить их и обработать. Если данных нет, функция read сразу же вернет управление. Затем то же самое проделаем со вторым дескрип-

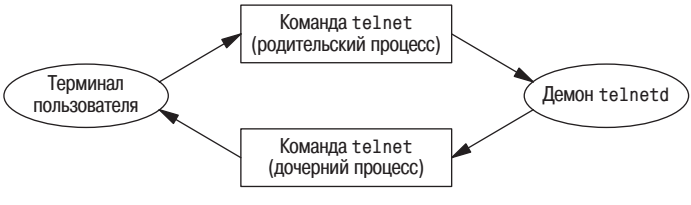


***Рис. 14.7.** Реализация программы telnet в виде двух процессов*

тором. После этого можно подождать некоторое время (возможно, несколько секунд) и повторить попытку чтения из первого дескриптора. Циклы такого рода называются *опросом*. Основная проблема такого решения – напрасный расход процессорного времени. Большую часть времени данные для чтения отсутствуют, и обращения к системному вызову `read` будут производиться вхолостую. Кроме того, мы должны решить, как долго ждать перед началом нового цикла. Несмотря на то, что такой подход пригоден для любой системы, которая поддерживает неблокирующие операции ввода-вывода, в многозадачных системах его следует избегать.

Еще одно решение связано с операциями асинхронного ввода-вывода. Чтобы воспользоваться ими, мы должны сообщить ядру о необходимости посылать процессу сигнал, когда дескриптор будет готов для ввода-вывода. С этим решением связаны две проблемы. Во-первых, не все системы поддерживают эту функциональность (в стандарте Single UNIX Specification она относится к разряду необязательных). В System V для этих целей предусмотрен сигнал `SIGPOLL`, но он посылается ядром только в том случае, если дескриптор ссылается на устройство `STREAMS`. В BSD есть похожий сигнал `SIGIO`, но и он имеет примерно такие же ограничения – сигнал посылается только в том случае, если дескриптор ссылается на терминальное устройство или сетевое соединение. Во-вторых, при использовании такой методики процесс может назначить сигнал (`SIGPOLL` или `SIGIO`) лишь для одного дескриптора. Если мы разрешим доставку сигнала для двух дескрипторов (в данном примере речь идет о чтении из двух дескрипторов), то, получив его, мы не сможем сказать, какой из дескрипторов готов к выполнению операции чтения. Чтобы проверить готовность дескрипторов, придется перевести каждый из них в неблокирующий режим и попытаться прочитать данные из обоих. Краткое описание асинхронного ввода-вывода приводится в разделе 14.6.

Наилучшим решением является *мультиплексирование ввода-вывода*. Для этого необходимо создать список дескрипторов, представляющих для нас определенный интерес (обычно список состоит более чем из одного дескриптора), и вызвать функцию, которая не вернет управление до тех пор, пока один из дескрипторов не будет готов к выполнению операции ввода-вывода. По возвращении из функции мы получим информацию о том, какие дескрипторы готовы для ввода-вывода.

Для выполнения мультиплексирования ввода-вывода предназначены три функции – `poll`, `pselect` и `select`. В табл. 14.5 приводятся сведения о том, какие платформы их поддерживают. Обратите внимание: функция `select` определяется базовым стандартом POSIX.1, а `poll` – как расширение XSI базового стандарта.

> Стандарт POSIX указывает, что для добавления всех определений, необходимых для обращения к функции `select`, программа должна подключать заголовочный файл `<sys/select.h>`. Однако исторически сложилось так, что некоторые системы еще не до конца соответствуют стандарту, и поэтому приходится подключать к программе три других заголовочных файла. Чтобы уточнить, какой из вариантов поддерживается в вашей системе, обращайтесь к странице справочного руководства к функции `select`.

Устаревшие системы требуют подключения заголовочных файлов <sys/types.h>, <sys/time.h> и <unistd.h>.

Возможность мультиплексирования ввода-вывода с помощью функции select появилась в 4.2BSD. Эта функция всегда могла работать с любыми дескрипторами, хотя основное ее предназначение – работа с дескрипторами терминалов и сетевых соединений. В SVR3, с появлением механизма STREAMS, была добавлена функция poll. Однако изначально функция poll могла работать только с устройствами STREAMS. Начиная с версии SVR4 в нее была добавлена поддержка любых типов дескрипторов.

Таблица 14.5. Поддержка мультиплексирования ввода-вывода различными платформами

Система	poll	pselect	select	<sys/select.h>
SUS	XSI	•	•	•
FreeBSD 5.2.1	•	•	•	
Linux 2.4.22	•	•	•	•
Mac OS X 10.3	•	•	•	
Solaris 9	•		•	•

14.5.1. Функции select и pselect

Функция select позволяет производить мультиплексирование ввода-вывода на любой POSIX-совместимой платформе. Аргументы, которые передаются функции select, сообщают ядру:

- Список интересующих нас дескрипторов
- Какие состояния каждого из дескрипторов нас интересуют (готовность к чтению, готовность к записи, наличие исключительной ситуации)
- Как долго ожидать изменения состояния дескриптора (не ограничивать время ожидания, определить некоторый интервал времени или вообще не ждать)

По возвращении из функции ядро сообщает:

- Общее количество дескрипторов, перешедших в требуемое состояние
- Какие из дескрипторов готовы для чтения, какие для записи, и для каких была обнаружена исключительная ситуация

Обладая этой информацией, мы можем производить соответствующие операции ввода-вывода (обычно чтение или запись), заранее зная, что они не будут заблокированы.

```
#include <sys/select.h>

int select(int maxfdp1, fd_set *restrict readfds, fd_set *restrict writefds,
           fd_set *restrict exceptfds, struct timeval *restrict tvptr);
```

Возвращает количество дескрипторов, готовых к выполнению операции, 0 – в случае истечения тайм-аута, –1 в случае ошибки

Для начала рассмотрим последний аргумент. Он определяет продолжительность времени ожидания:

```
struct timeval {
    long tv_sec;  /* секунды */
    long tv_usec; /* и микросекунды */
};
```

Возможны три различных состояния этого аргумента.

tvptr == NULL

Время ожидания не ограничено. Это бесконечное ожидание может быть прервано при перехвате сигнала. Возврат из функции возможен только тогда, когда хотя бы один из дескрипторов будет готов к выполнению операции или когда будет перехвачен сигнал. В последнем случае функция select возвращает значение –1 с кодом ошибки EINTR в переменной errno.

tvptr?>tv_sec == 0 && *tvptr?>tv_usec* == 0

Вообще не ждать. В этом случае просто производится проверка всех указанных дескрипторов, и управление тут же возвращается в вызывающую программу. Это один из способов запросить информацию об изменении состояния для целой группы дескрипторов, не блокируя процесс в функции select.

tvptr?>tv_sec != 0 || *tvptr?>tv_usec* != 0

Ждать не более заданного количества секунд и микросекунд. Возврат из функции возможен, когда хотя бы один из дескрипторов будет готов к выполнению операции или когда истечет время тайм-аута. По истечении тайм-аута, если ни один из дескрипторов не будет готов к выполнению операции, функция возвратит значение 0. (Если система не поддерживает измерение времени с точностью до микросекунд, то значение поля *tvptr->tv_usec* округляется до ближайшего поддерживаемого значения.) Как и в первом случае, ожидание может быть прервано перехваченным сигналом.

> Стандарт POSIX.1 позволяет реализациям изменять значения полей структуры timeval – таким образом, после возврата из функции select нельзя полагаться на то, что структура будет содержать значения, которые были записаны перед вызовом select. ОС FreeBSD 5.2.1, Mac OS X 10.3 и Solaris 9 оставляют эту структуру без изменений, а в ОС Linux 2.4.22 в случае возврата до истечения тайм-аута в этой структуре возвращается оставшееся время.

Второй, третий и четвертый аргументы – *readfds*, *writefds* и *exceptfds* – представляют собой указатели на *наборы дескрипторов*. Эти три набора определяют, какие дескрипторы нас интересуют и в каких состояниях (готовность к чтению, к записи или наличие исключительной ситуации). Для хранения набора дескрипторов предусмотрен тип данных fd_set. Этот тип данных выбирается реализациями таким образом, чтобы он мог хранить один бит для каждого возможного дескриптора. Можно рассматривать его как большой массив битов, как показано на рис. 14.8.

***Рис. 14.8.** Определение наборов дескрипторов для функции select*

Единственное, что можно сделать с переменными типа `fd_set`, это присвоить значение одной переменной этого типа другой переменной того же самого типа или передать переменную одной из следующих функций.

```
#include <sys/select.h>

int FD_ISSET(int fd, fd_set *fdset);
```

Возвращает ненулевое значение, если дескриптор `fd` включен в набор, 0 в противном случае

```
void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

Эти функции могут быть реализованы в виде макроопределений. Функция `FD_ZERO` сбрасывает все биты в наборе *`fdset`* в ноль. Функция `FD_SET` взводит один бит в наборе. Функция `FD_CLR` сбрасывает один бит. И наконец, с помощью функции `FD_ISSET` можно проверить состояние конкретного бита.

После объявления набора дескрипторов необходимо сбросить в нем все биты с помощью функции `FD_ZERO`. После этого нужно установить биты для интересующих нас дескрипторов – например, так:

```
fd_set rset;
int fd;

FD_ZERO(&rset);
FD_SET(fd, &rset);
FD_SET(STDIN_FILENO, &rset);
```

После возврата из функции `select` необходимо с помощью функции `FD_ISSET` проверить, какие биты в наборе остались установленными:

```
if (FD_ISSET(fd, &rset)) {
    ...
}
```

В любом (или во всех) из трех описанных аргументов (указатели на наборы дескрипторов) допускается передавать пустой указатель. Если во всех трех аргументах передать значение NULL, тогда в нашем распоряжении появится таймер с более высоким разрешением, чем предоставляемый функцией sleep. (В разделе 10.19 мы говорили, что функция sleep приостанавливает выполнение процесса на целое число секунд. С помощью функции select можно отмерять временные интервалы продолжительностью менее одной секунды – фактическая точность зависит от системных часов.) В упражнении 14.6 как раз говорится о таком применении функции.

Имя первого аргумента функции select, *maxfdp1*, происходит от выражения «maximum file descriptor plus 1» (максимальный номер дескриптора плюс 1). В качестве значения этого аргумента берется максимальный номер дескриптора, который нас интересует, увеличенный на единицу. Можно было бы просто передать в этом аргументе значение константы FD_SETSIZE, из заголовочного файла <sys/select.h>. Эта константа определяет максимально возможный номер дескриптора (часто 1024), но это значение слишком велико для большинства программ. В действительности большинство программ используют от 3 до 10 дескрипторов. (Некоторым программам требуется гораздо больше дескрипторов, но это нетипично для приложений UNIX.) Указав максимальный номер интересующего нас дескриптора, мы можем предотвратить просмотр ядром сотен неиспользуемых дескрипторов в трех наборах в поисках установленных битов.

В качестве примера на рис. 14.9 показаны два набора дескрипторов, которые были созданы следующим фрагментом программы:

```
fd_set readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);
FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);
select(4, &readset, &writeset, NULL, NULL);
```

Максимальный номер дескриптора необходимо увеличивать на единицу по той причине, что нумерация дескрипторов начинается с 0, а первый аргу-

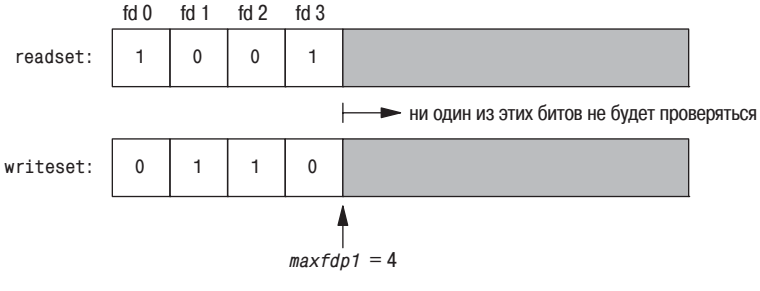


***Рис. 14.9.** Пример наборов дескрипторов для функции select*

мент функции на самом деле представляет собой счетчик количества дескрипторов, которые необходимо проверять (начиная с дескриптора 0).

Функция `select` может возвращать три различных значения:

1. Возвращаемое значение –1 свидетельствует об ошибке. Она может произойти, например, в случае перехвата сигнала, когда ни один из дескрипторов еще не готов для выполнения операции. В этой ситуации ни один из наборов дескрипторов не модифицируется.
2. Возвращаемое значение 0 свидетельствует о том, что ни один из дескрипторов не готов к выполнению операции. Это может произойти, если по истечении времени тайм-аута ни один из дескрипторов еще не готов для выполнения операции. Когда это происходит, все биты в наборах сбрасываются в ноль.
3. Положительное возвращаемое значение показывает количество дескрипторов, готовых к выполнению операции ввода-вывода. Это значение представляет собой сумму готовых дескрипторов из всех трех наборов. Таким образом, если один и тот же дескриптор готов как для чтения, так и для записи, в возвращаемом значении он будет посчитан дважды. Взведенными остаются только те биты в наборах, которые соответствуют дескрипторам, готовым к выполнению операций ввода-вывода.

Теперь необходимо уточнить смысл понятия «готов».

- Дескриптор из набора *`readfds`* считается готовым, если вызов функции `read` для этого дескриптора не будет заблокирован.
- Дескриптор из набора *`writefds`* считается готовым, если вызов функции `write` для этого дескриптора не будет заблокирован.
- Дескриптор из набора *`exceptfds`* считается готовым, если для данного дескриптора существует исключительная ситуация, ожидающая обработки. В настоящее время под исключительной ситуацией понимается либо поступление экстренных (out-of-band) данных через сетевое соединение, либо некоторые определенные события, возникающие на псевдотерминале, работающем в пакетном режиме. (Описание этих событий вы найдете в [Stevens 1990], раздел 15.10.)
- Для обычных файлов всегда возвращается признак готовности к чтению, к записи и наличия исключительной ситуации.

Очень важно понимать, что режим дескриптора (блокирующий или неблокирующий) не оказывает никакого влияния на то, будет ли заблокирован вызов функции `select`. Так, если у нас имеется дескриптор в неблокирующем режиме, который используется для чтения, и мы вызываем функцию `select` с тайм-аутом в 5 секунд, то `select` заблокирует работу процесса на 5 секунд. Аналогичным образом, если мы не ограничим время тайм-аута, то функция `select` заблокирует процесс до тех пор, пока не поступят ожидаемые данные или пока не будет перехвачен какой-либо сигнал.

Если для дескриптора будет встречен признак конца файла, функция `select` будет рассматривать этот дескриптор как готовый для чтения. После этого

вызов функции read вернет нам 0, что в UNIX расценивается как признак конца файла. (Многие неправильно полагают, что функция select рассматривает признак конца файла как исключительную ситуацию.)

Стандартом POSIX.1 определена разновидность функции select – функция pselect.

```
#include <sys/select.h>

int pselect(int maxfdp1, fd_set *restrict readfds, fd_set *restrict writefds,
            fd_set *restrict exceptfds, const struct timespec *restrict tsptr,
            const sigset_t *restrict sigmask);
```

Возвращает количество готовых дескрипторов,
0 в случае тайм-аута, –1 в случае ошибки

Функция pselect идентична функции select, со следующими исключениями:

- Значение тайм-аута в функции select задается в виде структуры timeval, в то время как в функции pselect – в виде структуры timespec. (Описание структуры timespec приводится в разделе 11.6.) Вместо секунд и микросекунд структура timespec представляет время в секундах и наносекундах. Это позволяет задавать время тайм-аута с более высокой точностью на платформах, которые предоставляют такой уровень точности измерения временных интервалов.
- Аргумент, в котором передается значение тайм-аута, объявлен со спецификатором const. Это гарантирует, что содержимое структуры не будет изменено в результате вызова функции pselect.
- Функция pselect имеет дополнительный аргумент – маску сигналов. Если в аргументе *sigmask* передается пустой указатель, функция pselect ведет себя по отношению к сигналам так же, как и функция select. В противном случае *sigmask* указывает на маску сигналов, которая будет автоматически установлена при вызове функции pselect. По возвращении из функции предыдущая маска сигналов будет восстановлена.

14.5.2. Функция poll

Функция poll напоминает функцию select, но ее программный интерфейс существенно отличается. Поскольку функция poll изначально появилась в System V, она тесно связана с механизмом STREAMS, хотя и допускает использование с любыми типами дескрипторов.

```
#include <poll.h>

int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```

Возвращает количество готовых дескрипторов,
0 в случае тайм-аута, –1 в случае ошибки

Вместо того, чтобы строить наборы дескрипторов для проверки трех возможных условий (готовность к чтению, готовность к записи, наличие исключи-

тельной ситуации), как это делается для функции select, при использовании функции poll мы должны создать массив структур pollfd, в котором каждый элемент соответствует определенному дескриптору и проверяемому условию:

```
struct pollfd {
    int fd;          /* номер дескриптора или число <0, */
                     /* если номер дескриптора игнорируется */
    short events;    /* интересующие события для заданного дескриптора */
    short revents;   /* произошедшие события для заданного дескриптора */
};
```

Количество элементов в массиве *fdarray* определяется аргументом *nfds*.

> Существовали некоторые различия в том, как объявлялся аргумент *nfds*. В SVR3 количество элементов в массиве определялось как unsigned long, что кажется излишним. В справочном руководстве к SVR4 [AT&T 1990d] второй аргумент в прототипе функции poll имел тип size_t. (Элементарные системные типы данных были приведены в табл. 2.16.) Но фактический прототип в заголовочном файле <poll.h> по-прежнему определял тип второго аргумента как unsigned long. Стандарт Single UNIX Specification определил новый тип – nfds_t, что позволяет реализациям выбирать для него соответствующий тип данных и скрывать детали реализации от приложений. Обратите внимание: этот тип должен быть достаточно большим, чтобы хранить целое число, так как возвращаемое значение представляет собой количество элементов в массиве, для которых возникли ожидаемые события.
>
> Документ SVID (определение интерфейса System V), соответствующий SVR4 [AT&T 1989], определяет первый аргумент функции poll, как struct pollfd *fdarray*[], тогда как справочное руководство SVR4 [AT&T 1990d] указывает, что этот аргумент определяется как struct pollfd **fdarray*. В языке C эти объявления эквивалентны. Однако мы будем использовать первое объявление, чтобы напомнить еще раз, что аргумент *fdarray* указывает на массив структур, а не на отдельную структуру.

Чтобы сообщить ядру об интересующих нас событиях, мы должны записать в поле events для каждого элемента массива одно или более значений, перечень которых приводится в табл. 14.6. По возвращении из функции poll ядро указывает, какие события произошли для каждого из дескрипторов. (Обратите внимание: функция poll не изменяет значение поля events. Это отличает ее от функции select, которая модифицирует значения входных аргументов, чтобы указать на готовые дескрипторы.)

Первые четыре строки проверяют готовность дескриптора для чтения, следующие три – готовность для записи и последние три – наличие исключительной ситуации. Последние три значения из табл. 14.6 устанавливаются ядром при возврате из функции. Эти три значения возвращаются в поле revents, когда возникает соответствующее событие, несмотря на то, что они не были указаны в поле events.

Когда для дескриптора обнаруживается обрыв связи (POLLHUP), мы уже не сможем ничего записать в него. Однако дескриптор еще может содержать данные, доступные для чтения.

Таблица 14.6. Значения флагов events и reevents для функции poll

Имя	events	revents	Описание
POLLIN	•	•	Данные, кроме высокоприоритетных, могут быть прочитаны, при этом операция чтения не будет заблокирована (эквивалент POLLRDNORM|POLLRDBAND).
POLLRDNORM	•	•	Обычные данные (с приоритетом 0) могут быть прочитаны, при этом операция чтения не будет заблокирована.
POLLRDBAND	•	•	Данные с ненулевым приоритетом могут быть прочитаны, при этом операция чтения не будет заблокирована.
POLLPRI	•	•	Высокоприоритетные данные могут быть прочитаны, при этом операция чтения не будет заблокирована.
POLLOUT	•	•	Обычные данные могут быть записаны, при этом операция записи не будет заблокирована.
POLLWRNORM	•	•	То же, что POLLOUT.
POLLWRBAND	•	•	Данные с ненулевым приоритетом могут быть записаны, при этом операция записи не будет заблокирована.
POLLERR		•	Возникла ошибка.
POLLHUP		•	Обрыв связи.
POLLNVAL		•	Дескриптор не соответствует открытому файлу.

Последний аргумент функции poll определяет, как долго ожидать наступления указанных событий. Как и в случае с функцией select, здесь возможны три различных значения аргумента.

timeout == **–1**

Время ожидания не ограничено. (В некоторых системах для этих целей в заголовочном файле <stropts.h>, определена константа INFTIM со значением –1.) Управление будет возвращено в вызывающую программу, если хотя бы для одного из дескрипторов произойдет ожидаемое событие или если процесс перехватит какой-либо сигнал. В последнем случае функция вернет значение –1 и код ошибки EINTR в переменной errno.

timeout == **0**

Не ждать. В этом случае просто производится проверка всех указанных дескрипторов, и управление сразу же возвращается в вызывающую программу. Это один из способов запросить информацию об изменении состояния целой группы дескрипторов, не блокируя процесс в функции poll.

timeout > **0**

Ожидать не более *timeout* миллисекунд. Управление будет возвращено в вызывающую программу, когда хотя бы один из дескрипторов будет готов или когда истечет время тайм-аута. Если время тайм-аута истечет раньше, функция возвратит значение 0. (В случае, если система не поддерживает

измерение временных интервалов с точностью до миллисекунды, значение *timeout* будет округлено до ближайшего поддерживаемого значения.)

Очень важно понимать различие между обрывом связи и получением признака конца файла. Если после ввода данных с терминала был введен символ конца файла, будет установлен флаг POLLIN, и благодаря этому мы сможем прочитать этот символ (функция read вернет значение 0). При этом флаг POLLHUP не будет выставлен в поле revents. Если во время чтения данных через модем происходит разрыв соединения, для дескриптора выставляется флаг POLLHUP.

Как и в случае с функцией select, неблокирующий режим дескриптора вовсе не определяет, будет ли блокироваться функция poll.

Прерываемость функций poll и select

Когда в 4.2BSD появилась возможность автоматического перезапуска прерванных системных вызовов (раздел 10.5), для функции select такая возможность не была предусмотрена. Это положение дел сохраняется в большинстве систем, даже если указывается флаг SA_RESTART. Но в SVR4, если флаг SA_RESTART указан, то даже функции select и poll перезапускаются автоматически. Чтобы воспрепятствовать такому поведению, которое может обернуться неприятными последствиями при переносе программного обеспечения на системы, происходящие от SVR4, мы всегда используем функцию signal_intr (листинг 10.13), если сигнал может прервать работу системного вызова select или poll.

> Ни одна из реализаций, рассматриваемых в данной книге, не предусматривает перезапуск системных вызовов select и poll при получении сигнала, даже если был установлен флаг SA_RESTART.

14.6. Асинхронный ввод-вывод

Функции select и poll, описанные в предыдущем разделе, представляют собой синхронную форму уведомления. Система ничего не сообщает нам о произошедших событиях, пока мы явно не спросим ее об этом (вызовом функции select или poll). В главе 10 мы видели, что сигналы являются асинхронной формой уведомления о происходящих событиях. Все системы, производные от BSD или System V, предоставляют возможность выполнения асинхронных операций ввода-вывода, используя сигналы (SIGPOLL – в System V и SIGIO – в BSD) для извещения процессов о том, что с дескриптором были произведены некоторые действия.

> Мы видели, что функции select и poll могут работать с любыми типами дескрипторов. Но теперь, при использовании операций асинхронного ввода-вывода, мы столкнемся с некоторыми ограничениями. В системах, производных от System V, асинхронный ввод-вывод работает только с устройствами и каналами STREAMS. В системах, производных от BSD, асинхронный ввод-вывод работает только с терминалами и сетевыми соединениями.

Одно из ограничений асинхронного ввода-вывода заключается в том, что процесс может назначить асинхронную форму ввода-вывода только для одного дескриптора. В противном случае, если мы разрешим асинхронный режим работы для нескольких дескрипторов, то при получении сигнала мы не сможем сказать, какому дескриптору соответствует полученный сигнал.

> Стандарт Single UNIX Specification включает обобщенный механизм асинхронного ввода-вывода, пришедший из предварительного стандарта на расширения реального времени. Он не связан с механизмами, которые мы описываем здесь. Этот механизм устраняет массу ограничений, существующих в обсуждаемом устаревшем механизме асинхронного ввода-вывода, но здесь мы не будем говорить о механизмах реального времени.

14.6.1. Асинхронный ввод–вывод в System V

В System V механизмы асинхронного ввода-вывода являются составной частью системы STREAMS и применимы только к устройствам и каналам STREAMS. Для задач асинхронного ввода-вывода в System V используется сигнал SIGPOLL.

Чтобы установить асинхронный режим ввода-вывода для устройства STREAMS, нужно вызвать функцию ioctl и передать ей в качестве второго аргумента (*request*) значение I_SETSIG. Третий аргумент функции в этом случае формируется из констант, перечисленных в табл. 14.7. Все эти константы определяются в заголовочном файле <stropts.h>.

Таблица 14.7. Условия, при которых генерируется сигнал SIGPOLL

Константа	Описание
S_INPUT	При получении сообщения, которое не относится к классу высокоприоритетных.
S_RDNORM	При получении обычного сообщения.
S_RDBAND	При получении сообщения с ненулевым приоритетом.
S_BANDURG	Если эта константа указывается совместно с S_RDBAND, то по прибытии сообщения с ненулевым приоритетом вместо сигнала SIGPOLL генерировать сигнал SIGURG.
S_HIPRI	При получении высокоприоритетного сообщения.
S_OUTPUT	При освобождении места в выходной очереди.
S_WRNORM	То же, что S_OUTPUT.
S_WRBAND	Можно передать сообщение с ненулевым приоритетом.
S_MSG	При получении сообщения, которое содержит в себе сигнал SIGPOLL.
S_ERROR	При получении сообщения M_ERROR.
S_HANGUP	При получении сообщения M_HANGUP.

Везде, где в табл. 14.7 встречается выражение «при получении», оно означает «при помещении в очередь чтения головы потока».

Кроме указания с помощью функции ioctl условий, при которых должен генерироваться сигнал SIGPOLL, мы также должны установить обработчик этого сигнала. В табл. 10.1 указывается, что действие по умолчанию для сигнала SIGPOLL состоит в завершении процесса, поэтому необходимо установить обработчик сигнала до вызова функции ioctl.

14.6.2. Асинхронный ввод–вывод в BSD

Асинхронный ввод-вывод в системах, производных от BSD, строится на комбинации сигналов SIGIO и SIGURG. Первый из них – это общий сигнал для всех операций асинхронного ввода-вывода, а второй используется для извещения процесса о прибытии экстренных данных через сетевое соединение.

Чтобы подготовиться к принятию сигнала SIGIO, необходимо выполнить следующие действия.

1. Установить обработчик сигнала SIGIO с помощью функции signal или sigaction.
2. Назначить командой F_SETOWN функции fcntl идентификатор процесса или идентификатор группы процессов, которым будет посылаться сигнал для дескриптора (раздел 3.14).
3. Разрешить асинхронный режим работы для дескриптора, вызвав функцию fcntl с командой F_SETFL, чтобы установить флаг состояния файла O_ASYNC (табл. 3.3).

Действие 3 может быть выполнено только в том случае, если дескриптор ссылается на терминальное устройство или сетевое соединение, что само по себе является фундаментальным ограничением механизма асинхронного ввода-вывода в BSD.

Для принятия сигнала SIGURG необходимо выполнить только действия 1 и 2. Этот сигнал генерируется только для дескрипторов, которые ссылаются на сетевые соединения, поддерживающие прием экстренных данных.

14.7. Функции readv и writev

Функции readv и writev предназначены для чтения и записи данных нескольких несмежных буферов одним обращением к функции. Эти операции называются *чтение вразброс* и *запись со слиянием*.

```
#include <sys/uio.h>

ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);

ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
```

Обе возвращают количество прочитанных или записанных байт, –1 в случае ошибки

Второй аргумент в обеих функциях – указатель на массив структур iovec:

```
struct iovec {
    void *iov_base;  /* адрес начала буфера */
    size_t iov_len;  /* размер буфера */
};
```

Количество элементов в массиве *iov* определяется аргументом *iovcnt* и ограничивается значением IOV_MAX (табл. 2.9). На рис. 14.10 показаны взаимоотношения между аргументами этих двух функций и структурой iovec.

Функция writev производит запись данных из буферов в порядке следования элементов в массиве – iov[0], iov[1] ... iov[iovcnt-1] – и возвращает общее количество записанных байт, которое обычно совпадает с суммой размеров всех буферов.

Функция readv разбрасывает данные по буферам в том же порядке, всегда до конца заполняя один буфер, прежде чем перейти к заполнению следующего. Она возвращает общее количество прочитанных байт. Если достигнут конец файла, функция readv возвращает значение 0.

> Эти две функции впервые появились в 4.2BSD и позднее были добавлены в SVR4. Они определяются стандартом Single UNIX Specification как расширения XSI.
>
> Несмотря на то, что стандарт Single UNIX Specification определяет указатель на буфер как void *, в большинстве реализаций, вопреки стандарту, эти указатели объявлены как char *.

Пример

В разделе 20.8, в функции _db_writeidx, нам потребуется записать в файл последовательно два буфера. Второй буфер передается в функцию из вызывающей программы в виде аргумента, а первый буфер создается внутри функции, он содержит длину второго буфера и смещение записи с данными от начала файла. Сделать это можно тремя способами.

1. Дважды вызвать функцию write – по разу для записи каждого буфера.
2. Разместить в динамической памяти общий буфер достаточного объема, скопировать в него оба буфера и затем одним вызовом функции write записать его в файл.
3. Записать оба буфера одним вызовом функции writev.

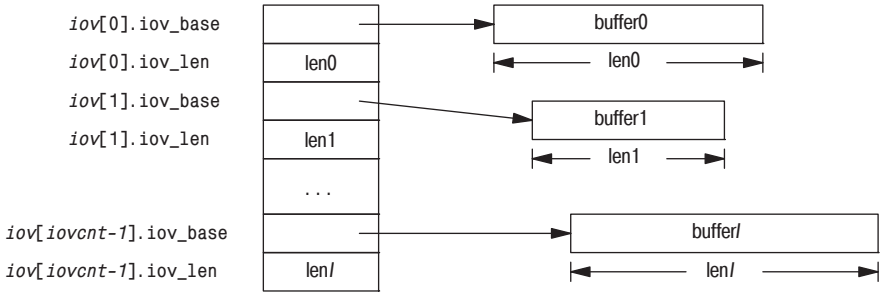


Рис. 14.10. Структура iovec для функций readv и writev

В разделе 20.8 мы используем функцию writev, но было бы любопытно сравнить все три способа.

В табл. 14.8 показаны результаты сравнения только что описанных методов.

Таблица 14.8. Результаты сравнения производительности функции writev с другими способами

Метод	Linux (Intel x86)			Mac OS X (Power PC)		
	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время
Две операции записи	1,29	3,15	7,39	1,60	17,40	19,84
Создание общего буфера и запись одним вызовом функции write	1,03	1,98	6,47	1,10	11,09	12,54
Запись функцией writev	0,70	2,72	6,41	0,86	13,58	14,72

Тестовая программа, с помощью которой проводились измерения, выводила 100-байтный заголовок и 200 байт данных. Запись выполнялась 1 048 576 раз, в результате был получен файл размером 300 мегабайт. Данная тестовая программа предусматривала запись по всем трем методикам, приведенным в табл. 14.8. Измерение времени производилось с помощью функции times (раздел 8.16), которая вызывалась до и после выполнения цикла записи. Все три значения времени (пользовательское, системное и общее время) приводятся в секундах.

Как и следовало ожидать, при двойном обращении к функции write системное время выполнения больше, чем в случае одного вызова функции write или writev. Это совпадает с результатами, приведенными в табл. 3.2.

Далее, обратите внимание, что процессорное время (сумма пользовательского и системного времени) при копировании буферов и при единственном вызове функции write меньше, чем при использовании функции writev. В случае с единственным вызовом функции write выполняется копирование буферов в промежуточный буфер в пространстве пользователя, и затем, при вызове функции write, ядро копирует его в свой внутренний буфер. В случае с использованием функции writev мы выигрываем в объеме копирования, потому что здесь необходимо только лишь скопировать данные во внутренний буфер. Однако фиксированная стоимость копирования таких небольших объемов данных сводит на нет все остальные преимущества функции writev. При увеличении объема данных, которые необходимо скопировать, вариант на основе функции writev будет выглядеть более привлекательно.

> Вас не должна смущать такая большая разница в производительности Linux и Mac OS X. Дело в том, что эти два компьютера слишком сильно отличаются друг от друга: они со-

браны на основе различных аппаратных архитектур, имеют различные объемы оперативной памяти и жесткие диски с различным быстродействием. Чтобы сравнение различных операционных систем было корректным, они должны работать на одинаковой аппаратуре.

Вывод: всегда нужно стараться делать как можно меньше системных вызовов. Если объемы данных невелики, то может оказаться так, что методика с единственным вызовом `write` окажется менее дорогостоящей по сравнению с методикой на основе функции `writev`. Иногда, однако, повышение производительности не оправдывает усложнения программы, связанного с необходимостью управления промежуточными буферами.

14.8. Функции readn и writen

Именованные и неименованные каналы и некоторые устройства, а именно терминалы, сетевые соединения и устройства STREAMS, обладают следующими двумя свойствами.

1. Функция `read` может вернуть меньшее количество байт, чем было запрошено, хотя конец файла не был достигнут. Это не является ошибкой, и мы можем продолжать чтение из устройства.
2. Функция `write` также может вернуть значение меньшее, чем мы указали. Это может произойти, например, из-за ограничений, накладываемых модулями в нисходящем потоке данных. Такое поведение также не следует расценивать как ошибочную ситуацию, а оставшиеся данные можно записать повторным обращением к `write`. (Обычно подобное случается только при использовании неблокирующего режима для дескрипторов или в результате перехвата сигнала.)

Такое никогда не происходит при работе с дисковыми файлами, за исключением случаев переполнения файловой системы или достижения предела выделенной квоты на дисковое пространство, когда система не в состоянии записать весь требуемый объем данных.

Вообще, при работе с терминалами, сетевыми соединениями или каналами всегда необходимо учитывать эти особенности. Чтобы прочитать или записать определенное количество байт, можно воспользоваться следующими двумя функциями. Они сами позаботятся об обслуживании ситуаций, когда операции чтения или записи выполняются лишь частично: они будут вызывать функции `read` или `write` столько раз, сколько потребуется для чтения или записи заданного количества байт.

```
#include "apue.h"

ssize_t readn(int filedes, void *buf, size_t nbytes);

ssize_t writen(int filedes, void *buf, size_t nbytes);
```

Обе возвращают количество прочитанных или записанных байт, −1 в случае ошибки

Мы даем описание этих двух функций, потому что они используются, например, в процедурах обработки ошибок, которые будут встречаться вам в дальнейших примерах. Функции `readn` и `writen` не являются частью какого-либо стандарта.

Функцию `writen` всегда можно использовать для типов файлов, о которых мы говорили выше, но функция `readn` должна вызываться только в том случае, когда заранее известно, что из данного файла может быть прочитано заданное количество байт. В листинге 14.11 показаны реализации функций `readn` и `writen`, которые будут использоваться в последующих примерах.

Обратите внимание, что в случае возникновения ошибки в процессе чтения или записи данных вместо признака ошибки возвращается количество переданных данных. Аналогичным образом, если в процессе чтения достигнут конец файла, функция возвращает количество байт, скопированных в буфер, предоставленный вызывающей программой, если некоторый объем данных удалось прочитать, но при этом он не равен запрошенному объему данных.

Листинг 14.11. Функции readn и writen

```
#include "apue.h"

ssize_t                     /* Прочитать n байт из дескриптора */
readn(int fd, void *ptr, size_t n)
{
    size_t nleft;
    ssize_t nread;

    nleft = n;
    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1);  /* ошибка, вернуть -1 */
            else
                break;       /* ошибка, вернуть количество прочитанных байт */
        } else if (nread == 0) {
            break;           /* конец файла */
        }
        nleft -= nread;
        ptr += nread;
    }
    return(n - nleft);       /* возвращаемое значение >= 0 */
}

ssize_t                     /* Записать n байт в дескриптор */
writen(int fd, const void *ptr, size_t n)
{
    size_t nleft;
    ssize_t nwritten;

    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) < 0) {
            if (nleft == n)
```

```
                return(-1);  /* ошибка, вернуть -1 */
            else
                break;       /* ошибка, вернуть количество записанных байт */
        } else if (nwritten == 0) {
            break;
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return(n - nleft);       /* возвращаемое значение >= 0 */
}
```

14.9. Операции ввода-вывода с отображаемой памятью

Операции ввода-вывода с отображаемой памятью (memory-mapped) позволяют отображать дисковые файлы на участок памяти таким образом, что при выборке данных из памяти производится чтение соответствующих байтов файла. Аналогично, при сохранении данных в отображенной памяти автоматически производится запись соответствующих байтов в файл. Это дает возможность производить ввод-вывод без использования функций `read` и `write`.

> Операции ввода-вывода с отображаемой памятью уже много лет используются для организации работы с виртуальной памятью. В 1981 году в 4.1BSD появился другой вариант ввода-вывода с отображаемой памятью, с использованием функций `vread` и `vwrite`. Позднее, в 4.2BSD, эти функции были удалены, их должна была заменить функция `mmap`. Однако функция `mmap` не вошла в состав 4.2BSD (по причинам, которые описаны в разделе 2.5 [McKusick et al. 1996]). Одна из реализаций функции `mmap` приводится в [Gingell, Moran, and Shannon 1987]. Функция `mmap` была включена в стандарт Single UNIX Specification, она является обязательной для реализации во всех XSI-совместимых системах и поддерживается в большинстве версий UNIX.

Чтобы воспользоваться этой возможностью, мы должны сообщить ядру о необходимости отобразить заданный файл в память. Делается это с помощью функции `mmap`.

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flag, int filedes, off_t off);
```

Возвращает адрес начала области отображаемой памяти в случае успеха, `MAP_FAILED` – в случае ошибки

В аргументе *addr* можно указать желаемый адрес начала участка отображенной памяти. Обычно в этом аргументе передается значение 0, что позволяет системе самой выбрать начальный адрес. Возвращаемое значение функции является адресом начала отображенной памяти.

Через аргумент *filedes* передается дескриптор отображаемого файла. Прежде чем отобразить файл в адресное пространство, необходимо открыть его. В ар-

гументе *len* передается количество байт, которые надо отобразить в памяти, а в аргументе *off* – смещение отображаемого участка от начала файла. (Далее будут описаны некоторые ограничения, существующие для аргумента *off*.)

Аргумент *prot* определяет степень защищенности отображенного участка.

Таблица 14.9. Защита области отображенной памяти

prot	Описание
PROT_READ	Область памяти доступна для чтения
PROT_WRITE	Область памяти доступна для записи
PROT_EXEC	Область памяти доступна для исполнения
PROT_NONE	Область памяти недоступна

Степень защищенности может быть указана либо как PROT_NONE, либо в виде объединения по ИЛИ (OR) любой комбинации из PROT_READ, PROT_WRITE и PROT_EXEC. Для области памяти не может использоваться степень защищенности, которая дает больше прав доступа, чем позволяет режим, в котором был открыт файл. Например, мы не сможем указать значение PROT_WRITE, если файл был открыт только для чтения.

Прежде чем перейти к описанию аргумента *flag*, рассмотрим рис. 14.11, на котором показан файл, отображенный в память. (Типичная раскладка памяти процесса была изображена на рис. 7.3). На данном рисунке «адрес начала» соответствует значению, возвращаемому функцией mmap. Область ото-

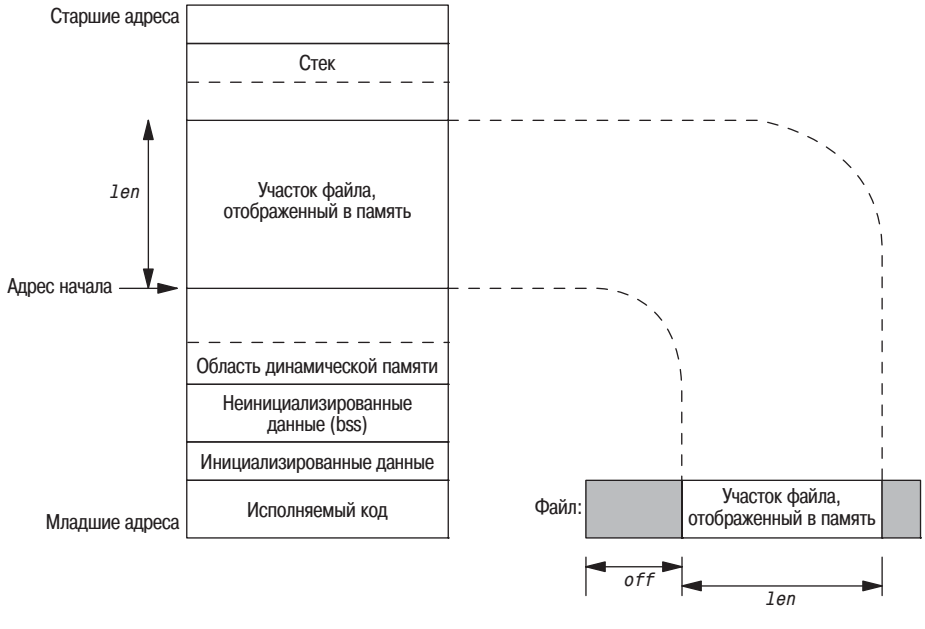


***Рис. 14.11.** Пример отображенного в память файла*

браженной памяти показана где-то между областью динамической памяти и стеком, но это зависит от конкретной реализации.

Аргумент *flag* оказывает воздействие на различные атрибуты области отображенной памяти.

MAP_FIXED Возвращаемое значение должно быть равно значению аргумента *addr*. Применение этого флага не рекомендуется, так как он снижает переносимость приложения. Если при использовании ненулевого значения в аргументе *addr* этот флаг не указывается, то ядро расценивает значение аргумента *addr* как желаемый адрес, но не дает никакой гарантии, что отображенная память будет размещена начиная с этого адреса. Максимальная переносимость достигается при указании значения 0 в аргументе *addr*.

> Поддержка флага MAP_FIXED является необязательной в POSIX-совместимых системах, но обязательна в XSI-совместимых системах.

MAP_SHARED Флаг определяет характер операций над областью отображенной памяти. При указании этого флага все операции записи в область отображенной памяти будут приводить к модификации самого файла, т. е. операции записи в память будут эквивалентны вызову функции write для файла. Допускается одновременная установка только одного флага – либо этого, либо следующего.

MAP_PRIVATE Этот флаг говорит о том, что все операции записи в область отображенной памяти будут приводить к созданию скрытой копии отображенного в память файла. Изменения в памяти не будут влиять на содержимое самого файла. (Флаг используется отладчиками, чтобы отобразить сегмент кода из файла программы в память и позволить пользователю модифицировать инструкции. Все модификации будут производиться только в памяти процесса и не будут затрагивать оригинальный файл программы.)

Каждая реализация поддерживает дополнительные, специфичные для нее значения MAP_xxx. За дополнительной информацией обращайтесь к странице справочного руководства по функции mmap(2).

Значения аргументов *off* и *addr* (если указан флаг MAP_FIXED) должны быть кратны размеру страницы виртуальной памяти. Это значение может быть получено с помощью функции sysconf (раздел 2.5.4) с аргументом _SC_PAGESIZE или _SC_PAGE_SIZE. Поскольку чаще всего в аргументах *off* и *addr* передается значение 0, это требование не представляет большой проблемы.

Поскольку смещение начала отображаемого участка файла привязано к размеру страницы виртуальной памяти, что случится, если длина отображаемого участка не кратна размеру страницы? Представим себе, что размер файла составляет 12 байт, а размер страницы виртуальной памяти – 512 байт. В этом случае система обычно выделяет область отображенной памяти размером 512 байт, но в последние 500 байт этой области будут записаны нули.

Мы можем вносить изменения в последние 500 байт, но эти изменения не будут отражаться на содержимом файла. Таким образом, с помощью функции mmap невозможно добавить новые данные в конец файла. Для этого необходимо сначала увеличить размер файла, как показано в листинге 14.12.

Для работы с отображенными областями памяти обычно используются два сигнала. Сигнал SIGSEGV, как правило, указывает на попытку обращения к недоступной области памяти. Этот сигнал также может быть сгенерирован при попытке записи в память, которая была определена как доступная только для чтения. Сигнал SIGBUS может быть сгенерирован при попытке обращения к части отображенной области, которая не имеет смысла к моменту обращения. Например, предположим, что мы отобразили в память весь файл целиком, но прежде чем мы смогли приступить к операциям с отображенной областью памяти, файл был усечен некоторым другим процессом. Тогда, если мы попытаемся обратиться к той части файла, которая была усечена, мы получим сигнал SIGBUS.

Области отображенной памяти наследуются дочерними процессами через функцию fork (поскольку отображенная память является частью адресного пространства родительского процесса), но по той же самой причине отображенная память не наследуется новыми программами через функцию exec.

Изменить права доступа к отображенной памяти можно с помощью функции mprotect.

```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

В аргументе *prot* могут передаваться те же самые значения, что и в аргументе *prot* функции mmap (табл. 14.9). Аргумент *addr* должен быть целым числом, кратным размеру страницы виртуальной памяти.

> Функция mprotect была включена в стандарт Single UNIX Specification как часть необязательной функциональности защиты памяти, но все XSI-совместимые системы обязаны поддерживать ее.

Если страницы в разделяемой области отображенной памяти были изменены, можно сбросить их в файл с помощью функции msync. Функция msync напоминает функцию fsync (раздел 3.13), но предназначена для работы с областями отображенной памяти.

```
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Если область отображенной памяти была создана с флагом MAP_PRIVATE, то содержимое отображаемого файла не изменяется. Как и в других функциях обслуживания отображенной памяти, аргумент *addr* должен содержать адрес, кратный размеру страницы виртуальной памяти.

Аргумент *flags* позволяет до некоторой степени управлять порядком сбрасывания памяти в файл. Чтобы просто запланировать запись данных, мы можем передать в этом аргументе значение MS_ASYNC. Если нам необходимо дождаться, пока данные будут записаны полностью, мы должны указать флаг MS_SYNC. Аргумент должен содержать одно из двух значений – либо MS_ASYNC, либо MS_SYNC.

Необязательный флаг MS_INVALIDATE предписывает аннулировать все изменения, произведенные в памяти, и синхронизировать ее содержимое в соответствии с содержимым отображаемого объекта (файла). Некоторые реализации аннулируют все измененные страницы в указанном диапазоне, но это совершенно необязательно.

Область отображенной памяти автоматически удаляется по завершении процесса или в результате вызова функции munmap. Закрытие файлового дескриптора не приводит к удалению этой области.

```
#include <sys/mman.h>

int munmap(caddr_t addr, size_t len);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Функция munmap не оказывает влияния на отображаемый объект, то есть вызов функции munmap не приводит к записи области отображенной памяти в файл. Обновление файла на диске при внесении изменений в область отображенной памяти, созданной с флагом MAP_SHARED, производится ядром автоматически. Все изменения, внесенные в область отображенной памяти, созданной с флагом MAP_PRIVATE, после вызова функции munmap будут утеряны.

Пример

Программа, представленная листингом 14.12, копирует файл (подобно команде cp(1)), используя для этого операции ввода-вывода с отображаемой памятью.

Листинг 14.12. Копирование файла с использованием операций ввода-вывода с отображаемой памятью

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

int
main(int argc, char *argv[])
{
    int fdin, fdout;
```

```
    void *src, *dst;
    struct stat statbuf;

    if (argc != 3)
        err_quit("Использование: %s <fromfile> <tofile>", argv[0]);

    if ((fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("невозможно открыть %s для чтения", argv[1]);

    if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) < 0)
        err_sys("невозможно создать %s для записи", argv[2]);

    if (fstat(fdin, &statbuf) < 0)    /* определить размер входного файла */
        err_sys("fstat error");

    /* установить размер выходного файла */
    if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
        err_sys("ошибка вызова функции lseek");
    if (write(fdout, "", 1) != 1)
        err_sys("ошибка вызова функции write");

    if ((src = mmap(0, statbuf.st_size, PROT_READ, MAP_SHARED,
      fdin, 0)) == MAP_FAILED)
        err_sys("ошибка вызова функции mmap для входного файла");

    if ((dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
      MAP_SHARED, fdout, 0)) == MAP_FAILED)
        err_sys("ошибка вызова функции mmap для выходного файла");

    memcpy(dst, src, statbuf.st_size); /* сделать копию файла */
    exit(0);
}
```

Прежде всего мы открываем оба файла и затем с помощью функции `fstat` получаем размер исходного файла. Этот размер необходим для вызова функции `mmap`, а также для того, чтобы установить размер выходного файла. Затем вызывается функция `lseek` и производится запись одного байта, в результате чего выходной файл обретает требуемый размер. Если мы не установим размер выходного файла таким способом, то вызов функции `mmap` завершится успехом, но при первой же попытке обратиться к отображенной памяти мы получим сигнал `SIGBUS`. Можно установить размер выходного файла с помощью функции `ftruncate`, но не все системы поддерживают возможность увеличения файла с помощью этой функции (раздел 4.13).

> Увеличение размера файла с помощью функции `ftruncate` возможно на всех четырех обсуждаемых в этой книге платформах.

Затем дважды вызывается функция `mmap` для отображения обоих файлов в память и, наконец, производится копирование содержимого входного буфера в выходной буфер с помощью функции `memcpy`. В момент выборки данных из входного буфера (`src`) ядро автоматически производит чтение данных из исходного файла. При сохранении данных в выходной буфер (`dst`) данные автоматически записываются в выходной файл.

Точный момент времени, когда данные записываются в файл, зависит от алгоритма обслуживания страниц виртуальной памяти. В некоторых системах запись измененных страниц производится отдельным демоном через продолжительные промежутки времени. Если вам необходимо, чтобы данные сразу же были записаны на диск, вызывайте перед выходом из программы функцию `msync` с флагом `MS_SYNC`.

А теперь сравним производительность копирования файла через отображение в память и копирования с помощью функций `read` и `write` (с размером буфера 8192 байта). В табл. 14.10 приводятся результаты. Размер копируемого файла составлял 300 мегабайт, результаты даны в секундах.

Таблица 14.10. Результаты измерения производительности копирования файла

Метод	Linux (Intel x86)			Solaris 9 (SPARC)		
	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время
`read/write`	0,04	1,02	39,76	0,18	9,70	41,66
`mmap/memcpy`	0,64	1,31	24,26	1,68	7,94	28,53

В Solaris 9 процессорное время (сумма пользовательского и системного времени) практически одинаково для обоих вариантов копирования: 9,88 и 9,62 секунды. В Linux 2.4.22 общее процессорное время при использовании связки `mmap/memcpy` увеличилось вдвое (1,06 и 1,95 секунды). Эта разница, скорее всего, обусловлена различной реализацией учета времени работы процесса в этих операционных системах.

Если принимать в расчет только общее время работы процесса, то вариант с использованием `mmap/memcpy` выглядит более предпочтительно, чем версия на основе `read/write`. В этом есть определенный смысл, потому что при использовании функций `mmap/memcpy` выполняется меньший объем работы. При использовании функций `read/write` данные копируются из буфера ядра в буфер приложения (`read`) и затем обратно в буфер ядра (`write`). При использовании функций `mmap/memcpy` данные копируются напрямую из одного буфера ядра, отображенного в адресное пространство процесса, в другой буфер ядра, также отображенный в адресное пространство процесса.

Операции ввода-вывода с отображаемой памятью производятся быстрее при копировании одного обычного файла в другой. Такой способ копирования невозможен для некоторых типов устройств (таких как сетевые или терминальные устройства) и, кроме того, мы должны проявлять осторожность, если размер файла после отображения может быть изменен. Однако некоторые приложения могут извлечь определенные выгоды из операций ввода-вывода с отображаемой памятью, так как их использование зачастую упрощает алгоритмы, поскольку вместо использования функций `read` и `write` мы манипулируем объектом в памяти. Один из примеров, когда подобные операции ввода-вывода могут быть полезны, – работа с буфером изображения, который связан с растровым дисплеем.

В [Kreiger, Stumm, and Unrau 1992] описывается альтернатива стандартной библиотеке ввода-вывода (глава 5), построенная на операциях ввода-вывода с отображаемой памятью.

Мы еще вернемся к вводу-выводу с отображаемой памятью в разделе 15.9 при рассмотрении примера, в котором показано, как можно использовать разделяемую память для взаимодействия процессов.

14.10. Подведение итогов

В этой главе мы описали многочисленные функции расширенных операций ввода-вывода, большинство из которых будут использоваться в примерах к следующим главам:

- Неблокирующий ввод-вывод – операции ввода-вывода, которые не могут заблокировать процесс.
- Блокировки записей (более подробно мы будем их рассматривать на примере реализации библиотеки для работы с базой данных в главе 20).
- Механизм System V STREAMS (понимание которого потребуется нам в главе 17, чтобы разобраться в каналах на основе STREAMS, проблемах передачи файловых дескрипторов и реализации соединений типа клиент-сервер в System V).
- Мультиплексирование ввода-вывода – функции `select` и `poll` (мы часто будем использовать их в последующих примерах).
- Функции `readv` и `writev` (которые также будут использоваться в последующих примерах).
- Операции ввода-вывода с отображаемой памятью (`mmap`).

Упражнения

14.1. Напишите тестовую программу, которая продемонстрирует поведение вашей системы в ситуации, когда процесс пытается получить блокировку для записи на участок файла, на который уже установлена блокировка для чтения, и при этом продолжают поступать запросы на установку блокировки для чтения. Будет ли подвешен процесс, запросивший блокировку для записи, процессами, которые устанавливают блокировки для чтения?

14.2. Просмотрите заголовочные файлы вашей системы и исследуйте реализацию функции `select` и четырех макросов `FD_`.

14.3. В системных заголовочных файлах обычно определено ограничение на количество дескрипторов, которое может храниться в типе `fd_set`. Предположим, что нам необходимо увеличить этот предел до 2048 дескрипторов. Как это можно сделать?

14.4. Сравните функции, предназначенные для работы с наборами сигналов (раздел 10.11), с функциями для работы с наборами дескрипторов `fd_set`. А также сравните реализацию тех и других в вашей системе.

14.5. Сколько различных типов информации возвращает функция `getmsg`?

14.6. Реализуйте функцию `sleep_us`, которая похожа на функцию `sleep`, но приостанавливает работу процесса на заданное количество микросекунд. Используйте для выполнения задержки функцию `select` или `poll`. Сравните эту функцию с функцией `usleep` систем BSD.

14.7. Возможно ли реализовать функции `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT` и `WAIT_CHILD` из листинга 10.17, используя рекомендательные блокировки записей вместо сигналов? Если да, то напишите программу и проверьте ее.

14.8. Определите емкость неименованного канала, используя для этого неблокирующую операцию записи. Сравните полученное значение с константой `PIPE_BUF` из главы 2.

14.9. Вспомните табл. 14.8. Определите для своей системы объем данных, при котором функция `writev` будет работать быстрее, чем `write`.

14.10. Запустите программу из листинга 14.12, скопируйте файл и посмотрите, изменилось ли время последнего обращения к исходному файлу.

14.11. В программе из листинга 14.12 попробуйте закрыть дескриптор исходного файла сразу же после вызова функции `mmap`, чтобы убедиться, что закрытие дескриптора не оказывает влияния на операции ввода-вывода с отображаемой памятью.

15

Межпроцессное взаимодействие

15.1. Введение

В главе 8 мы описали примитивы управления процессами и увидели, как можно создать несколько процессов. Но единственный способ обмена информацией между этими процессами заключался в передаче открытых файловых дескрипторов через функции `fork` или `exec` или через файловую систему. Теперь мы рассмотрим другие способы взаимодействия процессов друг с другом – механизмы IPC (Interprocess Communication), или механизмы межпроцессного взаимодействия.

В прошлом механизмы IPC в UNIX представляли собой смесь самых разных концепций, лишь немногие из которых были переносимы между различными реализациями. Благодаря усилиям по стандартизации, предпринятым POSIX и The Open Group (ранее X/Open), ситуация значительно улучшилась, но некоторые различия все еще существуют. В табл. 15.1 приводится список различных форм IPC, которые поддерживаются всеми четырьмя платформами, обсуждаемыми в этой книге.

Обратите внимание: стандарт Single UNIX Specification (колонка SUS) разрешает реализациям поддерживать дуплексные неименованные каналы, но лишь поддержка полудуплексных неименованных каналов является обязательной. В реализациях, которые поддерживают дуплексные каналы, попрежнему корректно работают приложения, написанные для реализаций, поддерживающих только полудуплексные каналы. В табл. 15.1 мы использовали обозначение «дуплекс», чтобы выделить реализации, которые поддерживают полудуплексные каналы через использование дуплексных каналов.

В табл. 15.1 поддержка базовых функциональных возможностей обозначена точкой (•). Для случая с дуплексными каналами, если эта функциональность может предоставляться через сокеты домена UNIX (раздел 17.3), в соответствующих ячейках таблицы указана аббревиатура UDS (UNIX domain socket). Некоторое реализации поддерживают как базовую функциональ-

ность, так и сокеты домена UNIX, поэтому в этих ячейках указаны точка, и аббревиатура UDS вместе.

Как уже упоминалось в разделе 14.4, поддержка механизма STREAMS определяется стандартом Single UNIX Specification как необязательная. Именованные дуплексные каналы, которые реализуются на базе неименованных каналов STREAMS, также определяются стандартом Single UNIX Specification как необязательные для реализации. В ОС Linux поддержка STREAMS доступна в виде отдельного пакета LiS (от «Linux STREAMS»), который не устанавливается по умолчанию. Если та или иная функциональность реализована на базе дополнительных пакетов, которые не устанавливаются по умолчанию, в соответствующей ячейке мы указываем сокращение «доп.».

Таблица 15.1. Перечень механизмов IPC, доступных в UNIX

Тип IPC	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Полудуплексные неименованные каналы	•	дуплекс	•	•	дуплекс
Именованные каналы	•	•	•	•	•
Дуплексные неименованные каналы	Допускается	•, UDS	доп., UDS	UDS	•, UDS
Именованные дуплексные каналы	XSI, доп.	UDS	доп., UDS	UDS	•, UDS
Очереди сообщений	XSI	•	•		•
Семафоры	XSI	•	•	•	•
Разделяемая память	XSI	•	•	•	•
Сокеты	•	•	•	•	•
STREAMS	XSI, доп.		доп.		•

Первые семь видов IPC из табл. 15.1 обычно предназначены для организации взаимодействий между процессами, работающими на одной и той же машине. Последние два – сокеты и STREAMS – единственные формы IPC, которые повсеместно используются для организации взаимодействий между процессами, работающими на разных машинах, объединенных в сеть.

Мы разделили обсуждение механизмов межпроцессного взаимодействия на три главы. В этой главе мы будем рассматривать классические формы IPC: именованные и неименованные каналы, очереди сообщений, семафоры и разделяемую память. В следующей главе мы обсудим механизмы организации взаимодействий через сеть с помощью сокетов. И в главе 17 расскажем о некоторых расширенных возможностях IPC.

15.2. Неименованные каналы

Неименованные каналы (pipes, далее для краткости просто каналы) – это старейшая форма организации взаимодействий между процессами, предоставляемая операционными системами UNIX. Каналы имеют два ограничения:

1. Исторически они являются полудуплексными (то есть данные могут передаваться по ним только в одном направлении). Некоторые современные системы предоставляют дуплексные каналы, но для сохранения переносимости приложений никогда не следует пользоваться этой возможностью.
2. Каналы могут использоваться только для организации взаимодействия между процессами, которые имеют общего предка. Обычно канал создается родительским процессом, который затем вызывает функцию `fork`, после чего этот канал может использоваться для общения между родительским и дочерним процессами.

Далее (в разделе 15.5) мы увидим, что именованные каналы не имеют второго ограничения, а сокеты домена UNIX (unix domain sockets, раздел 17.3) и именованные каналы на базе STREAMS (раздел 17.2.2) – обоих ограничений.

Несмотря на указанные ограничения, полудуплексные каналы по-прежнему являются одной из наиболее широко используемых форм IPC. Каждый раз, когда вы вводите в командной строке последовательность команд, объединенных в конвейер, оболочка создает отдельный процесс для каждой команды и связывает с помощью канала стандартный вывод предыдущей команды со стандартным вводом следующей команды.

Неименованный канал создается с помощью функции `pipe`.

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Через аргумент *filedes* возвращаются два файловых дескриптора: *filedes[0]* открыт для чтения, а *filedes[1]* – для записи. Данные, выводимые в *filedes[1]*, становятся входными данными для *filedes[0]*.

> В ОС 4.3BSD, 4.4BSD и Mac OS X 10.3 каналы реализованы с использованием сокетов домена UNIX. Даже несмотря на то, что сокеты по своей природе являются дуплексными, эти операционные системы ограничивают сокеты, используемые для организации каналов, таким образом, что они могут передавать информацию только в одном направлении.
>
> Стандарт POSIX.1 разрешает реализациям поддерживать дуплексные каналы. В таких реализациях дескрипторы *filedes[0]* и *filedes[1]* открываются как для чтения, так и для записи.

На рис. 15.1 изображены два примера использования полудуплексных каналов. Слева показан случай, когда канал обоими концами связан с одним и тем

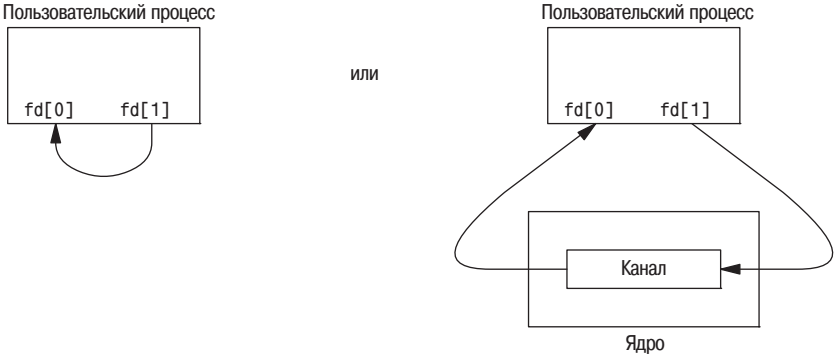


***Рис. 15.1.** Два примера использования полудуплексных каналов*

же процессом. Справа демонстрируется случай обмена данными между двумя процессами через ядро.

Функция `fstat` (раздел 4.2) для дескриптора любого конца канала возвращает тип файла FIFO. Убедиться в том, что дескриптор соответствует каналу, можно с помощью макроса `S_ISFIFO`.

> Стандарт POSIX.1 утверждает, что значение поля `st_size` структуры `stat` не определено для каналов. Но в большинстве систем вызов функции `fstat` для дескриптора, открытого на чтение, возвращает в поле `st_size` количество байт в канале, доступное для чтения. Однако это поведение не должно использоваться при разработке переносимых приложений.

Канал, который обоими концами связан с одним и тем же процессом, достаточно бесполезен. Обычно процесс, вызывающий функцию `pipe`, затем обращается к функции `fork`, создавая, таким образом, канал для передачи дан-

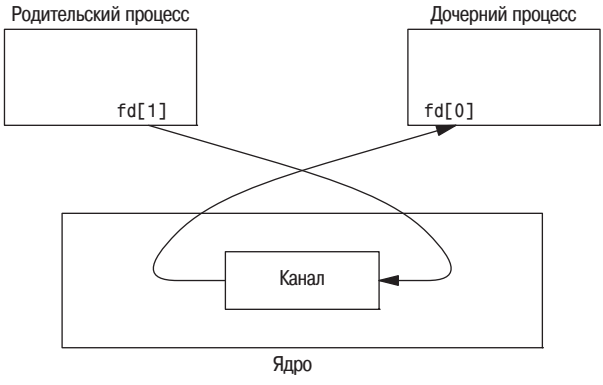


***Рис. 15.3.** Канал от родительского процесса к дочернему*

ных от родительского процесса к дочернему или наоборот. Этот сценарий показан на рис. 15.2.

Порядок действий, следующих за вызовом функции `fork`, зависит от того, в каком направлении мы желаем передавать данные. Если данные должны двигаться в направлении от родительского процесса к дочернему, тогда родитель закрывает дескриптор, открытый на чтение (`fd[0]`), а потомок закрывает дескриптор, открытый на запись (`fd[1]`). На рис. 15.3 показано окончательное состояние дескрипторов.

Чтобы организовать передачу в обратном направлении, родительский процесс закрывает `fd[1]`, а дочерний процесс – `fd[0]`.

Когда один из концов канала закрывается, в силу вступают следующие два правила.

1. Если попытаться прочитать данные из канала, в котором был закрыт дескриптор, открытый для записи, функция `read` вернет значение 0, чтобы сообщить о достижении конца файла после того, как все данные будут

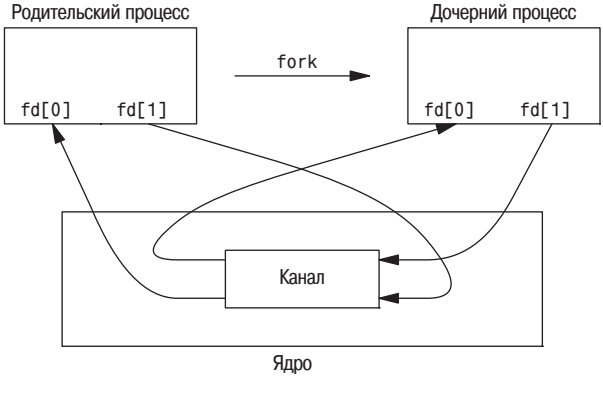


***Рис. 15.2.** Полудуплексные каналы после вызова функции fork*

прочитаны. (Технически, признак конца файла не будет сгенерирован до тех пор, пока не будут закрыты все дескрипторы, открытые для записи в канал. Такое возможно при создании дубликатов дескрипторов, благодаря чему сразу несколько процессов могут производить запись в канал. Однако обычно у канала имеется один дескриптор, открытый для чтения, и один дескриптор, открытый для записи. Когда в следующем разделе мы перейдем к изучению именованных каналов, то увидим, что зачастую в один именованный канал могут писать сразу несколько процессов.)

2. Если попытаться выполнить запись в канал, в котором был закрыт дескриптор, открытый для чтения, будет сгенерирован сигнал SIGPIPE. Если приложение игнорирует этот сигнал или перехватывает его и возвращает управление из обработчика сигнала нормальным образом, то функция write вернет значение –1 и код ошибки EPIPE в переменной errno.

При записи данных в канал размер буфера канала в ядре определяется константой PIPE_BUF. Если в канал записывается количество байт, не превышающее значения PIPE_BUF, то эти данные не будут перемежаться данными, записываемыми в канал (или FIFO) другими процессами. Если же мы попытаемся записать одним вызовом функции write большее, чем PIPE_BUF, количество байт, то записанные данные могут быть перемешаны с данными, поступившими от других процессов. Определить значение PIPE_BUF можно с помощью функции pathconf или fpathconf (раздел 2.11).

Пример

В листинге 15.1 представлена программа, которая создает канал между родительским и дочерним процессами и передает данные по этому каналу.

Листинг 15.1. Передача данных от родительского процесса к дочернему через канал

```
#include "apue.h"

int
main(void)
{
    int n;
    int fd[2];
    pid_t pid;
    char line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("ошибка вызова функции pipe");
    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) {              /* родительский процесс */
        close(fd[0]);
        write(fd[1], "привет, МИР\n", 12);
    } else {                           /* дочерний процесс */
```

```
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

В этом примере мы работали с дескрипторами канала напрямую, используя функции `write` и `read`. Но гораздо интереснее было бы продублировать тот или иной дескриптор на стандартный ввод или стандартный вывод. После этого дочерний процесс мог бы запустить некоторую программу, которая получает данные со стандартного ввода (из созданного нами канала) или производит запись на стандартный вывод (в канал).

Пример

Рассмотрим программу, которая должна выводить данные постранично. Вместо того, чтобы заново придумывать алгоритм постраничного вывода, который уже реализован некоторыми утилитами UNIX, мы попробуем воспользоваться программой постраничного просмотра, которую предпочитает пользователь. Чтобы избежать использования временного файла для хранения результатов и вызова функции `system` для отображения содержимого этого файла, мы воспользуемся каналом, по которому будем сразу же отправлять данные программе постраничного просмотра. Для этого мы сначала создадим канал, с помощью функции `fork` запустим дочерний процесс, переустановим дескриптор канала, открытый для чтения, на стандартный ввод и затем с помощью функции `exec` запустим программу постраничного просмотра. Листинг 15.2 показывает, как это можно сделать. (В этом примере программа принимает аргумент командной строки, определяющий имя файла, содержимое которого должно быть выведено. Но часто бывает, что данные, которые нужно вывести на терминал, уже находятся в памяти.)

Листинг 15.2. Передача файла программе постраничного просмотра

```
#include "apue.h"
#include <sys/wait.h>

#define DEF_PAGER "/bin/more" /* программа постраничного просмотра по умолчанию */

int
main(int argc, char *argv[])
{
    int n;
    int fd[2];
    pid_t pid;
    char *pager, *argv0;
    char line[MAXLINE];
    FILE *fp;

    if (argc != 2)
        err_quit("Использование: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
```

```
            err_sys("невозможно открыть %s", argv[1]);
        if (pipe(fd) < 0)
            err_sys("ошибка вызова функции pipe");

        if ((pid = fork()) < 0) {
            err_sys("ошибка вызова функции fork");
        } else if (pid > 0) {               /* родительский процесс */
            close(fd[0]);                   /* закрыть дескриптор для чтения */

            /* родительский процесс копирует argv[1] в канал */
            while (fgets(line, MAXLINE, fp) != NULL) {
                n = strlen(line);
                if (write(fd[1], line, n) != n)
                    err_sys("ошибка записи в канал");
            }
            if (ferror(fp))
                err_sys("ошибка вызова функции fgets");
            close(fd[1]);                   /* закрыть дескриптор для записи */
            if (waitpid(pid, NULL, 0) < 0)
                err_sys("ошибка вызова функции waitpid");
            exit(0);
        } else {                            /* дочерний процесс */
            close(fd[1]);                   /* закрыть дескриптор для записи */
            if (fd[0] != STDIN_FILENO) {
                if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                    err_sys("ошибка переназначения канала на stdin");
                close(fd[0]);               /* уже не нужен после вызова dup2 */
            }
            /* определить аргументы для execl() */
            if ((pager = getenv("PAGER")) == NULL)
                pager = DEF_PAGER;
            if ((argv0 = strrchr(pager, '/')) != NULL)
                argv0++;                    /* перейти за последний слэш */
            else
                argv0 = pager;              /* в имени программы нет слэша */

            if (execl(pager, argv0, (char *)0) < 0)
                err_sys("ошибка запуска программы %s", pager);
        }
        exit(0);
    }
```

Перед вызовом функции `fork` создается канал. После вызова функции `fork` родительский процесс закрывает дескриптор канала, открытый для чтения, а дочерний процесс – дескриптор, открытый для записи. После этого дочерний процесс вызывает функцию `dup2`, с помощью которой переназначает конец канала, открытый для чтения, на стандартный ввод.

Когда мы дублируем один дескриптор в другой (`fd[0]` – на стандартный ввод в дочернем процессе), необходимо убедиться в том, что номер дескриптора не совпадает с тем, который нам нужен. Если бы это был дескриптор с нужным нам номером, то в результате вызова функций `dup2` и `close` единственная

копия дескриптора была бы закрыта. (Поведение функции dup2, когда оба ее аргумента равны, обсуждалось в разделе 3.12). В этой программе, если бы стандартный ввод не был открыт командной оболочкой, то функция fopen, вызываемая в самом начале, все равно открыла бы для файла дескриптор с номером 0, как наименьшим неиспользуемым номером дескриптора, – таким образом, fd[0] никогда не должен совпадать с дескриптором стандартного ввода. Однако всякий раз, когда мы обращаемся к функциям dup2 и close, дублируя один дескриптор в другой, в качестве меры предосторожности мы будем сначала сравнивать эти дескрипторы.

Обратите внимание, как мы использовали переменную окружения PAGER, чтобы получить имя программы постраничного вывода, предпочитаемой пользователем. Если таковая не определена, мы запускаем программу по умолчанию. Это наиболее распространенное правило использования переменных окружения.

Пример

Давайте вспомним функции TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT и WAIT_CHILD из раздела 8.9. В листинге 10.17 была показана реализация этих функций на основе сигналов. В листинге 15.3 приводится реализация этих же функций, но уже на основе каналов.

Листинг 15.3. Процедуры синхронизации родительского и дочернего процессов

```
#include "apue.h"

static int pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("ошибка вызова функции pipe");
}

void
TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("ошибка вызова функции write");
}

void
WAIT_PARENT(void)
{
    char c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("ошибка вызова функции read");
    if (c != 'p')
        err_quit("WAIT_PARENT: получены некорректные данные");
}
```

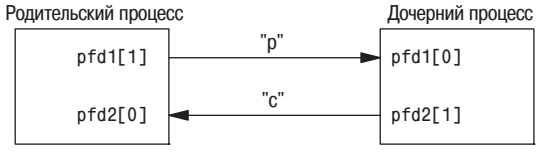


***Рис. 15.4.** Использование каналов для синхронизации родительского и дочернего процессов*

```
void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("ошибка вызова функции write");
}

void
WAIT_CHILD(void)
{
    char c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("ошибка вызова функции read");
    if (c != 'c')
        err_quit("WAIT_CHILD: получены некорректные данные");
}
```

Перед вызовом `fork` создаются два канала, как показано на рис. 15.4. Родительский процесс записывает в канал с помощью функции `TELL_CHILD` символ «`p`», а дочерний процесс с помощью функции `TELL_PARENT` записывает символ «`c`». Функции `WAIT_xxx` блокируются в системном вызове `read` до получения одиночного символа.

Обратите внимание, что в этой реализации каждый из каналов имеет два открытых для чтения дескриптора. Кроме того, что дочерний процесс может читать из дескриптора `pfd1[0]`, родительский процесс также имеет дескриптор, открытый для чтения. Но в данном случае это не имеет никакого значения, потому что родительский процесс не пытается читать из этого канала.

15.3. Функции popen и pclose

Поскольку чаще всего канал создается для взаимодействия с другим процессом, чтобы получать от него или отправлять ему данные, стандартная библиотека ввода-вывода традиционно поддерживает функции `pclose` и `popen`. Эти две функции берут на себя всю рутинную работу, которую мы до сих пор выполняли самостоятельно: создание канала, создание дочернего процесса,

закрытие неиспользуемых дескрипторов канала, запуск команды и ожидание завершения команды.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);

                                    Возвращает указатель на структуру FILE
                                    в случае успеха, NULL в случае ошибки

int pclose(FILE *fp);

                    Возвращает код завершения cmdstring, –1 в случае ошибки
```

Функция `popen` посредством функций `fork` и `exec` запускает на исполнение команду *`cmdstring`* и возвращает указатель на объект `FILE`. Если в аргументе *`type`* передается значение "`r`", указатель на файл будет связан со стандартным выводом *`cmdstring`* (рис. 15.5).

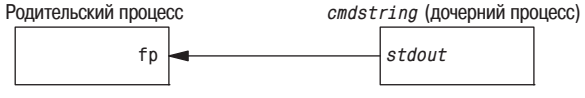


***Рис. 15.5.** Результат выполнения инструкции* `fp = popen(cmdstring, "r")`

Если в аргументе *`type`* передается значение "`w`", указатель на файл будет связан со стандартным вводом *`cmdstring`* (рис. 15.6).

Чтобы запомнить правила назначения второго аргумента функции `popen`, вспоминайте функцию `fopen`, которая возвращает объект `FILE`, открытый для чтения, если аргумент *`type`* имеет значение "`r`", и для записи, если аргумент *`type`* имеет значение "`w`".

Функция `pclose` закрывает поток ввода-вывода, ожидает завершения команды и возвращает код завершения командного интерпретатора, запущенного для выполнения команды *`cmdstring`*. (Код завершения рассматривался в разделе 8.6. Функция `system`, описанная в разделе 8.13, также возвращает код завершения.) Если командный интерпретатор не смог запуститься, функция `pclose` возвращает код завершения, как если бы командная оболочка вызвала функцию `exit(127)`.

Команда *`cmdstring`* запускается интерпретатором Bourne shell как

```
sh -c cmdstring
```

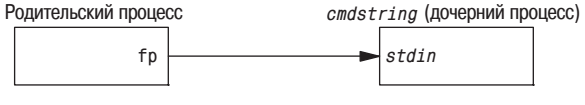


***Рис. 15.6.** Результат выполнения инструкции* `fp = popen(cmdstring, "w")`

Это означает, что командная оболочка производит интерпретацию всех специальных символов, которые встретятся в строке *cmdstring*. Это позволяет, например, выполнить команду

```
fp = popen("ls *.c", "r");
```

или

```
fp = popen("cmd 2>&1", "r");
```

Пример

Теперь попробуем переписать программу из листинга 15.2 таким образом, чтобы она использовала функцию popen. Текст новой программы приводится в листинге 15.4.

Листинг 15.4. Передача файла программе постраничного просмотра с использованием функции popen

```
#include "apue.h"
#include <sys/wait.h>

#define PAGER "${PAGER:-more}" /* либо значение переменной окружения, */
                               /* либо значение по умолчанию */
int
main(int argc, char *argv[])
{
    char line[MAXLINE];
    FILE *fpin, *fpout;

    if (argc != 2)
        err_quit("Использование: a.out <полное_имя_файла>");
    if ((fpin = fopen(argv[1], "r")) == NULL)
        err_sys("невозможно открыть %s", argv[1]);

    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("ошибка вызова функции popen");

    /* передать argv[1] программе постраничного просмотра */
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err_sys("ошибка записи в канал");
    }
    if (ferror(fpin))
        err_sys("ошибка вызова функции fgets");
    if (pclose(fpout) == -1)
        err_sys("ошибка вызова функции pclose");
    exit(0);
}
```

Использование функции popen позволило значительно уменьшить размер программы.

Команда `${PAGER:-more}` говорит о том, что следует использовать значение переменной окружения `PAGER`, если она определена и содержит непустую строку, в противном случае использовать строку `more`.

Пример – функции popen и pclose

В листинге 15.5 приводится наша реализация функций `popen` и `pclose`.

Листинг 15.5. Функции popen и pclose

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

/*
 * Указатель на массив, размещаемый во время выполнения.
 */
static pid_t *childpid = NULL;

/*
 * Будет получено из нашей функции open_max(), листинг 2.4.
 */
static int maxfd;

FILE *
popen(const char *cmdstring, const char *type)
{
    int i;
    int pfd[2];
    pid_t pid;
    FILE *fp;

    /* допустимы только "r" или "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;         /* требование стандарта POSIX */
        return(NULL);
    }
    if (childpid == NULL) {     /* самый первый вызов функции */
        /* разместить массив идентификаторов потомков, заполненный нулями */
        maxfd = open_max();
        if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }

    if (pipe(pfd) < 0)
        return(NULL);   /* значение errno будет установлено функцией pipe() */

    if ((pid = fork()) < 0) {
        return(NULL);   /* значение errno будет установлено функцией fork() */
    } else if (pid == 0) {  /* дочерний процесс */
        if (*type == 'r') {
            close(pfd[0]);
            if (pfd[1] != STDOUT_FILENO) {
```

```
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        } else {
            close(pfd[1]);
            if (pfd[0] != STDIN_FILENO) {
                dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
            }
        }

        /* закрыть все дескрипторы в childpid[] */
        for (i = 0; i < maxfd; i++)
            if (childpid[i] > 0)
                close(i);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);
    }

    /* родительский процесс... */
    if (*type == 'r') {
        close(pfd[1]);
        if ((fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
        close(pfd[0]);
        if ((fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }

    childpid[fileno(fp)] = pid; /* запомнить pid потомка для данного fd */
    return(fp);
}

int
pclose(FILE *fp)
{
    int fd, stat;
    pid_t pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);             /* функция popen() никогда не вызывалась */
}

    fd = fileno(fp);
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);             /* fp не был открыт функцией popen() */
    }

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
```

```
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* от waitpid получен код ошибки, отличный от EINTR */
    return(stat);              /* вернуть код завершения потомка */
}
```

В принципе функция popen похожа на тот код, который мы использовали ранее в этой главе, однако существует ряд моментов, которые необходимо принять во внимание. Прежде всего, каждый раз, когда вызывается функция popen, нужно запоминать идентификатор дочернего процесса и дескриптор файла либо указатель на объект FILE. Мы решили сохранять идентификатор дочернего процесса в массиве, который индексируется номерами файловых дескрипторов. Благодаря этому функция pclose, получая указатель на объект FILE, может восстановить по нему номер дескриптора файла с помощью функции fileno и затем извлечь из массива идентификатор дочернего процесса, чтобы передать его функции waitpid. Поскольку заданный процесс может вызывать функцию popen много раз, при первом обращении к функции popen мы размещаем в динамической памяти массив childpid максимального размера.

Вызовы функций pipe и fork и создание дубликатов дескрипторов для каждого процесса производятся практически так же, как мы это делали раньше.

Стандарт POSIX.1 требует, чтобы в дочернем процессе функция popen закрывала все потоки, которые были открыты предыдущими обращениями к ней. Для этого в дочернем процессе выполняется обход массива childpid и закрытие всех открытых дескрипторов.

Что случится, если процесс, вызывающий функцию pclose, установил обработчик сигнала SIGCHLD? В этом случае функция waitpid вернет код ошибки EINTR. Так как мы допускаем, что вызывающий процесс может перехватывать сигнал SIGCHLD (или любой другой сигнал, в результате чего может быть прервано выполнение системного вызова waitpid), то мы просто вызываем функцию waitpid еще раз, если ее выполнение было прервано в результате перехвата сигнала.

Обратите внимание: приложение может самостоятельно вызвать функцию waitpid и получить код завершения дочернего процесса, созданного функцией popen. В этом случае функция waitpid, вызываемая из pclose, обнаружит отсутствие дочернего процесса и вернет значение –1 с кодом ошибки ECHILD в переменной errno. Такое поведение регламентируется стандартом POSIX.1.

> Некоторые ранние версии pclose возвращали код ошибки EINTR, если выполнение функции wait было прервано перехваченным сигналом. Кроме того, в некоторых ранних версиях pclose игнорировались или блокировались сигналы SIGINT, SIGQUIT и SIGHUP. В стандарте POSIX.1 такое поведение считается недопустимым.

Обратите внимание, что функция popen никогда не должна вызываться из программ, для которых установлен бит set-user-ID или set-group-ID. Выполнение команды функцией popen эквивалентно выполнению инструкции

```
execl("/bin/sh", "sh", "-c", command, NULL);
```

которая запускает командный интерпретатор и команду *command* с окружением, унаследованным от вызывающей программы. В этом случае злоумышленник, манипулируя значениями переменных окружения, получает возможность запускать произвольные команды с повышенными привилегиями.

Функция popen особенно удобна для организации взаимодействия с простыми фильтрами, предназначенными для преобразования входных или выходных данных запускаемой команды. Это относится к случаям, когда программа сама выстраивает конвейер команд.

Пример

Рассмотрим пример программы, которая выводит на стандартный вывод приглашение и читает введенную строку со стандартного ввода. С помощью функции popen можно поместить некоторую программу между основным приложением и его стандартным вводом, чтобы выполнить первичную обработку входных данных. На рис. 15.7 показано, как взаимодействуют процессы в такой ситуации.

В качестве первичной обработки может выполняться, например, автоматическое дополнение имен файлов или предоставление истории команд (сохранение ранее введенных команд).

В листинге 15.6 показан пример подобного простого фильтра. Этот фильтр копирует данные со стандартного ввода на стандартный вывод, преобразуя символы верхнего регистра в нижний регистр. В следующем разделе, когда мы будем рассказывать о сопроцессах, вы узнаете, почему мы вставили вызов функции fflush после вывода символа перевода строки.

Листинг 15.6. Фильтр для преобразования символов верхнего регистра в нижний регистр

```
#include "apue.h"
#include <ctype.h>

int
main(void)
{
```

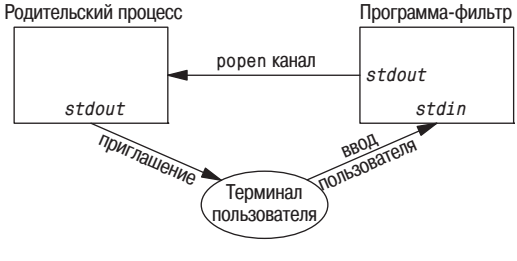


***Рис. 15.7.** Первичная обработка входных данных с помощью функции popen*

```
    int c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("ошибка вывода символа");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

Мы скомпилировали этот фильтр в исполняемый файл `myuclc`, который будет вызываться функцией `popen` из программы, представленной в листинге 15.7.

Листинг 15.7. Вызов фильтра преобразования регистра символов при чтении данных

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char line[MAXLINE];
    FILE *fpin;

    if ((fpin = popen("myuclc", "r")) == NULL)
        err_sys("ошибка вызова функции popen");

    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL)   /* чтение из канала */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("ошибка вызова функции fputs");
    }
    if (pclose(fpin) == -1)
        err_sys("ошибка вызова функции pclose");
    putchar('\n');
    exit(0);
}
```

Вызов `fflush` после вывода строки приглашения необходим по той простой причине, что стандартный вывод обычно буферизуется построчно, а строка приглашения не включает символ перевода строки.

15.4. Сопроцессы

В системе UNIX фильтрами называются программы, которые читают входные данные со стандартного ввода и выводят результаты на стандартный вы-

вод. Как правило, фильтры используются при конвейерной обработке данных. Фильтр является *сопроцессом*, если одна и та же программа предоставляет данные для фильтра и получает его вывод.

Командная оболочка Korn shell предоставляет возможность запуска сопроцессов (см. [Bolsky and Korn 1995]). Командные оболочки Bourne shell, Bourne-again shell и C shell такой возможности не имеют. Обычно сопроцесс запускается из командной оболочки в фоновом режиме, и его стандартный ввод и стандартный вывод соединены с другой программой посредством каналов. Несмотря на то, что синтаксис команд оболочки, необходимых для запуска сопроцесса и соединения его ввода и вывода с другим процессом, весьма запутан (за подробностями обращайтесь к [Bolsky and Korn 1995], стр. 62–63), работа с сопроцессами достаточно удобна из программ, написанных на C.

Учитывая однонаправленную природу каналов, для организации взаимодействия с сопроцессом необходимо создать два однонаправленных канала – один к стандартному вводу сопроцесса и другой от его стандартного вывода. После этого можно записать данные на стандартный ввод сопроцесса, обработать их и прочитать результат с его стандартного вывода.

Пример

Давайте рассмотрим пример сопроцесса. Основной процесс создает два канала: один связан со стандартным вводом сопроцесса, а второй – с его стандартным выводом. Эта ситуация показана на рис. 15.8.

Программа из листинга 15.8 представляет собой простейший сопроцесс, который принимает два числа со стандартного ввода, вычисляет сумму и выводит ее на стандартный вывод. (Сопроцессам обычно доверяют более серьезные задачи. Это во многом искусственный пример, он придуман лишь с целью изучить механизмы взаимодействия между процессами.)

Листинг 15.8. Простейший фильтр, который складывает два числа

```
#include "apue.h"

int
main(void)
{
    int n, int1, int2;
    char line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;      /* завершить строку нулевым символом */
```

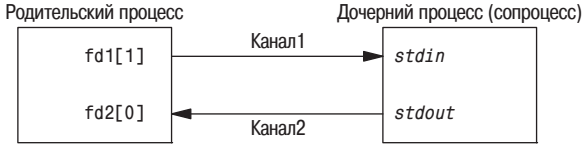


***Рис. 15.8.** Запись на стандартный ввод и чтение со стандартного вывода сопроцесса*

```
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("ошибка вызова функции write");
        } else {
            if (write(STDOUT_FILENO, "неверные аргументы\n", 19) != 19)
                err_sys("ошибка вызова функции write");
        }
    }
    exit(0);
}
```

Мы скомпилировали эту программу в исполняемый файл `add2`.

Программа, представленная листингом 15.9, читает два числа со стандартного ввода и вызывает сопроцесс `add2`. Значение, полученное от сопроцесса, выводится на стандартный вывод.

Листинг 15.9. Программа, которая использует фильтр add2

```
#include "apue.h"

static void sig_pipe(int);              /* обработчик сигнала */

int
main(void)
{
    int n, fd1[2], fd2[2];
    pid_t pid;
    char line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("ошибка вызова функции signal");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("ошибка вызова функции pipe");

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) {               /* родительский процесс */
        close(fd1[0]);
        close(fd2[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd1[1], line, n) != n)
                err_sys("ошибка записи в канал");
            if ((n = read(fd2[0], line, MAXLINE)) < 0)
                err_sys("ошибка чтения из канала");
            if (n == 0) {
                err_msg("канал был закрыт в дочернем процессе");
                break;
            }
            line[n] = 0; /* добавить завершающий нулевой символ */
            if (fputs(line, stdout) == EOF)
```

```
                err_sys("ошибка вызова функции fputs");
        }
        if (ferror(stdin))
            err_sys("ошибка получения данных со стандартного ввода");
        exit(0);
    } else {                            /* дочерний процесс */
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != STDIN_FILENO) {
            if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("ошибка вызова функции dup2 для stdin");
            close(fd1[0]);
        }
        if (fd2[1] != STDOUT_FILENO) {
            if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("ошибка вызова функции dup2 для stdout");
            close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *)0) < 0)
            err_sys("ошибка вызова функции execl");
    }
    exit(0);
}

static void
sig_pipe(int signo)
{
    printf("перехвачен сигнал SIGPIPE\n");
    exit(1);
}
```

Эта программа создает два канала, дочерний и родительский процессы закрывают ненужные дескрипторы каналов. Мы должны использовать два канала: один в качестве стандартного ввода сопроцесса, а второй в качестве его стандартного вывода. Затем, перед вызовом `execl`, дочерний процесс вызывает функцию `dup2`, чтобы перенести дескрипторы каналов на свои стандартные устройства ввода и вывода.

Если скомпилировать и запустить программу из листинга 15.9, она будет работать так, как мы и ожидали. Если в то время, когда основная программа ждет ввода двух чисел, завершить сопроцесс `add2` с помощью команды `kill`, то при попытке выполнить запись в канал, для которого отсутствует читающий процесс, основная программа получит сигнал `SIGPIPE` (упражнение 15.4).

В табл. 15.1 было указано, что функция `pipe` не во всех системах создает дуплексные каналы. В листинге 17.1 приводится еще одна версия этого примера. Вместо двух полудуплексных каналов она использует один дуплексный канал в тех системах, которые поддерживают дуплексные неименованные каналы.

Пример

В примере сопроцесса add2 (листинг 15.8) мы преднамеренно использовали низкоуровневые операции ввода-вывода (системные вызовы UNIX): read и write. А может ли сопроцесс использовать стандартную библиотеку ввода-вывода? Такая версия сопроцесса приводится в листинге 15.10.

Листинг 15.10. Простейший фильтр, складывающий два числа и реализованный с применением стандартной библиотеки ввода-вывода

```
#include "apue.h"

int
main(void)
{
    int int1, int2;
    char line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            if (printf("%d\n", int1 + int2) == EOF)
                err_sys("ошибка вызова функции printf");
        } else {
            if (printf("неверные аргументы\n") == EOF)
                err_sys("ошибка вызова функции printf");
        }
    }
    exit(0);
}
```

Если теперь попытаться вызвать этот новый сопроцесс из программы, представленной листингом 15.9, она перестанет работать. Проблема в том, что стандартная библиотека по умолчанию буферизует операции ввода-вывода. Когда вызывается программа из листинга 15.10, при первом обращении к функции fgets стандартная библиотека ввода-вывода размещает буфер и выбирает режим буферизации. Поскольку стандартный ввод является каналом, библиотека по умолчанию выбирает режим полной буферизации. То же самое происходит и со стандартным выводом. Пока программа add2 ожидает поступления данных со стандартного ввода, основная программа (из листинга 15.9) ожидает поступления данных из канала. В результате возникает тупиковая ситуация.

Можно изменить программу из листинга 15.10, добавив перед циклом while следующие четыре строки:

```
if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
    err_sys("ошибка вызова функции setvbuf");
if (setvbuf(stdout, NULL, _IOLBF, 0) != 0)
    err_sys("ошибка вызова функции setvbuf");
```

Эти строки заставят функцию fgets вернуть управление, когда строка символов будет записана в канал, а функцию printf – вызвать fflush после вывода символа перевода строки (подробное описание режимов буферизации

в стандартной библиотеке ввода-вывода приводилось в разделе 5.4). Добавив явные вызовы функций setvbuf, мы исправим ошибку в программе из листинга 15.10.

Иная методика требуется, если у нас нет возможности изменить программу, к стандартному выводу которой мы присоединяем канал. Например, при использовании в нашей программе в качестве сопроцесса программы awk(1) (вместо программы add2), следующий код не будет работать:

```
#! /bin/awk -f
{ print $1 + $2 }
```

Причина опять кроется в буферизации операций ввода-вывода. Но на этот раз мы не можем изменить программу awk (если, конечно, мы не имеем исходных текстов этой программы). У нас нет возможности внести изменения в исполняемый файл, чтобы изменить режим буферизации по умолчанию.

Решение этой проблемы заключается в том, чтобы заставить сопроцесс (в данном случае awk) думать, что его стандартный ввод и стандартный вывод соединены с терминалом. Это заставит функции стандартной библиотеки ввода-вывода в сопроцессе установить режим построчной буферизации для двух потоков ввода-вывода, как если бы функция setvbuf была вызвана явным образом. Для этого в главе 19 мы будем использовать псевдотерминалы.

15.5. FIFO

Каналы FIFO (First In First Out – первым пришел, первым ушел) иногда еще называют именованными каналами. Неименованные каналы могут использоваться только для организации взаимодействия процессов, которые имеют общего предка, создавшего каналы. (Исключение из этого правила составляют монтируемые каналы на основе механизма STREAMS, которые мы будем рассматривать в разделе 17.2.2.) С помощью каналов FIFO можно организовать взаимодействие между процессами, которые не связаны «родственными узами».

В главе 4 мы уже видели, что FIFO – это особый тип файлов. Определенный код в поле st_mode структуры stat (раздел 4.2) указывает, что файл является каналом FIFO. Проверить файл на принадлежность к типу FIFO можно с помощью макроса S_ISFIFO.

Создание канала FIFO очень похоже на создание обычного файла. Канал с полным именем *pathname* действительно создается в пределах файловой системы.

```
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Аргумент *mode* для функции mkfifo указывается точно так же, как и для функции open (раздел 3.3). Правила назначения пользователя и группы владельца FIFO совпадают с описанными в разделе 4.6.

После создания канала FIFO с помощью функции mkfifo мы можем открыть его функцией open. Все обычные функции ввода-вывода (close, read, write, unlink и др.) могут работать с каналами FIFO.

> Приложения могут создавать каналы FIFO с помощью функции mknod. Поскольку изначально стандарт POSIX.1 не включал в себя функцию mknod, специально для этого стандарта была придумана функция mkfifo. Сейчас функция mknod включена в стандарт в виде расширения XSI. В большинстве систем функция mkfifo создает канал FIFO с помощью функции mknod.
>
> Стандарт POSIX.1 также включает команду mkfifo(1). Все четыре платформы, обсуждаемые в этой книге, поддерживают данную команду. Она позволяет создавать каналы FIFO из командной оболочки, чтобы затем использовать их для перенаправления ввода-вывода.

При открытии FIFO наличие флага O_NONBLOCK оказывает следующее влияние.

- В обычной ситуации (флаг O_NONBLOCK не указан) операция открытия FIFO только для чтения будет заблокирована до тех пор, пока другой процесс не откроет канал для записи. Аналогично, операция открытия только для записи будет заблокирована, пока другой процесс не откроет канал для чтения.
- Если флаг O_NONBLOCK указан, при попытке открыть канал только для чтения функция open сразу же вернет управление. Но при попытке открыть канал только для записи функция open вернет значение –1 и код ошибки ENXIO в переменной errno, если канал не был открыт другим процессом для чтения.

Как и в случае с неименованными каналами, если мы попытаемся выполнить запись в FIFO, который не был открыт для чтения, процесс получит сигнал SIGPIPE. Когда последний пишущий в FIFO процесс закроет канал, читающий процесс получит признак конца файла.

Нередко запись данных в канал FIFO выполняется из нескольких процессов. Это означает, что необходимо побеспокоиться об атомарности операции записи, если мы хотим избежать смешивания данных, поступающих от разных процессов. (Решение этой проблемы мы увидим в разделе 17.2.2.) Максимальный объем данных, который может быть атомарно записан в канал FIFO, определяется, как и для неименованных каналов, константой PIPE_BUF.

Каналы FIFO применяются в двух случаях:

1. Каналы FIFO используются командными оболочками для передачи данных от одного конвейера команд другому без создания временных файлов для хранения промежуточных данных.
2. Каналы FIFO используются для организации взаимодействий типа клиент-сервер.

Каждый из этих случаев мы рассмотрим на конкретных примерах.

Пример – использование FIFO для дублирования вывода

Каналы FIFO могут использоваться для дублирования данных, передаваемых между сериями команд оболочки. Это помогает избежать создания временных файлов для хранения промежуточных данных (как и неименованные каналы). Но если неименованные каналы могут служить исключительно для линейного объединения процессов в конвейер, то каналы FIFO, благодаря наличию имени, могут использоваться для нелинейного объединения.

Рассмотрим ситуацию, когда необходимо обработать отфильтрованные данные дважды. Эта ситуация показана на рис. 15.9.

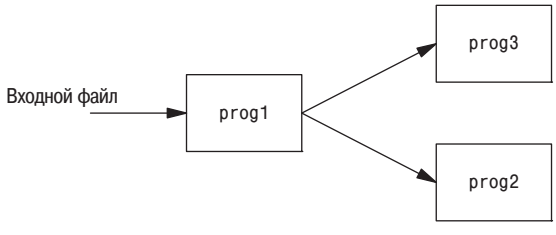


***Рис. 15.9.** Ситуация, когда необходимо обработать отфильтрованные данные дважды*

С помощью канала FIFO и программы `tee`(1) можно реализовать эту процедуру без использования временного файла. (Программа `tee` копирует данные, получаемые со стандартного ввода, на стандартный вывод и в файл, заданный в командной строке.)

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

Эта последовательность команд создает канал FIFO, после чего запускает в фоновом режиме программу `prog3`, которая читает данные из канала. Затем запускается фильтр `prog1`, вывод которого через команду `tee` поступает в канал и на вход программы `prog2`. На рис. 15.10 показана схема взаимодействия процессов в этой ситуации.

Пример – взаимодействия типа клиент-сервер

Еще одна область применения каналов FIFO – передача данных между клиентом и сервером. Если у нас имеется сервер, который обслуживает многочисленные клиентские приложения, каждый клиент может посылать запросы через канал FIFO с известным именем, заранее созданный сервером. (Имеется в виду, что полное имя канала FIFO заранее известно всем клиентам, которые нуждаются в услугах сервера.) На рис. 15.11 показана схема такого взаимодействия. Поскольку в этой ситуации писать в канал FIFO могут сразу несколько процессов, необходимо, чтобы размеры запросов не пре-

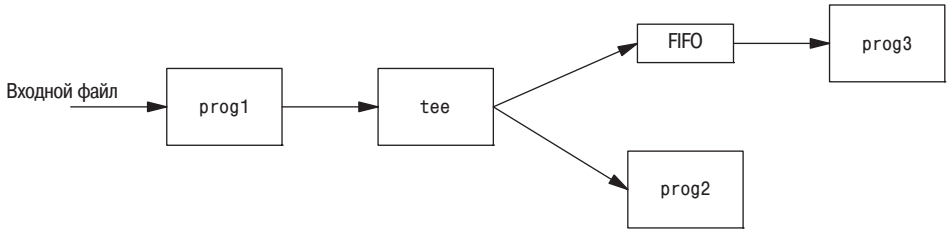


***Рис. 15.10.** Использование канала FIFO и команды tee для обработки потока данных двумя процессами*

вышали величины PIPE_BUF. Это позволит избежать смешивания данных, записываемых различными процессами.

При использовании каналов FIFO для организации взаимодействий такого типа возникает проблема с отправкой ответа сервера клиенту. Для этого не может использоваться единственный канал FIFO, поскольку клиенты не смогут отличить ответ сервера на свой запрос от ответов на запросы других клиентов. Одно из решений состоит в том, чтобы каждый клиент отсылал вместе с запросом идентификатор процесса. Тогда сервер мог бы создавать каналы FIFO для связи с каждым клиентом, генерируя имя канала на основе идентификатора процесса клиента. Например, сервер может создавать каналы FIFO с именами /tmp/serv1.XXXXX, где XXXXX – идентификатор процесса клиента. На рис. 15.12 показана схема такого взаимодействия.

Такая схема вполне работоспособна, хотя сервер не может обнаружить ситуацию аварийного завершения клиента. Это приводит к тому, что каналы FIFO, созданные для взаимодействия с конкретными клиентами, остаются в файловой системе. Кроме того, сервер должен предусматривать обработку сигнала SIGPIPE, поскольку клиент может послать запрос и завершить работу, не дожи-

***Рис. 15.11.** Обмен данными между клиентами и сервером с помощью канала FIFO*

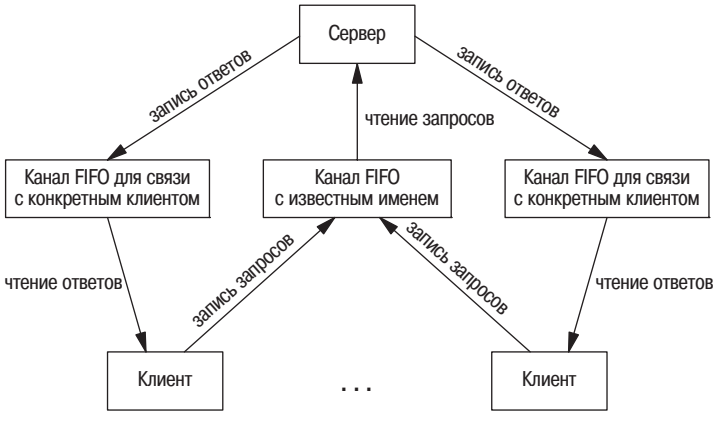


***Рис. 15.12.** Организация взаимодействий типа клиент-сервер с помощью каналов FIFO*

даясь получения ответа и оставляя свой канал FIFO с одним пишущим процессом (сервер), но без читающего процесса. Более элегантное решение этой проблемы мы увидим в разделе 17.2.2, когда будем обсуждать монтируемые каналы STREAMS.

В ситуации, показанной на рис. 15.12, если сервер откроет канал FIFO с заранее предопределенным именем только для чтения, то всякий раз, когда количество клиентов будет достигать 0, сервер будет получать из канала признак конца файла. Чтобы этого не происходило, сервер обычно открывает канал FIFO как для чтения, так и записи (упражнение 15.10).

15.6. XSI IPC

Три типа IPC, которые называются XSI IPC, – очереди сообщений, семафоры и разделяемая память – имеют много общего. В этом разделе мы рассмотрим общие для всех трех типов взаимодействий характеристики, а в следующем разделе – функции, специфичные для каждого из них.

> Функции XSI IPC целиком основаны на функциях System V IPC. Эти три типа взаимодействий появились в 70-х годах во внутренней версии UNIX AT&T, которая называлась Columbus UNIX. Позднее эти механизмы IPC были добавлены в System V. Их часто критикуют за то, что они используют свою собственную схему именования объектов, а не файловую систему.
>
> В табл. 15.1 было указано, что очереди сообщений, семафоры и разделяемая память определяются стандартом Single UNIX Specification как расширения XSI.

15.6.1. Идентификаторы и ключи

Каждой структуре IPC (очереди сообщений, семафору или сегменту разделяемой памяти) в ядре соответствует неотрицательное целое число – *идентификатор*. Например, чтобы отправить сообщение в очередь или получить его,

достаточно знать идентификатор очереди. В отличие от дескрипторов файлов, идентификаторы IPC не являются маленькими целыми числами. Каждый раз, когда создается какая-либо структура IPC, идентификатор, присваиваемый этой структуре, все время увеличивается, пока не достигнет максимально возможного значения для целых чисел, после чего сбрасывается в ноль.

Идентификатор – это внутреннее имя объекта IPC. Процессам же необходим механизм внешних имен, чтобы организовать взаимодействие через определенный объект IPC. Для этого каждому объекту IPC ставится в соответствие *ключ*, который и выступает в роли внешнего имени.

Всякий раз, когда создается структура IPC (`msgget`, `semget` или `shmget`), обязательно должен быть указан ключ. Тип данных ключа является одним из основных системных типов данных – `key_t`, который часто определяется в заголовочном файле `<sys/types.h>` как длинное целое со знаком. Ядро выполняет преобразование этого ключа во внутренний идентификатор.

Существуют разные способы организовать взаимодействие между клиентом и сервером через структуру IPC.

1. Сервер может создать новую структуру IPC с ключом `IPC_PRIVATE` и записать куда-нибудь (например, в файл) полученный идентификатор, чтобы сделать его доступным для клиента. Ключ `IPC_PRIVATE` гарантирует, что сервер создаст совершенно новую структуру IPC. Недостаток этого метода заключается в том, что в нем применяются операции с файловой системой – сервер должен записать идентификатор в файл, а клиент должен прочитать его из файла.

   Ключ `IPC_PRIVATE` также может использоваться для организации взаимодействия родительского и дочернего процессов. Родительский процесс создает новую структуру IPC с ключом `IPC_PRIVATE`, а полученный в результате идентификатор останется доступным для потомка после вызова функции `fork`. Дочерний процесс может передать полученный идентификатор новой программе в качестве аргумента функции `exec`.

2. Клиент и сервер могут договориться об использовании предопределенного ключа, задав его, например, в общем заголовочном файле. После этого сервер может создавать структуру IPC с заданным ключом. Но такое решение также не лишено недостатков – есть вероятность, что в системе уже существует некоторая структура IPC с точно таким же ключом. В этом случае функции создания структуры (`msgget`, `semget` или `shmget`) будут возвращать признак ошибки. Сервер должен правильно обработать ошибочную ситуацию, удалить существующую структуру IPC и попытаться создать ее снова.

3. Клиент и сервер могут договориться об использовании некоторого полного имени файла и идентификатора проекта (идентификатор проекта – это символ с кодом от 0 до 255), на основе которых с помощью функции `ftok`

можно получить значение ключа. После этого полученный ключ может использоваться точно так же, как и в предыдущем случае.

```
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

Возвращает ключ в случае успеха, (key_t) −1 в случае ошибки

Аргумент *path* должен содержать имя существующего файла. В создании ключа используются только 8 младших бит аргумента *id*.

Обычно при создании ключа используются значения полей st_dev и st_ino структуры stat (раздел 4.2), соответствующей файлу с заданным именем, в комбинации с идентификатором проекта. Если имена файлов различаются, то функция ftok обычно возвращает разные ключи. Однако, поскольку и номера индексных узлов, и ключи часто хранятся в виде длинных целых чисел со знаком, то при создании ключа может происходить некоторая потеря информации. Это означает, что существует вероятность, когда при использовании различных имен файлов будут сгенерированы одинаковые ключи, если в обоих случаях использовались одинаковые значения идентификатора проекта.

Все три функции get (msgget, semget и shmget) имеют одинаковые аргументы: *key* и *flag*. Новая структура IPC создается (обычно сервером) в том случае, если в аргументе *key* передается значение IPC_PRIVATE либо заданный ключ не соответствует какой-либо существующей структуре IPC данного типа, и в аргументе *flag* передается флаг IPC_CREAT. Чтобы получить ссылку на существующую очередь (обычно со стороны клиента), в аргументе *key* нужно передать значение, совпадающее с ключом, использовавшимся при создании этой очереди, а флаг IPC_CREAT не должен быть установлен.

Обратите внимание: если используется ключ IPC_PRIVATE, то получить ссылку на существующую очередь невозможно, так как с помощью этого специального ключа всегда создается новая очередь. Чтобы иметь возможность обращаться к существующей очереди, которая была создана с ключом IPC_PRIVATE, мы должны узнать связанный с ней идентификатор и затем использовать его во всех остальных функциях работы с объектом IPC (таких как msgsnd или msgrcv) в обход функции get.

Если нужно создать новую структуру IPC, а не получить ссылку на существующую, мы должны в аргументе *flag* вместе с флагом IPC_CREAT указать флаг IPC_EXCL. В результате, если данная структура IPC уже существует, функция вернет признак ошибки с кодом EEXIST. (Очень напоминает правила определения флагов O_CREAT и O_EXCL функции open.)

15.6.2. Структура прав доступа

С каждой структурой IPC механизм XSI IPC связывает структуру ipc_perm. Эта структура определяет права доступа к объекту и его владельца. Она содержит как минимум следующие поля:

```
struct ipc_perm {
    uid_t uid;     /* эффективный идентификатор пользователя владельца */
    gid_t gid;     /* эффективный идентификатор группы владельца */
    uid_t cuid;    /* эффективный идентификатор пользователя создателя */
    gid_t cgid;    /* эффективный идентификатор группы создателя */
    mode_t mode;   /* режим доступа */
    .
    .
    .
};
```

Каждая реализация включает в эту структуру дополнительные поля. Полное определение этой структуры в своей системе вы найдете в заголовочном файле <sys/ipc.h>.

Все поля структуры инициализируются при создании структуры IPC. Позднее можно изменить состояние полей uid, gid и mode с помощью функций msgctl, semctl или shmctl. Чтобы иметь возможность изменять эти значения, процесс должен обладать правами создателя структуры или правами суперпользователя. Изменение этих полей структуры похоже на вызов функций chown или chmod для обычного файла.

Значения поля mode напоминают значения, которые мы уже видели в табл. 4.5, за исключением права на исполнение. Кроме того, применительно к очередям сообщений и разделяемой памяти используются термины «право на чтение» и «право на запись», а применительно к семафорам – «право на чтение» и «право на изменение». В табл. 15.2 приводится список различных прав доступа к каждой из структур IPC.

Таблица 15.2. Права доступа XSI IPC

Право доступа	Бит
user-read – доступно пользователю для чтения	0400
user-write – доступно пользователю для записи (изменения)	0200
group-read – доступно группе для чтения	0040
group-write – доступно группе для записи (изменения)	0020
other-read – доступно остальным для чтения	0004
other-write – доступно остальным для записи (изменения)	0002

Некоторые реализации определяют символические константы для каждого бита прав доступа, однако имена этих констант не стандартизированы в Single UNIX Specification.

15.6.3. Конфигурируемые пределы

Мы можем столкнуться с встроенными пределами для всех трех форм XSI IPC. Большинство из них могут быть изменены при переконфигурировании ядра. Мы будем рассматривать эти пределы при обсуждении каждой из трех форм IPC.

Каждая из платформ предоставляет свой собственный способ получения и изменения конкретных пределов. В ОС FreeBSD 5.2.1, Linux 2.4.22 и Mac OS X 10.3 имеется команда `sysctl`, с помощью которой можно просмотреть и изменить конфигурационные параметры ядра. В Solaris 9 для изменения конфигурационных параметров ядра нужно отредактировать файл `/etc/system` и перезагрузить систему.

В Linux можно просмотреть пределы, связанные с IPC, запустив команду `ipcs -l`. В FreeBSD ей соответствует команда `ipcs -T`. В Solaris можно просмотреть значения настраиваемых параметров, запустив команду `sysdef -i`.

15.6.4. Преимущества и недостатки

Фундаментальная проблема всех форм XSI IPC заключается в том, что структуры IPC привязаны к системе в целом, а не к конкретному процессу, и не имеют счетчика ссылок. Например, если мы создадим очередь сообщений, поместим в нее некоторое сообщение и завершим процесс, то очередь не будет удалена. Сообщение останется в системе до тех пор, пока не будет прочитано каким либо процессом с помощью функции `msgrcv`, удалено с помощью функции `msgctl` или команды `ipcrm`(1), или пока система не будет перезагружена. Сравните это с неименованными каналами, которые удаляются автоматически, когда завершается последний процесс, имеющий ссылку на этот канал. В случае с FIFO имя канала остается в системе до тех пор, пока явно не будет удалено, но данные удаляются из канала FIFO автоматически, когда завершается последний процесс, имеющий ссылку на этот канал.

Другая проблема, связанная с механизмами XSI IPC, заключается в том, что они не имеют имен в файловой системе. Мы не можем получить к ним доступ или изменить их свойства с помощью функций, описанных в главах 3 и 4. Для поддержки этих объектов IPC в ядро была добавлена почти дюжина новых системных вызовов (`msgget`, `semop`, `shmat` и другие). Мы не можем получить список существующих объектов IPC с помощью команды `ls`, удалить их с помощью команды `rm` и изменить права доступа к ним с помощью команды `chmod`. Вместо них должны использоваться две новые команды – `ipcs`(1) и `ipcrm`(1).

Так как эти формы IPC не используют файловые дескрипторы, нельзя использовать для работы с ними функции мультиплексирования ввода-вывода (`select` и `poll`). Это осложняет одновременную работу с более чем одной структурой IPC или использование какой-либо из этих структур совместно с файлами или устройствами ввода-вывода. Например, мы не можем организовать на стороне сервера ожидание сообщения, которое может поступить по одной из двух очередей, не применяя для этого ту или иную форму цикла активного ожидания.

Краткий обзор системы диалоговой обработки запросов, построенной на основе System V IPC, приводится в [Andrade, Carges, and Kovach 1989]. Авторы этой книги утверждают, что принцип именования, используемый System V IPC (идентификаторы), является преимуществом, а не недостатком, как мы говорили выше, потому что идентификаторы позволяют процессам посылать сообщения в очередь сообщений, используя для этого всего одну функцию (`msgsnd`), тогда как другие формы IPC требуют вызова трех функций:

open, write и close. Значение идентификатора, присвоенного конкретной очереди, зависит от количества существующих очередей сообщений и от того, сколько раз создавались новые очереди с момента последней перезагрузки ядра. Это динамическое значение – его невозможно предугадать или предопределить в заголовочном файле. Как мы уже говорили в разделе 15.6.1, в простейшем случае сервер должен записать идентификатор очереди в файл, который может быть прочитан клиентами.

Среди других преимуществ очередей сообщений, на которые указывают авторы упоминавшейся выше книги, можно назвать надежность, управление ходом исполнения, ориентированность на отдельные записи и возможность извлекать сообщения не только в порядке их помещения в очередь. Как мы уже видели в разделе 14.4, всеми этими свойствами обладает механизм STREAMS, хотя он и требует вызова функции open перед передачей данных в поток и вызова функции close по окончании работы с потоком. В таблице 15.3 приводятся некоторые сравнительные характеристики различных форм IPC.

Таблица 15.3. Сравнение некоторых характеристик различных форм IPC

Тип IPC	Ориентированность на установление соединения	Надежность	Управление ходом исполнения	Записи	Типы сообщений или свойства
Очереди сообщений	Да	Да	Да	Да	Да
STREAMS	Да	Да	Да	Да	Да
Сокеты домена UNIX, ориентированные на потоки	Да	Да	Да	Нет	Нет
Сокеты домена UNIX, ориентированные на дейтаграммы	Нет	Да	Нет	Да	Нет
FIFO (не STREAMS)	Да	Да	Да	Нет	Нет

(Сокеты, ориентированные на потоки и на дейтаграммы, будут описаны в главе 16. Сокеты домена UNIX будут описаны в разделе 17.3.) Под понятием «ориентированность на установление соединения» подразумевается необходимость предварительного вызова некоторой функции открытия механизма IPC. Как было сказано ранее, мы считаем, что очереди сообщений ориентированы на установление соединения, поскольку для получения идентификатора очереди должны быть предварительно выполнены некоторые действия. Так как область применения всех этих механизмов IPC ограничивается одним хостом, их можно отнести к разряду надежных. Возможность потери сообщений возникает, если сообщения передаются через сеть. Под «управлением ходом исполнения» подразумевается, что передающий процесс может быть приостановлен, если в приемном буфере недостаточно места или принимающий процесс в данное время не может принять сообщение. Когда появится возможность принять сообщение от передающего процесса, его работа будет возобновлена автоматически.

Одна из характеристик, которую мы не упомянули в табл. 15.3, – это возможность автоматически создавать уникальное соединение между сервером и каждым из клиентов. В главе 17 мы увидим, что механизм STREAMS и сокеты, ориентированные на потоки, предоставляют такую возможность.

В следующих трех разделах подробно описываются все три формы XSI IPC.

15.7. Очереди сообщений

Очередь сообщений – это связный список сообщений, который хранится в памяти ядра и идентифицируется идентификатором очереди сообщений. Далее мы будем называть очередь сообщений просто *очередью*, а ее идентификатор – *идентификатором очереди*.

> Стандарт Single UNIX Specification включает определение альтернативной реализации механизма очередей сообщений в расширениях реального времени. Но в этой книге мы не будем рассматривать расширения реального времени.

Создание новой очереди или открытие существующей производится с помощью функции `msgget`. Новые сообщения добавляются в конец очереди функцией `msgsnd`. Каждое сообщение содержит тип (положительное длинное целое число), неотрицательное значение длины и собственно данные (объем которых определяется длиной сообщения). Все эти значения передаются функции `msgsnd` при его добавлении в очередь. Сообщения могут извлекаться из очереди не только в порядке «первым пришел – первым ушел», но и на основе типа сообщения.

С каждой очередью связывается структура `msqid_ds`:

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* раздел 15.6.2 */
    msgqnum_t msg_qnum;       /* количество сообщений в очереди */
    msglen_t msg_qbytes;      /* максимальное количество байт в очереди */
    pid_t msg_lspid;   /* идентификатор процесса, последним вызвавшего msgsnd() */
    pid_t msg_lrpid;   /* идентификатор процесса, последним вызвавшего msgrcv() */
    time_t msg_stime;         /* время последнего вызова msgsnd()*/
    time_t msg_rtime;         /* время последнего вызова msgrcv()*/
    time_t msg_ctime;         /* время последнего изменения */
    .
    .
    .
};
```

Эта структура определяет текущее состояние очереди. Поля структуры, показанные здесь, определяются стандартом Single UNIX Specification. Реализации, как правило, включают в эту структуру дополнительные поля, которые не включаются в стандарт.

В табл. 15.4 перечислены системные пределы, имеющие отношение к очередям сообщений. Если система не поддерживает ту или иную возможность, в соответствующей ячейке указано «не поддерживается». Значение «производное» указывается там, где предел является производным от других преде-

лов. Например, максимальное количество сообщений в Linux зависит от максимального количества очередей и максимального объема данных, которые могут быть помещены в очередь. Если считать, что минимальный размер сообщения составляет 1 байт, тогда максимальное количество сообщений для данной системы может быть вычислено по формуле: *максимальное_количество_очередей* × *максимальный_размер_одной_очереди*. Учитывая значения пределов, которые даны в табл. 15.4, ОС Linux в конфигурации по умолчанию ограничивает количество сообщений числом 262 144. (Даже если сообщение вообще не содержит данных, ОС Linux все равно считает, что такое сообщение имеет размер 1 байт, чтобы ограничить количество сообщений, которые могут быть помещены в очереди.)

Таблица 15.4. Системные пределы, связанные с очередями сообщений

Описание	Типичные значения			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Максимальный размер сообщения	16 384	8 192	Не поддерживается	2 048
Максимальный размер очереди в байтах (т. е. сумма всех сообщений в очереди)	2 048	16 384	Не поддерживается	4 096
Максимальное количество очередей сообщений в системе	40	16	Не поддерживается	50
Максимальное количество сообщений в системе	40	Производное	Не поддерживается	40

> В табл. 15.1 было указано, что Mac OS X 10.3 не поддерживает очереди сообщений XSI. Поскольку Mac OS X частично основана на FreeBSD, а FreeBSD поддерживает очереди сообщений, Mac OS X в принципе также может поддерживать очереди сообщений. С помощью поисковых систем вы наверняка найдете ссылки на реализации очередей сообщений XSI для Mac OS X от сторонних производителей.

Обычно при работе с очередями прежде всего вызывается функция `msgget`, которая открывает существующую или создает новую очередь.

```
#include <sys/msg.h>

int msgget(key_t key, int flag);
```

Возвращает идентификатор очереди в случае успеха, −1 в случае ошибки

В разделе 15.6.1 мы рассмотрели правила преобразования ключа в идентификатор и обсудили вопрос, когда создается новая очередь, а когда открывается существующая. При создании новой очереди инициализируются следующие поля структуры `msqid_ds`:

- Структура `msqid_ds` инициализируется, как описано в разделе 15.6.2. Поле `mode` устанавливается в соответствии со значениями битов прав доступа

в аргументе *flag*. Значения для каждого конкретного права доступа приведены в табл. 15.2.

- В поля msg_qnum, msg_lspid, msg_lrpid, msg_stime и msg_rtime записывается значение 0.
- В поле msg_ctime записывается значение текущего времени.
- В поле msg_qbytes записывается значение соответствующего системного предела.

В случае успеха функция msgget возвращает неотрицательный идентификатор очереди. Это значение может использоваться в других трех функциях, предназначенных для работы с очередями сообщений.

Функция msgctl производит различные операции над очередью. Она и аналогичные ей функции для семафоров и разделяемой памяти (semctl и shmctl) являются аналогами функции ioctl для механизмов XSI IPC.

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf );
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Аргумент *cmd* представляет собой код операции, которая должна быть выполнена над очередью, определяемой аргументом *msqid*.

IPC_STAT Получить структуру msqid_ds данной очереди и сохранить ее по адресу *buf*.

IPC_SET Скопировать из *buf* в структуру msqid_ds, которая связана с очередью, значения следующих полей: msg_perm.uid, msg_perm.gid, msg_perm.mode и msg_qbytes. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением msg_perm.cuid или msg_perm.uid или если процесс обладает привилегиями суперпользователя. Значение поля msg_qbytes может увеличить только суперпользователь.

IPC_RMID Удалить очередь сообщений и все данные, которые в ней имеются. Удаление очереди происходит немедленно. Все процессы, которые продолжают использовать очередь, получат код ошибки EIDRM при первой же попытке обращения к ней. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением msg_perm.cuid или msg_perm.uid или если процесс обладает привилегиями суперпользователя.

Позже мы увидим, что те же самые команды (IPC_STAT, IPC_SET и IPC_RMID) используются и для управления семафорами и сегментами разделяемой памяти.

Помещение данных в очередь сообщений производится с помощью функции msgsnd.

```
#include <sys/msg.h>

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Как мы уже упоминали ранее, каждое сообщение состоит из значения, определяющего тип сообщения, поля длины сообщения (*nbytes*) и собственно данных (объем которых равен значению поля длины сообщения).

Через аргумент *ptr* передается указатель на длинное целое со знаком, которое содержит положительное значение, определяющее тип сообщения, за которым сразу же размещаются данные сообщения. (Считается, что сообщение не имеет данных, если в аргументе *nbytes* передается значение 0.) Если максимальный размер отправляемых сообщений составляет 512 байт, то можно определить следующую структуру:

```
struct mymesg {
    long mtype;       /* тип сообщения – положительное число */
    char mtext[512]; /* данные сообщения, объем которых равен nbytes */
};
```

В этом случае в аргументе *ptr* можно передавать указатель на структуру `mymesg`. Тип сообщения может использоваться принимающим процессом для извлечения сообщений в порядке, отличном от порядка помещения сообщений в очередь.

> Некоторые платформы имеют как 32-битную, так и 64-битную реализации. Это сказывается на размере длинных целых чисел и указателей. Например, в 64-битной реализации ОС Solaris допускает сосуществование как 32-битных, так и 64-битных приложений. Если 32-битное приложение попытается выполнить обмен такими структурами данных с 64-битным приложением через неименованный канал или сокет, могут возникнуть проблемы, поскольку размер длинного целого для 32-битных приложений составляет 4 байта, а для 64-битных приложений – 8 байт. Это означает, что 32-битное приложение будет считать, что поле `mtext` отстоит на 4 байта от начала структуры, тогда как 64-битное приложение – что оно отстоит от начала структуры на 8 байт. В этой ситуации часть поля `mtype` 64-битного приложения будет расценена 32-битным приложением как часть поля `mtext`, а первые 4 байта поля `mtext` 32-битного приложения будут расценены 64-битным приложениям как часть поля `mtype`.
>
> Однако эта проблема отсутствует в механизме очередей сообщений XSI. ОС Solaris реализует 32-битные и 64-битные версии системных вызовов IPC с различными точками входа. Системные вызовы заранее предусматривают возможность корректного обмена данными между 32-битными и 64-битными приложениями и правильно интерпретируют размер поля типа сообщения. Единственная проблема, которая здесь может возникнуть, – это потеря информации о типе, когда 64-битное приложение посылает сообщение 32-битному приложению, поскольку 8-байтное поле типа сообщения нельзя без потерь уместить в 4-байтное поле, используемое в 32-битных приложениях. В этом случае 32-битное приложение будет получать усеченное значение типа.

В аргументе *flag* может быть указано значение `IPC_NOWAIT`. Это аналог флага `O_NONBLOCK`, который используется для определения неблокирующего режи-

ма операций файлового ввода-вывода (раздел 14.2). Если очередь сообщений заполнена до отказа (количество сообщений в очереди или общее количество байт в очереди достигло системного предела), при указании флага IPC_NOWAIT функция msgsnd будет сразу же возвращать управление с кодом ошибки EAGAIN. Если флаг IPC_NOWAIT не указан, то процесс будет заблокирован до тех пор, пока не освободится место в очереди, пока очередь не будет удалена из системы или пока не будет перехвачен какой-либо сигнал и обработчик вернет управление. Во втором случае будет возвращен код ошибки EIDRM (identifier removed – идентификатор был удален), а в последнем – код ошибки EINTR.

Обратите внимание на то, как неудачно обрабатывается ситуация удаления очереди. Поскольку для очередей сообщений не поддерживается счетчик ссылок (как для открытых файлов), удаление очереди просто приводит к появлению ошибок при последующих попытках выполнить какие-либо действия с нею. В случае семафоров ситуация удаления обслуживается точно так же. При удалении файла, напротив, его содержимое остается в неприкосновенности до тех пор, пока не будет закрыт последний дескриптор этого файла.

В случае успеха функция msgsnd изменяет содержимое структуры msqid_ds, ассоциированной с заданной очередью: в поле msg_lspid заносится идентификатор вызывающего процесса, в поле msg_stime – время вызова, а значение поля msg_qnum (количество сообщений в очереди) увеличивается на единицу.

Выборка сообщений из очереди производится функцией msgrcv.

```
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

Возвращает объем данных в сообщении в случае успеха, –1 в случае ошибки

Как и в функции msgsnd, аргумент *ptr* содержит адрес, по которому будет сохранено длинное целое число – тип сообщения, за которым сразу же следует буфер для размещения данных сообщения. Если размер полученного сообщения превышает значение *nbytes* и при этом в аргументе *flag* установлен бит MSG_NOERROR, сообщение будет усечено до размера nbytes. (В этом случае приложение никогда не узнает, что сообщение было усечено.) Если размер полученного сообщения превышает значение *nbytes* и в аргументе *flag* не установлен бит MSG_NOERROR, то вместо сообщения будет возвращен признак ошибки с кодом E2BIG (сообщение при этом останется в очереди).

Аргумент *type* позволяет определить желаемый тип сообщения.

`type == 0` Будет возвращено первое сообщение в очереди.

`type > 0` Будет возвращено первое сообщение, имеющее заданный тип.

`type < 0` Будет возвращено первое сообщение, значение типа которого меньше или равно абсолютному значению аргумента *type*.

Ненулевое значение аргумента *type* используется в том случае, когда необходимо извлекать сообщения из очереди не в порядке их помещения в очередь. Например, значение *type* может указывать приоритет сообщения. Еще один вариант использования поля *type* – клиент может передавать в нем идентификатор своего процесса, если сервер использует единственную очередь для обмена данными со всеми клиентами (разумеется, если идентификатор процесса умещается в длинное целое со знаком).

В аргументе *flag* можно указать значение IPC_NOWAIT, чтобы выполнить операцию в неблокирующем режиме. При наличии этого флага, если в очереди отсутствуют сообщения заданного типа функция msgrcv будет возвращать значение –1 с кодом ошибки ENOMSG в переменной errno. Если флаг IPC_NOWAIT не указан, операция будет заблокирована до тех пор, пока не станет доступно сообщение указанного типа, пока очередь не будет удалена (msgrcv вернет –1 и код ошибки EIDRM в переменной errno) или пока не будет перехвачен сигнал и обработчик сигнала вернет управление (msgrcv вернет –1 и код ошибки EINTR в переменной errno).

В случае успешного завершения функции msgrcv ядро обновит содержимое структуры msqid_ds, ассоциированной с заданной очередью: в поле msg_lrpid будет записан идентификатор вызывающего процесса, в поле msg_rtime – время вызова, а значение поля msg_qnum уменьшится на единицу.

Пример – сравнение производительности очередей сообщений и дуплексных каналов

Для организации двустороннего обмена между клиентом и сервером можно использовать либо очереди сообщений, либо дуплексные каналы. (В табл. 15.1 мы уже упоминали, что дуплексные каналы могут быть организованы на основе сокетов домена UNIX (раздел 17.3), хотя на некоторых платформах имеется поддержка механизма дуплексных каналов на основе функции pipe.)

В табл. 15.5 приводятся результаты сравнения производительности в ОС Solaris трех механизмов: очередей сообщений, каналов STREAMS и сокетов домена UNIX. В процессе измерений тестовая программа создавала канал IPC, вызывала функцию fork, а затем родительский процесс передавал порядка 200 мегабайт данных дочернему процессу. Всего отправлялось 100 000 сообщений по 2 000 байт в каждом. Время приводится в секундах.

Таблица 15.5. Результаты сравнения производительности альтернативных форм IPC в Solaris

Механизм IPC	Пользовательское время	Системное время	Общее время
Очередь сообщений	0,57	3,63	4,22
Канал STREAMS	0,50	3,21	3,71
Сокет домена UNIX	0,43	4,45	5,59

Результаты показывают нам, что очереди сообщений, которые изначально задумывались как скоростной механизм обмена данными, таковым более не являются (фактически, каналы STREAMS имеют более высокую производительность, чем очереди сообщений). (Когда были реализованы очереди сообщений, единственной доступной альтернативной формой IPC были полудуплексные каналы.) При рассмотрении проблем, связанных с очередями сообщений (раздел 15.6.4), мы пришли к выводу, что их не следует использовать в новых приложениях.

15.8. Семафоры

Семафоры не похожи на те формы межпроцессного взаимодействия, которые мы уже описали (именованные и неименованные каналы и очереди сообщений). Семафор – это счетчик, который используется для предоставления доступа к данным, совместно используемым несколькими процессами.

> Стандарт Single UNIX Specification включает определение альтернативного набора функций для работы с семафорами в расширениях реального времени. Но здесь мы не будем обсуждать эти функции.

Чтобы получить доступ к ресурсу, находящемуся в совместном использовании, процесс должен:

1. Проверить состояние семафора, который регулирует доступ к этому ресурсу.
2. Если семафор имеет положительное значение, процесс может обратиться к ресурсу. В этом случае процесс уменьшает значение семафора на 1, указывая тем самым, что он использовал единицу ресурса.
3. В противном случае, если семафор имеет значение 0, процесс приостанавливается до тех пор, пока значение семафора не станет больше 0. После этого процесс возобновит работу и вернется к шагу 1.

По окончании работы с ресурсом, доступ к которому регулируется семафором, значение семафора будет увеличено на 1. Если в этот момент какой-либо другой процесс находится в ожидании освобождения семафора, он возобновит свою работу.

Для корректной работы семафоров необходимо, чтобы проверка состояния семафора и его уменьшение выполнялись в виде атомарной операции. По этой причине семафоры обычно реализуются внутри ядра.

Чаще всего используется разновидность семафоров, которая получила название *двоичный семафор*. Семафоры этого типа регулируют доступ к единственному ресурсу и инициализируются значением 1. Однако вообще семафоры могут инициализироваться любым положительным значением, которое определяет, сколько единиц ресурса, может одновременно использоваться несколькими процессами.

К сожалению, на практике семафоры XSI имеют более сложную организацию. Эта сложность обусловлена следующими тремя особенностями.

1. Семафор – это не просто одиночное неотрицательное значение. Чтобы определить семафор, необходимо определить набор из одного или более семафоров. Количество семафоров в наборе задается при создании этого набора.
2. Создание набора семафоров (`semget`) происходит независимо от его инициализации (`semctl`). Это очень серьезный недостаток, поскольку невозможно атомарно создать новый набор семафоров и инициализировать их значения.
3. Поскольку семафоры, как и все формы XSI IPC, продолжают существовать даже после завершения процессов, их использующих, необходимо предусматривать в программах алгоритмы освобождения размещенных ранее наборов семафоров. В этом может помочь операция *undo*, которая будет описана немного позднее.

Каждому набору семафоров ядро ставит в соответствие структуру `semid_ds`:

```
struct semid_ds {
    struct ipc_perm sem_perm; /* раздел 15.6.2 */
    unsigned short sem_nsems; /* количество семафоров в наборе */
    time_t sem_otime;         /* время последнего вызова функции semop() */
    time_t sem_ctime;         /* время последнего изменения */
    .
    .
    .
};
```

Указанные поля структуры определены стандартом Single UNIX Specification, но реализации могут дополнять структуру `semid_ds` собственными полями.

Каждый из семафоров представлен в наборе анонимной структурой, которая содержит как минимум следующие поля:

```
struct {
    unsigned short semval;  /* значение семафора, всегда >= 0 */
    pid_t sempid;           /* идентификатор процесса, выполнившего */
                            /* последнюю операцию */
    unsigned short semncnt; /* количество процессов, */
                            /* ожидающих выполнения условия semval>curval */
    unsigned short semzcnt; /* количество процессов, */
                            /* ожидающих выполнения условия semval==0 */
    .
    .
    .
};
```

В табл. 15.6 перечислены системные пределы, которые имеют отношение к наборам семафоров.

Обычно при работе с семафорами прежде всего вызывается функция `semget`, которая возвращает идентификатор набора семафоров.

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);
```

Возвращает идентификатор набора семафоров в случае успеха, −1 в случае ошибки

Таблица 15.6. Системные пределы, которые имеют отношение к наборам семафоров

Описание	Типичные значения			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Максимальное значение любого семафора	32 767	32 767	32 767	32 767
Максимальное значение корректировки (adjust-on-exit) для любого семафора (это значение добавляется к семафору при завершении процесса)	16 384	32 767	16 384	16 384
Максимальное количество наборов семафоров в системе	10	128	87 381	10
Максимальное количество семафоров в системе	60	32 000	87 381	60
Максимальное количество емафоров в наборе	60	250	87 381	25
Максимальное количество структур `undo` в системе	30	32 000	87 381	30
Максимальное количество записей в структуре `undo`	10	32	10	10
Максимальное количество операций, выполняемых одним вызовом `semop`	100	32	100	10

В разделе 15.6.1 мы рассмотрели правила преобразования ключа в идентификатор и обсудили вопрос, когда создается новый набор семафоров, а когда открывается существующий. При создании нового набора инициализируются следующие поля структуры `semid_ds`:

- Структура `ipc_perm` инициализируется, как описано в разделе 15.6.2. Поле `mode` устанавливается в соответствии со значениями битов прав доступа в аргументе *flag*. Значения для каждого конкретного права доступа приведены в табл. 15.2.
- В поле `sem_otime` записывается значение 0.
- В поле `sem_ctime` записывается значение текущего времени.
- В поле `sem_nsems` записывается значение аргумента *nsems*.

Количество семафоров в наборе определяется аргументом *nsems*. Если создается новый набор семафоров (обычно на стороне сервера), мы должны указать значение *nsems*. Если открывается существующий набор семафоров, допускается в аргументе *nsems* передавать значение 0.

Операции над набором семафоров выполняются с помощью функции semctl.

```
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd,
           ... /* union semun arg */);
```

Возвращаемые значения описаны ниже

Четвертый аргумент функции является необязательным и зависит от выполняемой команды; если он присутствует, то представляет собой объединение semun различных аргументов команд:

```
union semun {
    int val;               /* для SETVAL */
    struct semid_ds *buf;  /* для IPC_STAT и IPC_SET */
    unsigned short *array; /* для GETALL и SETALL */
};
```

Обратите внимание, что четвертый аргумент является объединением, а не указателем на объединение.

Аргумент *cmd* определяет одну из следующих десяти операций, которые могут выполняться над набором семафоров, представленным аргументом *semid*. Пять команд, которые используются для работы с отдельным семафором, получают номер семафора в наборе из аргумента *semnum*. Значение *semnum* должно находиться в пределах от 0 до *nsems*–1 включительно.

IPC_STAT Получить структуру semid_ds, которая соответствует заданному набору семафоров, и сохранить ее по адресу *arg.buf*.

IPC_SET Установить значения полей sem_perm.uid, sem_perm.gid и sem_perm.mode в соответствии со значениями этих же полей в структуре, на которую указывает *arg.buf*. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением sem_perm.cuid или sem_perm.uid или если процесс обладает привилегиями суперпользователя.

IPC_RMID Удалить набор семафоров. Удаление происходит немедленно. Все процессы, которые продолжают использовать набор семафоров, получат код ошибки EIDRM при первой же попытке обращения к нему. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением sem_perm.cuid или sem_perm.uid или если процесс обладает привилегиями суперпользователя.

GETVAL Вернуть значение поля semval для семафора с номером *semnum*.

SETVAL Установить значение поля semval для семафора с номером *semnum*. Значение определяется в *arg.val*.

GETPID Вернуть значение поля sempid для семафора с номером *semnum*.

GETNCNT Вернуть значение поля semncnt для семафора с номером *semnum*.

GETZNCNT Вернуть значение поля semzcnt для семафора с номером *semnum*.

GETALL Вернуть значения всех семафоров в наборе. Значения сохраняются в массиве, на который указывает *arg.array*.

SETALL Установить значения всех семафоров в наборе. Значения берутся из массива, на который указывает *arg.array*.

В случае всех команд GET, за исключением GETALL, функция возвращает соответствующее значение вызывающему процессу. Для остальных команд возвращается 0.

Функция semop выполняет сразу несколько операций над набором семафоров.

```
#include <sys/sem.h>

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Аргумент *semoparray* представляет собой массив указателей на операции с семафорами, каждая из которых представлена в виде структуры sembuf:

```
struct sembuf {
    unsigned short sem_num; /* количество семафоров в наборе */
                            /* (0, 1, ..., nsems-1) */
    short sem_op;           /* операция (<0, 0 или >0) */
    short sem_flg;          /* IPC_NOWAIT, SEM_UNDO */
};
```

Аргумент *nops* определяет количество операций (элементов) в массиве.

Операция, выполняемая над каждым семафором из набора, определяется значением sem_op. Это значение может быть отрицательным, положительным или равным нулю. (Ниже мы будем упоминать флаг «undo». Этот флаг соответствует биту SEM_UNDO в поле sem_flg.)

1. Самый простой случай – положительное значение поля sem_op. Он соответствует случаю, когда процесс освобождает занятые ресурсы. Значение sem_op добавляется к значению семафора. Если указан флаг SEM_UNDO, это значение также вычитается из значения корректировки (adjust-on-exit) процесса.
2. Если значение sem_op отрицательное, это означает, что процесс желает получить ресурс, доступ к которому регулируется семафором.

Если значение семафора больше или равно абсолютному значению sem_op (ресурс доступен), абсолютное значение sem_op вычитается из значения семафора. Это гарантирует, что значение семафора ни при каких обстоятельствах не будет меньше нуля. Если указан флаг SEM_UNDO, абсолютное значение sem_op также прибавляется к величине корректировки семафора для данного процесса.

Если значение семафора меньше, чем абсолютное значение sem_op (ресурс недоступен), то вступают в силу следующие условия:

а) Если указан флаг IPC_NOWAIT, функция semop возвращает управление с кодом ошибки EAGAIN.

б) Если флаг IPC_NOWAIT не указан, то для данного семафора увеличивается значение semncnt, а выполнение вызывающего процесса приостанавливается до тех пор, пока не будет соблюдено одно из следующих условий:

- Значение семафора стало больше или равно абсолютному значению sem_op (то есть другой процесс освободил требуемый ресурс). Значение semncnt для этого семафора уменьшается (поскольку ожидание освобождения семафора можно считать законченным), и абсолютное значение sem_op вычитается из значения семафора. Если был указан флаг SEM_UNDO, то абсолютное значение sem_op также добавляется к величине корректировки семафора.
- Семафор был удален из системы. В этом случае функция semop вернет признак ошибки с кодом EIDRM.
- Процессом был перехвачен сигнал, и обработчик сигнала вернул управление. В этом случае значение semncnt уменьшается (поскольку вызывающий процесс больше не ждет освобождения ресурса), и функция semop вернет признак ошибки с кодом EINTR.

3. Если значение sem_op равно нулю, это означает, что процесс желает дождаться момента, когда значение семафора достигнет нуля.

Если значение семафора уже равно нулю, функция сразу же вернет управление.

Если значение семафора больше нуля, тогда вступают в силу следующие условия:

а) Если указан флаг IPC_NOWAIT, функция semop возвращает управление с кодом ошибки EAGAIN.

б) Если флаг IPC_NOWAIT не указан, то для данного семафора увеличивается значение semzcnt и выполнение вызывающего процесса приостанавливается до тех пор, пока не будет соблюдено одно из следующих условий:

- Значение семафора стало равным нулю. В этом случае значение semzcnt уменьшается (поскольку ожидание освобождения семафора можно считать законченным).
- Семафор был удален из системы. В этом случае функция semop вернет признак ошибки с кодом EIDRM.

- Процессом был перехвачен сигнал, и обработчик сигнала вернул управление. В этом случае значение semzcnt уменьшается (поскольку вызывающий процесс прекращает ожидание), и функция semop вернет признак ошибки с кодом EINTR.

Функция semop выполняет все операции атомарно – будут выполнены либо все запрошенные действия, либо ни одно из них.

Корректировка семафора по завершении

Как уже упоминалось ранее, завершение процесса в то время, когда он захватил какие-либо ресурсы посредством семафора, может стать достаточно серьезной проблемой. Всякий раз, когда мы устанавливаем для операции над семафором флаг SEM_UNDO (значение sem_op меньше нуля), ядро запоминает, как много ресурсов было захвачено процессом с помощью конкретного семафора (абсолютное значение sem_op). Когда процесс завершается, добровольно или принудительно, ядро проверяет, имеет ли процесс какие-либо невыполненные корректировки семафоров и, если таковые имеются, корректирует значения соответствующих семафоров.

Когда начальное значение семафора устанавливается функцией semctl с помощью команды SETVAL или SETALL, значение корректировки семафора сбрасывается в 0.

Пример – сравнение производительности семафоров и блокировки записей в файлах

При совместном использовании одного ресурса несколькими процессами порядок доступа к ресурсу может регулироваться с помощью семафора или блокировки записей в файле. Было бы интересно сравнить производительность этих двух методов.

В случае семафоров мы создали набор семафоров, в состав которого входит единственный семафор. Он инициализируется значением 1. Чтобы захватить ресурс, процесс должен вызвать semop со значением sem_op, равным –1. Чтобы освободить ресурс, процесс должен вызвать semop со значением sem_op, равным +1. Кроме того, для каждой операции мы указывали флаг SEM_UNDO на случай завершения процесса, который не успел освободить ресурс.

В случае с блокировками мы создали пустой файл и использовали его первый байт (который не обязательно должен существовать) для установки блокировки. Чтобы захватить ресурс, процесс должен установить блокировку для записи на этот байт, чтобы освободить ресурс – снять блокировку с байта. Одно из свойств блокировок заключается в том, что по завершении процесса, который удерживает блокировку, она будет автоматически снята ядром.

В табл. 15.7 показано время выполнения этих двух методов блокировок в Linux. В каждом случае три тестовых процесса захватывали и освобождали ресурс 100 000 раз. Цифры, приводимые в табл. 15.7, представляют собой общее время для всех трех процессов в секундах.

Таблица 15.7. Производительность двух альтернативных механизмов блокировки в Linux

Механизм IPC	Пользовательское время	Системное время	Общее время
Семафоры с флагом SEM_UNDO	0,38	0,48	0,86
Рекомендательная блокировка записи в файле	0,41	0,95	1,36

В ОС Linux проигрыш при использовании механизма блокировок записей в файлах составил почти 60 процентов по сравнению с семафорами.

Но даже несмотря на то, что механизм блокировок записей медленнее семафоров, в тех случаях, когда речь идет об одном ресурсе (таком как сегмент разделяемой памяти) и вам не нужны все причудливые особенности семафоров XSI, мы все-таки рекомендуем использовать блокировки записей. Причина состоит в том, что они намного проще в использовании и система сама заботится о блокировках, которые не были сняты по завершении процесса.

15.9. Разделяемая память

Механизм разделяемой памяти позволяет двум и более процессам совместно использовать одну и ту же область памяти. Это самый скоростной вид IPC, поскольку при его использовании данные не нужно лишний раз копировать между клиентом и сервером. Единственный сложный момент при работе с разделяемой памятью – это синхронизация доступа к ней. Если сервер размещает некоторые данные в области разделяемой памяти, клиент не должен пытаться читать данные до тех пор, пока сервер не выполнит всю работу. Очень часто для синхронизации используются семафоры. (Но, как мы видели в конце предыдущего раздела, блокировки записей в файлах также могут использоваться.)

> Стандарт Single UNIX Specification включает определение альтернативного набора функций для организации доступа к разделяемой памяти в расширениях реального времени. Но в этой книге мы не будем рассматривать расширения реального времени.

Каждому сегменту разделяемой памяти ядро ставит в соответствие структуру, которая содержит как минимум следующий набор полей:

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* раздел 15.6.2 */
    size_t shm_segsz;  /* размер сегмента в байтах */
    pid_t shm_lpid;    /* идентификатор процесса, последним вызвавшего shmop() */
    pid_t shm_cpid;      /* идентификатор процесса-создателя */
    shmatt_t shm_nattch; /* текущее количество подключений */
    time_t shm_atime;    /* время последнего подключения */
    time_t shm_dtime;    /* время последнего отключения */
    time_t shm_ctime;    /* время последнего изменения */
    .
    .
```

```
};
```

(Каждая реализация при необходимости может добавлять собственные поля в эту структуру.)

Тип `shmatt_t` определен как беззнаковое целое, по меньшей мере — `unsigned short`. В табл. 15.8 перечислены системные пределы (раздел 15.6.3), которые имеют отношение к разделяемой памяти.

Таблица 15.8. Системные пределы, имеющие отношение к разделяемой памяти

Описание	Типичные значения			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Максимальный размер сегмента разделяемой памяти в байтах	33 554 432	33 554 432	4 194 304	8 388 608
Минимальный размер сегмента разделяемой памяти в байтах	1	1	1	1
Максимальное количество сегментов разделяемой памяти в системе	192	4 096	32	100
Максимальное количество сегментов разделяемой памяти для процесса	128	4 096	8	6

Обычно при работе с разделяемой памятью прежде всего вызывается функция `shmget`, которая возвращает идентификатор сегмента разделяемой памяти.

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int flag);
```

Возвращает идентификатор сегмента разделяемой памяти в случае успеха, –1 в случае ошибки

В разделе 15.6.1 мы рассмотрели правила преобразования ключа в идентификатор и обсудили вопрос, когда создается новый сегмент, а когда открывается существующий. При создании нового сегмента инициализируются следующие поля структуры `shmid_ds`.

- Структура `ipc_perm` инициализируется, как описано в разделе 15.6.2. Поле `mode` устанавливается в соответствии со значениями битов прав доступа в аргументе *flag*. Значения для каждого конкретного права доступа приводятся в табл. 15.2.
- В поля `shm_lpid`, `shm_nattach`, `shm_atime` и `shm_dtime` записывается значение 0.
- В поле `shm_ctime` записывается значение текущего времени.
- В поле `shm_segsz` записывается значение аргумента *size*.

Аргумент *size* определяет размер сегмента разделяемой памяти в байтах. Обычно реализации округляют это число так, чтобы оно было кратно размеру страницы памяти в системе, но если приложение определяет в аргументе size число, не кратное размеру страницы памяти, то остаток последней страницы будет недоступен для использования. Если должен быть создан новый сегмент разделяемой памяти (обычно на стороне сервера), то его размер необходимо определить в аргументе *size*. Если нам нужно лишь получить ссылку на существующий сегмент (в случае клиента), то мы можем передать в аргументе *size* значение 0. Когда создается новый сегмент, его содержимое очищается.

Функция shmctl выполняет различные операции над сегментом разделяемой памяти.

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf );
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Аргумент *cmd* представляет собой код операции, которая должна быть выполнена над сегментом, определяемым аргументом *shmid*.

IPC_STAT Получить структуру shmid_ds для данного сегмента памяти и сохранить ее по адресу *buf*.

IPC_SET Скопировать значения полей shm_perm.uid, shm_perm.gid и shm_perm.mode из *buf* в структуру shmid_ds, связанную с сегментом разделяемой памяти. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением shm_ perm.cuid или shm_perm.uid или если процесс обладает привилегиями суперпользователя.

IPC_RMID Удалить сегмент разделяемой памяти. Поскольку для сегментов разделяемой памяти поддерживается счетчик ссылок (поле shm_nattach в структуре shmod_ds), сегмент не будет удален до тех пор, пока последний использующий его процесс не завершится или не отсоединит этот сегмент. Независимо от того, находится ли сегмент в использовании, его идентификатор немедленно удаляется из системы, что предотвращает возможность новых подключений сегмента вызовом функции shmat. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением shm_perm.cuid или shm_perm.uid или если процесс обладает привилегиями суперпользователя.

ОС Linux и Solaris предоставляют две дополнительные команды, которые не являются частью стандарта Single UNIX Specification.

SHM_LOCK Заблокировать сегмент разделяемой памяти. Эта команда может быть выполнена, только если процесс обладает привилегиями суперпользователя.

SHM_UNLOCK Разблокировать сегмент разделяемой памяти. Эта команда может быть выполнена, только если процесс обладает привилегиями суперпользователя.

После создания сегмента разделяемой памяти процесс может присоединить его к своему адресному пространству с помощью функции shmat.

```
#include <sys/shm.h>

void *shmat(int shmid, const void *addr, int flag);
```

Возвращает указатель на сегмент разделяемой памяти в случае успеха, –1 в случае ошибки

Адрес, начиная с которого будет присоединен сегмент разделяемой памяти, зависит от значения аргумента *addr* и наличия флага SHM_RND в аргументе *flag*.

- Если в аргументе *addr* передано значение 0, сегмент будет присоединен к первому доступному адресу, который выберет ядро. Это рекомендуемая методика.
- Если в аргументе *addr* передано ненулевое значение и флаг SHM_RND не указан, сегмент присоединяется, начиная с адреса *addr*.
- Если в аргументе *addr* передано ненулевое значение и указан флаг SHM_RND, сегмент будет присоединен с адреса, который вычисляется по формуле: (addr – (addr mod SHMLBA)). Имя константы SHM_RND происходит от слова «round» (округлить), а имя константы SHMLBA, величина которой всегда представлена степенью числа 2, – от «low boundary address multiple» (множитель адреса нижней границы). Приведенная выше формула округляет адрес вниз до ближайшего кратного числу SHMLBA.

Если мы не планируем, что приложение будет работать на единственной аппаратной платформе (что в наши дни весьма маловероятно), мы не должны указывать адрес присоединения сегмента разделяемой памяти. Вместо этого следует передавать в аргументе *addr* значение 0, позволяя системе самой выбрать адрес.

Если в аргументе *flag* указан флаг SHM_RDONLY, присоединенный сегмент будет доступен только для чтения. В противном случае присоединенный сегмент доступен для чтения и записи.

Значение, возвращаемое функцией shmat, представляет собой адрес, начиная с которого был присоединен сегмент разделяемой памяти. В случае ошибки возвращается значение –1. Если вызов shmat завершился успехом, ядро увеличит счетчик shm_nattch в структуре shmid_ds, связанной с данным сегментом.

По окончании работы с сегментом разделяемой памяти следует вызывать функцию shmdt для его отсоединения. Обратите внимание: эта функция не удаляет из системы идентификатор и структуры данных, ассоциированные с сегментом памяти. Идентификатор продолжает существовать до тех пор, пока какой-либо процесс (зачастую сервер) специально не удалит его вызовом функции shmctl с командой IPC_RMID.

```
#include <sys/shm.h>

int shmdt(void *addr);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

В аргументе *addr* передается значение, полученное от функции `shmat`. В случае успеха `shmdt` уменьшает значение счетчика `shm_nattch` в структуре `shmid_ds`.

Пример

Адрес, к которому будет подключен сегмент разделяемой памяти, когда в аргументе `addr` передается значение 0, в значительной степени зависит от операционной системы. Листинг 15.11 содержит текст программы, которая выводит сведения о том, где размещаются различного рода данные в конкретной системе.

Листинг 15.11. Вывод сведений о размещении различного рода данных

```
#include "apue.h"
#include <sys/shm.h>

#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE 0600                   /* чтение и запись для владельца */

char array[ARRAY_SIZE];                 /* неинициализированные данные = bss */

int
main(void)
{
    int shmid;
    char *ptr, *shmptr;

    printf("array[] от %lx до %lx\n", (unsigned long)&array[0],
           (unsigned long)&array[ARRAY_SIZE]);
    printf("стек примерно %lx\n", (unsigned long)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("ошибка вызова функции malloc");
    printf("динамически выделенная область от %lx до %lx\n",
           (unsigned long)ptr, (unsigned long)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("ошибка вызова функции shmget");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("ошибка вызова функции shmat");
    printf("сегмент разделяемой памяти присоединен в адресах от %lx до %lx\n",
           (unsigned long)shmptr, (unsigned long)shmptr+SHM_SIZE);
    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("ошибка вызова функции shmctl");
    exit(0);
}
```

Запуск этой программы в ОС Linux на платформе Intel дал следующие результаты:

```
$ ./a.out
array[] от 804a080 до 8053cc0
стек примерно bffff9e4
динамически выделенная область от 8053cc8 до 806c368
сегмент разделяемой памяти присоединен в адресах от 40162000 до 4017a6a0
```

На рис. 15.13 показана раскладка памяти, соответствующая полученным результатам. Обратите внимание: сегмент разделяемой памяти присоединен в адресах, расположенных значительно ниже стека.

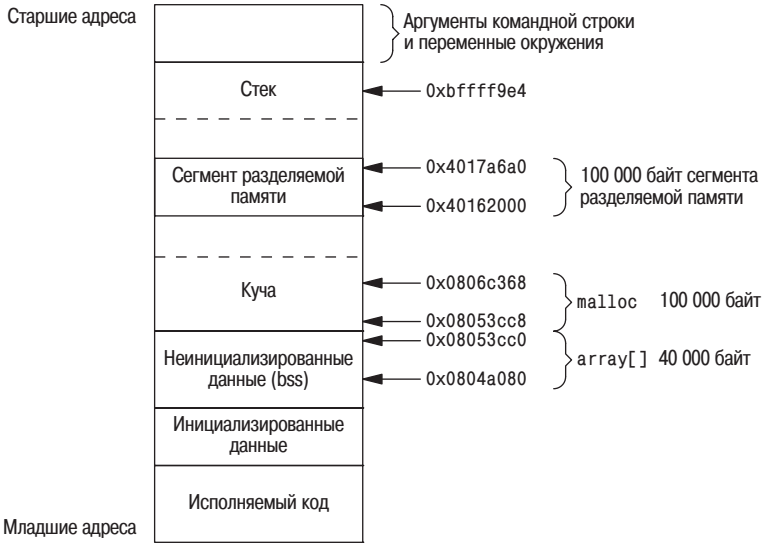


***Рис. 15.13.** Раскладка памяти в ОС Linux на платформе Intel*

В разделе 14.9 мы говорили о том, что с помощью функции `mmap` можно отобразить определенный участок файла в адресное пространство процесса. Концептуально это очень похоже на присоединение сегмента разделяемой памяти с помощью функции `shmat` XSI IPC. Главное отличие состоит в том, что сегмент памяти, отображенный с помощью функции `mmap`, связан с файлом, тогда как сегмент разделяемой памяти XSI вообще никак не связан с файлами.

Пример – отображение в память файла /dev/zero

Разделяемая память может использоваться для организации взаимодействия между процессами, которые не связаны родственными отношениями. Но если процессы взаимосвязаны, то некоторые реализации предоставляют иную методику.

> Следующий прием работает в ОС FreeBSD 5.2.1, Linux 2.4.22 и Solaris 9. В Mac OS X 10.3 в настоящее время отображение символьных устройств в память процесса не поддерживается.

Устройство /dev/zero при чтении из него служит неиссякаемым источником нулевых байтов. Оно также может принимать любые объемы данных, совершенно игнорируя их. Это устройство представляет для нас интерес из-за особых свойств, которые оно проявляет при отображении в память.

- Создается неименованная область памяти, размер которой передается функции mmap во втором аргументе. Это число округляется до ближайшего целого, кратного размеру страницы.
- Область памяти инициализируется нулями.
- Эта область может совместно использоваться несколькими процессами, если их общий предок передал функции mmap флаг MAP_SHARED.

Пример работы с этим устройством приводится в листинге 15.12.

Листинг 15.12. Взаимодействие между родительским и дочерним процессами с использованием операций ввода-вывода над устройством /dev/zero, отображенным в память

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define NLOOPS 1000
#define SIZE sizeof(long) /* размер сегмента разделяемой памяти */

static int
update(long *ptr)
{
    return((*ptr)++);     /* вернуть значение до увеличения */
}

int
main(void)
{
    int fd, i, counter;
    pid_t pid;
    void *area;

    if ((fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("ошибка вызова функции open");
    if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, 0)) == MAP_FAILED)
        err_sys("ошибка вызова функции mmap");
    close(fd); /* теперь, после отображения, /dev/zero можно закрыть */

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) { /* родительский процесс */
        for (i = 0; i < NLOOPS; i += 2) {
            if ((counter = update((long *)area)) != i)
                err_quit("предок: ожидалось %d, получено %d", i, counter);

            TELL_CHILD(pid);
```

```
            WAIT_CHILD();
        }
    } else { /* дочерний процесс */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();

            if ((counter = update((long *)area)) != i)
                err_quit("потомок: ожидалось %d, получено %d", i, counter);

            TELL_PARENT(getppid());
        }
    }

    exit(0);
}
```

Эта программа открывает устройство `/dev/zero` и вызывает функцию `mmap`, указывая ей размер отображаемой области. Обратите внимание: когда участок этого специального файла отображен, мы можем закрыть его. После этого создается дочерний процесс. Поскольку при отображении был указан флаг `MAP_SHARED`, данные, которые запишет в эту область один процесс, сможет прочитать другой. (Если бы при отображении мы указали флаг `MAP_PRIVATE`, то этот пример не работал бы.)

Затем родительский и дочерний процессы поочередно начинают увеличивать число, находящееся в разделяемой области отображенной памяти, используя для синхронизации функции из раздела 8.9. Число, находящееся в разделяемой памяти, инициализируется значением 0. Родительский процесс увеличивает его до значения 1, затем дочерний процесс увеличивает его до 2, потом родительский процесс увеличивает его до 3 и т. д. Обратите внимание, что в функции `update` используются круглые скобки, потому что нам нужно увеличить число в памяти, а не сам указатель.

Основное преимущество такого подхода заключается в том, что отпадает необходимость в существовании файла перед созданием отображенной области вызовом `mmap`. Отображение устройства `/dev/zero` автоматически создает область отображенной памяти заданного размера. Недостаток же состоит в том, что такой прием работает только с процессами, которые связаны родственными отношениями. Однако для родственных процессов, вероятно, более простым и эффективным решением было бы использование потоков (главы 11 и 12). Обратите внимание: независимо от выбранной методики, все равно необходимо синхронизировать доступ к разделяемым данным.

Пример – анонимные области отображаемой памяти

Большинство реализаций предоставляют возможность создавать анонимные области отображаемой памяти – примерно так же, как это делается при отображении устройства `/dev/zero`. Чтобы воспользоваться этой возможностью, нужно передать функции `mmap` флаг `MAP_ANON` и число –1 вместо дескриптора файла. В результате мы получим анонимную (поскольку она не связана

с именем какого-либо файла) область памяти, которая может совместно использоваться родственными процессами.

> Возможность создания анонимных областей отображенной памяти имеется во всех четырех платформах, обсуждаемых в этой книге. Обратите внимание на то, что ОС Linux определяет флаг, поддерживающий эту возможность, как MAP_ANONYMOUS, но при этом также определяет и флаг MAP_ANON с тем же самым значением для сохранения совместимости.

Чтобы программа из листинга 15.12 использовала эту возможность, в нее нужно внести три изменения: (а) убрать операцию открытия устройства /dev/zero, (б) убрать операцию закрытия дескриптора и (в) изменить обращение к функции mmap следующим образом:

```
if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                 MAP_ANON | MAP_SHARED, -1, 0)) == MAP_FAILED)
```

В этом вызове мы указали флаг MAP_ANON и передали значение -1 вместо дескриптора файла. Остальная часть программы из листинга 15.12 остается без изменений.

Последние два примера демонстрируют совместное использование области памяти двумя родственными процессами. Если необходимо использовать разделяемую память для организации взаимодействия между процессами, которые не связаны родственными отношениями, мы можем выбрать один из двух вариантов. Приложения могут использовать функции XSI, предназначенные для работы с разделяемой памятью, или функцию mmap с флагом MAP_SHARED для отображения одного и того же файла в собственные адресные пространства.

15.10. Свойства взаимодействий типа клиент-сервер

Рассмотрим подробнее некоторые свойства клиентов и серверов, которые имеют отношение к различным механизмам IPC, используемым для организации взаимодействия между ними. Самый простой тип взаимоотношений – когда клиент с помощью функций fork и exec запускает требуемый сервер. В этом случае перед вызовом функции fork могут быть созданы два полудуплексных канала, чтобы организовать движение данных в обе стороны. На рис. 15.8 показан пример такой организации взаимодействий. Запускаемый сервер может быть программой с установленным битом set-user-ID, что дает ему специальные привилегии. Кроме того, сервер может идентифицировать клиента, получив свой собственный реальный идентификатор пользователя. (В разделе 8.10 мы говорили, что реальные идентификаторы пользователя и группы не изменяются при запуске новой программы с помощью функции exec.)

На основе этой схемы мы можем разработать *сервер открытия файлов*. (Реализация его будет показана в разделе 17.5.) Он будет открывать файлы по запросу клиента. Таким образом, мы можем добавить проверку дополни-

тельных прав доступа, кроме обычных для UNIX прав пользователь/группа/остальные. Этот подход предполагает, что программа-сервер должна иметь установленный бит set-user-ID, чтобы получить дополнительные привилегии (возможно, привилегии суперпользователя). Сервер на основе реального идентификатора пользователя клиента определяет, разрешен ли ему доступ к запрошенному файлу. Благодаря этому мы можем создать сервер, который предоставляет определенным пользователям дополнительные привилегии, которых они обычно не имеют.

В этом примере, поскольку сервер является дочерним процессом по отношению к клиенту, он может передать родительскому процессу только содержимое файла. Хотя такой подход вполне применим к обычным файлам, он не может быть использован, например, для специальных файлов устройств. Было бы лучше, если бы сервер открывал требуемый файл и передавал клиенту дескриптор этого файла. Родительский процесс может передать дескриптор потомку, но передать дескриптор в обратном направлении, от дочернего процесса родительскому, невозможно (если не использовать специальные приемы, о которых мы расскажем в главе 17).

Следующий тип сервера был показан на рис. 15.12. Этот сервер представляет собой процесс-демон, который взаимодействует со всеми клиентами посредством некоторого механизма IPC. Для такого рода взаимодействий между клиентами и сервером нельзя использовать неименованные каналы. Здесь требуется именованная форма IPC – например, именованные каналы (FIFO) или очереди сообщений. В случае именованных каналов, как мы уже видели, необходимо создавать отдельный именованный канал для связи с каждым из клиентов, если предполагается передача данных клиенту от сервера. Если же данные передаются только от клиента, то достаточно будет создать единственный именованный канал с предопределенным именем. (Такую форму взаимодействия использует демон печати в System V. В этом случае в роли клиента выступает команда `lp`(1), а сервер представлен демоном `lpsched`. Данные в этой схеме передаются только от клиента к серверу, обратная связь полностью отсутствует.)

При использовании очередей сообщений мы получаем дополнительные возможности.

1. Для взаимодействия сервера со всеми клиентами достаточно одной очереди. Поле `type` в сообщении может служить для идентификации получателя сообщения. Например, клиенты могут отправлять запросы, указывая в поле `type` число 1. При этом каждый клиент должен включать в сообщение идентификатор своего процесса. В результате сервер может принимать только сообщения, в которых поле `type` имеет значение 1 (четвертый аргумент функции `msgrcv`), а клиенты – принимать только сообщения, в которых значение поля `type` совпадает с идентификаторами их процессов.
2. Для каждого из клиентов также может быть создана отдельная очередь сообщений. Перед отправкой первого запроса клиент создает собственную очередь сообщений с ключом `IPC_PRIVATE`. Сервер также должен создать очередь с ключом или идентификатором, которые известны клиен-

там. Первый запрос клиент передает через предопределенную очередь сообщений, отсылая серверу идентификатор своей очереди, а весь последующий обмен данными уже будет происходить через отдельную очередь, созданную клиентом. Свой первый и все последующие отклики сервер передает через очередь сообщений клиента.

Один из недостатков такого подхода заключается в том, что каждая клиентская очередь может содержать всего одно сообщение – либо запрос клиента, либо ответ сервера. Это выглядит слишком расточительно из-за ограничений на количество очередей в системе, поэтому вместо отдельных очередей лучше использовать именованные каналы. Другая проблема состоит в том, что сервер вынужден получать сообщения из нескольких очередей сразу, но ни `select`, ни `poll` не могут работать с очередями сообщений.

Любая из этих двух методик, основанных на очередях сообщений, может быть реализована на базе разделяемой памяти с применением методов синхронизации (семафоры или блокировка записей в файле).

Проблема с таким видом взаимодействий клиента и сервера (когда они не связаны родственными отношениями) состоит в том, что сервер должен точно идентифицировать клиента. Если сервер выполняет привилегированные операции, он должен точно знать, кто является клиентом. Это совершенно необходимо, если сервер, например, представляет собой программу с установленным битом set-user-ID. Хотя все эти формы IPC проходят через ядро, оно не предоставляет никаких средств идентификации отправителя.

В случае очередей сообщений, когда для передачи данных между сервером и клиентом используется единственная очередь, в которой может одновременно находиться только одно сообщение, поле `msg_lspid` будет содержать идентификатор процесса отправителя. Но это не совсем то, что нам нужно, желательно было бы иметь эффективный идентификатор пользователя заданного процесса. К сожалению, переносимого способа получения эффективного идентификатора пользователя по идентификатору процесса не существует. (Естественно, ядро хранит оба эти значения в таблице процессов, но, обладая одним, мы не можем получить другой без прямого поиска в памяти ядра.)

В разделе 17.3 мы будем применять следующую методику идентификации клиента на стороне сервера. Этот прием также может использоваться при работе с именованными каналами, очередями сообщений, семафорами или разделяемой памятью. Предположим, что для организации взаимодействий, схема которых представлена на рис. 15.12, используются именованные каналы. Клиент должен создать свой собственный канал FIFO и установить права доступа к нему таким образом, чтобы он был доступен на чтение и на запись только владельцу. Здесь мы исходим из предположения, что сервер обладает привилегиями суперпользователя (в противном случае нет большого смысла беспокоиться по поводу идентификации клиента), таким образом, сервер может выполнять операции чтения и записи с данным каналом. Когда по предопределенному каналу FIFO поступает первый запрос от клиента (который должен содержать идентификатор канала клиента), сервер вызывает функцию `stat` или `fstat` для канала клиента. Предполагается, что эф-

фективный идентификатор пользователя клиента – это идентификатор владельца FIFO (поле st_uid структуры stat). Сервер должен убедиться в том, что доступ к каналу разрешен только для его владельца. Дополнительно сервер должен проверить, имеют ли три поля времени, связанные с FIFO (поля st_atime, st_mtime и st_ctime структуры stat), допустимые значения (например, не более 15 или 30 секунд). Если злоумышленник сможет создать канал FIFO с другим эффективным идентификатором и установить право только на чтение и на запись для владельца, значит, система имеет весьма серьезные проблемы с безопасностью.

Чтобы реализовать эту методику для XSI IPC, вспомните, что с каждой очередью сообщений, семафором и сегментом разделяемой памяти ассоциируется структура ipc_perm, которая идентифицирует создателя объекта IPC (поля cuid и cgid). Как и в случае FIFO, сервер должен требовать от клиента, чтобы создаваемая им структура IPC имела права доступа только для владельца. Кроме того, сервер должен убедиться в том, что все характеристики времени имеют надлежащие значения (поскольку эти структуры IPC могут существовать в системе до тех пор, пока явно не будут удалены).

В разделе 17.2.2 мы увидим, что существует более надежный способ идентификации, когда эффективные идентификаторы пользователя и группы клиента предоставляются ядром. Сделать это можно с помощью подсистемы STREAMS, передавая дескрипторы файлов между процессами.

15.11. Подведение итогов

Мы рассмотрели разнообразные формы взаимодействий между процессами: именованные и неименованные каналы и три формы IPC, которые обычно называют XSI IPC (очереди сообщений, семафоры и разделяемую память). Семафоры в действительности представляют собой механизм синхронизации, а не обмена данными, и часто используются для синхронизации доступа к разделяемым ресурсам, таким как сегменты разделяемой памяти. При обсуждении неименованных каналов мы рассмотрели реализацию функции popen, понятие сопроцессов и возможные ловушки, связанные с режимом буферизации в стандартной библиотеке ввода-вывода.

После сравнения производительности очередей сообщений с дуплексными каналами и семафоров с механизмом блокировки записей в файлах мы можем дать следующие рекомендации. Изучайте именованные и неименованные каналы, поскольку эти два механизма по-прежнему остаются эффективным средством организации обмена данными для большинства приложений. Избегайте использования очередей сообщений и семафоров в новых приложениях. Вместо них следует применять дуплексные каналы и блокировки записей в файлах, так как они намного проще. Разделяемая память может найти применение, хотя те же самые возможности предоставляются функцией mmap (раздел 14.9).

В следующей главе мы рассмотрим механизмы сетевых взаимодействий, которые помогают организовать обмен информацией между разными машинами.

Упражнения

15.1. В программе из листинга 15.2 удалите вызов функции `close` перед вызовом `waitpid` в конце кода родителя. Объясните, что произойдет.

15.2. В программе из листинга 15.2 удалите обращение к функции `waitpid` в конце кода родителя. Объясните, что произойдет.

15.3. Что произойдет, если функции `popen` передать имя несуществующей команды? Напишите небольшую программу, чтобы проверить эту ситуацию.

15.4. В программе из листинга 15.9 удалите обработчик сигнала, запустите программу и завершите дочерний процесс. Каким образом можно убедиться, что родительский процесс завершился при получении сигнала `SIGPIPE` после ввода строки?

15.5. Попробуйте в программе из листинга 15.9 использовать для работы с неименованными каналами вместо функций `read` и `write` функции чтения и записи из стандартной библиотеки ввода-вывода.

15.6. В пояснениях к стандарту POSIX.1 в качестве одной из причин появления функции `waitpid` приводится описание ситуации, которая не может быть обработана без этой функции:

```
if ((fp = popen("/bin/true", "r")) == NULL)
    ...
if ((rc = system("sleep 100")) == -1)
    ...
if (pclose(fp) == -1)
    ...
```

Что получится в результате выполнения этого кода, если вместо функции `waitpid` использовать функцию `wait`?

15.7. Объясните, как функции `select` и `poll` обрабатывают ситуацию закрытия неименованного канала пишущим процессом. Чтобы ответить на этот вопрос, напишите две небольшие программы: одну с использованием функции `select`, другую с использованием функции `poll`.

15.8. Что произойдет, если команда *`cmdstring`*, запущенная функцией `popen` со значением `"r"` в аргументе *`type`*, попытается вывести что-нибудь на стандартный вывод сообщений об ошибках?

15.9. Для выполнения команды из аргумента *`cmdstring`* функция `popen` вызывает командный интерпретатор. Что происходит по завершении *`cmdstring`*? (Подсказка: нарисуйте схему происходящего.)

15.10. Стандарт POSIX.1 особо отмечает, что возможность открытия канала FIFO с помощью функции `open` одновременно для чтения и записи не предусмотрена, хотя большинство версий UNIX допускают это. Продемонстрируйте другой метод открытия FIFO для чтения и записи без использования блокировок.

15.11. Если файл не содержит секретной информации, то его доступность на чтение для всех пользователей не несет никакого вреда. (Хотя обычно попытки совать нос в чужие файлы не одобряются.) Но что может произойти, если злонамеренный процесс получит доступ на чтение к очереди сообщений, которая используется для взаимодействия сервера и нескольких клиентов? Какой информацией должен обладать злонамеренный процесс, чтобы прочитать содержимое очереди сообщений?

15.12. Напишите программу, которая выполняет следующие действия: пять раз в цикле создает очередь сообщений, выводит идентификатор очереди, удаляет очередь сообщений; затем в другом цикле пять раз создает очередь сообщений с ключом `IPC_PRIVATE` и размещает в очереди одно сообщение. После завершения программы просмотрите очереди сообщений с помощью команды `ipcs`(1). Объясните, что происходит с идентификаторами очередей.

15.13. Опишите, как можно создать связанный список объектов данных в сегменте разделяемой памяти. Что следует хранить в качестве указателей в списке?

15.14. Нарисуйте временную диаграмму работы программы из листинга 15.12, показывающую значение переменной `i` в родительском и дочернем процессах, значения числа в разделяемой памяти и возвращаемые значения функции `update`. Исходите из предположения, что после вызова функции `fork` первым получает управление дочерний процесс.

15.15. Перепишите программу из листинга 15.12 таким образом, чтобы она вместо отображаемой памяти использовала функции для работы с разделяемой памятью XSI из раздела 15.9.

15.16. Перепишите программу из листинга 15.12 таким образом, чтобы она использовала семафоры для синхронизации родительского и дочернего процессов.

15.17. Перепишите программу из листинга 15.12 таким образом, чтобы она использовала механизм блокировки записей в файле для синхронизации родительского и дочернего процессов.

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru - Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-089-8, название «UNIX. Профессиональное программирование», 2-е издание – покупка в Интернет-магазине «Books.Ru - Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

16

Межпроцессное взаимодействие в сети: сокеты

16.1. Введение

В предыдущей главе мы рассмотрели именованные и неименованные каналы, очереди сообщений, семафоры и разделяемую память – классические механизмы межпроцессного взаимодействия, предоставляемые различными версиями UNIX. Эти механизмы позволяют организовать взаимодействие между процессами, работающими на одной машине. В этой главе мы рассмотрим сетевые механизмы IPC, которые позволяют процессам, выполняющимся на разных машинах (объединенных в общую сеть), взаимодействовать друг с другом.

В этой главе будет описан интерфейс сетевых сокетов, который может использоваться для организации взаимодействий между процессами независимо от того, где они работают – на одной машине или на разных. Это было одной из основных целей при разработке интерфейса сокетов: один и тот же набор функций должен был использоваться как для внутримашинного, так и для межмашинного обмена данными. Несмотря на то, что интерфейс сокетов может использоваться для работы по многим сетевым протоколам, в этой главе мы ограничимся обсуждением только протоколов TCP/IP, поскольку де-факто они стали стандартом для взаимодействий через Интернет.

Как указывает стандарт POSIX.1, прикладной программный интерфейс сокетов основан на интерфейсе сокетов 4.4BSD. Хотя за прошедшие годы и были внесены некоторые изменения, тем не менее современный интерфейс весьма напоминает тот, что впервые появился в начале 80-х годов в 4.2BSD.

Эта глава представляет собой лишь краткий обзор прикладного программного интерфейса сокетов. Детальное обсуждение сокетов вы найдете в книге, посвященной сетевому программированию в UNIX [Stevens, Fenner, and Rudoff 2004].

16.2. Дескрипторы сокетов

Сокет – это абстракция конечной точки взаимодействия. Подобно тому как для работы с файлами приложения используют дескрипторы файлов, для работы с сокетами они используют дескрипторы сокетов. В UNIX дескрипторы сокетов реализованы так же, как дескрипторы файлов. В действительности большинство функций, работающих с дескрипторами файлов, таких как `read` или `write`, будут работать и с дескрипторами сокетов.

Создается дескриптор сокета с помощью функции `socket`.

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Возвращает дескриптор файла (сокета) в случае успеха, –1 в случае ошибки

Аргумент *domain* определяет природу взаимодействия, включая формат адреса (более подробно он будет описан в следующем разделе). В табл. 16.1 приводится список доменов, которые определены стандартом POSIX.1. Имена констант начинаются с префикса `AF_` (от *address family – семейство адресов*), потому что каждый домен обладает своим собственным форматом представления адресов.

Таблица 16.1. Домены сокетов

Домен	Описание	Домен	Описание
AF_INET	Домен Интернета IPv4	AF_UNIX	Домен UNIX
AF_INET6	Домен Интернета IPv6	AF_UNSPEC	Неопределенный домен

Домен UNIX будет обсуждаться в разделе 17.3. Большинство систем определяют дополнительный домен `AF_LOCAL`, который представляет собой псевдоним домена `AF_UNIX`. Константа `AF_UNSPEC` обозначает неопределенный домен, который может представлять любой домен. Некоторые платформы традиционно реализуют поддержку дополнительных сетевых протоколов, таких как `AF_IPX` для семейства проколов NetWare, но стандарт POSIX.1 не определяет константы доменов для этих протоколов.

В аргументе *type* указывается тип сокета, который в свою очередь определяет характеристики взаимодействия. Типы сокетов, определенные стандартом POSIX.1, перечислены в табл. 16.2, но реализации могут добавлять поддержку дополнительных типов.

Через аргумент *protocol* обычно передается значение 0, чтобы выбрать протокол по умолчанию для данного домена и типа сокета. Если для одного и того же домена и типа сокета поддерживается несколько протоколов, можно использовать этот аргумент для выбора конкретного протокола. Протокол по умолчанию для сокетов типа `SOCK_STREAM` из домена `AF_INET` – TCP (Transmission Control Protocol – протокол управления передачей данных). Прото-

кол по умолчанию для сокетов типа SOCK_DGRAM из домена AF_INET – UDP (User Datagram Protocol – протокол пользовательских дейтаграмм).

Таблица 16.2. Типы сокетов

Тип	Описание
SOCK_DGRAM	Не ориентированы на создание логического соединения, сообщения фиксированной длины, доставка сообщений не гарантируется
SOCK_RAW	Интерфейс дейтаграмм к протоколу IP (необязателен в POSIX.1)
SOCK_SEQPACKET	Ориентированы на создание логического соединения, упорядоченность передачи данных, сообщения фиксированной длины, гарантируется доставка сообщений
SOCK_STREAM	Ориентированы на создание логического соединения, упорядоченность передачи данных, гарантируется доставка сообщений, двунаправленный поток байтов

При использовании интерфейса дейтаграмм (SOCK_DGRAM) не требуется устанавливать логическое соединение, чтобы обмениваться данными между конечными точками взаимодействия. Все, что нужно сделать – это передать сообщение по адресу сокета, который используется процессом на другом конце.

Таким образом, дейтаграммы представляют собой службу, не ориентированную на установление логического соединения. Потоки байтов (SOCK_STREAM), с другой стороны, требуют, чтобы перед началом обмена данными между нашим сокетом и сокетом, принадлежащим сетевому узлу, с которым предполагается взаимодействовать, было установлено логическое соединение.

Дейтаграмма представляет собой самостоятельное сообщение. Передача дейтаграммы напоминает отправку письма по почте. Можно отправить множество писем, но нельзя гарантировать, что они будут доставлены в определенном порядке и что некоторые из них не потеряются по дороге. Каждое письмо содержит адрес получателя, благодаря чему оно не зависит от других писем. Письма даже могут быть отправлены разным получателям.

Напротив, протоколы, ориентированные на создание логического соединения, напоминают телефонный звонок. Прежде всего необходимо установить соединение, набрав номер телефона, и после того, как соединение будет установлено, вы сможете общаться с удаленным абонентом. Такого рода соединение, через которое вы имеете возможность общаться, является соединением типа «точка-точка». Ваши слова не содержат адресной информации, так как подключение этого типа логически связывает оба конца коммуникационного канала и само по себе подразумевает однозначную идентификацию отправителя и получателя.

При использовании сокетов типа SOCK_STREAM приложения не распознают границ отдельных сообщений, поскольку сокеты такого типа реализуют услугу передачи потока байтов. Это означает, что операция чтения данных из сокета может вернуть не то количество байт, которое было записано передающим процессом. В конечном счете будет получено все, что было отправлено, но для этого может потребоваться несколько обращений к функциям.

Сокеты типа SOCK_SEQPACKET очень похожи на сокеты типа SOCK_STREAM, за исключением того, что вместо услуги приема/передачи данных в виде потока байтов они реализуют услугу передачи отдельных сообщений. Это означает, что объем данных, полученных из сокета типа SOCK_SEQPACKET, всякий раз в точности совпадает с объемом отправленных данных. Служба передачи последовательности пакетов в домене Интернет реализуется на базе протокола SCTP (Stream Control Transmission Protocol – протокол передачи с управлением потоком).

Сокеты типа SOCK_RAW представляют собой интерфейс дейтаграмм на сетевом уровне (то есть интерфейс к протоколу IP в домене Интернет). При использовании этого интерфейса вся ответственность за построение заголовков пакетов возлагается на приложения, поскольку сокеты этого типа не используют протоколы транспортного уровня (такие как TCP или UDP). Чтобы предотвратить использование сокетов типа SOCK_RAW в неблаговидных целях, для их создания приложение должно обладать привилегиями суперпользователя.

Вызов функции socket напоминает вызов функции open. В обоих случаях мы получаем дескриптор файла, который затем используется в операциях ввода-вывода. По окончании работы с сокетом вызывается функция close, которая закрывает соединение и освобождает номер дескриптора для повторного использования.

Хотя дескриптор сокета и является файловым дескриптором, тем не менее его можно использовать не во всех функциях, которые принимают дескриптор файла в качестве аргумента. В табл. 16.3 приводится перечень большинства описанных нами функций, которые работают с файловыми дескрипторами, и дается описание их поведения при обслуживании дескрипторов сокетов. Если в ячейке таблицы указано «не определено» или «зависит от реализации», это означает, что, как правило, данная функция не может работать с дескрипторами сокетов. Например, функция lseek не может работать с сокетами, поскольку сокеты не поддерживают понятие текущей позиции файла.

Таблица 16.3. Поведение некоторых функций при работе с сокетами

Функция	Поведение при работе с сокетами
close (раздел 3.3)	Освобождает сокет
dup, dup2 (раздел 3.12)	Создают дубликат дескриптора
fchdir (раздел 4.22)	Завершается с кодом ошибки ENOTDIR а переменной errno
fchmod (раздел 4.9)	Не определено
fchown (раздел 4.11)	Зависит от реализации
fcntl (раздел 3.14)	Поддерживает некоторые команды, включая F_DUPFD, F_GETFD, F_GETFL, F_GETOWN, F_SETFD, F_SETFL и F_SETOWN
fdatasync, fsync (раздел 3.13)	Зависит от реализации
fstat (раздел 4.2)	Поддерживает некоторые поля структуры stat, но правила поддержки определяются реализацией

Таблица 16.3 (продолжение)

Функция	Поведение при работе с сокетами
`ftruncate` (раздел 4.13)	Не определено
`getmsg`, `getpmsg` (раздел 14.4)	Работает с сокетами, реализованными на базе STREAMS (т. е. в Solaris)
`ioctl` (раздел 3.15)	Выполняет ограниченный набор команд, который зависит от реализации драйвера устройства
`lseek` (раздел 3.6)	Зависит от реализации (обычно завершается с кодом ошибки `ESPIPE`)
`mmap` (раздел 14.9)	Не определено
`poll` (раздел 14.5.2)	Работает так, как и следует ожидать
`putmsg`, `putpmsg` (раздел 14.4)	Работает с сокетами, реализованными на базе STREAMS (т. е. в Solaris)
`read` (раздел 3.7) и `readv` (раздел 14.7)	Эквивалентны вызову функции `recv` (раздел 16.5) без каких-либо флагов
`select` (раздел 14.5.1)	Работает так, как и следует ожидать
`write` (раздел 3.8) и `writev` (раздел 14.7)	Эквивалентны вызову функции `send` (раздел 16.5) без каких-либо флагов

Обмен данными через сокеты является двунаправленным. Выполнение отдельных операций над сокетами можно запретить с помощью функции `shutdown`.

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Если в аргументе *how* передается значение `SHUT_RD`, то будет запрещена операция чтения из сокета. Если в аргументе *how* передается значение `SHUT_WR`, то будет запрещена операция записи в сокет. Если через аргумент *how* передать значение `SHUT_RDWR`, то будет запрещена возможность передачи данных в обоих направлениях.

Зачем же нужна функция `shutdown`, если функция `close` умеет работать с сокетами? На то есть несколько причин. Во-первых, функция `close` закрывает соединение и освобождает дескриптор только тогда, когда будет закрыта последняя активная ссылка на сокет. Это означает, что если мы создали дубликат дескриптора сокета (например, с помощью функции `dup`), то функция `close` не сможет закрыть сокет до тех пор, пока не будет закрыт последний файловый дескриптор, ссылающийся на него. Функция `shutdown` позволяет деактивировать сокет независимо от количества ссылающихся на него активных дескрипторов. Во-вторых, иногда возникает потребность запретить

передачу данных в одном из направлений. Например, мы можем запретить операцию записи, чтобы дать возможность процессу, с которым мы взаимодействуем, определить момент окончания передачи данных, но при этом мы хотели бы сохранить возможность приема данных, которые еще могут быть посланы удаленным процессом.

16.3. Адресация

В предыдущем разделе мы рассмотрели порядок создания и удаления сокетов. Прежде чем двинуться дальше, нам необходимо узнать, как производится идентификация процесса, с которым мы собираемся взаимодействовать. Идентификационная информация состоит из двух частей. Сетевой адрес компьютера позволяет идентифицировать сетевой узел, с которым мы предполагаем вступить в контакт, а номер службы помогает идентифицировать конкретный процесс на этом компьютере.

16.3.1. Порядок байтов

При организации взаимодействий между процессами, работающими на одной машине, мы обычно не задумываемся о порядке следования байтов. Порядок байтов – это характеристика аппаратной архитектуры процессора, определяющая, в каком порядке следуют байты в данных длинных типов, таких как целые числа. На рис. 16.1 показан порядок байтов в 32-битном целом числе.

Если процессор поддерживает *обратный* (*big-endian*) порядок байтов, то в старшем адресе будет располагаться младший значащий байт (МЗБ). В случае *прямого* (*little-endian*) порядка байтов младший значащий байт будет храниться в младшем адресе. Обратите внимание: независимо от порядка байтов старший значащий байт (СЗБ) всегда располагается слева, а младший значащий байт – справа. Таким образом, если присвоить переменной 32-битное целое значение `0x04030201`, то старший значащий байт будет иметь значение 4, а младший значащий байт – значение 1, независимо от порядка байтов. Если теперь привести адрес переменной к типу `char *` (`cp`), то мы сможем наблюдать различия в порядке байтов на разных аппаратных архитектурах. Если

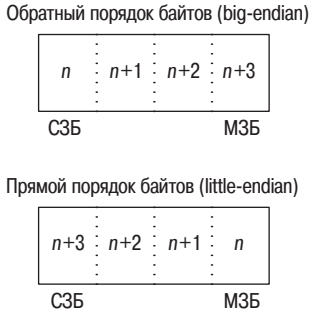


Рис. 16.1. *Порядок байтов в 32-битном целом числе*

процессор поддерживает прямой (*little-endian*) порядок байтов, `cp[0]` будет ссылаться на младший значащий байт, который содержит значение 1, а `cp[3]` будет ссылаться на старший значащий байт, имеющий значение 4. Если процессор поддерживает обратный (*big-endian*) порядок байтов, `cp[0]` будет ссылаться на старший значащий байт со значением 4, а `cp[3]` – на младший значащий байт со значением 1. В табл. 16.4 показано, какие платформы какой порядок байтов поддерживают.

Таблица 16.4. Порядок байтов на тестовых платформах

Операционная система	Архитектура процессора	Порядок байтов
FreeBSD 5.2.1	Intel Pentium	Прямой (*little-endian*)
Linux 2.4.22	Intel Pentium	Прямой
Mac OS X 10.3	PowerPC	Обратный (*big-endian*)
Solaris 9	Sun SPARC	Обратный

> Некоторые типы процессоров допускают возможность конфигурирования порядка байтов, что вносит еще большую путаницу.

Чтобы не возникало путаницы с порядком байтов при обмене данными между разнородными компьютерными системами, сетевые протоколы жестко задают порядок байтов. Набор протоколов TCP/IP использует сетевой (обратный, *big-endian*) порядок байтов. Порядок байтов приобретает важность, когда приложения начинают обмениваться форматированными данными. При использовании протоколов TCP/IP адреса имеют сетевой порядок байтов, поэтому в приложениях иногда возникает необходимость преобразовать порядок байтов, поддерживаемый аппаратной архитектурой, в сетевой порядок байтов. Такое преобразование обычно производится, например, при выводе адреса в удобочитаемой форме.

Преобразования между сетевым и аппаратным порядком байтов производятся с помощью следующих четырех функций.

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostint32);
                        Возвращает 32-битное целое с сетевым порядком байтов
uint16_t htons(uint16_t hostint16);
                        Возвращает 16-битное целое с сетевым порядком байтов
uint32_t ntohl(uint32_t netint32);
                    Возвращает 32-битное целое с аппаратным порядком байтов
uint16_t ntohs(uint16_t netint16);
                    Возвращает 16-битное целое с аппаратным порядком байтов
```

В именах функций буква n означает «network» (сетевой порядок байтов), а h – «host» (аппаратный). Буква l означает «long» (длинное, то есть 4-байтное, целое), а s – «short» (короткое, то есть 2-байтное, целое). Эти четыре функции определены в заголовочном файле <arpa/inet.h>, хотя в некоторых устаревших системах их определения размещались в заголовочном файле <netinet/in.h>.

16.3.2. Форматы адресов

Адреса используются для идентификации сокетов в конкретном домене. Для каждого домена определен свой формат представления адреса. Чтобы адреса различных форматов могли передаваться функциям, работающим с сокетами, выполняется приведение адресов к обобщенной структуре адреса sockaddr:

```
struct sockaddr {
    sa_family_t sa_family; /* семейство адресов */
    char        sa_data[]; /* адрес переменной длины */
    .
    .
    .
};
```

Реализации могут дополнять эту структуру своими полями и определять размер поля sa_data. Например, в ОС Linux эта структура определена как

```
struct sockaddr {
    sa_family_t sa_family;   /* семейство адресов */
    char        sa_data[14]; /* адрес переменной длины */
};
```

а в ОС FreeBSD как

```
struct sockaddr {
    unsigned char sa_len;      /* общая длина */
    sa_family_t   sa_family;   /* семейство адресов */
    char          sa_data[14]; /* адрес переменой длины */
};
```

Формат представления адресов Интернета определен в заголовочном файле <netinet/in.h>. Адреса сокетов из домена IPv4 (AF_INET) представлены структурой sockaddr_in:

```
struct in_addr {
    in_addr_t s_addr; /* адрес IPv4 */
};

struct sockaddr_in {
    sa_family_t    sin_family;  /* семейство адресов */
    in_port_t      sin_port;    /* номер порта */
    struct in_addr sin_addr;    /* адрес IPv4 */
};
```

Тип данных in_port_t определен как uint16_t, а тип in_addr_t – как uint32_t. Эти целочисленные типы данных задают количество используемых бит и определены в заголовочном файле <stdint.h>.

В отличие от домена AF_INET, адреса сокетов домена Интернет IPv6 (AF_INET6) представлены структурой sockaddr_ip6:

```
struct in6_addr {
    uint8_t s6_addr[16]; /* адрес IPv6 */
};

struct sockaddr_in6 {
    sa_family_t     sin6_family; /* семейство адресов */
    in_port_t       sin6_port;   /* номер порта */
    uint32_t sin6_  flowinfo;    /* класс трафика и сведения о потоке */
    struct in6_addr sin6_addr;   /* адрес IPv6 */
    uint32_t sin6_  scope_id;    /* идентификатор области видимости */
};
```

Это определения, которые требует стандарт Single UNIX Specification. Реализации могут добавлять в эти структуры дополнительные поля. Например, в ОС Linux структура sockaddr_in определена как

```
struct sockaddr_in {
    sa_family_t    sin_family;  /* семейство адресов */
    in_port_t      sin_port;    /* номер порта */
    struct in_addr sin_addr;    /* адрес IPv4 */
    unsigned char  sin_zero[8]; /* заполнитель */
};
```

где поле sin_zero является заполнителем и должно содержать только нулевые значения.

Обратите внимание, что хотя структуры sockaddr_in и sockaddr_in6 совершенно различны, тем не менее обе они приводятся к типу sockaddr при передаче функциям, работающим с сокетами. В разделе 17.3 мы увидим, что структура представления адресов сокетов домена UNIX отличается от обеих структур представления адресов домена Интернет.

Иногда возникает необходимость выводить адреса в виде, удобном для человека. Сетевое программное обеспечение BSD включало функции inet_ntoa и inet_addr, которые выполняли преобразование адресов между двоичным представлением и представлением в виде строки в десятично-точечной нотации (a.b.c.d). Однако эти функции могут работать только с адресами IPv4. Позднее появились две новые функции – inet_ntop и inet_pton, которые имели аналогичную функциональность, но могли работать также с адресами IPv6.

Функция inet_ntop преобразует адрес из двоичного представления с сетевым порядком байтов в текстовую строку. Функция inet_pton преобразует текстовую строку в двоичное представление с сетевым порядком байтов. Эти функции поддерживают только два значения аргумента *domain*: AF_INET и AF_INET6.

```
#include <arpa/inet.h>

const char *inet_ntop(int domain, const void *restrict addr,
                      char *restrict str, socklen_t size);
```

Возвращает указатель на строку с адресом в случае успеха, NULL в случае ошибки

```
int inet_pton(int domain, const char *restrict str, void *restrict addr);
```

Возвращает 1 в случае успеха, 0 при неверном формате, −1 в случае ошибки

Аргумент *size* функции inet_ntop задает размер буфера (*str*), в котором будет размещена строка. Для удобства существуют две константы: INET_ADDRSTRLEN, которая определяет размер буфера, достаточный для хранения строки с адресом IPv4, и INET6_ADDRSTRLEN, которая определяет размер буфера, достаточный для хранения строки с адресом IPv6. Аргумент *addr* функции inet_pton должен содержать адрес буфера достаточного размера для хранения 32-битного адреса, если в аргументе *domain* передается значение AF_INET, и 128-битного адреса, если в аргументе *domain* передается значение AF_INET6.

16.3.3. Определение адреса

В идеале приложения ничего не должны знать о внутренней структуре адреса сокета. Если приложение просто передает адреса в виде структуры sockaddr и не использует какие-либо специфические для протокола особенности, то оно сможет работать с самыми разными протоколами, которые предоставляют один и тот же вид услуги.

Сетевая подсистема BSD традиционно предоставляла интерфейсы для доступа к различной информации о конфигурации сети. В разделе 6.7 мы уже вкратце рассмотрели некоторые файлы с сетевой информацией и функции для работы с этими файлами. В этом разделе мы обсудим их подробнее и рассмотрим новые функции, применяемые для поиска адресной информации.

Информация о конфигурации сети может храниться в статических файлах (/etc/hosts, /etc/services и другие) или предоставляться различными сетевыми службами, такими как DNS (Domain Name System – система доменных имен) и NIS (Network Information Service – сетевая информационная служба). Независимо от того, где хранится информация, для доступа к ней используются одни и те же функции.

Адреса хостов, известных заданной системе, могут быть получены с помощью функции gethostent.

Функция gethostent возвращает очередную запись из файла с данными об адресах. Если файл еще не открыт, функция gethostent откроет его. Функция sethostent открывает файл или переходит в его начало, если он уже открыт. Функция endhostent закрывает файл.

```
#include <netdb.h>

struct hostent *gethostent(void);
```

Возвращает указатель в случае успеха, NULL в случае ошибки

```
void sethostent(int stayopen);

void endhostent(void);
```

Когда функция gethostent возвращает управление, мы получаем указатель на структуру hostent, которая может размещаться в области статической памяти, которая будет затерта при следующем обращении к этой функции. Структура hostent содержит как минимум следующие поля:

```
struct hostent {
    char  *h_name;      /* имя хоста */
    char **h_aliases;   /* указатель на массив псевдонимов */
    int    h_addrtype;  /* тип адреса */
    int    h_length;    /* длина адреса в байтах */
    char **h_addr_list; /* указатель на массив сетевых адресов */
    .
    .
    .
};
```

Возвращаемые адреса имеют сетевой порядок байтов.

Существуют еще две функции, gethostbyname и gethostbyaddr, которые также работают со структурами hostent, но в настоящее время они считаются устаревшими. Вскоре мы рассмотрим функции, которые пришли им на смену.

С помощью следующего набора функций можно получить имена сетей и их номера.

```
#include <netdb.h>

struct netent *getnetbyaddr(uint32_t net, int type);

struct netent *getnetbyname(const char *name);

struct netent *getnetent(void);
```

Все возвращают указатель в случае успеха, NULL в случае ошибки

```
void setnetent(int stayopen);

void endnetent(void);
```

Структура netent содержит как минимум следующие поля:

```
struct netent {
    char    *n_name;     /* имя сети */
    char   **n_aliases;  /* указатель на массив псевдонимов сети */
    int      n_addrtype; /* тип адреса */
    uint32_t n_net;      /* номер сети */
```

```
    .
    .
    .
};
```

Номер сети имеет сетевой порядок байтов. Тип адреса – одна из констант, определяющих семейство адресов (например, AF_INET).

Преобразования между именами протоколов и их номерами производятся с помощью следующих функций.

```
#include <netdb.h>

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotoent(void);
```

Все возвращают указатель в случае успеха, NULL в случае ошибки

```
void setprotoent(int stayopen);

void endprotoent(void);
```

Структура protoent определена стандартом POSIX.1 и должна содержать как минимум следующие поля:

```
struct protoent {
    char  *p_name;    /* имя протокола */
    char **p_aliases; /* указатель на массив псевдонимов протокола */
    int    p_proto;   /* номер протокола */
    .
    .
    .
};
```

Службы определяются номером порта, который является частью адреса. Каждой сетевой службе присвоен свой уникальный номер порта. Получить номер порта по имени службы можно с помощью функции getservbyname, а имя службы по номеру порта – с помощью функции getservbyport. С помощью функции getservent можно последовательно просмотреть все записи в базе данных служб.

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

struct servent *getservent(void);
```

Все возвращают указатель в случае успеха, NULL в случае ошибки

```
void setservent(int stayopen);

void endservent(void);
```

Структура servent содержит как минимум следующие поля:

```
struct servent {
    char  *s_name;    /* имя службы */
    char **s_aliases; /* указатель на массив псевдонимов службы */
    int    s_port;    /* номер порта */
    char  *s_proto;   /* имя протокола */
    .
    .
    .
};
```

Стандарт POSIX.1 определяет ряд новых функций, которые позволяют получать сетевой адрес из имени хоста и имени службы, и наоборот. Эти функции заменили устаревшие gethostbyname и gethostbyaddr.

Функция getaddrinfo позволяет получить адрес по имени хоста и сетевой службы.

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict host, const char *restrict service,
                const struct addrinfo *restrict hint,
                struct addrinfo **restrict res);
```

Возвращает 0 в случае успеха, неотрицательный код ошибки в случае неудачи

```
void freeaddrinfo (struct addrinfo *ai);
```

Мы должны передать функции имя хоста, имя службы или и то и другое. Если мы передаем только одно имя, второе должно быть пустым указателем. Имя хоста может быть как именем сетевого узла, так и адресом в десятичноточечной нотации.

Функция getaddrinfo возвращает связанный список структур addrinfo. Функция freeaddrinfo используется для освобождения памяти, занимаемой списком этих структур.

Структура addrinfo содержит как минимум следующие поля:

```
struct addrinfo {
    int              ai_flags;     /* флаги */
    int              ai_family;    /* семейство адресов */
    int              ai_socktype;  /* тип сокета */
    int              ai_protocol;  /* протокол */
    socklen_t        ai_addrlen;   /* длина адреса в байтах */
    struct sockaddr *ai_addr;      /* адрес */
    char            *ai_canonname; /* каноническое имя хоста */
    struct addrinfo *ai_next;      /* следующий элемент списка */
    .
    .
    .
};
```

Аргумент *hint* может использоваться для задания дополнительных критериев выбора адресов. Этот аргумент представляет собой шаблон, используемый для фильтрации адресов, в котором используются только поля `ai_family`, `ai_flags`, `ai_protocol` и `ai_socktype`. Остальные поля целочисленного типа должны содержать значения 0, а указатели – `NULL`. В табл. 16.5 приводится перечень флагов, которые могут быть использованы в поле `ai_flags`, и их назначение.

Таблица 16.5. Флаги для структуры addrinfo

Флаг	Описание
`AI_ADDRCONFIG`	Запрос типа адреса (IPv4 или IPv6).
`AI_ALL`	Поиск обоих типов адресов – IPv4 и IPv6 (используется только вместе с флагом `AI_V4MAPPED`).
`AI_CANONNAME`	Запрос канонического имени (в противоположность псевдониму).
`AI_NUMERICHOST`	Вернуть адрес в числовом формате.
`AI_NUMERICSERV`	Вернуть службу в виде номера порта.
`AI_PASSIVE`	Сокет предназначен для работы в режиме прослушивания.
`AI_V4MAPPED`	Если адреса IPv6 не найдены, возвращать адреса IPv4 в формате IPv6.

Если вызов `getaddrinfo` завершается ошибкой, мы не можем воспользоваться функциями `perror` или `strerror` для генерации текста сообщения об ошибке. Вместо них для преобразования кода ошибки в текстовое представление нужно пользоваться функцией `gai_strerror`.

```
#include <netdb.h>

const char *gai_strerror(int error);
```

Возвращает указатель на строку с описанием ошибки

Функция `getnameinfo` преобразует адрес в имя хоста и имя сетевой службы.

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *restrict addr,
                socklen_t alen, char *restrict host,
                socklen_t hostlen, char *restrict service,
                socklen_t servlen, unsigned int flags);
```

Возвращает 0 в случае успеха, ненулевое значение в случае ошибки

Адрес сокета (*addr*) преобразуется в имя хоста и имя сетевой службы. Если в аргументе *host* передается непустой указатель, он должен указывать на буфер, размер которого указывается в аргументе *hostlen*. Имя хоста будет возвращено в этом буфере. Аналогично, если *service* не является пустым указа-

телем, он указывает на буфер размером *servlen* байт, в котором будет возвращено имя сетевой службы.

С помощью аргумента *flags* можно влиять на порядок преобразования. В табл. 16.6 перечислены поддерживаемые значения этого аргумента.

Таблица 16.6. Флаги для функции getnameinfo

Флаг	Описание
NI_DGRAM	Служба основана на интерфейсе дейтаграмм, а не потоков.
NI_NAMEREQD	Если имя хоста не найдено, считать это ошибкой.
NI_NOFQDN	Для локальных хостов вместо полного доменного имени возвращать только имя узла.
NI_NUMERICHOST	Вместо имени хоста возвращать его адрес в числовой форме.
NI_NUMERICSERV	Возвращать имя службы в числовом представлении (то есть номер порта).

Пример

В листинге 16.1 показан пример использования функции getaddrinfo.

Листинг 16.1. Вывод сведений о хостах и сетевых службах

```
#include "apue.h"
#include <netdb.h>
#include <arpa/inet.h>
#if defined(BSD) || defined(MACOS)
#include <sys/socket.h>
#include <netinet/in.h>
#endif

void
print_family(struct addrinfo *aip)
{
    printf(" семейство ");
    switch (aip->ai_family) {
    case AF_INET:
        printf("inet");
        break;
    case AF_INET6:
        printf("inet6");
        break;
    case AF_UNIX:
        printf("unix");
        break;
    case AF_UNSPEC:
        printf("не определено");
        break;
    default:
        printf("неизвестно");
    }
}
```

```
void
print_type(struct addrinfo *aip)
{
    printf(" тип ");
    switch (aip->ai_socktype) {
    case SOCK_STREAM:
        printf("stream");
        break;
    case SOCK_DGRAM:
        printf("datagram");
        break;
    case SOCK_SEQPACKET:
        printf("seqpacket");
        break;
    case SOCK_RAW:
        printf("raw");
        break;
    default:
        printf("неизвестный (%d)", aip->ai_socktype);
    }
}

void
print_protocol(struct addrinfo *aip)
{
    printf(" протокол ");
    switch (aip->ai_protocol) {
    case 0:
        printf("по умолчанию");
        break;
    case IPPROTO_TCP:
        printf("TCP");
        break;
    case IPPROTO_UDP:
        printf("UDP");
        break;
    case IPPROTO_RAW:
        printf("raw");
        break;
    default:
        printf("неизвестный (%d)", aip->ai_protocol);
    }
}

void
print_flags(struct addrinfo *aip)
{
    printf("флаги");
    if (aip->ai_flags == 0) {
        printf(" 0");
    } else {
        if (aip->ai_flags & AI_PASSIVE)
            printf(" passive");
```

```
        if (aip->ai_flags & AI_CANONNAME)
            printf(" canon");
        if (aip->ai_flags & AI_NUMERICHOST)
            printf(" numhost");
#if defined(AI_NUMERICSERV)
        if (aip->ai_flags & AI_NUMERICSERV)
            printf(" numserv");
#endif
#if defined(AI_V4MAPPED)
        if (aip->ai_flags & AI_V4MAPPED)
            printf(" v4mapped");
#endif
#if defined(AI_ALL)
        if (aip->ai_flags & AI_ALL)
            printf(" all");
#endif
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    struct sockaddr_in *sinp;
    const char *addr;
    int err;
    char abuf[INET_ADDRSTRLEN];

    if (argc != 3)
        err_quit("Использование: %s имя_узла служба", argv[0]);
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = 0;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], argv[2], &hint, &ailist)) != 0)
        err_quit("ошибка вызова функции getaddrinfo: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        print_flags(aip);
        print_family(aip);
        print_type(aip);
        print_protocol(aip);
        printf("\n\tхост %s", aip->ai_canonname?aip->ai_canonname:"-");
        if (aip->ai_family == AF_INET) {
            sinp = (struct sockaddr_in *)aip->ai_addr;
            addr = inet_ntop(AF_INET, &sinp->sin_addr, abuf,
                             INET_ADDRSTRLEN);
            printf(" адрес %s", addr?addr:"не известен");
            printf(" порт %d", ntohs(sinp->sin_port));
```

```
            }
            printf("\n");
        }
        exit(0);
    }
```

Эта программа иллюстрирует работу с функцией `getaddrinfo`. Если заданный хост предоставляет заданную службу по нескольким протоколам, программа выведет несколько записей. В нашем примере выводится адресная информация только для протоколов IPv4 (`ai_family` имеет значение `AF_INET`). Если необходимо ограничиться только семейством протоколов `AF_INET`, следует записать это значение в поле `ai_family` аргумента *`hint`*.

После запуска программы на одной из наших тестовых систем мы получили:

```
$ ./a.out harry nfs
флаги canon семейство inet тип stream протокол TCP
     хост harry адрес 192.168.1.105 порт 2049
флаги canon семейство inet тип datagram протокол UDP
     хост harry адрес 192.168.1.105 порт 2049
```

16.3.4. Присвоение адресов сокетам

Адрес, присваиваемый клиентскому сокету, не представляет для нас особого интереса, и потому мы можем позволить системе выбирать адрес по умолчанию. Однако для сервера очень важно присвоить сокету предопределенный адрес, на который клиенты будут присылать запросы. Клиентам необходимо заранее знать требуемый адрес, чтобы войти в контакт с сервером, и самое простое решение заключается в том, чтобы зарезервировать адрес сервера в файле `/etc/services` или в службе имен.

Чтобы ассоциировать адрес с сокетом, используется функция `bind`.

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Существует несколько ограничений, касающихся адресов:

- Указываемый адрес должен быть действительным адресом для машины, на которой выполняется процесс – мы не можем задать адрес, который принадлежит другой машине.
- Формат адреса должен совпадать с форматом, который поддерживается семейством адресов, указанным при создании сокета.
- Номер порта не может быть меньше 1024, если процесс не имеет соответствующих привилегий (например, привилегий суперпользователя).
- Обычно каждый конкретный адрес может быть связан только с одним сокетом, хотя некоторые протоколы допускают присвоение одного и того же адреса нескольким сокетам.

В домене Интернет имеется специальный IP-адрес INADDR_ANY, который соответствует адресам всех сетевых интерфейсов в системе. Это означает, что существует возможность принимать пакеты с любого сетевого интерфейса, установленного в системе. В следующем разделе мы увидим, что система сама может присваивать адрес сокету при обращении к функциям connect и listen.

Чтобы получить адрес, присвоенный сокету, можно использовать функцию getsockname.

```
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Перед вызовом функции getsockname необходимо занести в аргумент *alenp* адрес целого числа, которое определяет размер буфера *addr*. По возвращении из функции это число будет содержать фактический размер полученного адреса. Если адрес не умещается в предоставленный буфер, он будет усечен. Если сокету не присвоен адрес, результат функции не определен.

Если сокет соединен с удаленным узлом, мы можем получить адрес удаленного узла, обратившись к функции getpeername.

```
#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Кроме того, что эта функция возвращает адрес удаленной стороны, она полностью идентична функции getsockname.

16.4. Установление соединения

Если мы имеем дело с сетевой службой, которая ориентирована на установление логического соединения (SOCK_STREAM или SOCK_SEQPACKET), то прежде чем начать обмениваться данными, необходимо установить соединение между сокетом процесса, посылающего запрос (клиентом), и процессом, предоставляющим услугу (сервером). Для создания соединения используется функция connect.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Адрес, который передается функции connect, – это адрес сервера, с которым предполагается установить связь. Если сокет *sockfd* не связан с каким-либо адресом, функция присвоит ему адрес по умолчанию.

Попытка соединения с сервером может потерпеть неудачу по нескольким причинам. Машина, с которой устанавливается соединение, должна быть включена и связана с сетью. Серверу должен быть присвоен адрес, с которым мы пытаемся соединиться, и в очереди запросов на соединение на стороне сервера должно быть достаточно места, чтобы поставить в очередь и наш запрос (вскоре мы поговорим об этом более подробно). Таким образом, приложение должно уметь обрабатывать возможные ошибки соединения.

Пример

В листинге 16.2 показан пример обработки ошибочных ситуаций, возникающих при попытке установления соединения. Такие ошибки наиболее вероятны при попытке связаться с сервером, работающим в сильно загруженной системе.

Листинг 16.2. Попытка соединения с повторением

```
#include "apue.h"
#include <sys/socket.h>

#define MAXSLEEP 128

int
connect_retry(int sockfd, const struct sockaddr *addr, socklen_t alen)
{
    int nsec;
    /*
     * Попытаться установить соединение с экспоненциальной задержкой.
     */
    for (nsec = 1; nsec <= MAXSLEEP; nsec <<= 1) {
        if (connect(sockfd, addr, alen) == 0) {
            /*
             * Соединение установлено.
             */
            return(0);
        }
        /*
         * Задержка перед следующей попыткой.
         */
        if (nsec <= MAXSLEEP/2)
            sleep(nsec);
    }
    return(-1);
}
```

Эта функция демонстрирует известный *алгоритм с экспоненциальной задержкой*. Если функция connect терпит неудачу, процесс приостанавливается на короткое время и затем повторяет попытку, всякий раз увеличивая время задержки до тех пор, пока оно не достигнет максимума – около 2 минут.

Если сокет находится в неблокирующем режиме, который мы обсудим в разделе 16.8, и соединение не может быть установлено немедленно, функция connect вернет значение –1 и код ошибки EINPROGRESS в переменной errno. Приложение может определить, когда дескриптор станет доступен для записи, с помощью функции poll или select. В этот момент установление соединения будет завершено.

Функция connect также может использоваться для работы со службами, которые не требуют установления соединения (SOCK_DGRAM). Казалось бы, здесь кроется какое-то противоречие, но это не так, поскольку это своего рода оптимизация. Если мы вызовем функцию connect для сокета SOCK_DGRAM, то для всех исходящих пакетов будет установлен адрес, который мы передадим функции connect, что освобождает нас от необходимости указывать адрес при передаче каждой дейтаграммы. Кроме того, мы будем получать дейтаграммы только с указанного адреса.

С помощью функции listen сервер заявляет о своем желании принимать запросы на установление соединения.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Аргумент *backlog* определяет желаемое количество ожидающих обработки запросов, которые должны быть поставлены в очередь от имени процесса. Фактическое значение определяется самой системой, но верхний предел определен под именем SOMAXCONN в заголовочном файле <sys/socket.h>.

> В ОС Solaris предел из <sys/socket.h> игнорируется системой. Значение этого предела зависит от реализации каждого конкретного протокола. Так, для TCP этот предел равен 128.

После заполнения очереди система будет отвергать дополнительные запросы на соединение, поэтому значение *backlog* должно выбираться с учетом возможной нагрузки на сервер и объема ресурсов, необходимого для принятия запроса и запуска службы.

После вызова функции listen указанный сокет будет использоваться для приема запросов на соединение. Функция accept принимает запрос и преобразует его в соединение.

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict len);
```

Возвращает дескриптор файла (сокета) в случае успеха, –1 в случае ошибки

Функция accept возвращает дескриптор сокета, соединенного с клиентом, вызвавшим функцию connect. Этот новый сокет имеет тот же тип и семейство адресов, что и сокет *sockfd*. Первоначальный сокет, который передается

функции accept, не связан с установленным соединением, он остается свободным для приема последующих запросов на соединение.

Если нас не беспокоит проблема идентификации клиента, мы можем передать в аргументах *addr* и *len* значение NULL. В противном случае необходимо записать в аргумент *addr* адрес буфера достаточного размера для хранения адреса, а в аргумент *len* – адрес целого числа, которое определяет размер буфера. По возвращении из функции accept в буфере будет находиться адрес клиента, а по адресу *len* будет сохранен фактический размер адреса.

Если запросы, ожидающие обработки, отсутствуют, то функция accept будет заблокирована до тех пор, пока не поступит хотя бы один запрос. Если *sockfd* находится в неблокирующем режиме, функция accept вернет значение –1 и код ошибки EAGAIN или EWOULDBLOCK в переменной errno.

> На всех четырех платформах, обсуждаемых в этой книге, константа EAGAIN определена с тем же значением, что и EWOULDBLOCK.

Если сервер вызовет функцию accept при отсутствии запросов на соединение, он окажется заблокированным, пока не придет хотя бы один запрос. Как вариант, сервер может использовать функцию poll или select для ожидания прибытия запросов. В этом случае сокет, который ожидает соединения, будет выступать как доступный для чтения.

Пример

В листинге 16.3 приводится исходный код функции, которая размещает и инициализирует сокет для серверного процесса.

Листинг 16.3. Инициализация сокета для сервера

```
#include "apue.h"
#include <errno.h>
#include <sys/socket.h>

int
initserver(int type, const struct sockaddr *addr, socklen_t alen, int qlen)
{
    int fd;
    int err = 0;

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (bind(fd, addr, alen) < 0) {
        err = errno;
        goto errout;
    }
    if (type == SOCK_STREAM || type == SOCK_SEQPACKET) {
        if (listen(fd, qlen) < 0) {
            err = errno;
            goto errout;
        }
    }
    return(fd);
```

```
errout:
    close(fd);
    errno = err;
    return(-1);
}
```

Позднее мы увидим, что несколько странные правила протокола TCP, касающиеся многократного использования адреса, делают этот пример неадекватным. В листинге 16.9 приводится другая версия функции, которая обходит эти правила и исправляет главный недостаток проблему данной версии.

16.5. Передача данных

Поскольку сокет представлен файловым дескриптором, мы можем использовать функции `read` и `write`, когда он соединен с удаленной стороной. Мы уже говорили о том, что сокеты типа `SOCK_DGRAM` также могут находиться в состоянии «установленного соединения», если с помощью функции `connect` был задан адрес удаленного узла по умолчанию. Возможность использовать функции `read` и `write` для работы с дескрипторами сокетов является большим плюсом, так как это означает, что мы можем передавать дескрипторы сокетов функциям, которые изначально проектировались для работы с локальными файлами. Кроме того, можно передавать дескрипторы сокетов дочерним процессам, которые запускают программы, ничего не знающие о сокетах.

Хотя мы и можем использовать функции `read` и `write` для обмена данными через сокеты, но это практически все, что возможно сделать с их помощью. Если нам понадобится определить какие-либо дополнительные возможности, принимать пакеты от нескольких клиентов или передавать экстренные данные, то придется использовать одну из шести функций, специально разработанных для передачи данных через сокеты.

Три функции из этих шести предназначены для передачи данных, а три – для приема. В первую очередь мы рассмотрим функции, которые используются для передачи данных.

Самая простая из них – функция `send`. Она похожа на функцию `write`, но в отличие от нее позволяет указать дополнительные флаги, влияющие на процесс передачи данных.

```
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
```

Возвращает количество отправленных байтов в случае успеха, –1 в случае ошибки

Как и в случае функции `write`, к моменту вызова функции `send` сокет должен быть соединен с удаленной стороной. Аргументы *buf* и *nbytes* имеют тот же смысл, что и для функции `write`.

Однако, в отличие от write, функция send поддерживает дополнительный аргумент *flags*. Стандарт Single UNIX Specification определяет два флага, но большинство реализаций поддерживают дополнительные флаги. Перечень флагов приводится в табл. 16.7.

Таблица 16.7. Флаги, используемые функцией send

Флаг	Описание	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
MSG_DONTROUTE	Не отправлять пакет за пределы локальной сети		•	•	•	•
MSG_DONTWAIT	Разрешить неблокирующий режим выполнения операции (эквивалент флага O_NONBLOCK)		•	•	•	
MSG_EOR	Обозначает конец записи, если поддерживается протоколом	•	•	•	•	
MSG_OOB	Обозначает передачу экстренных данных, если поддерживается протоколом (раздел 16.7)	•	•	•	•	•

Успешное завершение функции send еще не означает, что процесс на другом конце соединения получил отправленные данные. Все, что гарантирует функция send в случае успеха, – это отсутствие ошибок при передаче данных сетевым драйверам.

Если при использовании протокола, который ограничивает размер сообщения, попытаться послать сообщение размером больше максимально допустимого, функция send вернет признак ошибки с кодом EMSGSIZE в переменной errno. При использовании протоколов, которые поддерживают обмен данными в виде потоков байтов, функция send будет заблокирована до тех пор, пока не будет передан весь объем данных.

Функция sendto напоминает функцию send. Отличие состоит в том, что sendto позволяет указать адрес получателя при работе с сокетами типа SOCK_DGRAM.

```
#include <sys/socket.h>

ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags,
               const struct sockaddr *destaddr, socklen_t destlen);
```

Возвращает количество отправленных байтов в случае успеха, –1 в случае ошибки

При использовании сокетов, ориентированных на установление соединения, адрес получателя игнорируется, так как он определяется самим соединением. Для обслуживания сокетов, которые не создают соединение, нельзя исполь-

зовать функцию send, если предварительно не была вызвана функция connect, поэтому sendto предоставляет альтернативный способ передачи данных.

В нашем распоряжении имеется еще одна функция, предназначенная для передачи данных через сокет. Функция sendmsg принимает структуру msghdr, которая определяет сразу несколько буферов с данными для передачи, что делает ее похожей на writev (раздел 14.7).

```
#include <sys/socket.h>

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Возвращает количество отправленных байтов в случае успеха, −1 в случае ошибки

Согласно стандарту POSIX.1, структура msghdr должна содержать как минимум следующие поля:

```
struct msghdr {
    void         *msg_name;       /* необязательный адрес */
    socklen_t     msg_namelen;    /* размер адреса в байтах */
    struct iovec *msg_iov;        /* массив буферов ввода-вывода */
    int           msg_iovlen;     /* количество элементов в массиве */
    void         *msg_control;    /* вспомогательные данные */
    socklen_t     msg_controllen; /* объем вспомогательных данных в байтах */
    int           msg_flags;      /* флаги принятого сообщения */
    .
    .
    .
};
```

Мы уже рассматривали структуру iovec в разделе 14.7. Назначение вспомогательных данных будет рассмотрено в разделе 17.4.2.

Функция recv похожа на функцию read, но в отличие от нее позволяет указать дополнительные флаги, влияющие на процесс приема данных.

```
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

Возвращает длину сообщения в байтах, 0 при отсутствии доступных сообщений и на удаленном конце соединения запрещена операция записи в сокет, −1 в случае ошибки

Перечень флагов, которые могут быть переданы функции recv, приводится в табл. 16.8. Только три из них определены стандартом Single UNIX Specification.

Если указан флаг MSG_PEEK, можно «подсмотреть» содержимое следующего сообщения, не удаляя его из приемной очереди. Эти данные будут повторно получены при следующем обращении к функции read или к одной из функций recv.

Таблица 16.8. Флаги, используемые функцией recv

Флаг	Описание	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
MSG_OOB	Принять экстренные данные, если поддерживается протоколом (раздел 16.7)	•	•	•	•	•
MSG_PEEK	Вернуть содержимое пакета, но не удалять его из приемной очереди	•	•	•	•	•
MSG_TRUNC	Запросить, чтобы был возвращен реальный размер пакета, даже если он был обрезан			•		
MSG_WAITALL	Ждать, пока не будут получены все данные (только для SOCK_STREAM)	•	•	•	•	•

При использовании сокетов типа SOCK_STREAM мы можем получить меньший объем данных, чем было запрошено. Флаг MSG_WAITALL запрещает такое поведение функции recv, заставляя ее дожидаться, пока запрошенный объем данных не будут получен полностью. Для сокетов типа SOCK_DGRAM и SOCK_SEQPACKET флаг MSG_WAITALL не оказывает влияния на поведение функции recv, поскольку для них за одну операцию чтения всегда возвращается сообщение целиком.

Если отправитель вызвал функцию shutdown (раздел 16.2), чтобы завершить передачу данных, или если протокол поддерживает завершение передачи по умолчанию и отправитель закрыл свой сокет, то функция recv вернет значение 0 после получения всех данных.

Если нас интересует идентификация отправителя, мы можем воспользоваться функцией recvfrom, которая возвращает адрес, с которого была произведена передача данных.

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                 struct sockaddr *restrict addr, socklen_t *restrict addrlen);
```

Возвращает длину сообщения в байтах, 0 при отсутствии доступных сообщений и на удаленном конце соединения запрещена операция записи в сокет, −1 в случае ошибки

Если аргумент *addr* содержит непустой указатель, то по указанному адресу будет записан адрес сокета, с которого данные были отправлены. При вызове recvfrom необходимо передать в аргументе *addrelen* указатель на целое число, которое содержит размер буфера *addr*. По возвращении из функции это число будет содержать фактический размер адреса в байтах.

Так как функция позволяет получить адрес отправителя, она обычно используется для работы с сокетами типа SOCK_DGRAM. В остальном функция recvfrom ничем не отличается от recv.

Чтобы принять данные сразу в несколько буферов, как это делает функция readv (раздел 14.7), или получить вспомогательные данные (раздел 17.4.2), мы можем использовать функцию recvmsg.

```
#include <sys/socket.h>

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

Возвращает длину сообщения в байтах, 0 при отсутствии доступных сообщений и на удаленном конце соединения запрещена операция записи в сокет, −1 в случае ошибки

Перечень приемных буферов определяется структурой msghdr (которая также используется с функцией sendmsg). Для изменения поведения функции recvmsg по умолчанию можно установить дополнительные флаги в аргументе *flags*. По возвращении из функции поле msg_flags структуры msghdr будет указывать различные характеристики полученных данных. (На входе в recvmsg поле msg_flags игнорируется.) Перечень флагов, которые могут быть возвращены из recvmsg, приводится в табл. 16.9. Пример использования этой функции мы увидим в главе 17.

Таблица 16.9. Флаги, возвращаемые функцией recvmsg в поле msg_flags

Флаг	Описание	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
MSG_CTRUNC	Управляющая информация была обрезана	•	•	•	•	•
MSG_DONTWAIT	Был задан неблокирующий режим вызова recvmsg			•		•
MSG_EOR	Был получен признак конца записи	•	•	•	•	•
MSG_OOB	Были приняты экстренные данные	•	•	•	•	•
MSG_TRUNC	Данные были обрезаны	•	•	•	•	•

Пример – клиент, ориентированный на создание соединения

В листинге 16.4 приводится исходный код клиентской программы, которая запрашивает у сервера результат выполнения команды uptime. Мы назвали эту службу «remote uptime» (удаленный uptime) или, для краткости, «ruptime».

Листинг 16.4. Клиент, получающий результат выполнения команды uptime на сервере

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define MAXADDRLEN 256
#define BUFLEN 128

extern int connect_retry(int, const struct sockaddr *, socklen_t);

void
print_uptime(int sockfd)
{
    int n;
    char buf[BUFLEN];

    while ((n = recv(sockfd, buf, BUFLEN, 0)) > 0)
        write(STDOUT_FILENO, buf, n);
    if (n < 0)
        err_sys("ошибка вызова функции recv");
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int sockfd, err;

    if (argc != 2)
        err_quit("Использование: ruptime hostname");
    hint.ai_flags = 0;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;

    if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
        err_quit("ошибка вызова функции getaddrinfo: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = socket(aip->ai_family, SOCK_STREAM, 0)) < 0)
            err = errno;
        if (connect_retry(sockfd, aip->ai_addr, aip->ai_addrlen) < 0) {
            err = errno;
        } else {
            print_uptime(sockfd);
            exit(0);
        }
```

```
        }
        fprintf(stderr, "невозможно соединиться с %s: %s\n", argv[1],
            strerror(err));
        exit(1);
    }
```

Эта программа соединяется с сервером, читает переданную сервером строку и выводит ее на стандартный вывод. Так как в программе используется сокет типа SOCK_STREAM, мы не можем быть уверены в том, что за одно обращение к функции recv строка будет прочитана целиком, поэтому попытки получения данных повторяются в цикле до тех пор, пока функция не вернет значение 0.

Функция getaddrinfo может вернуть несколько адресов, если сервер поддерживает несколько сетевых интерфейсов или несколько сетевых протоколов. Мы пробуем соединиться поочередно с каждым из них, пока не будет найден адрес требуемой службы. Для установления соединения с сервером используется функция connect_retry из листинга 16.2.

Пример – сервер, ориентированный на создание соединения

В листинге 16.5 приводится исходный код сервера, который возвращает результат выполнения команды uptime по запросу клиента из листинга 16.4.

Листинг 16.5. Сервер, предоставляющий результат выполнения команды uptime по запросу клиента

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUFLEN 128
#define QLEN 10
#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int clfd;
    FILE *fp;
    char buf[BUFLEN];

    for (;;) {
        clfd = accept(sockfd, NULL, NULL);
        if (clfd < 0) {
            syslog(LOG_ERR, "ruptimed: ошибка вызова функции accept: %s",
```

```
                    strerror(errno));
            exit(1);
        }
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "ошибка: %s\n", strerror(errno));
            send(clfd, buf, strlen(buf), 0);
        } else {
            while (fgets(buf, BUFLEN, fp) != NULL)
                send(clfd, buf, strlen(buf), 0);
            pclose(fp);
        }
        close(clfd);
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int sockfd, err, n;
    char *host;

    if (argc != 1)
        err_quit("Использование: ruptimed");

#ifdef _SC_HOST_NAME_MAX
    n = sysconf(_SC_HOST_NAME_MAX);
    if (n < 0)        /* лучшее, что можно сделать */
#endif
        n = HOST_NAME_MAX;
    host = malloc(n);
    if (host == NULL)
        err_sys("ошибка вызова функции malloc");
    if (gethostname(host, n) < 0)
        err_sys("ошибка вызова функции gethostname");
    daemonize("ruptimed");
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: ошибка вызова функции getaddrinfo: %s",
               gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
```

```
            aip->ai_addrlen, QLEN)) >= 0) {
              serve(sockfd);
              exit(0);
            }
        }
        exit(1);
    }
```

Чтобы определить собственный адрес, сервер должен получить сетевое имя компьютера, на котором он запущен. Некоторые системы не определяют константу _SC_HOST_NAME_MAX, в этой ситуации мы используем константу HOST_NAME_MAX. Если система не определяет и константу HOST_NAME_MAX, то мы задаем ее самостоятельно. Стандарт POSIX.1 указывает, что минимальное значение длины сетевого имени хоста должно быть равно 255 байтам без учета завершающего нулевого символа, поэтому мы определяем константу HOST_NAME_MAX со значением 256 – с учетом завершающего нулевого символа.

Сервер получает сетевое имя хоста с помощью функции gethostname и отыскивает адрес службы uptime. Функция getaddrinfo может вернуть несколько адресов, но для простоты мы выбираем первый из них, на котором будет возможно установить пассивный сокет. Обработку нескольких адресов мы оставляем вам в качестве упражнения.

Для инициализации сокета мы использовали функцию initserver из листинга 16.3. Этот сокет будет ожидать поступления запросов на соединение. (Фактически мы использовали версию функции из листинга 16.9, почему – вы узнаете, когда мы перейдем к обсуждению параметров сокетов в разделе 16.6.)

Пример – альтернативный сервер, ориентированный на создание соединения

Ранее мы уже говорили, что возможность использования файловых дескрипторов для организации доступа к сокетам играет важную роль, так как позволяет использовать для работы в сети программы, которые ничего не знают о сетях. Листинг 16.6 как раз демонстрирует такой сервер. Вместо того, чтобы читать данные со стандартного вывода команды uptime и передавать их клиенту, сервер связывает стандартный вывод и стандартный вывод сообщений об ошибках команды uptime с сокетом, который соединен с клиентом.

Листинг 16.6. Сервер, демонстрирующий запись вывода команды прямо в сокет

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define QLEN 10
#ifndef HOST_NAME_MAX
```

```
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int clfd, status;
    pid_t pid;

    for (;;) {
        clfd = accept(sockfd, NULL, NULL);
        if (clfd < 0) {
            syslog(LOG_ERR, "ruptimed: ошибка вызова функции accept: %s",
                   strerror(errno));
            exit(1);
        }
        if ((pid = fork()) < 0) {
            syslog(LOG_ERR, "ruptimed: ошибка вызова функции fork: %s",
                   strerror(errno));
            exit(1);
        } else if (pid == 0) { /* дочерний процесс */
            /*
             * Родительский процесс вызвал функцию daemonize (листинг 13.1),
             * поэтому STDIN_FILENO, STDOUT_FILENO и STDERR_FILENO
             * уже открыты на устройстве /dev/null. В результате
             * нет необходимости защищать вызов close проверкой
             * на равенство clfd одному из этих значений.
             */
            if (dup2(clfd, STDOUT_FILENO) != STDOUT_FILENO ||
              dup2(clfd, STDERR_FILENO) != STDERR_FILENO) {
                syslog(LOG_ERR, "ruptimed: неожиданная ошибка");
                exit(1);
            }
            close(clfd);
            execl("/usr/bin/uptime", "uptime", (char *)0);
            syslog(LOG_ERR, "ruptimed: неожиданный возврат из exec: %s",
                   strerror(errno));
        } else { /* родительский процесс */
            close(clfd);
            waitpid(pid, &status, 0);
        }
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int sockfd, err, n;
    char *host;
```

```
    if (argc != 1)
        err_quit("Использование: ruptimed");
#ifdef _SC_HOST_NAME_MAX
    n = sysconf(_SC_HOST_NAME_MAX);
    if (n < 0) /* лучшее, что можно сделать */
#endif
        n = HOST_NAME_MAX;
    host = malloc(n);
    if (host == NULL)
        err_sys("ошибка вызова функции malloc");
    if (gethostname(host, n) < 0)
        err_sys("ошибка вызова функции gethostname");
    daemonize("ruptimed");
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: ошибка вызова функции getaddrinfo: %s",
               gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
          aip->ai_addrlen, QLEN)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

Вместо использования функции `popen` для запуска команды `uptime` и получения ее вывода мы вызвали функцию `fork`, чтобы запустить дочерний процесс, который затем связывает дескрипторы `STDOUT_FILENO` и `STDERR_FILENO` с сокетом. Когда запускается команда `uptime`, она выводит результаты на стандартный вывод, который связан с сокетом, и данные отправляются клиенту.

Родительский процесс может закрыть дескриптор сокета, соединенный с клиентом, поскольку дочерний процесс удерживает его открытым. Родительский процесс ожидает завершения дочернего процесса, что предотвращает появление зомби. Так как время работы команды `uptime` невелико, родительский процесс может позволить себе дождаться завершения потомка, прежде чем перейти к приему следующего запроса на соединение. Такая стратегия может оказаться неприемлемой, если дочерний процесс выполняется достаточно продолжительное время.

В предыдущем примере использовался сокет, ориентированный на создание логического соединения. Но как правильно выбрать тип сокета? В каких случаях следует использовать сокеты, ориентированные либо не ориентированные на создание соединения? Ответ зависит от того, какой объем работы предполагается выполнить и насколько приложение чувствительно к ошибкам.

При использовании сокетов, не ориентированных на создание соединений, пакеты могут прибывать не в том порядке, в каком они были отправлены – поэтому, если все данные не смогут уместиться в один пакет, придется побеспокоиться о порядке доставки пакетов. Максимальный размер пакета является характеристикой используемого протокола. Кроме того, следует учитывать, что при использовании сокетов, не ориентированных на создание соединений, пакеты могут теряться. Если логика приложения не допускает таких потерь, то необходимо использовать сокеты, ориентированные на создание соединений.

Есть два способа сделать приложение нечувствительным к потере пакетов. Если необходимо обеспечить надежную связь с удаленной стороной, нам придется пронумеровать пакеты и запрашивать повторную передачу отсутствующего пакета при обнаружении потери. Кроме того, необходимо предусмотреть обработку дубликатов пакетов, поскольку пакет может задержаться, а приложение может посчитать, что он потерян, и запросить повторную передачу, после чего могут быть доставлены оба пакета.

Второй вариант – разрешить пользователю повторить команду при появлении ошибки. Для простых приложений такой вариант может оказаться вполне приемлемым, но для сложных программ он не подходит – лучше использовать сокеты, ориентированные на создание логических соединений.

Один из недостатков сокетов, ориентированных на создание логических соединений, состоит в том, что для установления соединения необходим больший объем работы и требуется больше времени, а кроме того, каждое соединение потребляет больше ресурсов операционной системы.

Пример – клиент, не ориентированный на создание соединения

Программа из листинга 16.7 является версией клиента из листинга 16.4, использующей интерфейс дейтаграмм.

Листинг 16.7. Клиент, использующий интерфейс дейтаграмм

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define BUFLEN 128
#define TIMEOUT 20

void
sigalrm(int signo)
```

```
{
}

void
print_uptime(int sockfd, struct addrinfo *aip)
{
    int n;
    char buf[BUFLEN];

    buf[0] = 0;
    if (sendto(sockfd, buf, 1, 0, aip->ai_addr, aip->ai_addrlen) < 0)
        err_sys("ошибка вызова функции sendto");
    alarm(TIMEOUT);
    if ((n = recvfrom(sockfd, buf, BUFLEN, 0, NULL, NULL)) < 0) {
        if (errno != EINTR)
            alarm(0);
        err_sys("ошибка вызова функции recv");
    }
    alarm(0);
    write(STDOUT_FILENO, buf, n);
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int sockfd, err;
    struct sigaction sa;

    if (argc != 2)
        err_quit("Использование: ruptime hostname");
    sa.sa_handler = sigalrm;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGALRM, &sa, NULL) < 0)
        err_sys("ошибка вызова функции sigaction");
    hint.ai_flags = 0;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_DGRAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
        err_quit("ошибка вызова функции getaddrinfo: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = socket(aip->ai_family, SOCK_DGRAM, 0)) < 0) {
            err = errno;
        } else {
            print_uptime(sockfd, aip);
            exit(0);
```

```
        }
    }
    fprintf(stderr, "невозможно соединиться с %s: %s\n", argv[1],
            strerror(err));
    exit(1);
}
```

Функция `main` в этой версии клиента практически не изменилась, добавилась только установка обработчика сигнала `SIGALRM`. Функция `alarm` используется для того, чтобы предотвратить блокировку вызова функции `recvfrom` на длительное время.

При использовании протокола, ориентированного на создание соединения, мы должны были подключиться к серверу до начала обмена данными. Для сервера было достаточно получить запрос на соединение, чтобы понять, что он должен обслужить клиента. Но при использовании протокола передачи дейтаграмм необходимо оповестить сервер о том, что мы хотим получить услугу. В этом примере мы просто посылаем серверу 1-байтное сообщение. Сервер принимает его, извлекает из пакета наш адрес и по этому адресу отправляет ответ. Если бы сервер предоставлял несколько услуг, то мы могли бы в запросе посылать идентификатор требуемой услуги, но поскольку наш сервер выполняет всего одну команду, содержимое 1-байтного сообщения не играет никакой роли.

Если сервер не запущен, клиент может оказаться заблокированным на неопределенное время в функции `recvfrom`. В предыдущем примере, ориентированном на создание соединения, функция `connect` возвращает управление с признаком ошибки, если сервер не отвечает. Во избежание блокировки на неопределенное время мы устанавливаем таймер перед вызовом функции `recvfrom`.

Пример – сервер, не ориентированный на создание соединения

В листинге 16.8 приводится исходный код версии сервера uptime, которая реализует обмен дейтаграммами.

Листинг 16.8. Сервер, который реализует службу uptime на основе обмена дейтаграммами

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUFLEN 128
#define MAXADDRLEN 256
#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, struct sockaddr *, socklen_t, int);
```

```
void
serve(int sockfd)
{
    int n;
    socklen_t alen;
    FILE *fp;
    char buf[BUFLEN];
    char abuf[MAXADDRLEN];

    for (;;) {
        alen = MAXADDRLEN;
        if ((n = recvfrom(sockfd, buf, BUFLEN, 0,
          (struct sockaddr *)abuf, &alen)) < 0) {
            syslog(LOG_ERR, "ruptimed: ошибка вызова функции recvfrom: %s",
                   strerror(errno));
            exit(1);
        }
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "ошибка: %s\n", strerror(errno));
            sendto(sockfd, buf, strlen(buf), 0,
                   (struct sockaddr *)abuf, alen);
        } else {
            if (fgets(buf, BUFLEN, fp) != NULL)
                sendto(sockfd, buf, strlen(buf), 0,
                       (struct sockaddr *)abuf, alen);
            pclose(fp);
        }
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int sockfd, err, n;
    char *host;

    if (argc != 1)
        err_quit("Использование: ruptimed");
#ifdef _SC_HOST_NAME_MAX
    n = sysconf(_SC_HOST_NAME_MAX);
    if (n < 0)        /* лучшее, что можно сделать */
#endif
        n = HOST_NAME_MAX;
    host = malloc(n);
    if (host == NULL)
        err_sys("ошибка вызова функции malloc");
    if (gethostname(host, n) < 0)
        err_sys("ошибка вызова функции gethostname");
    daemonize("ruptimed");
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
```

```
        hint.ai_socktype = SOCK_DGRAM;
        hint.ai_protocol = 0;
        hint.ai_addrlen = 0;
        hint.ai_canonname = NULL;
        hint.ai_addr = NULL;
        hint.ai_next = NULL;
        if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
            syslog(LOG_ERR, "ruptimed: ошибка вызова функции getaddrinfo: %s",
                   gai_strerror(err));
            exit(1);
        }
        for (aip = ailist; aip != NULL; aip = aip->ai_next) {
            if ((sockfd = initserver(SOCK_DGRAM, aip->ai_addr,
              aip->ai_addrlen, 0)) >= 0) {
                serve(sockfd);
                exit(0);
            }
        }
        exit(1);
    }
```

Сервер блокируется в функции `recvfrom` в ожидании прибытия запроса. Когда приходит запрос, сервер извлекает адрес отправителя и с помощью функции `popen` запускает команду `uptime`. Результат работы команды с помощью функции `sendto` отправляется клиенту по адресу, откуда пришел запрос.

16.6. Параметры сокетов

Механизм сокетов предоставляет две функции доступа к параметрам сокетов, которые управляют их поведением. Одна функция используется для изменения параметров, а другая возвращает текущие значения параметров. Мы можем получить и изменить три типа параметров:

1. Универсальные параметры, которые присущи всем типам сокетов.
2. Параметры, которые поддерживаются на уровне сокета, но зависят от используемого протокола.
3. Параметры, уникальные для каждого отдельно взятого протокола.

Стандарт Single UNIX Specification определяет только те параметры, которые поддерживаются на уровне сокетов (первые два типа параметров из предыдущего списка).

Изменить параметры сокета можно с помощью функции `setsockopt`.

```
#include <sys/socket.h>

int setsockopt(int sockfd, int level, int option, const void *val,
               socklen_t len);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Аргумент *level* определяет протокол, на который будет воздействовать параметр. Если параметр относится к разряду универсальных, в аргументе *level* передается значение SOL_SOCKET. В противном случае в аргумент *level* должен быть записан номер протокола, например IPPROTO_TCP для протокола TCP и IPPROTO_IP для протокола IP. В табл. 16.10 приводится перечень универсальных параметров, которые определены стандартом Single UNIX Specification.

Таблица 16.10. Параметры сокетов

Параметр	Тип аргумента *val*	Описание
SO_ACCEPTCONN	int	Определяет, находится ли сокет в режиме приема запросов на соединение (только для getsockopt).
SO_BROADCAST	int	Допускается передача широковещательных дейтаграмм, если значение *val не равно нулю.
SO_DEBUG	int	Сетевому драйверу разрешена запись отладочной информации, если значение *val не равно нулю.
SO_DONTROUTE	int	Передавать сообщения в обход процедуры маршрутизации, если значение *val не равно нулю.
SO_ERROR	int	Получить и сбросить значение ошибки, ожидающей обработки (только для getsockopt).
SO_KEEPALIVE	int	Разрешена периодическая передача служебных сообщений для поддержания соединения в активном состоянии, если значение *val не равно нулю.
SO_LINGER	struct linger	Время задержки закрытия сокета, если в нем имеются неотправленные сообщения.
SO_OOBINLINE	int	Экстренные сообщения помещаются во входной поток, если значение *val не равно нулю.
SO_RCVBUF	int	Размер приемного буфера в байтах.
SO_RCVLOWAT	int	Минимальный объем данных, который должен возвращаться функциями приема.
SO_RCVTIMEO	struct timeval	Максимальное время ожидания для операций чтения из сокета.
SO_REUSEADDR	int	Разрешает повторное использование локальных адресов функцией bind, если значение *val не равно нулю.
SO_SNDBUF	int	Размер буфера передачи в байтах.
SO_SNDTIMEO	struct timeval	Максимальное время ожидания для операций записи в сокет.
SO_TYPE	int	Тип сокета (только для getsockopt).

Аргумент *val* может содержать указатель на целое число или на структуру, в зависимости от параметра. Некоторые параметры являются флагами, которые могут иметь только два значения – *включено* и *выключено*. Если целое

число не равно нулю, то параметр включен. Если целое число равно нулю, то параметр выключен. Аргумент *len* определяет размер объекта, на который указывает *val*.

Текущие значения параметров можно получить с помощью функции getsockopt.

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int option, void *restrict val,
               socklen_t *restrict lenp);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Обратите внимание: аргумент *lenp* – это указатель на целое число. Перед вызовом функции getsockopt нужно установить это число равным размеру буфера, куда будет скопировано текущее значение параметра. Если фактический размер параметра больше размера буфера, параметр будет обрезан. Если фактический размер параметра меньше размера буфера, по адресу *lenp* будет записано фактическое значение размера параметра.

Пример

Функция, исходный код которой приводится в листинге 16.3, не в состоянии отработать должным образом, если попытаться перезапустить сервер сразу же после его завершения. Как правило, реализация протокола TCP не допускает присваивать тот же самый адрес другому сокету, пока не пройдет определенный промежуток времени, который обычно составляет несколько минут. К счастью, это ограничение легко обойти с помощью параметра SO_REUSEADDR, как показано в листинге 16.9.

Листинг 16.9. Инициализация сокета для сервера с разрешенной возможностью повторного использования адреса

```
#include "apue.h"
#include <errno.h>
#include <sys/socket.h>

int
initserver(int type, const struct sockaddr *addr, socklen_t alen, int qlen)
{
    int fd, err;
    int reuse = 1;

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &reuse,
      sizeof(int)) < 0) {
        err = errno;
        goto errout;
    }
    if (bind(fd, addr, alen) < 0) {
```

```
            err = errno;
            goto errout;
        }
        if (type == SOCK_STREAM || type == SOCK_SEQPACKET) {
            if (listen(fd, qlen) < 0) {
                err = errno;
                goto errout;
            }
        }
        return(fd);
        errout:
        close(fd);
        errno = err;
        return(-1);
    }
```

Чтобы разрешить возможность повторного использования адреса, необходимо установить параметр *SO_REUSEADDR*, для этого мы записываем в переменную ненулевое значение и передаем ее адрес функции `setsockopt` в аргументе *val*. В аргументе *len* передается размер целого числа, определяющего размер объекта, на который указывает *val*.

16.7. Экстренные данные

Передача экстренных данных – это необязательная функциональная возможность, поддерживаемая некоторыми протоколами, которая позволяет производить доставку высокоприоритетных данных. Экстренные данные отправляются в первую очередь. Протокол TCP поддерживает такую возможность, а UDP – нет. Интерфейс доступа к экстренным данным в сокетах очень тесно связан с реализацией экстренных данных в протоколе TCP.

Протокол TCP называет экстренные данные «urgent» (срочные). Он поддерживает только однобайтные срочные данные, но позволяет доставлять их в первую очередь. Чтобы послать срочные данные, нужно передать флаг `MSG_OOB` любой из трех функций `send`. Если с флагом `MSG_OOB` передается более одного байта срочных данных, то только последний байт в сообщении будет воспринят как срочные данные.

Можно предусмотреть посылку сигнала `SIGURG` при получении срочных данных. В разделах 3.14 и 14.6.2 мы уже видели, что для этого можно использовать команду `F_SETOWN` функции `fcntl`, которая назначает владельца дескриптора. Третий аргумент функции `fcntl` задает идентификатор процесса, если он является положительным числом, и группу процессов, если он является отрицательным числом, отличным от –1. Таким образом, мы можем указать, что процесс должен получать сигнал от сокета, вызвав

```
fcntl(sockfd, F_SETOWN, pid);
```

Чтобы узнать, какой процесс владеет сокетом, можно использовать команду `F_GETOWN`. Как и в случае команды `F_SETOWN`, отрицательное значение представ-

ляет идентификатор группы процессов, а положительное – идентификатор процесса. Таким образом, вызов

```
owner = fcntl(sockfd, F_GETOWN, 0);
```

запишет в переменную owner идентификатор процесса, который будет получать сигналы от сокета, если возвращаемое значение положительное, а если оно отрицательное, абсолютное значение переменной owner будет соответствовать идентификатору группы процессов, которая будет получать сигналы от сокета.

Протокол TCP поддерживает понятие *маркера срочности*: позиция в потоке обычных данных, куда помещаются срочные данные. Можно задать такой режим приема срочных данных, когда они размещаются в потоке обычных данных, для чего необходимо включить параметр SO_OOBINLINE. Проверить достижение маркера срочных данных можно с помощью функции sockatmark.

```
#include <sys/socket.h>

int sockatmark(int sockfd);
```

Возвращает 1, если маркер достигнут, 0 – если нет, –1 в случае ошибки

Если следующий доступный для чтения байт размещается в позиции маркера срочности, функция sockatmark вернет значение 1.

При наличии в сокете срочных данных функция select (раздел 14.5.1) возвратит дескриптор файла как имеющий исключительную ситуацию, ожидающую обработки. У нас есть возможность выбора: получать срочные данные в потоке обычных данных или же с помощью одной из функций recv, используя флаг MSG_OOB для выборки срочных данных в первую очередь. Протокол TCP помещает в очередь только один байт срочных данных. Если другой срочный байт прибудет до того, как мы получим текущий, существующий байт будет утерян.

16.8. Неблокирующий и асинхронный ввод-вывод

Если в сокете нет данных, доступных для чтения, выполнение функции recv обычно блокируется. Аналогичным образом блокируется и функция send, если в выходной очереди сокета не хватает места для отправляемого сообщения. Если сокет находится в неблокирующем режиме, поведение функций изменяется. В этом случае данные функции не блокируются, вместо этого они возвращают признак ошибки с кодом EWOULDBLOCK или EAGAIN в переменной errno. В такой ситуации для определения момента, когда можно будет принять или послать данные, мы можем использовать функцию poll или select.

Расширения реального времени в стандарте Single UNIX Specification включают поддержку обобщенного механизма асинхронного ввода-вывода. Механизм сокетов обрабатывает асинхронный ввод-вывод собственным способом, но он не стандартизован в Single UNIX Specification. В некоторых книгах

классический механизм асинхронного ввода-вывода для сокетов называется «ввод-вывод, основанный на сигналах», чтобы отличать его от механизма асинхронного ввода-вывода в расширениях реального времени.

При использовании механизма асинхронного ввода-вывода для сокетов мы можем организовать посылку сигнала SIGIO, когда в сокете появятся доступные для чтения данные или освободится место в выходной очереди. Процесс включения механизма асинхронного ввода-вывода состоит из двух действий:

1. Назначить владельца сокета так, чтобы сигнал доставлялся соответствующему процессу.
2. Информировать сокет о том, чтобы он посылал сигнал, когда выполнение операций ввода-вывода не будет блокировать процесс.

Первое действие можно выполнить тремя способами:

1. Воспользоваться командой F_SETOWN функции fcntl.
2. Воспользоваться командой FIOSETOWN функции ioctl.
3. Воспользоваться командой SIOCSPGRP функции ioctl.

Чтобы выполнить второе действие, есть две альтернативы:

1. С помощью команды F_SETFL функции fcntl установить флаг O_ASYNC.
2. Воспользоваться командой FIOASYNC функции ioctl.

Существует несколько вариантов, но они не универсальны. В табл. 16.11 показано, какие из вариантов на каких платформах поддерживаются. Точка свидетельствует о наличии поддержки, а знак «+» говорит о том, что поддержка зависит от домена сокета. Например, в ОС Linux отсутствует поддержка команд FIOSETOWN и SIOCSPGRP для сокетов домена UNIX.

Таблица 16.11. Команды управления режимом асинхронного ввода-вывода для сокетов

Механизм	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
fcntl(fd, F_SETOWN, pid)	•	•	•	•	•
ioctl(fd, FIOSETOWN, pid)		•	+	•	•
ioctl(fd, SIOCSPGRP, pid)		•	+	•	•
fcntl(fd, F_SETFL, flags|O_ASYNC)		•	•	•	
ioctl(fd, FIOASYNC, &n)		•	•	•	•

16.9. Подведение итогов

В этой главе мы рассмотрели механизмы IPC, которые позволяют процессам обмениваться данными с другими процессами, выполняющимися как на других машинах, так и на той же самой машине. Мы узнали, как назначить сокету адрес и как получить адреса, используемые для соединения с серверами.

Мы привели примеры клиентов и серверов, которые используют сокеты, ориентированные и не ориентированные (интерфейс дейтаграмм) на создание логического соединения. Коротко обсудили неблокирующий и асинхронный ввод-вывод для сокетов и функции, которые используются для работы с параметрами сокетов.

В следующей главе мы рассмотрим ряд более сложных тем, связанных с IPC, включая передачу дескрипторов файлов между процессами, выполняющимися на одной машине.

Упражнения

16.1. Напишите программу, которая определяла бы порядок байтов.

16.2. Напишите программу для вывода полей структуры `stat`, которые поддерживаются для сокетов. Запустите ее хотя бы на двух различных платформах и опишите, какие различия вы обнаружили.

16.3. Программа из листинга 16.5 предоставляет услугу только через один сокет. Модифицируйте программу таким образом, чтобы она поддерживала обслуживание через несколько сокетов (каждый из которых имеет свой адрес) одновременно.

16.4. Напишите программы клиента и сервера, с помощью которых можно было бы получать количество процессов, работающих на заданной машине.

16.5. В программе из листинга 16.6 сервер, прежде чем принять очередной запрос на соединение, ожидает, пока дочерний процесс запустит команду `uptime` и завершится. Перепишите сервер таким образом, чтобы он мог принимать входящие запросы на соединение без задержки.

16.6. Напишите две библиотечные процедуры, первая из которых должна включать асинхронный режим ввода-вывода для сокета, а вторая – выключать его. Воспользуйтесь табл. 16.11, чтобы обеспечить работоспособность этих функций на всех платформах и возможность обрабатывать как можно больше типов сокетов.

17

Расширенные возможности IPC

17.1. Введение

В предыдущих двух главах мы обсудили различные формы IPC, включая каналы и сокеты. В этой главе рассматриваются две дополнительные формы IPC – каналы на основе механизма STREAMS и сокеты домена UNIX. С помощью этих форм IPC процессы могут передавать друг другу открытые файловые дескрипторы, серверы могут присваивать имена своим файловым дескрипторам, а клиенты – использовать эти имена для взаимодействия с серверами. Кроме того, мы увидим, как операционная система может обеспечить уникальность канала связи с каждым клиентом. Большая часть идей, которые легли в основу описываемых здесь методик, была позаимствована из статьи [Presotto and Ritchie 1990].

17.2. Каналы на основе STREAMS

Каналы на основе механизма STREAMS (далее для краткости – каналы STREAMS) представляют собой двунаправленные (дуплексные) каналы. Чтобы получить двунаправленный поток данных между родительским и дочерним процессами, достаточно создать всего один канал STREAMS.

> В разделе 15.1 мы уже говорили, что поддержка каналов STREAMS имеется в ОС Solaris и, в виде дополнительного пакета, в Linux.

Рис. 17.1 демонстрирует два взгляда на канал STREAMS. Единственное отличие этого рисунка от рис. 15.1 состоит в том, что стрелки указывают в обе стороны, поскольку каналы STREAMS являются дуплексными, то есть данные могут перемещаться в обоих направлениях.

Если заглянуть внутрь канала STREAMS (рис. 17.2), мы увидим, что это просто головы двух потоков, у которых входная очередь (WQ, write queue) одного канала связана с выходной очередью (RQ, read queue) другого канала. Данные, записываемые во входную очередь одного канала, перемещаются в виде сообщений в выходную очередь другого канала.

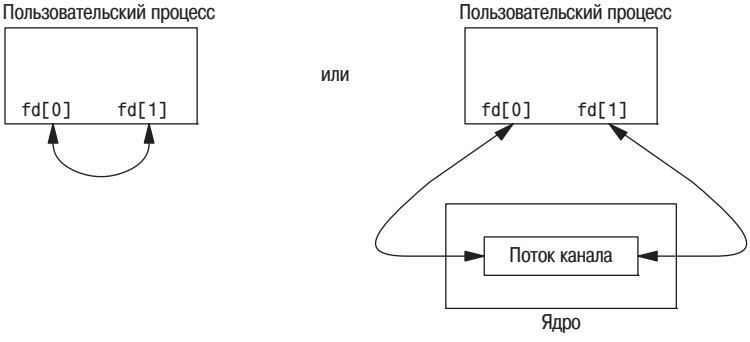


Рис. 17.1. *Два взгляда на канал STREAMS*

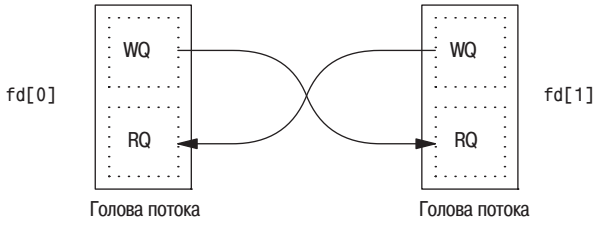


Рис. 17.2. *Внутреннее устройство канала STREAMS*

Поскольку каналы STREAMS являются потоками STREAMS, мы можем с любого конца канала добавлять модули STREAMS для промежуточной обработки данных, записываемых в канал (рис. 17.3). Но модули, добавленные в поток с одного конца, не могут быть удалены с другого конца. При необходимости удалить модуль сделать это возможно только с той стороны, с которой модуль был добавлен.

Если забыть на время о дополнительных особенностях каналов STREAMS, таких как возможность добавления промежуточных модулей, то можно сказать, что каналы STREAMS ведут себя подобно обычным каналам, за исключением поддержки специфических команд `ioctl`, описанных в `streamio`(7). В разделе 17.2.2 мы рассмотрим пример добавления в канал STREAMS модуля, который обеспечивает уникальность соединения, когда каналу присваивается имя в файловой системе.

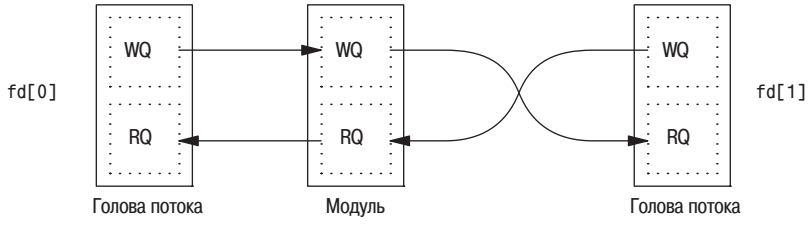


Рис. 17.3. *Дополнительный модуль внутри канала STREAMS*

Пример

Давайте вернемся к примеру сопроцесса из листинга 15.9 и перепишем его так, чтобы он использовал единственный канал STREAMS. В листинге 17.1 приводится новый вариант функции `main`. Сопроцесс `add2` остался без изменений (листинг 15.8). Канал STREAMS создается вызовом функции `s_pipe`. (Вскоре мы продемонстрируем обе версии этой функции – на основе каналов STREAMS и сокетов домена UNIX.)

Листинг 17.1. Программа, управляющая фильтром add2 с помощью канала STREAMS

```
#include "apue.h"

static void sig_pipe(int);              /* наш обработчик сигнала */

int
main(void)
{
    int n;
    int fd[2];
    pid_t pid;
    char line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("ошибка вызова функции signal");

    if (s_pipe(fd) < 0)                 /* необходим только один канал */
        err_sys("ошибка вызова функции pipe");

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) {               /* родительский процесс */
        close(fd[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd[0], line, n) != n)
                err_sys("ошибка записи в канал");
            if ((n = read(fd[0], line, MAXLINE)) < 0)
                err_sys("ошибка чтения из канала");
            if (n == 0) {
                err_msg("потомок закрыл канал ");
                break;
            }
            line[n] = 0;                /* завершающий нулевой символ */
            if (fputs(line, stdout) == EOF)
                err_sys("ошибка вызова функции fputs");
        }
        if (ferror(stdin))
            err_sys("ошибка чтения строки со стандартного ввода");
        exit(0);
    } else {                            /* дочерний процесс */
        close(fd[0]);
        if (fd[1] != STDIN_FILENO &&
```

```
            dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
              err_sys("ошибка вызова функции dup2 для stdin");
          if (fd[1] != STDOUT_FILENO &&
            dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
              err_sys("ошибка вызова функции dup2 для stdout");
          if (execl("./add2", "add2", (char *)0) < 0)
              err_sys("ошибка вызова функции execl");
      }
      exit(0);
  }

  static void
  sig_pipe(int signo)
  {
      printf("перехвачен сигнал SIGPIPE\n");
      exit(1);
  }
```

Родительский процесс использует только дескриптор `fd[0]`, а дочерний – только `fd[1]`. Так как каждый из концов канала STREAMS является дуплексным, родительский процесс читает и пишет в дескриптор `fd[0]`, а дочерний процесс связывает дескриптор `fd[1]` как со стандартным вводом, так и со стандартным выводом. На рис. 17.4 показаны получившиеся в итоге дескрипторы. Обратите внимание, что этот пример также может работать с дуплексными неименованными каналами, не основанными на механизме STREAMS, потому что он не использует никаких особенностей, присущих только каналам STREAMS, кроме их дуплексной природы.

> Более подробно каналы STREAMS описываются в [Rago 1993]. В табл. 15.1 указывается, что FreeBSD поддерживает дуплексные каналы, но они не основаны на механизме STREAMS.

Мы определили функцию `s_pipe` так, чтобы она была похожа на стандартную функцию `pipe`. Обе функции принимают один и тот же аргумент, но дескрипторы, возвращаемые функцией `s_pipe`, открыты как для записи, так и для чтения.

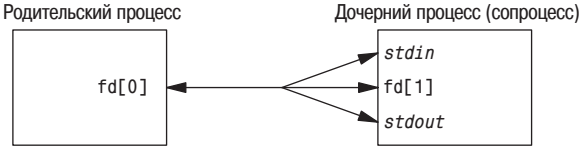


***Рис. 17.4.** Схема размещения дескрипторов для сопроцесса*

Пример – версия функции s_pipe на основе STREAMS

В листинге 17.2 приводится версия функции `s_pipe`, основанная на механизме STREAMS. Эта версия просто вызывает стандартную функцию `pipe`, которая создает дуплексный канал.

Листинг 17.2. Версия функции s_pipe на основе STREAMS

```
#include "apue.h"

/*
 * Возвращает канал STREAMS с двумя дескрипторами fd[0] и fd[1].
 */
int
s_pipe(int fd[2])
{
    return(pipe(fd));
}
```

17.2.1. Именованные каналы STREAMS

Обычно каналы могут использоваться только с процессами, которые связаны родственными отношениями: дочерний процесс наследует дескрипторы неименованных каналов от своего предка. В разделе 15.5 мы узнали, как можно организовать взаимодействие между независимыми процессами с помощью именованных каналов (FIFO), но они предоставляют возможность обмена только в одном направлении. Механизм STREAMS позволяет наделить неименованный канал именем в файловой системе. Это решает проблему однонаправленности именованных каналов FIFO.

Чтобы присвоить имя каналу STREAMS, используется функция `fattach`.

```
#include <stropts.h>

int fattach(int filedes, const char *path);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

В аргументе *path* должно передаваться имя существующего файла, а вызывающий процесс должен либо быть владельцем этого файла и обладать правом на запись в него, либо иметь привилегии суперпользователя.

После того как канал STREAMS будет связан с именем из пространства имен файловой системы, заданный файл станет недоступен. Любой процесс, который попытается открыть объект с данным именем, получит доступ к каналу, а не к самому файлу. Однако процессы, которые к моменту вызова `fattach` уже открыли этот файл, смогут продолжать работу с ним. В действительности эти процессы вообще не будут знать, что теперь это имя соответствует совсем другому объекту.

На рис. 17.5 показан канал, связанный с именем `/tmp/pipe`. Только один конец канала связывается с именем из файловой системы. Другой конец открывается процессами по заданному имени файла. Хотя таким образом можно связать с именем в файловой системе дескриптор любого объекта STREAMS, наиболее часто этот прием используется для присвоения имен именно каналам STREAMS.

Чтобы разорвать связь между именем и объектом STREAMS, используется функция `fdetach`.

```
#include <stropts.h>
int fdetach(const char *path);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

После вызова функции `fdetach` все процессы, которые удерживали именованный канал STREAMS открытым, смогут продолжать обмениваться через него данными, как ни в чем не бывало, но все последующие попытки открыть объект с именем *path* будут завершаться открытием оригинального файла с данным именем.

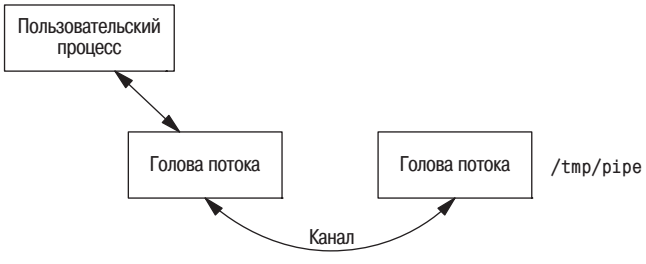


Рис. 17.5. Канал, связанный с именем из файловой системы

17.2.2. Уникальные соединения

Несмотря на возможность связать канал STREAMS с именем файла из файловой системы, по-прежнему существует проблема организации взаимодействий между одним сервером и несколькими клиентами посредством каналов STREAMS. Данные одного клиента могут смешиваться с данными другого клиента. Даже если клиенты не будут записывать в канал больше чем `PIPE_BUF` байт за одну операцию, что, безусловно, гарантирует атомарность записи, у нас по-прежнему нет возможности отправлять данные клиентам и гарантировать при этом, что данные будут прочитаны именно тем процессом, для которого они были предназначены. При наличии нескольких клиентов нельзя заранее сказать, какой из них первым получит управление и прочитает переданные данные.

Эта проблема решается с помощью модуля STREAMS – `connld`. Прежде чем связать канал с именем из файловой системы, серверный процесс должен добавить в канал модуль `connld` с того конца, которому будет присвоено имя. В результате будет получена конфигурация, изображенная на рис. 17.6.

На рис. 17.6 показано, что серверный процесс связал один конец канала с именем `/tmp/pipe`. Пунктирной стрелкой мы обозначили процесс открытия канала STREAMS со стороны клиента. После того как канал будет открыт, мы получим конфигурацию, изображенную на рис. 17.7.

Клиентский процесс никогда не сможет открыть дескриптор для того конца канала, который уже был открыт. Вместо этого при попытке открыть объект `/tmp/pipe` операционная система создаст новый канал и вернет один из кон-

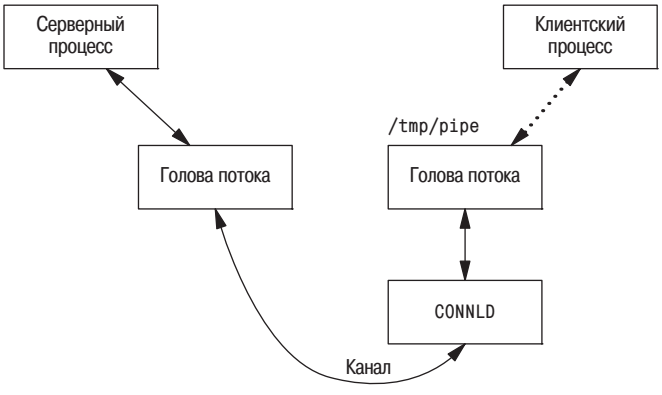


***Рис. 17.6.** Установка модуля connld для создания уникальных соединений*

цов клиенту. Другой конец вновь созданного канала система передаст серверному процессу через дескриптор существующего канала (связанного с данным именем из файловой системы). В результате будет получено уникальное соединение между серверным и клиентским процессами. Механизм передачи файловых дескрипторов через каналы STREAMS мы рассмотрим в разделе 17.4.1.

> Функция `fattach` построена на основе системного вызова `mount`. Поэтому именованные каналы STREAMS еще называют *смонтированными потоками*. Концепция смонтированных потоков и модуль `connld` были разработаны Пресотто (Presotto) и Ритчи (Ritchie) в 1990 году для системы Research UNIX. Позднее эти механизмы были перенесены в SVR4.

Теперь попробуем написать три функции, которые могли бы использоваться для создания уникальных соединений между независимыми процессами. Они будут имитировать описанные в разделе 16.4 функции для работы с сокетами, ориентированными на создание логических соединений. Сейчас мы реализуем эти функции на основе каналов STREAMS, а затем (в разделе 17.3)

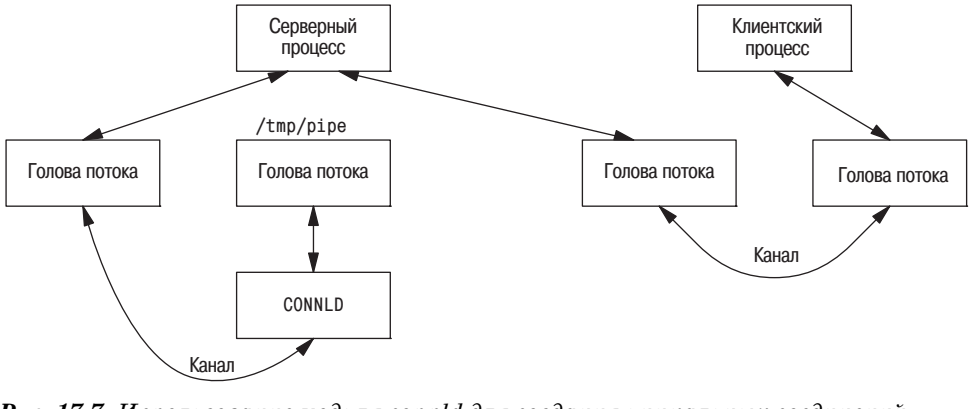


***Рис. 17.7.** Использование модуля connld для создания уникальных соединений*

мы рассмотрим альтернативные версии, реализованные на основе сокетов домена UNIX.

```
#include "apue.h"

int serv_listen(const char *name);
```

Возвращает в случае успеха файловый дескриптор, ожидающий запросы на соединение, отрицательное значение – в случае ошибки

```
int serv_accept(int listenfd, uid_t *uidptr);
```

Возвращает новый файловый дескриптор в случае успеха, отрицательное значение – в случае ошибки

```
int cli_conn(const char *name);
```

Возвращает файловый дескриптор в случае успеха, отрицательное значение – в случае ошибки

Сервер вызывает функцию `serv_listen` (листинг 17.3), чтобы объявить о своей готовности принимать от клиентов запросы на соединение через заранее предопределенное имя в файловой системе (полное имя некоторого файла). Клиенты будут использовать это имя для открытия соединения с сервером. Возвращаемое значение – дескриптор канала STREAMS на стороне сервера.

Листинг 17.3. Функция serv_listen, реализованная на основе каналов STREAMS

```
#include "apue.h"
#include <fcntl.h>
#include <stropts.h>

/* Права доступа к каналу: владелец rw, группа rw, остальные rw */
#define FIFO_MODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

/*
 * Устанавливает точку приема запросов на соединение.
 * Возвращает fd в случае успеха, <0 в случае ошибки.
 */
int
serv_listen(const char *name)
{
    int tempfd;
    int fd[2];

    /*
     * Создать файл - точку монтирования для функции fattach().
     */
    unlink(name);
    if ((tempfd = creat(name, FIFO_MODE)) < 0)
        return(-1);
    if (close(tempfd) < 0)
        return(-2);
    if (pipe(fd) < 0)
        return(-3);

    /*
```

```
     * Добавить модуль connld и вызвать fattach() для fd[1].
     */
    if (ioctl(fd[1], I_PUSH, "connld") < 0) {
        close(fd[0]);
        close(fd[1]);
        return(-4);
    }
    if (fattach(fd[1], name) < 0) {
        close(fd[0]);
        close(fd[1]);
        return(-5);
    }
    close(fd[1]);   /* функция fattach удерживает этот конец открытым */

    return(fd[0]);  /* fd[0] через этот дескриптор будут поступать */
                    /* запросы на соединение от клиентов */
}
```

Функция `serv_accept` (листинг 17.4) используется сервером для ожидания запросов на соединение от клиентов. При поступлении запроса система автоматически создаст новый канал STREAMS, и функция вернет один конец серверу. Кроме того, функция сохраняет эффективный идентификатор пользователя клиентского процесса по адресу `uidptr`.

Листинг 17.4. Функция serv_accept, реализованная на основе каналов STREAMS

```
#include "apue.h"
#include <stropts.h>

/*
 * Ожидает прибытия запросов на соединение и принимает их.
 * Кроме того, запоминает идентификатор пользователя клиентского процесса.
 * Возвращает новый fd в случае успеха, <0 в случае ошибки.
 */
int
serv_accept(int listenfd, uid_t *uidptr)
{
    struct strrecvfd recvfd;

    if (ioctl(listenfd, I_RECVFD, &recvfd) < 0)
        return(-1);          /* в случае перехвата сигнала может быть EINTR */
    if (uidptr != NULL)
        *uidptr = recvfd.uid;  /* эффективный uid вызывающего процесса */
    return(recvfd.fd);         /* вернуть новый дескриптор */
}
```

Чтобы подключиться к серверу, клиент вызывает функцию `cli_conn` (листинг 17.5). В аргументе *name* клиент должен указать то же самое имя, которое было объявлено сервером при обращении к функции `serv_listen`. В случае успешного завершения клиент получает файловый дескриптор канала STREAMS, соединенного с сервером.

Листинг 17.5. Функция cli_conn, реализованная на основе каналов STREAMS

```
#include "apue.h"
#include <fcntl.h>
```

```
#include <stropts.h>

/*
 * Создает клиентскую точку соединения и связывает ее с сервером.
 * Возвращает fd в случае успеха, <0 в случае ошибки.
 */
int
cli_conn(const char *name)
{
    int fd;

    /* открыть смонтированный поток */
    if ((fd = open(name, O_RDWR)) < 0)
        return(-1);
    if (isastream(fd) == 0) {
        close(fd);
        return(-2);
    }
    return(fd);
}
```

Проверка возвращаемого дескриптора на принадлежность к устройству STREAMS необходима на тот случай, если сервер не был запущен, а файл с заданным именем продолжает существовать в файловой системе. Порядок применения этих трех функций рассматривается в разделе 17.6.

17.3. Сокеты домена UNIX

Сокеты домена UNIX используются для организации взаимодействия процессов, работающих на одной и той же машине. Сокеты домена Интернет также могут служить для этих целей, но сокеты домена UNIX выполняют эту работу более эффективно. Сокеты домена UNIX просто копируют данные – они никак не обрабатывают сетевые протоколы, не удаляют и не добавляют никаких заголовков пакетов, не вычисляют контрольные суммы, не генерируют последовательные номера и не высылают подтверждения о приеме.

Сокеты домена UNIX реализуют как интерфейс дейтаграмм, так и интерфейс потоков. Однако интерфейс дейтаграмм домена UNIX гарантирует доставку пакета получателю. Сообщения никогда не будут утеряны или доставлены не в том порядке. Сокеты домена UNIX сочетают в себе особенности сокетов и неименованных каналов. Для взаимодействия с сокетом домена UNIX можно использовать интерфейс сетевого сокета или создать пару неименованных, связанных между собой сокетов домена UNIX с помощью функции `socketpair`.

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sockfd[2]);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Несмотря на то, что интерфейс выглядит достаточно общим, чтобы использовать функцию socketpair для создания сокетов произвольного домена, в большинстве операционных систем эта функция поддерживает только домен UNIX.

Пример – версия функции s_pipe на основе сокетов домена UNIX

В листинге 17.6 приводится версия функции s_pipe (из листинга 17.2), реализованная на основе сокетов. Она создает пару соединенных между собой сокетов домена UNIX.

Листинг 17.6. Версия функции s_pipe, основанная на сокетах

```
#include "apue.h"
#include <sys/socket.h>

/*
* Возвращает дуплексный канал (сокет домена UNIX)
* с двумя файловыми дескрипторами fd[0] и fd[1].
*/
int
s_pipe(int fd[2])
{
    return(socketpair(AF_UNIX, SOCK_STREAM, 0, fd));
}
```

> Некоторые системы, происходящие от BSD, используют сокеты домена UNIX для реализации неименованных каналов. Но при вызове функции pipe открытый для записи конец первого дескриптора и открытый для чтения конец второго дескриптора закрываются. Чтобы получить дуплексный канал, необходимо напрямую вызвать функцию socketpair.

17.3.1. Именованные сокеты домена UNIX

Хотя функция socketpair и создает соединенные друг с другом сокеты, но у них нет имен. Это означает, что они не могут использоваться для взаимодействия независимых процессов.

В разделе 16.3.4 мы узнали, как присваиваются адреса сокетам домена Интернет. Точно так же можно присваивать адреса сокетам домена UNIX и использовать их для получения запросов на соединение. Однако формат адреса сокетов домена UNIX отличается от формата адреса сокетов домена Интернет.

В разделе 16.3 мы уже говорили о том, что форматы адресов сокетов отличаются в различных реализациях. Адрес сокетов домена UNIX представлен структурой sockaddr_un. В ОС Linux 2.4.22 и Solaris 9 структура sockaddr_un объявлена в заголовочном файле <sys/un.h> следующим образом:

```
struct sockaddr_un {
    sa_family_t sun_family;    /* AF_UNIX */
    char        sun_path[108]; /* полное имя */
};
```

Однако в FreeBSD 5.2.1 и Mac OS X 10.3 структура `sockaddr_un` объявлена несколько иначе:

```
struct sockaddr_un {
    unsigned char sun_len;      /* длина, включая завершающий нулевой символ */
    sa_family_t   sun_family;   /* AF_UNIX */
    char          sun_path[104]; /* полное имя */
};
```

Поле `sun_path` содержит полное имя файла. Присваивая имя сокету домена UNIX, система создает файл типа `S_IFSOCK` с этим именем.

Этот файл существует только для того, чтобы сообщить имя сокета клиентам. Сам файл не может быть открыт или как-то иначе использован для взаимодействия приложений.

Если во время попытки присвоить имя сокету файл уже существует, вызов функции `bind` завершается с признаком ошибки. При закрытии сокета этот файл не удаляется автоматически, поэтому мы должны сами побеспокоиться о его удалении перед завершением приложения.

Пример

Программа, представленная листингом 17.7, демонстрирует порядок присваивания адреса сокету домена UNIX.

Листинг 17.7. Присваивание адреса сокету домена UNIX

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>

int
main(void)
{
    int fd, size;
    struct sockaddr_un un;

    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, "foo.socket");
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        err_sys("ошибка вызова функции socket");
    size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, size) < 0)
        err_sys("ошибка вызова функции bind");
    printf("имя сокету домена UNIX присвоено\n");
    exit(0);
}
```

Когда эта программа будет запущена в первый раз, вызов функции `bind` завершится успехом, но если запустить программу повторно, мы получим сообщение об ошибке, потому что файл с заданным именем уже существует. Программа будет терпеть неудачу до тех пор, пока файл не будет удален.

```
$ ./a.out                                   запустить программу
имя сокету домена UNIX присвоено
$ ls -l foo.socket                          проверить наличие файла сокета
srwxrwxr-x 1 sar 0 Aug 22 12:43 foo.socket
$ ./a.out                                   попытаться запустить программу
ошибка вызова функции bind: Address already in use
$ rm foo.socket                             удалить файл сокета
$ ./a.out                                   запустить программу в третий раз
имя сокету домена UNIX присвоено            теперь опять все в порядке
```

Чтобы получить размер адреса мы определяем смещение поля sun_path в структуре sockaddr_un и добавляем к нему длину имени без учета завершающего нулевого символа. Так как в различных реализациях полю sun_path в структуре sockaddr_un может предшествовать разное количество полей, для определения его смещения от начала структуры мы воспользовались макроопределением offsetoff, которое объявлено в заголовочном файле <stddef.h> (подключается в заголовочном файле apue.h). В файле <stddef.h> вы найдете примерно такое определение:

```
#define offsetof(TYPE, MEMBER) ((int)&((TYPE *)0)->MEMBER)
```

Это выражение вычисляет адрес начала заданного поля при условии, что сама структура начинается с адреса 0.

17.3.2. Уникальные соединения

Сервер может создавать через сокеты домена UNIX уникальные соединения с клиентами, используя для этого стандартные функции bind, listen и accept. Для соединения с сервером клиент может использовать функцию connect; после того как сервер примет запрос на соединение, между клиентом и сервером будет установлено уникальное соединение. Этот способ взаимодействия аналогичен тому, что использовался в листингах 16.4 и 16.5.

В листинге 17.8 приводится версия функции serv_listen, основанная на сокетах домена UNIX.

Листинг 17.8. Функция serv_listen, реализованная на основе сокетов домена UNIX

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>

#include <errno.h>
#define QLEN 10

/*
 * Создает точку соединения на стороне сервера.
 * Возвращает fd в случае успеха, <0 в случае ошибки.
 */
int
serv_listen(const char *name)
{
    int fd, len, err, rval;
```

```
    struct sockaddr_un un;

    /* создать сокет домена UNIX типа SOCK_STREAM */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);
    unlink(name); /* если name уже существует */

    /* заполнить структуру с адресом */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    /* присвоить имя дескриптору */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -2;
        goto errout;
    }

    if (listen(fd, QLEN) < 0) { /* сообщить ядру, что процесс является сервером */
        rval = -3;
        goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(rval);
}
```

Сначала с помощью функции `socket` создается сокет домена UNIX. Затем в структуру `sockaddr_un` заносится предопределенное имя, которое будет связано с сокетом. Эта структура будет служить аргументом функции `bind`. Обратите внимание: не нужно записывать значение в поле `sun_len`, которое присутствует на некоторых платформах, потому что операционная система сама сделает это, используя аргумент длины адреса, передаваемый функции `bind`.

В заключение вызывается функция `listen` (раздел 16.4), которая сообщает ядру, что процесс будет выступать в роли сервера, ожидая запросов на соединение от клиентов. Чтобы принять запрос на соединение, сервер должен вызвать функцию `serv_accept` (листинг 17.9).

Листинг 17.9. Функция serv_accept, реализованная на основе сокетов домена UNIX

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <time.h>
#include <errno.h>

#define STALE 30    /* имя, используемое клиентом, не должно быть старше */
                    /* этого значения (сек) */
```

```
/*
 * Дождаться запроса на соединение от клиента и принять его.
 * Мы также получаем идентификатор пользователя клиента из характеристик файла,
 * который должен быть связан с сокетом перед вызовом сервера.
 * Возвращает новый fd в случае успеха, <0 в случае ошибки.
 */
int
serv_accept(int listenfd, uid_t *uidptr)
{
    int clifd, len, err, rval;
    time_t staletime;
    struct sockaddr_un un;
    struct stat statbuf;

    len = sizeof(un);
    if ((clifd = accept(listenfd, (struct sockaddr *)&un, &len)) < 0)
        return(-1); /* чаще всего errno=EINTR, если был перехвачен сигнал */

    /* получить идентификатор пользователя клиента из имени файла */
    len -= offsetof(struct sockaddr_un, sun_path); /* длина имени */
    un.sun_path[len] = 0;  /* завершающий нулевой символ */
    if (stat(un.sun_path, &statbuf) < 0) {
        rval = -2;
        goto errout;
    }

#ifdef S_ISSOCK /* не определен в SVR4 */
    if (S_ISSOCK(statbuf.st_mode) == 0) {
        rval = -3;          /* это не сокет */
        goto errout;
    }
#endif
    if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
      (statbuf.st_mode & S_IRWXU) != S_IRWXU) {
        rval = -4;          /* не rwx------ */
        goto errout;
    }

    staletime = time(NULL) - STALE;
    if (statbuf.st_atime < staletime ||
      statbuf.st_ctime < staletime ||
      statbuf.st_mtime < staletime) {
        rval = -5;          /* индексный узел слишком стар */
        goto errout;
    }

    if (uidptr != NULL)
        *uidptr = statbuf.st_uid; /* вернуть идентификатор пользователя клиента */
    unlink(un.sun_path);        /* работа с файлом закончена */
    return(clifd);

errout:
    err = errno;
    close(clifd);
    errno = err;
```

```
    return(rval);
}
```

Сервер блокируется в функции accept, ожидая, пока клиент не вызовет функцию cli_conn. Когда accept вернет управление, мы получим от нее дескриптор, который будет соединен с клиентом. (Примерно то же самое делает модуль connld в подсистеме STREAMS.) Кроме того, имя файла, которое клиент присвоит своему сокету (содержащее идентификатор клиентского процесса), также будет возвращено функцией accept во втором аргументе (указатель на структуру sockaddr_un). Далее мы добавляем завершающий нулевой символ в конец имени файла и вызываем функцию stat. Это даст нам возможность убедиться в том, что файл действительно является сокетом, и права доступа к нему разрешают чтение, запись и исполнение только для владельца файла. Кроме того, мы проверяем три значения времени, которые не должны превосходить текущее время более чем на 30 секунд. (В разделе 6.10 мы уже упоминали, что функция time возвращает текущее время и дату в виде количества секунд, прошедших с начала Эпохи.)

Если все эти проверки увенчались успехом, мы предполагаем, что клиент действительно является владельцем сокета. Хотя эти проверки далеки от совершенства, но это лучшее, что можно сделать в современных системах. (Было бы лучше, если бы эффективный идентификатор пользователя возвращался ядром через функцию accept, как это делает команда I_RECVFD функции ioctl.)

Клиент инициирует соединение с сервером, вызывая функцию cli_conn (листинг 17.10).

Листинг 17.10. Функция cli_conn, реализованная на основе сокетов домена UNIX

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define CLI_PATH "/var/tmp/" /* +5 для идентификатора процесса = 14 символов */
#define CLI_PERM S_IRWXU     /* rwx только для владельца */

/*
 * Создать точку соединения на стороне клиента и соединить ее с сервером.
 * Возвращает fd в случае успеха, <0 в случае ошибки.
 */
int
cli_conn(const char *name)
{
    int fd, len, err, rval;
    struct sockaddr_un un;

    /* создать сокет домена UNIX типа SOCK_STREAM */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    /* заполнить структуру с адресом */
    memset(&un, 0, sizeof(un));
```

```
    un.sun_family = AF_UNIX;
    sprintf(un.sun_path, "%s%05d", CLI_PATH, getpid());
    len = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
    unlink(un.sun_path);      /* если файл уже существует */

    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -2;
        goto errout;
    }

    if (chmod(un.sun_path, CLI_PERM) < 0) {
        rval = -3;
        goto errout;
    }

    /* записать в структуру адрес сервера */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    if (connect(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -4;
        goto errout;
    }

    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(rval);
}
```

Для создания сокета на стороне клиента вызывается функция `socket`. После этого в структуру `sockaddr_un` заносится имя файла, созданного клиентом.

Мы не можем позволить системе выбирать адрес по умолчанию, поскольку в этом случае сервер не сможет отличить одного клиента от другого. Вместо этого мы присваиваем сокету указанный нами адрес, что обычно не делается при разработке клиентских приложений с использованием сокетов.

Последние пять символов имени файла отводятся под идентификатор процесса клиента. На всякий случай мы вызываем функцию `unlink`, чтобы гарантировать отсутствие сгенерированного имени в файловой системе. Затем вызывается функция `bind`, которая связывает сокет с заданным именем. При этом она создает файл сокета с тем же именем. После этого функция `chmod` устанавливает права доступа к файлу таким образом, чтобы право на чтение, запись и исполнение имел только владелец файла. Функция `serv_accept`, запускаемая сервером, проверяет эти права и идентифицирует клиента по идентификатору пользователя сокета.

Затем мы заполняем вторую структуру `sockaddr_un`, записывая в нее предопределенное имя сокета сервера. В завершение вызывается функция `connect`, которая инициирует соединение с сервером.

17.4. Передача дескрипторов файлов

Способность передавать дескрипторы файлов между процессами дает неоценимую возможность по-иному подойти к разработке архитектуры клиент-серверных приложений. Она позволяет одному из процессов (обычно серверу) выполнять все действия, связанные с открытием файла (включая преобразование сетевых имен в адреса, соединение через модем, установку блокировок на файл и тому подобное), и возвращать вызывающему процессу дескриптор, который может быть использован в операциях ввода-вывода. Такой подход помогает скрыть от клиента все сложные механизмы, связанные с открытием файла.

Нам необходимо уточнить смысл понятия «передача открытого дескриптора файла от одного процесса другому». На рис. 3.2 показан случай, когда два процесса открыли один и тот же файл. Хотя они и используют один и тот же виртуальный узел (v-node), но при этом каждый из процессов обращается к нему через свою собственную запись в таблице файлов.

Когда открытый дескриптор файла передается от одного процесса другому, необходимо, чтобы передающий и принимающий процессы совместно использовали одну и ту же запись в таблице файлов. На рис. 17.8 показано, что мы хотим получить.

Технически нам нужно передать указатель на запись в таблице открытых файлов от одного процесса другому. Этот указатель должен быть связан

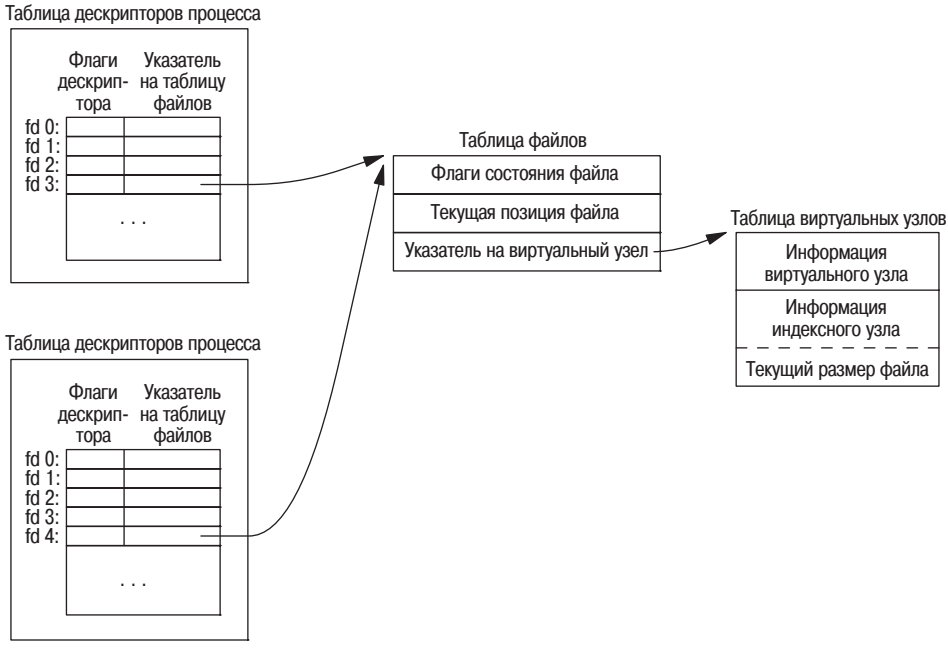


Рис. 17.8. *Передача открытого дескриптора файла от верхнего процесса нижнему*

с первым доступным дескриптором в принимающем процессе. (Выражение «передача открытого дескриптора файла» создает ошибочное впечатление, что номер дескриптора в принимающем процессе должен совпадать с номером дескриптора в передающем процессе – такое возможно, но это не всегда верно.) Ситуация, когда два процесса совместно используют одну и ту же запись в таблице файлов, в точности совпадает с ситуацией, возникающей после вызова функции `fork` (рис. 8.1).

Как правило, после передачи дескриптора от одного процесса другому передающий процесс закрывает свой дескриптор. Закрытие дескриптора в этом случае не означает закрытие файла или устройства, так как считается, что файл остается открытым в принимающем процессе (даже если принимающий процесс еще не успел получить дескриптор).

Сейчас мы определим три функции, которые будут использоваться в этой главе для передачи и приема дескрипторов файлов. Далее в этом разделе мы продемонстрируем реализации этих трех функций, основанные как на механизме STREAMS, так и на сокетах.

```
#include "apue.h"

int send_fd(int fd, int fd_to_send);

int send_err(int fd, int status, const char *errmsg);
```

Обе возвращают 0 в случае успеха, –1 в случае ошибки

```
int recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t));
```

Возвращает дескриптор файла в случае успеха, отрицательное значение – в случае ошибки

Процесс (обычно сервер), который желает передать дескриптор другому процессу, вызывает `send_fd` или `send_err`. Процесс, ожидающий получения дескриптора (обычно клиент), вызывает `recv_fd`.

Функция `send_fd` передает дескриптор *fd_to_send* посредством канала STREAMS или сокета домена UNIX, представленного дескриптором *fd*.

> Для обозначения двунаправленного канала, который может быть реализован как на базе каналов STREAMS, так и на базе сокетов домена UNIX, мы будем использовать термин *s-pipe*.

Функция `send_err` передает сообщение об ошибке *errmsg*, сопровождаемое кодом ошибки *status*. Значение аргумента *status* должно находиться в диапазоне от –1 до –255.

Для приема дескриптора клиент вызывает функцию `recv_fd`. Если все в порядке (передающий процесс обратился к функции `send_fd`), функция вернет клиенту неотрицательный дескриптор. В противном случае возвращаемое значение будет представлять собой код ошибки *status*, отправленный функцией `send_err` (отрицательное число в диапазоне от –1 до –255). Кроме того, если сервер передал сообщение об ошибке, для его обработки будет вызвана кли-

ентская функция *userfunc*. В первом аргументе этой функции передается константа STDERR_FILENO, во втором – указатель на строку сообщения, а в третьем – длина сообщения. Возвращаемое значение *userfunc* – количество записанных байт или отрицательный код ошибки. Часто клиентские приложения используют стандартную функцию write в качестве *userfunc*.

Мы реализуем собственный протокол, который будет использоваться этими тремя функциями. Чтобы передать дескриптор, функция send_fd посылает два байта со значением 0, за которыми следует фактический дескриптор. Чтобы передать сообщение об ошибке, функция send_err посылает строку *errmsg*, за которой следуют байт со значением 0 и абсолютное значение байта с кодом ошибки (1–255). Функция recv_fd читает все, что поступает по каналу s-pipe, до тех пор, пока не встретится нулевой байт. Все символы, прочитанные до этого момента, передаются функции *userfunc*. Следующий байт, который будет прочитан функцией recv_fd, – это код ошибки. Если он равен 0, следовательно, был передан дескриптор файла, в противном случае дескриптор не был получен.

После записи сообщения об ошибке в канал s-pipe функция send_err вызывает send_fd. Исходный текст функции send_err приводится в листинге 17.11.

Листинг 17.11 Функция send_err

```
#include "apue.h"
/*
 * Эта функция используется для передачи сообщения об ошибке с помощью протокола
 * send_fd()/recv_fd(), если при передаче дескриптора возникла ошибка.
 */
int
send_err(int fd, int errcode, const char *msg)
{
    int n;

    if ((n = strlen(msg)) > 0)
        if (writen(fd, msg, n) != n) /* передать сообщение об ошибке */
            return(-1);

    if (errcode >= 0)
        errcode = -1;    /* код ошибки должен быть отрицательным числом */

    if (send_fd(fd, errcode) < 0)
        return(-1);
    return(0);
}
```

В следующем разделе мы рассмотрим реализацию функций send_fd и recv_fd.

17.4.1. Передача дескрипторов с помощью каналов STREAMS

Обмен дескрипторами посредством каналов STREAMS производится с помощью двух команд ioctl: I_SENDFD и I_RECVFD. Чтобы отправить дескриптор,

нужно передать его функции ioctl в третьем аргументе. Это показано в листинге 17.12.

Листинг 17.12 Функция send_fd, реализованная на основе каналов STREAMS

```
#include "apue.h"
#include <stropts.h>

/*
 * Передает дескриптор файла другому процессу.
 * Если fd<0, то в качестве кода ошибки, отправляется -fd.
 */
int
send_fd(int fd, int fd_to_send)
{
    char buf[2]; /* 2-байтный протокол send_fd()/recv_fd() */

    buf[0] = 0;  /* нулевой байт - флаг для recv_fd() */
    if (fd_to_send < 0) {
        buf[1] = -fd_to_send; /* ненулевое значение означает наличие ошибки */
        if (buf[1] == 0)
            buf[1] = 1; /* протокол преобразует в -256 */
    } else {
        buf[1] = 0;     /* нулевое значение означает отсутствие ошибки */
    }

    if (write(fd, buf, 2) != 2)
        return(-1);
    if (fd_to_send >= 0)
        if (ioctl(fd, I_SENDFD, fd_to_send) < 0)
            return(-1);
    return(0);
}
```

При получении дескриптора в третьем аргументе функции ioctl будет возвращен указатель на структуру strrecvfd:

```
struct strrecvfd {
    int   fd;    /* новый дескриптор */
    uid_t uid; /* эффективный идентификатор пользователя передающего процесса */
    gid_t gid; /* эффективный идентификатор группы передающего процесса */
    char  fill[8];
};
```

Функция recv_fd считывает данные из канала STREAMS до тех пор, пока не будет принят первый байт 2-байтного протокола (нулевой байт). К моменту вызова функции ioctl с командой I_RECVFD следующее сообщение в очереди чтения головы потока должно быть дескриптором, посланным командой I_SENDFD, в противном случае произойдет ошибка. Исходный текст функции приводится в листинге 17.13.

Листинг 17.13. Функция recv_fd, реализованная на основе каналов STREAMS

```
#include "apue.h"
#include <stropts.h>
```

```
/*
 * Принимает дескриптор файла от другого процесса (сервера).
 * Кроме того, любые данные, принятые от сервера, передаются
 * функции (*userfunc)(STDERR_FILENO, buf, nbytes). Чтобы принять дескриптор,
 * мы должны соблюдать 2-байтный протокол.
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int newfd, nread, flag, status;
    char *ptr;
    char buf[MAXLINE];
    struct strbuf dat;
    struct strrecvfd recvfd;

    status = -1;
    for ( ; ; ) {
        dat.buf = buf;
        dat.maxlen = MAXLINE;
        flag = 0;
        if (getmsg(fd, NULL, &dat, &flag) < 0)
            err_sys("ошибка вызова функции getmsg");
        nread = dat.len;
        if (nread == 0) {
            err_ret("соединение закрыто сервером");
            return(-1);
        }

        /*
         * Проверить, являются ли два последних байта нулевым байтом
         * и кодом ошибки. Нулевой байт должен быть предпоследним,
         * а код ошибки - последним байтом в буфере.
         * Нулевой код ошибки означает, что мы должны принять дескриптор.
         */
        for (ptr = buf; ptr < &buf[nread]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nread-1])
                    err_dump("нарушение формата сообщения");
                status = *ptr & 0xFF; /* предотвратить расширение знакового бита */
                if (status == 0) {
                    if (ioctl(fd, I_RECVFD, &recvfd) < 0)
                        return(-1);
                    newfd = recvfd.fd; /* новый дескриптор */
                } else {
                    newfd = -status;
                }
                nread -= 2;
            }
        }

        if (nread > 0)
            if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
                return(-1);
```

```
        if (status >= 0)   /* доставлены заключительные данные */
            return(newfd); /* дескриптор или код ошибки */
    }
}
```

17.4.2. Передача дескрипторов с помощью сокетов домена UNIX

Для передачи дескрипторов файлов между процессами с помощью сокетов домена UNIX можно использовать функции recvmsg(2) и sendmsg(2) (раздел 16.5). Обе функции принимают указатель на структуру msghdr, которая содержит всю необходимую информацию о том, что передается и принимается. Эта структура может выглядеть примерно таким образом:

```
struct msghdr {
    void         *msg_name;       /* необязательный адрес */
    socklen_t     msg_namelen;    /* размер адреса в байтах */
    struct iovec *msg_iov;        /* массив буферов ввода-вывода */
    int           msg_iovlen;     /* количество элементов в массиве */
    void         *msg_control;    /* вспомогательные данные */
    socklen_t     msg_controllen; /* объем вспомогательной информации в байтах */
    int           msg_flags;      /* флаги принятого сообщения */
};
```

Первые два поля обычно используются при передаче дейтаграмм через сетевое соединение, когда адрес назначения можно указать для каждой дейтаграммы. Следующие два поля позволяют определить массив буферов ввода-вывода, как для функций readv и writev (чтение вразброс и запись со слиянием, раздел 14.7). Поле msg_flags содержит флаги, описывающие принятое сообщение (перечень флагов был приведен в табл. 16.9).

Два поля имеют отношение к передаче и приему управляющей информации. Поле msg_control содержит указатель на структуру cmsghdr (заголовок блока управляющей информации), а msg_controllen – количество байт управляющей информации.

```
struct cmsghdr {
    socklen_t cmsg_len;   /* количество байт данных, включая заголовок */
    int       cmsg_level; /* определяет протокол */
    int       cmsg_type;  /* тип управляющей информации */
    /* далее следует управляющая информация */
};
```

Чтобы передать дескриптор, необходимо записать в поле cmsg_len размер структуры cmsghdr плюс размер целого числа (дескриптора). В поле cmsg_level записывается значение SOL_SOCKET, а в поле cmsg_type – значение SCM_RIGHTS, которое указывает на то, что передаются права доступа. (Аббревиатура SCM означает *socket-level control message* – управляющее сообщение уровня сокетов.) Права доступа могут передаваться только через сокеты домена UNIX. Дескриптор следует сразу же за полем cmsg_type, а чтобы получить указатель на него, можно воспользоваться макросом CMSG_DATA.

Три макроса используются для доступа к управляющей информации и один – для вычисления значения, которое заносится в поле `cmsg_len`.

```
#include <sys/socket.h>

unsigned char *CMSG_DATA(struct cmsghdr *cp);
```

Возвращает указатель на данные, связанные со структурой `cmsghdr`

```
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mp);
```

Возвращает указатель на первую структуру `cmsghdr`, связанную со структурой `msghdr`, или `NULL`, если таковых не существует

```
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mp,
                            struct cmsghdr *cp);
```

Возвращает указатель на следующую структуру `cmsghdr`, связанную со структурой `msghdr`, относительно заданной структуры `cmsghdr` или `NULL`, если таковой не существует

```
unsigned int CMSG_LEN(unsigned int nbytes);
```

Возвращает объем памяти, который необходимо выделить для хранения объекта размером *nbytes*

В Single UNIX Specification определены первые три макроса, но отсутствует `CMSG_LEN`.

Макрос `CMSG_LEN` возвращает количество байт, необходимое для хранения данных объемом *nbytes* после добавления размера структуры `cmsghdr` с учетом всех ограничений по выравниванию полей, накладываемых аппаратной архитектурой процессора.

В листинге 17.14 приводится исходный код функции `send_fd`, реализованной на основе сокетов домена UNIX.

Листинг 17.14. Функция send_fd, реализованная на основе сокетов домена UNIX

```
#include "apue.h"
#include <sys/socket.h>

/* размер буфера с управляющей информацией для приема/передачи одного дескриптора */
#define CONTROLLEN CMSG_LEN(sizeof(int))

static struct cmsghdr *cmptr = NULL; /* размещается при первом вызове */

/*
 * Передает дескриптор файла другому процессу.
 * Если fd<0, то в качестве кода ошибки, отправляется -fd.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct iovec iov[1];
    struct msghdr msg;
    char buf[2]; /* 2-байтный протокол send_fd()/recv_fd() */
```

```
    iov[0].iov_base = buf;
    iov[0].iov_len = 2;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;
    msg.msg_name = NULL;
    msg.msg_namelen = 0;

    if (fd_to_send < 0) {
        msg.msg_control = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send; /* ненулевое значение означает ошибку */
        if (buf[1] == 0)
            buf[1] = 1;       /* протокол преобразует в -256 */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level = SOL_SOCKET;
        cmptr->cmsg_type = SCM_RIGHTS;
        cmptr->cmsg_len = CONTROLLEN;
        msg.msg_control = cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd_to_send; /* записать дескриптор */
        buf[1] = 0;    /* нулевое значение означает отсутствие ошибки */
    }
    buf[0] = 0;        /* нулевой байт - флаг для recv_fd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Функции `sendmsg` передаются как данные протокола (нулевой байт и код ошибки), так и дескриптор.

Чтобы принять дескриптор (листинг 17.15), мы выделяем достаточный объем памяти для размещения структуры `cmsghdr` и дескриптора, затем помещаем в поле `msg_control` указатель на выделенную память и вызываем функцию `recvmsg`. Для расчета объема выделяемого пространства мы воспользовались макросом `CMSG_LEN`.

Чтение данных из сокета производится до тех пор, пока не будет получен нулевой байт, предшествующий заключительному байту с кодом ошибки. Все, что было получено до этого байта, рассматривается как сообщение об ошибке от отправителя. Это демонстрирует листинг 17.15.

Листинг 17.15. Функция recv_fd, реализованная на основе сокетов домена UNIX

```
#include "apue.h"
#include <sys/socket.h> /* struct msghdr */

/* размер буфера с управляющей информацией для приема/передачи одного дескриптора */
#define CONTROLLEN CMSG_LEN(sizeof(int))

static struct cmsghdr *cmptr = NULL; /* размещается при первом вызове */
```

```
/*
 * Принимает дескриптор файла от серверного процесса. Кроме того, любые
 * принятые данные передаются функции (*userfunc)(STDERR_FILENO, buf, nbytes).
 * Чтобы принять дескриптор, мы должны соблюдать 2-байтный протокол.
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int newfd, nr, status;
    char *ptr;
    char buf[MAXLINE];
    struct iovec iov[1];
    struct msghdr msg;

    status = -1;
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len = sizeof(buf);
        msg.msg_iov = iov;
        msg.msg_iovlen = 1;
        msg.msg_name = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
            err_sys("ошибка вызова функции recvmsg");
        } else if (nr == 0) {
            err_ret("соединение закрыто сервером");
            return(-1);
        }
        /*
         * Проверить, являются ли два последних байта нулевым байтом
         * и кодом ошибки. Нулевой байт должен быть предпоследним,
         * а код ошибки - последним байтом в буфере.
         * Нулевой код ошибки означает, что мы должны принять дескриптор.
         */
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("нарушение формата сообщения");
                status = *ptr & 0xFF; /* предотвратить расширение знакового бита */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("получен код 0, но отсутствует fd");
                    newfd = *(int *)CMSG_DATA(cmptr);
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
```

```
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)     /* доставлены заключительные данные */
            return(newfd);   /* дескриптор или код ошибки */
    }
}
```

Обратите внимание: перед каждым вызовом `recvmsg` мы готовимся к приему дескриптора (устанавливаем поля `msg_control` и `msg_controllen`), но действительно принимаем дескриптор только тогда, когда в поле `msg_contollen` после возвращения из функции содержится значение, отличное от нуля.

Единственное различие при получении дескрипторов посредством каналов STREAMS и сокетов домена UNIX заключается в том, что при использовании каналов STREAMS мы получаем информацию, идентифицирующую передающий процесс. Некоторые версии сокетов домена UNIX предоставляют аналогичную функциональность, но их интерфейсы различаются в разных реализациях.

> ОС Linux 2.4.22 и FreeBSD 5.2.1 поддерживают передачу идентификационной информации о процессе через сокеты домена UNIX, но эта возможность реализована по-разному. Mac OS X 10.3 является производной от FreeBSD, однако передача идентификационных сведений в ней отключена. Solaris 9 вообще не поддерживает передачу идентификационной информации о процессе через сокеты домена UNIX.

В FreeBSD идентификационная информация передается в виде структуры `cmsgcred`:

```
#define CMGROUP_MAX 16

struct cmsgcred {
    pid_t cmcred_pid;          /* идентификатор передающего процесса */
    uid_t cmcred_uid;          /* реальный идентификатор пользователя */
                               /* передающего процесса */
    uid_t cmcred_euid;         /* эффективный идентификатор пользователя */
                               /* передающего процесса */
    gid_t cmcred_gid;          /* реальный идентификатор группы */
                               /* передающего процесса */
    short cmcred_ngroups;              /* количество групп */
    gid_t cmcred_groups[CMGROUP_MAX];  /* список групп */
};
```

При передаче идентификационной информации нужно только зарезервировать место в памяти под структуру `cmsgcred`. Ядро само заполняет ее, чтобы предотвратить подделку этой информации.

В Linux идентификационные сведения передаются в виде структуры `ucred`:

```
struct ucred {
    uint32_t pid; /* идентификатор передающего процесса */
    uint32_t uid; /* идентификатор пользователя передающего процесса */
    uint32_t gid; /* идентификатор группы передающего процесса */
};
```

В отличие от FreeBSD, ОС Linux требует, чтобы приложение само инициализировало структуру перед ее отправкой. Ядро лишь гарантирует, что предоставленные данные соответствуют вызывающему процессу либо он обладает соответствующими привилегиями, чтобы идентифицировать себя таким образом.

В листинге 17.16 приводится исходный текст функции `send_fd`, в которую добавлена возможность передачи идентификационной информации о передающем процессе.

Листинг 17.16. Передача идентификационной информации через сокеты домена UNIX

```
#include "apue.h"
#include <sys/socket.h>

#if defined(SCM_CREDS) /* интерфейс BSD */
#define CREDSTRUCT cmsgcred
#define SCM_CREDTYPE SCM_CREDS
#elif defined(SCM_CREDENTIALS) /* интерфейс Linux */
#define CREDSTRUCT ucred
#define SCM_CREDTYPE SCM_CREDENTIALS
#else
#error передача идентификационной информации не поддерживается!
#endif

/* размер буфера с управляющей информацией для приема/передачи одного дескриптора */
#define RIGHTSLEN CMSG_LEN(sizeof(int))
#define CREDSLEN CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN (RIGHTSLEN + CREDSLEN)
static struct cmsghdr *cmptr = NULL; /* размещается при первом вызове */

/*
 * Передает дескриптор файла другому процессу.
 * Если fd<0, то в качестве кода ошибки отправляется -fd.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct CREDSTRUCT *credp;
    struct cmsghdr *cmp;
    struct iovec iov[1];
    struct msghdr msg;
    char buf[2]; /* 2-байтный протокол send_fd()/recv_fd() */

    iov[0].iov_base = buf;
    iov[0].iov_len = 2;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;
    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_flags = 0;

    if (fd_to_send < 0) {
```

```
        msg.msg_control = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send; /* ненулевое значение означает наличие ошибки */
        if (buf[1] == 0)
            buf[1] = 1;       /* протокол преобразует в -256 */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control = cmptr;
        msg.msg_controllen = CONTROLLEN;
        cmp = cmptr;
        cmp->cmsg_level = SOL_SOCKET;
        cmp->cmsg_type = SCM_RIGHTS;
        cmp->cmsg_len = RIGHTSLEN;
        *(int *)CMSG_DATA(cmp) = fd_to_send; /* дескриптор для передачи */

        cmp = CMSG_NXTHDR(&msg, cmp);
        cmp->cmsg_level = SOL_SOCKET;
        cmp->cmsg_type = SCM_CREDTYPE;
        cmp->cmsg_len = CREDSLEN;
        credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
#if defined(SCM_CREDENTIALS)
        credp->uid = geteuid();
        credp->gid = getegid();
        credp->pid = getpid();
#endif
        buf[1] = 0; /* нулевое значение означает отсутствие ошибки */
    }
    buf[0] = 0;     /* нулевой байт - флаг для recv_ufd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Обратите внимание: инициализация структуры с идентификационной информацией должна производиться только в ОС Linux.

В листинге 17.17 приводится модифицированная версия функции `recv_fd`, которая называется `recv_ufd`. Она возвращает идентификатор пользователя серверного процесса через аргумент, передаваемый по ссылке.

Листинг 17.17. Прием идентификационной информации через сокеты домена UNIX

```
#include "apue.h"
#include <sys/socket.h>         /* struct msghdr */
#include <sys/un.h>

#if defined(SCM_CREDS)          /* интерфейс BSD */
#define CREDSTRUCT cmsgcred
#define CR_UID cmcred_uid
#define CREDOPT LOCAL_PEERCRED
#define SCM_CREDTYPE SCM_CREDS
```

```
#elif defined(SCM_CREDENTIALS)  /* интерфейс Linux */
#define CREDSTRUCT ucred
#define CR_UID uid
#define CREDOPT SO_PASSCRED
#define SCM_CREDTYPE SCM_CREDENTIALS
#else
#error передача идентификационной информации не поддерживается!
#endif

/* размер буфера с управляющей информацией для приема/передачи одного дескриптора */
#define RIGHTSLEN CMSG_LEN(sizeof(int))
#define CREDSLEN CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN (RIGHTSLEN + CREDSLEN)

static struct cmsghdr *cmptr = NULL;   /* размещается при первом вызове */

/*
 * Принимает дескриптор файла от серверного процесса. Кроме того, любые
 * принятые данные передаются функции (*userfunc)(STDERR_FILENO, buf, nbytes).
 * Чтобы принять дескриптор, мы должны соблюдать 2-байтный протокол.
 */
int
recv_ufd(int fd, uid_t *uidptr,
ssize_t (*userfunc)(int, const void *, size_t))
{
    struct cmsghdr *cmp;
    struct CREDSTRUCT *credp;
    int newfd,nr, status;
    char *ptr;
    char buf[MAXLINE];
    struct iovec iov[1];
    struct msghdr msg;

    const int on = 1;
    status = -1;
    newfd = -1;

    if (setsockopt(fd, SOL_SOCKET, CREDOPT, &on, sizeof(int)) < 0) {
        err_ret("ошибка вызова функции setsockopt");
        return(-1);
    }

    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len = sizeof(buf);
        msg.msg_iov = iov;
        msg.msg_iovlen = 1;
        msg.msg_name = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
```

```
            err_sys("ошибка вызова функции recvmsg");
        } else if (nr == 0) {
            err_ret("соединение закрыто сервером");
            return(-1);
        }

        /*
         * Проверить, являются ли два последних байта нулевым байтом
         * и кодом ошибки. Нулевой байт должен быть предпоследним,
         * а код ошибки - последним байтом в буфере.
         * Нулевой код ошибки означает, что мы должны принять дескриптор.
         */
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("нарушение формата сообщения");
                status = *ptr & 0xFF; /* предотвратить расширение знакового бита */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("получен код 0, но отсутствует fd");

                    /* обработка управляющей информации */
                   for (cmp = CMSG_FIRSTHDR(&msg);
                     cmp != NULL; cmp = CMSG_NXTHDR(&msg, cmp)) {
                       if (cmp->cmsg_level != SOL_SOCKET)
                           continue;
                       switch (cmp->cmsg_type) {
                       case SCM_RIGHTS:
                           newfd = *(int *)CMSG_DATA(cmp);
                           break;
                       case SCM_CREDTYPE:
                           credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
                           *uidptr = credp->CR_UID;
                       }
                   }
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)   /* доставлены заключительные данные */
            return(newfd); /* дескриптор или код ошибки */
    }
}
```

В FreeBSD при обмене идентификационной информацией используется константа `SCM_CREDS`**, а в Linux –** `SCM_CREDENTIALS`**.**

17.5. Сервер открытия файлов, версия 1

Теперь, используя возможность передачи дескрипторов файлов между процессами, мы напишем сервер открытия файлов – программу, которая запускается процессом и открывает один или более файлов. Но вместо того, чтобы отправлять вызывающему процессу содержимое файла, сервер будет передавать ему открытый дескриптор файла. Это позволит серверу работать с любыми типами файлов (такими как устройства или сокеты), а не только с обычными файлами. Это также означает, что через механизмы IPC будет передаваться минимум информации – имя файла и режим открытия (от клиента серверу) и дескриптор открытого файла (от сервера клиенту). Содержимое файла передаваться не будет.

Такая архитектура, когда сервер работает в виде отдельного процесса (запускаемого клиентской программой, как в этом разделе, либо в виде демона – как в следующем разделе), имеет свои преимущества:

- Любой клиент может соединиться с сервером так же просто, как если бы он вызывал библиотечную функцию. Мы не «зашиваем» в программу службу с жестко заданным алгоритмом, вместо этого мы создаем универсальный инструмент, который может использоваться другими программами.
- Если необходимо будет внести изменения в сервер, то они коснутся только одной программы, тогда как обновление одной библиотечной функции может потребовать внесения изменений во все программы, которые вызывают эту функцию (точнее, потребуется заново пересобрать приложения). Хотя подобное обновление можно упростить за счет использования динамических библиотек (раздел 7.7).
- Сервер может быть программой с установленным битом set-user-ID, что дает ему дополнительные права, которыми не обладает клиент. Обратите внимание, что библиотечные функции (или функции динамических библиотек) не предоставляют такой возможности.

Клиентский процесс создает канал s-pipe (на основе либо механизмов STREAMS, либо сокетов домена UNIX) и затем с помощью функций `fork` и `exec` вызывает сервер. После этого клиент передает серверу запрос и через канал s-pipe получает от него ответ.

Определим следующий протокол обмена данными между сервером и клиентом.

1. Клиент отправляет серверу через канал s-pipe запрос вида «`open` <*pathname*> <*openmode*>`\0`», где <*openmode*> – это значение в виде набора цифр ASCII, представляющее второй аргумент функции `open`. Строка запроса завершается нулевым символом.
2. В ответ сервер отправляет вызывающему процессу дескриптор открытого файла или сообщение об ошибке, вызывая для этого функции `send_fd` или `send_err` соответственно.

В данном примере дескриптор файла передается от дочернего процесса родительскому. В разделе 17.6 мы изменим этот пример таким образом, чтобы сервер работал в виде отдельного процесса-демона, и тогда передача дескриптора будет осуществляться между независимыми друг от друга процессами.

Для начала создадим заголовочный файл open.h (листинг 17.18), который подключает необходимые заголовочные файлы и содержит некоторые определения.

Листинг 17.18. Заголовочный файл open.h

```
#include "apue.h"
#include <errno.h>

#define CL_OPEN "open" /* текст запроса, отправляемого серверу клиентом */

int csopen(char *, int);
```

Функция main клиента (листинг 17.19) представляет собой цикл, который считывает имя файла со стандартного ввода и копирует содержимое файла на стандартный вывод. Она вызывает функцию csopen, чтобы соединиться с сервером и получить от него дескриптор открытого файла.

Листинг 17.19. Функция main клиента, версия 1

```
#include "open.h"
#include <fcntl.h>

#define BUFFSIZE 8192

int
main(int argc, char *argv[])
{
    int n, fd;
    char buf[BUFFSIZE], line[MAXLINE];

    /* прочитать имя файла со стандартного ввода */
    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (line[strlen(line) - 1] == '\n')
            line[strlen(line) - 1] = 0; /* заменить символ перевода строки */
                                        /* нулевым символом */
        /* открыть файл */
        if ((fd = csopen(line, O_RDONLY)) < 0)
            continue;   /* csopen() выведет сообщение, полученное от сервера */

        /* и вывести его содержимое на стандартный вывод */
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("ошибка вызова функции write");
        if (n < 0)
            err_sys("ошибка вызова функции read");
        close(fd);
    }

    exit(0);
}
```

Функция `csopen` **с помощью** `fork` **и** `exec` **запускает сервер, после чего создает канал s-pipe.**

Листинг 17.20. Функция csopen, версия 1

```
#include "open.h"
#include <sys/uio.h> /* struct iovec */

/*
 * Передает серверу аргументы name и oflag
 * и получает от него дескриптор открытого файла.
 */
int
csopen(char *name, int oflag)
{
    pid_t pid;
    int len;
    char buf[10];
    struct iovec iov[3];
    static int fd[2] = { -1, -1 };

    if (fd[0] < 0) { /* при первом обращении запустить сервер */
        if (s_pipe(fd) < 0)
            err_sys("ошибка вызова функции s_pipe");
        if ((pid = fork()) < 0) {
            err_sys("ошибка вызова функции fork");
        } else if (pid == 0) { /* дочерний процесс */
            close(fd[0]);
            if (fd[1] != STDIN_FILENO &&
              dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                err_sys("ошибка переназначения stdin с помощью dup2");
            if (fd[1] != STDOUT_FILENO &&
              dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("ошибка переназначения stdout с помощью dup2");
            if (execl("./opend", "opend", (char *)0) < 0)
                err_sys("ошибка вызова функции execl");
        }
        close(fd[1]); /* родительский процесс */
    }
    sprintf(buf, " %d", oflag); /* перевести oflag в строковое представление*/
    iov[0].iov_base = CL_OPEN " ";      /* конкатенация строк */
    iov[0].iov_len = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len = strlen(buf) + 1;   /* +1 - для нулевого символа */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(fd[0], &iov[0], 3) != len)
        err_sys("ошибка вызова функции writev");

    /* получить дескриптор, сообщение об ошибке обработать функцией write() */
    return(recv_fd(fd[0], write));
}
```

Дочерний процесс закрывает один конец канала, а родительский процесс – другой. Кроме того, дочерний процесс перенаправляет стандартный ввод и стандартный вывод в канал. (Как вариант можно было бы передавать имя файла и режим его открытия в виде аргументов командной строки.)

Родительский процесс передает серверу запрос, в котором содержится имя файла и режим открытия. В заключение родитель вызывает `recv_fd` и получает дескриптор либо признак ошибки. Для вывода сообщения об ошибке на стандартный вывод сообщений об ошибках вызывается функция `write`.

Теперь перейдем к реализации сервера. Эта программа, которую мы назвали `opend`, запускается клиентом из листинга 17.20. В первую очередь создадим заголовочный файл `opend.h` (листинг 17.21), который подключает необходимые заголовочные файлы и содержит ряд определений глобальных переменных и прототипов функций.

Листинг 17.21. Заголовочный файл opend.h, версия 1

```
#include "apue.h"
#include <errno.h>

#define CL_OPEN "open"  /* текст запроса, отправляемого серверу клиентом */

extern char  errmsg[];  /* строка сообщения об ошибке, возвращаемая клиенту */
extern int   oflag;     /* флаги функции open(): O_xxx ... */
extern char *pathname;  /* имя файла, полученное от клиента */
int          cli_args(int, char **);
void         request(char *, int, int);
```

Функция `main` (листинг 17.22) считывает текст запроса из канала s-pipe (со своего стандартного ввода) и вызывает функцию `request`.

Листинг 17.22. Функция main сервера, версия 1

```
#include "opend.h"

char errmsg[MAXLINE];
int oflag;
char *pathname;

int
main(void)
{
    int nread;
    char buf[MAXLINE];

    for ( ; ; ) {    /* прочитать аргументы в буфер и обработать запрос */
        if ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
            err_sys("ошибка чтения из канала");
        else if (nread == 0)
            break;   /* клиент закрыл канал */
        request(buf, nread, STDOUT_FILENO);
    }
    exit(0);
}
```

Вся основная работа выполняется в функции `request` (листинг 17.23). Она вызывает функцию `buf_args` для извлечения и преобразования запроса клиента в `argv`-подобный список и передает его для обработки функции `cli_args`. Если ошибок не обнаружено, то вызывается функция `open`, которая открывает файл, и затем функция `send_fd` отправляет клиенту дескриптор открытого файла через канал s-pipe (стандартный вывод). Если возникла ошибка, вызывается функция `send_err`, которая отправляет клиенту сообщение об ошибке, используя описанный ранее протокол.

Листинг 17.23. Функция request, версия 1

```
#include "opend.h"
#include <fcntl.h>

void
request(char *buf, int nread, int fd)
{
    int newfd;

    if (buf[nread-1] != 0) {
        sprintf(errmsg,
                "текст запроса не завершается нулевым символом: %*.*s\n",
                nread, nread, buf);
        send_err(fd, -1, errmsg);
        return;
    }
    if (buf_args(buf, cli_args) < 0) { /* разбор аргументов */
        send_err(fd, -1, errmsg);
        return;
    }
    if ((newfd = open(pathname, oflag)) < 0) {
        sprintf(errmsg, "невозможно открыть файл %s: %s\n", pathname,
                strerror(errno));
        send_err(fd, -1, errmsg);
        return;
    }
    if (send_fd(fd, newfd) < 0)        /* отправить дескриптор */
        err_sys("ошибка вызова функции send_fd");
    close(newfd); /* сервер завершил работу с дескриптором */
}
```

Запрос клиента представляет собой завершающуюся нулевым символом строку, в которой все аргументы разделены пробельными символами. Функция `buf_args` из листинга 17.24 извлекает аргументы и передает их пользовательской функции в виде `argv`-подобного списка для дальнейшей обработки. Далее в этой главе мы еще будем использовать функцию `buf_args`. Для извлечения отдельных аргументов из строки используется функция `strtok`, определяемая стандартом ISO C.

Листинг 17.24. Функция buf_args

```
#include "apue.h"

#define MAXARGC 50    /* максимальное количество аргументов в буфере */
```

```
#define WHITE " \t\n" /* пробельные символы, разделяющие аргументы */

/*
 * В buf[] содержатся аргументы, разделенные пробельными символами.
 * Содержимое буфера преобразуется в argv-подобный массив указателей
 * и передается пользовательской функции (optfunc) для дальнейшей обработки.
 * В случае ошибки при разборе содержимого буфера возвращается
 * значение -1, иначе возвращается результат работы функции optfunc().
 * Обратите внимание: содержимое буфера buf[] модифицируется
 * (после каждого аргумента вставляется нулевой символ).
 */

int
buf_args(char *buf, int (*optfunc)(int, char **))
{
    char *ptr, *argv[MAXARGC];
    int argc;

    if (strtok(buf, WHITE) == NULL) /* аргумент argv[0] обязателен */
        return(-1);
    argv[argc = 0] = buf;

    while ((ptr = strtok(NULL, WHITE)) != NULL) {
        if (++argc >= MAXARGC-1)   /* -1 - предусмотреть место */
                                   /* для пустого указателя в конце списка */
            return(-1);
        argv[argc] = ptr;
    }
    argv[++argc] = NULL;

    /*
     * Поскольку массив argv[] содержит указатели, ссылающиеся на строки
     * в массиве buf[], пользовательская функция может просто скопировать
     * указатели, даже несмотря на то, что массив argv[] исчезнет
     * после выхода из функции.
     */
    return((*optfunc)(argc, argv));
}
```

Пользовательская функция, которую вызывает `buf_args`**, называется** `cli_args` **(листинг 17.25). Она проверяет количество полученных аргументов и сохраняет их в глобальных переменных.**

Листинг 17.25. Функция cli_args

```
#include "opend.h"

/*
 * Эта функция вызывается из buf_args(), которая в свою очередь вызывается
 * функцией request(). Функция buf_args() преобразует содержимое буфера
 * в argv[]-подобный массив, который мы сейчас должны обработать.
 */
int
cli_args(int argc, char **argv)
```

```
{
    if (argc != 3 || strcmp(argv[0], CL_OPEN) != 0) {
        strcpy(errmsg, "Использование: <pathname> <oflag>\n");
        return(-1);
    }
    pathname = argv[1]; /* сохранить указатель на имя файла */
    oflag = atoi(argv[2]);
    return(0);
}
```

На этом разработку сервера открытия файлов, запускаемого клиентской программой, можно считать завершенной. Перед вызовом функции `fork` клиент создает единственный канал s-pipe, который используется для взаимодействия клиента и сервера. Благодаря такой архитектуре мы имеем по серверу для каждого клиента.

17.6. Сервер открытия файлов, версия 2

В предыдущем разделе мы разработали сервер открытия файлов, который запускается клиентским приложением с помощью функций `fork` и `exec`. Этот пример демонстрирует порядок передачи дескриптора от дочернего процесса родительскому. Теперь мы создадим сервер открытия файлов, который будет работать как демон. Один сервер будет обслуживать множество клиентов. Мы предполагаем, что такой вариант более эффективен, поскольку в нем отсутствует обращение к функциям `fork` и `exec`. Для взаимодействия между клиентом и сервером мы по-прежнему будем использовать канал s-pipe и при этом продемонстрируем возможность передачи дескриптора файла между независимыми процессами. В этом примере будут использоваться функции `serv_listen`, `serv_accept` и `cli_conn`, о которых мы говорили в разделе 17.2.2. Кроме того, эта версия демонстрирует возможность обслуживания множества клиентов единственным сервером с помощью функций `select` и `poll` (раздел 14.5).

Новый клиент очень похож на программу, исходный текст которой был приведен в разделе 17.5. Функция `main` осталась без изменений (листинг 17.19). А в заголовочный файл `open.h` (листинг 17.18) мы добавили одну строку:

```
#define CS_OPEN "/home/sar/opend"    /* предопределенное имя сервера */
```

Содержимое файла `open.c` (листинг 17.20) претерпело некоторые изменения, поскольку теперь вместо функций `fork` и `exec` вызывается функция `cli_conn`. Содержимое этого файла приводится в листинге 17.26.

Листинг 17.26. Функция csopen, версия 2

```
#include "open.h"
#include <sys/uio.h> /* struct iovec */

/*
 * Передает аргументы name и oflag серверу
 * и получает от него дескриптор открытого файла.
```

```
 */
int
csopen(char *name, int oflag)
{
    int len;
    char buf[10];
    struct iovec iov[3];
    static int csfd = -1;

    if (csfd < 0) { /* открыть соединение с сервером */
        if ((csfd = cli_conn(CS_OPEN)) < 0)
            err_sys("ошибка вызова функции cli_conn");
    }

    sprintf(buf, " %d", oflag);       /* преобразовать oflag в строку ascii */
    iov[0].iov_base = CL_OPEN " ";    /* конкатенация строк */
    iov[0].iov_len = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len = strlen(buf) + 1; /* нулевой символ передается всегда */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(csfd, &iov[0], 3) != len)
        err_sys("ошибка вызова функции writev");

    /* получить дескриптор, сообщение об ошибке обработать функцией write() */
    return(recv_fd(csfd, write));
}
```

Протокол взаимодействия клиента и сервера остался без изменений.

Теперь перейдем к реализации сервера. Заголовочный файл `opend.h` **(листинг 17.27) подключает необходимые заголовочные файлы и содержит определения глобальных переменных и прототипов функций.**

Листинг 17.27. Заголовочный файл opend.h, версия 2

```
#include "apue.h"
#include <errno.h>

#define CS_OPEN "/home/sar/opend" /* предопределенное имя сервера */
#define CL_OPEN "open"   /* текст запроса, отправляемого серверу клиентом */

extern int debug; /* ненулевое значение для запуска в интерактивном
                  /* режиме (не демон) */
extern char errmsg[];  /* строка сообщения об ошибке, возвращаемая клиенту */
extern int oflag;      /* флаги функции open: O_xxx ... */
extern char *pathname; /* имя файла, полученное от клиента */

typedef struct {       /* по одной структуре Client для каждого клиента */
    int fd;            /* fd или -1, если недоступно */
    uid_t uid;
} Client;

extern Client *client; /* указатель на массив в динамической памяти */
```

```
extern int client_size;   /* количество элементов в массиве client[] */

int cli_args(int, char **);
int client_add(int, uid_t);
void client_del(int);
void loop(void);
void request(char *, int, int, uid_t);
```

Поскольку теперь сервер будет обслуживать сразу несколько клиентов, он должен отслеживать состояние соединения с каждым из них. Он будет делать это с помощью массива `client`**, объявленного в заголовочном файле** `opend.h`**. В листинге 17.28 приводятся исходные тексты трех функций, которые обслуживают этот массив.**

Листинг 17.28. Функции обслуживания массива client

```
#include "opend.h"

#define NALLOC 10  /* количество структур в массиве client для alloc/realloc */

static void
client_alloc(void) /* разместить дополнительные элементы в массиве client[] */
{
    int i;

    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
        client = realloc(client, (client_size+NALLOC)*sizeof(Client));
    if (client == NULL)
        err_sys("невозможно выделить пространство для массива клиентов");

    /* инициализировать новые элементы */
    for (i = client_size; i < client_size + NALLOC; i++)
        client[i].fd = -1;    /* fd = -1 означает, что элемент не занят */

    client_size += NALLOC;
}

/*
 * Вызывается из функции loop() по прибытии нового запроса от клиента.
 */
int
client_add(int fd, uid_t uid)
{
    int i;

    if (client == NULL) /* первое обращение к функции */
        client_alloc();
again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) { /* найти незанятый элемент */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i); /* вернуть индекс в массиве client[] */
        }
```

```
        }

        /* массив полон, выделить дополнительное пространство */
        client_alloc();
        goto again;   /* и повторить поиск (на этот раз все будет в порядке) */
    }

    /*
     * Вызывается функцией loop() по завершении работы с клиентом.
     */
    void
    client_del(int fd)
    {
        int i;

        for (i = 0; i < client_size; i++) {
            if (client[i].fd == fd) {
                client[i].fd = -1;
                return;
            }
        }
        log_quit("невозможно отыскать запись о клиенте по дескриптору %d", fd);
    }
```

Функция `client_add` при первом вызове обращается к функции `client_alloc`, которая выделяет пространство для десяти записей с помощью функции `malloc`. Когда все десять записей будут заполнены, следующий же вызов `client_add` приведет к выделению дополнительного пространства в массиве посредством функции `realloc`. Используя такой способ хранения данных в динамической памяти, мы избежали необходимости ограничения размера массива во время компиляции и определения соответствующих значений в заголовочном файле. При появлении ошибок эти функции обращаются к функциям семейства `log_` (приложение B), поскольку предполагается, что сервер будет работать как демон.

Функция `main` (листинг 17.29) определяет ряд глобальных переменных, обрабатывает аргументы командной строки и вызывает функцию `loop`. Если сервер вызван с ключом `-d`, то он запускается в интерактивном режиме. Это может потребоваться для отладки сервера.

Листинг 17.29. Функция main сервера, версия 2

```
#include "opend.h"
#include <syslog.h>

int     debug, oflag, client_size, log_to_stderr;
char    errmsg[MAXLINE];
char   *pathname;
Client *client = NULL;

int
main(int argc, char *argv[])
{
    int c;
```

```
    log_open("open.serv", LOG_PID, LOG_USER);
    opterr = 0;  /* функция getopt() не должна выводить сообщения на stderr */
    while ((c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
        case 'd':        /* отладка */
            debug = log_to_stderr = 1;
            break;
        case '?':
            err_quit("недопустимая опция: -%c", optopt);
        }
    }

    if (debug == 0)
        daemonize("opend");

    loop();     /* никогда не вернет управление */
}
```

Функция `loop` **организует бесконечный цикл. Мы продемонстрируем две версии этой функции. В листинге 17.30 приводится версия, реализованная на основе функции** `select`**, а в листинге 17.31 – на основе функции** `poll`**.**

Листинг 17.30. Функция loop на основе функции select

```
#include "opend.h"
#include <sys/time.h>
#include <sys/select.h>

void
loop(void)
{
    int i, n, maxfd, maxi, listenfd, clifd, nread;
    char buf[MAXLINE];
    uid_t uid;
    fd_set rset, allset;

    FD_ZERO(&allset);

    /* получить fd, на котором сервер будет ожидать поступления запросов */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("ошибка вызова функции serv_listen");
    FD_SET(listenfd, &allset);
    maxfd = listenfd;
    maxi = -1;

    for ( ; ; ) {
        rset = allset; /* rset модифицируется на каждом проходе цикла */
        if ((n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0)
            log_sys("ошибка вызова функции select");
        if (FD_ISSET(listenfd, &rset)) {

            /* принять новый запрос на соединение с клиентом */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("ошибка вызова функции serv_accept: %d", clifd);
            i = client_add(clifd, uid);
```

```
            FD_SET(clifd, &allset);
            if (clifd > maxfd)
                maxfd = clifd; /* максимальный номер fd для функции select() */
            if (i > maxi)
                maxi = i;      /* максимальный индекс в массиве client[] */
            log_msg("новое соединение: uid %d, fd %d", uid, clifd);
            continue;
        }
        for (i = 0; i <= maxi; i++) {     /* обход массива client[] */
            if ((clifd = client[i].fd) < 0)
            continue;
            if (FD_ISSET(clifd, &rset)) {

                /* прочитать содержимое буфера с аргументами */
                if ((nread = read(clifd, buf, MAXLINE)) < 0) {
                    log_sys("ошибка вызова функции read для fd %d", clifd);
                } else if (nread == 0) {
                    log_msg("закрыто: uid %d, fd %d",
                    client[i].uid, clifd);
                    client_del(clifd);    /* клиент закрыл соединение */
                    FD_CLR(clifd, &allset);
                    close(clifd);
                } else {                  /* обработать запрос от клиента */
                    request(buf, nread, clifd, client[i].uid);
                }
            }
        }
    }
}
```

Эта функция создает точку соединения на стороне сервера с помощью функции `serv_listen`. Остальная часть функции представляет собой цикл, начинающийся с вызова функции `select`, после возврата из которой возможны два состояния.

1. Дескриптор `listenfd` готов для чтения. Это означает, что новый клиент вызвал функцию `cli_conn`. Для приема поступившего запроса на соединение вызывается функция `serv_accept`, а затем в массив `client` добавляется информация о клиенте. (Мы отслеживаем значения самого большого номера дескриптора для передачи его в качестве первого аргумента функции `select`. Мы также отслеживаем значения самого большого индекса в массиве клиентов.)
2. Дескриптор существующего соединения с клиентом готов для чтения. Это означает, что клиент либо закрыл соединение, либо прислал новый запрос. Если функция `read` вернула значение 0 (признак конца файла), следовательно, клиент закрыл соединение. Если же функция `read` вернула значение больше 0, это значит, что клиент прислал новый запрос, который мы передаем на обработку функции `request`.

Мы запоминаем используемые дескрипторы в наборе `allset`. Как только новый клиент соединяется с сервером, мы включаем соответствующий бит в на-

боре. После того, как клиент закроет соединение, соответствующий бит будет выключен.

Мы всегда будем знать, когда клиент закрыл соединение (неважно, добровольно или в результате аварийного завершени), поскольку все дескрипторы клиента (включая дескриптор, поддерживающий соединение с сервером) в этом случае будут закрыты ядром автоматически. В этом состоит одно из отличий дескрипторов от механизмов XSI IPC.

Версия функции `loop`, реализованная на основе функции `poll`, приводится в листинге 17.31.

Листинг 17.31. Функция loop на основе функции poll

```
#include "opend.h"
#include <poll.h>

#if !defined(BSD) && !defined(MACOS)
#include <stropts.h>
#endif

void
loop(void)
{
    int i, maxi, listenfd, clifd, nread;
    char buf[MAXLINE];
    uid_t uid;
    struct pollfd *pollfd;

    if ((pollfd = malloc(open_max() * sizeof(struct pollfd))) == NULL)
        err_sys("malloc error");

    /* получить fd, на котором сервер будет ожидать поступления запросов */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("ошибка вызова функции serv_listen");
    client_add(listenfd, 0);   /* нулевой индекс используется для listenfd */
    pollfd[0].fd = listenfd;
    pollfd[0].events = POLLIN;
    maxi = 0;

    for ( ; ; ) {
        if (poll(pollfd, maxi + 1, -1) < 0)
            log_sys("ошибка вызова функции poll");
        if (pollfd[0].revents & POLLIN) {

            /* принять новый запрос на соединение */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("ошибка вызова функции serv_accept: %d", clifd);
            i = client_add(clifd, uid);
            pollfd[i].fd = clifd;
            pollfd[i].events = POLLIN;
            if (i > maxi)
                maxi = i;
            log_msg("новое соединение: uid %d, fd %d", uid, clifd);
        }
```

```
            for (i = 1; i <= maxi; i++) {
                if ((clifd = client[i].fd) < 0)
                    continue;
                if (pollfd[i].revents & POLLHUP) {
                    goto hungup;
                } else if (pollfd[i].revents & POLLIN) {

                    /* прочитать содержимое буфера с аргументами */
                    if ((nread = read(clifd, buf, MAXLINE)) < 0) {
                        log_sys("ошибка чтения из fd %d", clifd);
                    } else if (nread == 0) {
    hungup:
                        log_msg("закрыто: uid %d, fd %d",
                        client[i].uid, clifd);
                        client_del(clifd);   /* клиент закрыл соединение */
                        pollfd[i].fd = -1;
                        close(clifd);
                    } else {     /* обработать запрос от клиента */
                        request(buf, nread, clifd, client[i].uid);
                    }
                }
            }
        }
    }
```

Чтобы иметь возможность обслуживать столько клиентов, сколько может быть открытых дескрипторов, мы динамически распределяем пространство под массив структур `pollfd`. (Вспомните функцию `open_max` из листинга 2.4.)

Первая запись в массиве `client` (с индексом 0) используется для хранения дескриптора `listenfd`. Таким образом, индекс клиента в массиве `client` для одного и того же клиента совпадает с индексом в массиве `pollfd`. Поступление нового запроса на соединение определяется событием `POLLIN` дескриптора `listenfd`. Как и прежде, для приема запроса на соединение вызывается функция `serv_accept`.

Для существующего клиента мы должны обрабатывать два различных события функции `poll`: разрыв соединения с клиентом (событие `POLLHUP`) и поступление нового запроса от клиента (событие `POLLIN`). В упражнении 15.7 мы упоминали, что к моменту поступления сообщения о разрыве соединения в голове потока еще могут находиться данные, доступные для чтения. При использовании неименованных каналов мы могли бы прочитать все данные перед обработкой закрытия соединения. Однако в данном случае при закрытии соединения со стороны клиента можно просто удалить все данные, находящиеся в потоке. Нет смысла обрабатывать запрос, если некому отправить ответ.

Как и в версии на основе функции `select`, запрос клиента обслуживается функцией `request` (листинг 17.32). Эта версия функции похожа на предыдущую (листинг 17.23). Она вызывает ту же самую функцию `buf_args` (листинг 17.24), которая в свою очередь вызывает функцию `cli_args` (листинг 17.24), но поскольку теперь она вызывается из демона, все сообщения вместо стандартного потока сообщений об ошибках выводятся в системный журнал.

Листинг 17.32. Функция request, версия 2

```
#include "opend.h"
#include <fcntl.h>

void
request(char *buf, int nread, int clifd, uid_t uid)
{
    int newfd;

    if (buf[nread-1] != 0) {
        sprintf(errmsg,
          "строка запроса от uid %d не завершается нулевым символом: %*.*s\n",
          uid, nread, nread, buf);
        send_err(clifd, -1, errmsg);
        return;
    }
    log_msg("запрос: %s, от uid %d", buf, uid);

    /* разбор аргументов */
    if (buf_args(buf, cli_args) < 0) {
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }
    if ((newfd = open(pathname, oflag)) < 0) {
        sprintf(errmsg, "невозможно открыть %s: %s\n",
                pathname, strerror(errno));
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    /* передать дескриптор */
    if (send_fd(clifd, newfd) < 0)
        log_sys("ошибка вызова функции send_fd");
    log_msg("передан fd %d через fd %d для %s", newfd, clifd, pathname);
    close(newfd);    /* работа с дескриптором завершена */
}
```

На этом мы завершаем разработку второй версии сервера открытия файлов, которая работает в виде демона и в состоянии обслуживать запросы от множества клиентов.

17.7. Подведение итогов

Ключевые темы этой главы – передача дескрипторов открытых файлов между процессами и создание уникальных соединений между сервером и клиентами. Мы увидели, как это можно реализовать на базе каналов STREAMS и сокетов домена UNIX. Несмотря на то, что все платформы обеспечивают поддержку сокетов домена UNIX (табл. 15.1), их реализация на разных платформах различна, что существенно осложняет их использование при разработке переносимых приложений.

Мы представили две версии сервера открытия файлов. Первая версия запускается прямо из клиентского приложения с помощью функций `fork` и `exec`. Вторая версия реализована в виде демона, который способен обрабатывать запросы множества клиентов. В обеих версиях были использованы функции передачи и приема дескрипторов. Кроме того, последняя версия использует функции обслуживания соединений между клиентом и сервером, которые обсуждались в разделе 17.2.2, а также функции мультиплексирования ввода-вывода, о которых говорилось в разделе 14.5.

Упражнения

17.1. Перепишите пример из листинга 17.1 таким образом, чтобы для работы с каналом STREAMS вместо функций `read` и `write` использовались функции стандартной библиотеки ввода-вывода.

17.2. Используя функции приема/передачи дескрипторов из этой главы и функции синхронизации родительского и дочернего процессов из раздела 8.9, напишите следующую программу. Программа вызывает функцию `fork`, дочерний процесс открывает существующий файл и передает дескриптор родительскому процессу. После этого дочерний процесс изменяет текущую позицию файла с помощью функции `lseek` и извещает об этом родителя. Родительский процесс читает данные из файла, начиная с текущей позиции, и выводит их для проверки. Если дескриптор был передан описанным нами способом, оба процесса должны совместно использовать одну и ту же запись в таблице файлов. Таким образом, изменение текущей позиции в дочернем процессе должно отразиться на дескрипторе родительского процесса. После этого дочерний процесс должен переместить текущую позицию файла в другое место и опять сообщить об этом родительскому процессу.

17.3. В листингах 17.22 и 17.23 мы по-разному объявили и описали глобальные переменные. В чем суть этих различий?

17.4. Перепишите функцию `buf_args` (листинг 17.24) так, чтобы убрать ограничение времени компиляции на размер массива `argv`. Используйте динамическую память для размещения этого массива.

17.5. Подумайте, как можно оптимизировать функцию `loop` из листингов 17.30 и 17.31. Реализуйте оптимизированные версии.

18

Терминальный ввод-вывод

18.1. Введение

Вопросы, связанные с терминальным вводом-выводом, относятся к разряду наиболее запутанных независимо от типа операционной системы. ОС UNIX не исключение. Самые объемные страницы справочного руководства обычно посвящены именно терминальному вводу-выводу.

Первые противоречия начали проявляться в конце 70-х годов, когда при создании System III были разработаны процедуры для работы с терминалами, в корне отличавшиеся от тех процедур, которые были в Version 7. Процедуры System III далее перекочевали в System V, а процедуры из Version 7 стали стандартом для BSD-систем. Как и в случае с сигналами, противоречия между этими двумя мирами были преодолены благодаря стандарту POSIX.1. В этой главе мы рассмотрим все функции, предназначенные для работы с терминалами, а также некоторые дополнительные функции, характерные для конкретных платформ.

Основная сложность системы терминального ввода-вывода связана с тем, что ее функции используются для выполнения самых разнообразных задач: для управления терминалами, для взаимодействия между компьютерами, соединенными кабелем, для работы с модемами, принтерами и т. п.

18.2. Обзор

Терминальный ввод-вывод имеет два режима работы:

1. Канонический режим обслуживания ввода. В этом режиме ввод с терминала обслуживается построчно. Драйвер терминала возвращает не более одной строки за один запрос.
2. Неканонический режим обслуживания ввода. Вводимые символы не собираются в строки.

Канонический режим действует по умолчанию, если мы не делаем что-то особенное. Например, если в командной оболочке стандартное устройство

ввода связано с терминалом, и мы копируем данные со стандартного ввода на стандартный вывод с помощью функций `read` и `write`, то при работе терминала в каноническом режиме функция `read` будет возвращать данные построчно. Программы, которые работают в полноэкранном режиме, например редактор `vi`, используют неканонический режим, поскольку команды редактора могут состоять всего из одного символа и не содержать перевода строки. Кроме того, редактор не должен позволять системе обслуживать специальные символы, поскольку в самом редакторе они могут обозначать вполне определенные команды редактирования. Например, символ Control-D, который большинством терминалов воспринимается как признак конца файла, в редакторе `vi` обозначает команду прокрутки на пол-экрана вниз.

> Драйверы терминалов из Version 7 и первых версий BSD поддерживали три режима обслуживания ввода с терминала: (а) *подготовленный* (*cooked mode* – вводимые символы собираются в строки и производится обработка специальных символов), (б) *прозрачный* (*raw mode* – вводимые символы не собираются в строки и обработка специальных символов не выполняется) и (в) *посимвольный* (*cbreak mode* – вводимые символы не собираются в строки, но обрабатываются некоторые специальные символы). В листинге 18.10 показаны функции стандарта POSIX.1, которые используются для перевода терминала в прозрачный и в посимвольный режимы.

Стандарт POSIX.1 определяет 11 специальных символов, 9 из которых можно изменить. С некоторыми из них мы уже встречались в предыдущих главах, например символ конца файла (обычно Control-D) и символ приостановки (обычно Control-Z). В разделе 18.3 будут даны описания всех этих символов.

Терминал можно представить себе как некоторое устройство, управляемое драйвером, обычно расположенным в ядре. Каждое терминальное устройство имеет входную и выходную очереди, как показано на рис. 18.1. Обратите особое внимание на следующие моменты:

- Наличие эхо-вывода подразумевает связь между входной и выходной очередями.
- Размер входной очереди ограничивается значением `MAX_INPUT` (табл. 2.8). Реакция системы на переполнение входной очереди зависит от конкрет-

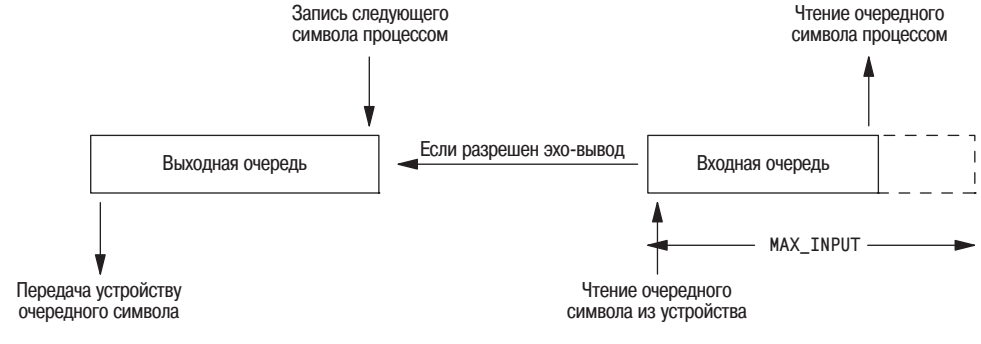


***Рис. 18.1.** Логическое изображение входной и выходной очередей устройства терминала*

ной реализации. В большинстве версий UNIX, если это происходит, ввод последующих символов сопровождается звуковым сигналом.

- Существует еще один предел, ограничивающий размер входной очереди, который здесь не показан, – `MAX_CANON`. Этот предел определяет максимальный размер строки в байтах при работе терминала в каноническом режиме.
- Хотя размер выходной очереди также ограничен, тем не менее константы, которые определяли бы конкретное значение, отсутствуют, потому что, когда выходная очередь начинает переполняться, ядро просто приостанавливает процесс, выполняющий запись, до тех пор, пока в выходной очереди не освободится место.
- Позже мы увидим, как с помощью функции `tcflush` можно сбросить содержимое входной и выходной очередей. Аналогичным образом при рассмотрении функции `tcsetattr` мы узнаем, как с ее помощью можно изменить характеристики терминала, но только после опустошения выходной очереди. (Это может понадобиться, например, для изменения параметров вывода.) Мы также можем заставить систему очистить очередь ввода при изменении параметров терминала. (Это может пригодиться при изменении параметров ввода или при переходе от канонического режима к неканоническому и обратно, чтобы предотвратить неверную интерпретацию ранее введенных символов.)

В большинстве версий UNIX реализация канонического режима выполнена в виде модуля, который называется *terminal line discipline* (дисциплина обслуживания линии связи с терминалом). Этот модуль можно представить себе как некий черный ящик, расположенный между универсальными функциями чтения/записи ядра и фактическим драйвером устройства (рис. 18.2).

Обратите внимание на то, как похож рис. 18.2 на схему потока STREAMS с промежуточным модулем из рис. 14.5. Мы еще вернемся к этому рисунку в главе 19, когда будем обсуждать псевдотерминалы.

Все характеристики терминала, которые можно узнать или изменить, содержатся в структуре `termios`. Определение этой структуры находится в заголовочном файле `<termios.h>`, который будет постоянно использоваться на протяжении всей главы:

```
struct termios {
    tcflag_t c_iflag;    /* флаги режима ввода */
    tcflag_t c_oflag;    /* флаги режима вывода */
    tcflag_t c_cflag;    /* флаги режима управления */
    tcflag_t c_lflag;    /* флаги локального режима */
    cc_t     c_cc[NCCS]; /* управляющие символы */
};
```

Можно сказать, что флаги режима ввода управляют вводом символов, который производится драйвером терминального устройства (очистка восьмого бита, проверка бита четности и т. д.), флаги режима вывода контролируют вывод драйвера (обработка выходного потока данных, замена символа пере-

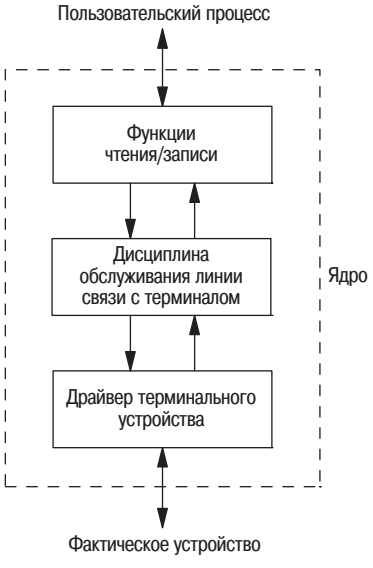


***Рис. 18.2.** Модуль, реализующий дисциплину обслуживания линии связи с терминалом*

вода строки комбинацией CR/LF и тому подобное), а флаги режима управления определяют параметры последовательного порта RS-232 (игнорировать строки состояния модема, количество стоповых битов и т. д.). И наконец, флаги локального режима оказывают влияние на интерфейс между драйвером и пользователем (включение/выключение эхо-вывода, отображение символа забоя, разрешение/запрет генерации сигналов терминалом и т. д.).

Тип `tcflag_t` достаточно велик, чтобы переменные этого типа могли хранить значения сразу всех флагов. Зачастую он определен как `unsigned int` или `unsigned long`. Массив `c_cc` хранит все специальные символы, которые можно изменить. Константа `NCCS` представляет количество элементов этого массива, обычно ее значение находится в диапазоне от 15 до 20 (поскольку большинство версий UNIX поддерживают более 11 определяемых стандартом POSIX.1 управляющих символов). Тип `cc_t` достаточно велик, чтобы переменные этого типа могли хранить любой из управляющих символов, и обычно определен как `unsigned char`.

> В версиях System V, предшествовавших появлению стандарта POSIX.1, были заголовочный файл `<termio.h>` и структура `termio`. Стандарт POSIX.1 добавил к именам букву s, чтобы отличить современные определения от их предшественников.

В табл. 18.1–18.4 приводится список всех флагов, с помощью которых можно воздействовать на характеристики терминального устройства. Обратите внимание, что хотя стандарт Single UNIX Specification определяет базовый набор, тем не менее все платформы расширяют его своими собственными флагами. Большинство дополнительных флагов появились в результате ис-

торически сложившихся различий между системами. Более подробно назначение каждого из этих флагов мы рассмотрим в разделе 18.5.

Таблица 18.1. Флаги режима управления терминалом

Флаг	Описание	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	MacOS X 10.3	Solaris 9
CBAUDEXT	Расширенное управление скоростью					•
CCAR_OFLOW	Использовать линию DCD для управления выходным потоком		•		•	
CCTS_OFLOW	Использовать линию CTS для управления выходным потоком		•		•	•
CDSR_OFLOW	Использовать линию DSR для управления выходным потоком		•		•	•
CDTR_IFLOW	Использовать линию DTR для управления входным потоком		•		•	
CIBAUDEXT	Расширенное управление скоростью приема					•
CIGNORE	Игнорировать флаги управления режимами		•		•	
CLOCAL	Игнорировать строки состояния модема	•	•	•	•	•
CREAD	Разрешить прием	•	•	•	•	•
CRTSCTS	Разрешить аппаратное управление потоком данных		•	•	•	•
CRTS_IFLOW	Использовать линию RTS для управления входным потоком		•		•	•
CRTSXOFF	Разрешить аппаратное управление входным потоком данных					•
CSIZE	Маска размера символов	•	•	•	•	•
CSTOPB	Передавать два стоповых бита или один	•	•	•	•	•
HUPCL	Разорвать связь при закрытии устройства последним процессом	•	•	•	•	•
MDMBUF	То же самое, что и CCAR_OFLOW		•		•	

Таблица 18.1 (продолжение)

Флаг	Описание	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	MacOS X 10.3	Solaris 9
PARENB	Разрешить контроль четности	•	•	•	•	•
PAREXT	Контроль четности по схеме mark или space					•
PARODD	Контроль четности по схеме odd или even	•	•	•	•	•

Таблица 18.2. Флаги режима ввода

Флаг	Описание	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	MacOS X 10.3	Solaris 9
BRKINT	Генерировать сигнал SIGINT при получении символа BREAK	•	•	•	•	•
ICRNL	Преобразовывать символ CR (возврат каретки) в символ NL (перевод строки) при вводе	•	•	•	•	•
IGNBRK	Игнорировать символ BREAK	•	•	•	•	•
IGNCR	Игнорировать символ CR	•	•	•	•	•
IGNPAR	Игнорировать символы с ошибками контроля четности	•	•	•	•	•
IMAXBEL	Выдавать звуковой сигнал при переполнении очереди ввода		•	•	•	•
INLCR	Преобразовывать символ NL в символ CR при вводе	•	•	•	•	•
INPCK	Разрешить проверку бита паритета при вводе	•	•	•	•	•
ISRIP	Сбрасывать восьмой бит во вводимых символах	•	•	•	•	•
IUCLC	Преобразовывать при вводе символы верхнего регистра в нижний регистр			•		•
IXANY	Разрешить перезапуск вывода по любому символу	XSI	•	•	•	•
IXOFF	Разрешить управление входным потоком данных с помощью символов START/STOP	•	•	•	•	•
IXON	Разрешить управление выходным потоком данных с помощью символов START/STOP	•	•	•	•	•
PARMRK	Отмечать ошибки контроля четности	•	•	•	•	•

Таблица 18.3. Флаги локального режима

Флаг	Описание	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	MacOS X 10.3	Solaris 9
ALTWERASE	Использовать альтернативный алгоритм обработки символа WERASE		•		•	
ECHO	Разрешить эхо-вывод	•	•	•	•	•
ECHOCTL	Выводить управляющие символы как ^ (символ)		•	•	•	•
ECHOE	Отображать забой	•	•	•	•	•
ECHOK	Отображать удаление строки	•	•	•	•	•
ECHOKE	Отображать забой каждого символа при удалении строки		•	•	•	•
ECHONL	Отображать символ перевода строки	•	•	•	•	•
ECHOPRT	Отображать удаление символов для вывода на принтер		•	•	•	•
EXTPROC	Внешний обработчик символов		•		•	
FLUSHO	Сбрасывать очередь вывода		•	•	•	•
ICANON	Канонический режим ввода	•	•	•	•	•
IEXTEN	Разрешить расширенную обработку вводимых символов	•	•	•	•	•
ISIG	Разрешить генерацию сигналов терминалом	•	•	•	•	•
NOFLSH	Запретить сброс очередей после прерывания по сигналам SIGINT и SIGQUIT	•	•	•	•	•
NOKERNINFO	Не выводить информацию при вводе символа STATUS		•		•	
PENDIN	Вывод символов из очереди ввода		•	•	•	•
TOSTOP	Послать сигнал SIGTTOU фоновому источнику вывода	•	•	•	•	•
XCASE	Каноническое представление символов верхнего и нижнего регистров			•		•

Итак, флаги нам известны, но как можно изменить те или иные характеристики терминального устройства? В табл. 18.5 приводится список различных функций, определяемых стандартом Single UNIX Specification для взаимодействия с терминальными устройствами. (Все перечисленные функции являются частью базовых спецификаций стандарта POSIX.1, за исключени-

ем функции tcgetsid, которая определена в составе расширений XSI. Функции tcgetpgrp, tcgetsid и tcsetpgrp были описаны в разделе 9.7.)

Таблица 18.4. Флаги режима вывода

Флаг	Описание	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	MacOS X 10.3	Solaris 9
BSDLY	Маска задержки символа забоя	XSI		•		•
CMSPAR	Контроль четности по схеме mark или space			•		
CRDLY	Маска задержки символа CR (возврат каретки)	XSI		•		•
FFDLY	Маска задержки символа FF (перевод страницы)	XSI		•		•
NLDLY	Маска задержки символа NL (перевод строки)	XSI		•		•
OCRNL	Преобразовывать символ CR в NL при выводе	XSI	•	•		•
OFDEL	Использовать символ DEL в качестве заполнителя, иначе – символ NUL	XSI		•		•
OFILL	Использовать для задержки символы заполнения	XSI		•		•
OLCUC	Преобразовывать символы нижнего регистра в верхний при выводе			•		•
ONLCR	Преобразовывать символы NL в последовательности символов CR-NL	XSI	•	•	•	•
ONLRET	Символ NL выполняет функции символа CR	XSI	•	•		•
ONOCR	Не выводить символ CR в нулевой позиции строки	XSI	•	•		•
ONOEOT	Не выводить символ EOT (^D)	•	•		•	
OPOST	Выполнять обработку вывода	•	•	•	•	•
OXTABS	Заменять символы табуляции пробелами		•		•	
TABDLY	Маска задержки символа горизонтальной табуляции	XSI		•		•
VTDLY	Маска задержки символа вертикальной табуляции	XSI		•		•

Обратите внимание: стандарт Single UNIX Specification не предусматривает использования классической функции `ioctl` для работы с терминальными устройствами. Вместо нее должны использоваться 13 функций, перечисленных в табл. 18.5. Причина состоит в том, что тип последнего аргумента функции `ioctl` зависит от выполняемой операции, что делает невозможным контроль соответствия типов.

Хотя для работы с терминальными устройствами определено всего 13 функций, тем не менее первые две функции из табл. 18.5 могут использоваться для управления почти 70 параметрами (табл. 18.1–18.4). Обслуживание терминальных устройств осложняется большим количеством параметров и необходимостью определять, какие из них требуются для работы с конкретным устройством (терминалом, модемом, принтером или любым другим).

Таблица 18.5. Перечень функций, предназначенных для работы с терминалами

Функция	Описание
`tcgetattr`	Получить характеристики терминала (структура `termios`)
`tcsetattr`	Изменить характеристики терминала (структура `termios`)
`cfgetispeed`	Получить скорость ввода
`cfgetospeed`	Получить скорость вывода
`csetispeed`	Установить скорость ввода
`csetospeed`	Установить скорость вывода
`tcdrain`	Ожидать, пока не будут отправлены все выходные данные
`tcflow`	Приостановить прием или передачу
`tcflush`	Сбросить содержимое очереди ввода или вывода
`tcsendbreak`	Отправить символ BREAK
`tcgetpgrp`	Получить идентификатор группы процессов переднего плана
`tcsetpgrp`	Перевести группу процессов с заданным идентификатором на передний план
`tcgetsid`	Получить идентификатор группы процессов лидера сессии для заданного управляющего терминала

Взаимосвязь функций из табл. 18.5 показана на рис. 18.3.

> Стандарт POSIX.1 не оговаривает, в каком поле структуры `termios` сожержится информация о скорости обмена – это оставлено на усмотрение реализации. Некоторые системы, такие как Linux и Solaris, хранят сведения о скорости в поле `c_cflag`. Системы, производные от BSD, такие как FreeBSD и Mac OS X, предусматривают в структуре два дополнительных поля: одно для скорости ввода, другое для скорости вывода.

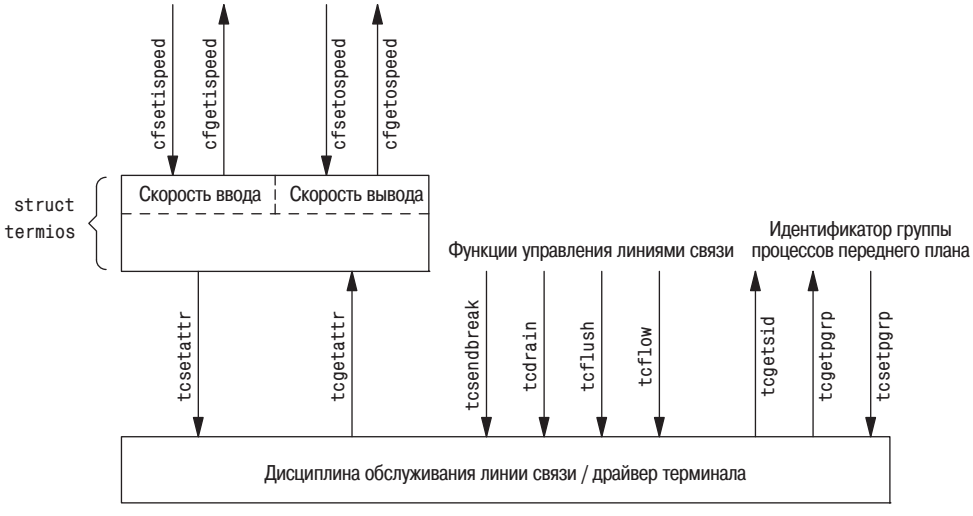


Рис. 18.3. *Взаимосвязь функций, предназначенных для работы с терминалами*

18.3. Специальные символы ввода

Стандарт POSIX.1 определяет 11 специальных (или служебных) символов ввода. Каждая реализация может дополнять этот список своими символами. В табл. 18.6 приводится список специальных символов.

Таблица 18.6. Список специальных символов ввода

Символ	Описание	Индекс в массиве c_cc	Разрешается		Типичное значение	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	MacOS X 10.3	Solaris 9
			поле	флаг						
CR	Возврат каретки	(не может быть изменено)	c_lflag	ICANON	\r	•	•	•	•	•
DISCARD	Отменить вывод	VDISCARD	c_lflag	IEXTEN	^O		•	•	•	•
DSUSP	Отложенная приостановка (SIGTSTP)	VDSUSP	c_lflag	ISIG	^Y		•		•	•
EOF	Конец файла	VEOF	c_lflag	ICANON	^D	•	•	•	•	•
EOL	Конец строки	VEOL	c_lflag	ICANON		•	•	•	•	•
EOL2	Альтернативный конец строки	VEOL2	c_lflag	ICANON			•	•	•	•
ERASE	Забой	VERASE	c_lflag	ICANON	^H, ^?	•	•	•	•	•

Символ	Описание	Индекс в массиве c_cc	Разрешается		Типичное значение	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	MacOS X 10.3	Solaris 9
			поле	флаг						
ERASE2	Альтернативный забой	VERASE2	c_lflag	ICANON	^H, ^?		•			
INTR	Сигнал прерывания (SIGINT)	VINTR	c_lflag	ISIG	^?, ^C	•	•	•	•	•
KILL	Стирание строки	VKILL	c_lflag	ISIG	^U	•	•	•	•	•
LNEXT	Экранирует следующий символ	VLNEXT	c_lflag	ICANON	^V		•	•	•	•
NL	Перевод строки	(не может быть изменено)	c_lflag	ICANON	\n	•	•	•	•	•
QUIT	Сигнал завершения (SIGQUIT)	VQUIT	c_lflag	ISIG	^\	•	•	•	•	•
REPRINT	Перепечатать входную строку	VREPRINT	c_lflag	ICANON	^R		•	•	•	•
START	Продолжить вывод	VSTART	c_lflag	IXON/ IXOFF	^Q	•	•	•	•	•
STATUS	Запрос состояния	VSTATUS	c_lflag	ICANON	^T		•		•	
STOP	Остановить вывод	VSTOP	c_lflag	IXON/ IXOFF	^S	•	•	•	•	•
SUSP	Сигнал приостановки (SIGTSTP)	VSUSP	c_lflag	ISIG	^Z	•	•	•	•	•
WERASE	Стереть одно слово	VWERASE	c_lflag	ICANON	^W		•	•	•	•

Стандарт POSIX.1 определяет 11 специальных символов, 9 из которых мы можем заменить практически любыми символами по своему желанию. Исключение составляют символы возврата каретки и перевода строки (\r и \n соответственно) и, возможно, символы STOP и START (зависит от реализации). Чтобы выполнить замену, нужно изменить соответствующие элементы массива c_cc в структуре termios. Элементы этого массива индексируются константами, имена которых начинаются с буквы V (третья колонка в табл. 18.6).

Стандарт POSIX.1 позволяет запретить действие этих символов. Для этого в соответствующий элемент массива нужно записать значение _POSIX_VDISABLE.

В прежних версиях стандарта Single UNIX Specification поддержка константы _POSIX_VDISABLE была необязательной. Современная версия стандарта требует, чтобы эта константа поддерживалась всеми реализациями.

Все четыре платформы, обсуждаемые в этой книге, поддерживают такую возможность. В Linux 2.4.22 и Solaris 9 константа _POSIX_VDISABLE определена со значением 0, в FreeBSD 5.2.1 и Mac OS X 10.3 – со значением 0xff.

В некоторых ранних версиях UNIX действие того или иного специального символа можно было запретить, записав в соответствующий элемент массива значение 0.

Пример

Прежде чем приступить к подробному описанию специальных символов, рассмотрим небольшую программу, которая изменяет их. Программа из листинга 18.1 запрещает символ прерывания и устанавливает символ Control-B в качестве символа конца файла.

Листинг 18.1. Запрет символа прерывания и изменение символа конца файла

```
#include "apue.h"
#include <termios.h>

int
main(void)
{
    struct termios term;
    long vdisable;

    if (isatty(STDIN_FILENO) == 0)
        err_quit("стандартное устройство ввода не является терминалом");

    if ((vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
        err_quit("ошибка fpathconf или _POSIX_VDISABLE не поддерживается");

    if (tcgetattr(STDIN_FILENO, &term) < 0) /* получить характеристики терминала */
        err_sys("ошибка вызова функции tcgetattr");

    term.c_cc[VINTR] = vdisable; /* запретить действие символа INTR */
    term.c_cc[VEOF] = 2;         /* символ конца файла теперь Control-B */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
        err_sys("ошибка вызова функции tcsetattr");

    exit(0);
}
```

В этой программе обратите внимание на следующее:

- Изменение значений служебных символов выполняется только в случае, если стандартное устройство ввода является терминалом. Для проверки вызывается функция isatty (раздел 18.9).
- Значение константы _POSIX_VDISABLE мы получаем с помощью функции fpathconf.
- Сначала функция tcgetattr (раздел 18.4) получает от ядра структуру termios. После модификации ее содержимого вызывается функция tcsetattr,

которая устанавливает новые значения. Изменятся только те значения, которые были явным образом модифицированы.

- Запрет действия клавиши прерывания имеет иной смысл, нежели изменение диспозиции сигнала SIGINT. Программа из листинга 18.1 просто запрещает действие служебного символа, который заставляет драйвер терминала генерировать сигнал SIGINT. Но мы по-прежнему можем прервать работу процесса, используя функцию kill для посылки сигнала.

Теперь более подробно опишем каждый служебный символ. Мы называем эти символы служебными символами ввода, но два из них – START и STOP (Control-Q и Control-S) – также имеют специальное назначение и при выводе. Обратите внимание на то, что эти символы распознаются драйвером терминала и обрабатываются отдельно, после чего большинство из них уничтожается – они не передаются процессу при выполнении операции чтения. Исключение из этого правила составляют символы перевода строки (NL, EOL, EOL2) и возврата каретки (CR).

CR Символ возврата каретки. Мы не можем изменить его значение. Символ распознается в каноническом режиме ввода. Когда одновременно установлены флаги ICANON (канонический режим) и ICRNL (преобразование CR в NL) и сброшен флаг IGNCR (игнорировать CR), символ CR преобразуется в символ NL и воспринимается как символ NL. Этот символ передается читающему процессу (возможно, преобразованный в символ NL).

DISCARD Символ распознается в расширенном режиме ввода (IEXTEN) и уничтожает все вводимые символы до тех пор, пока не будет встречен другой символ DISCARD или пока состояние терминала не будет изменено (флаг FLUSHO). После обработки этот символ уничтожается (т. е. не передается процессу).

DSUSP Символ отложенной приостановки выполнения задания. Он распознается в расширенном режиме ввода (IEXTEN), если поддерживается управление заданиями и установлен флаг ISIG. Аналогично символу SUSP, символ отложенной приостановки приводит к генерации сигнала SIGTSTP, который передается всем процессам из группы процессов переднего плана (рис. 9.7). Единственное отличие – сигнал посылается не тогда, когда будет введен символ, а когда процесс начнет чтение из управляющего терминала. Этот символ в процессе обработки уничтожается (т. е. не передается процессу).

EOF Символ конца файла. Этот символ распознается в каноническом режиме ввода (ICANON). При вводе этого символа все данные во входной очереди немедленно передаются читающему процессу. Если очередь ввода была пуста, процессу возвращается счетчик прочитанных байтов, равный нулю. Как правило, чтобы передать программе признак конца файла, символ EOF вводится в начале новой строки. Когда этот символ вводится в каноническом режиме, он уничтожается после обработки (т. е. не передается процессу).

EOL Дополнительный символ – разделитель строк, подобный символу NL. Символ распознается при вводе в каноническом режиме (ICANON) и передается читающему процессу. Однако обычно этот символ не используется.

EOL2 Еще один символ – разделитель строк, подобный символу NL. Он интерпретируется точно так же, как символ EOL.

ERASE Символ забоя. Этот символ распознается в каноническом режиме ввода (ICANON). Он стирает предыдущий символ в строке, но не переходит через начало строки. Когда этот символ вводится в каноническом режиме, он уничтожается после обработки (т. е. не передается процессу).

ERASE2 Альтернативный символ забоя. Он интерпретируется точно так же, как символ ERASE.

INTR Символ прерывания. Этот символ распознается при вводе, если установлен флаг ISIG, и приводит к генерации сигнала SIGINT, который посылается всем процессам в группе процессов переднего плана (рис. 9.7). Этот символ уничтожается после обработки (т. е. не передается процессу).

KILL Символ стирания строки. (Имя «kill» подобрано не совсем правильно, потому что оно напоминает имя функции kill, которая используется для посылки сигналов процессу. Этот символ лучше было бы назвать line-erase (стирание строки), т. к. он не имеет никакого отношения к сигналам.) Этот символ распознается в каноническом режиме ввода (ICANON). Он удаляет всю строку и уничтожается после обработки (т. е. не передается процессу).

LNEXT Экранирует следующий символ. Этот символ распознается в расширенном режиме ввода (IEXTEN); он отменяет специальное назначение следующего за ним символа. Это относится к любым специальным символам из тех, что описываются в этом разделе. С помощью этого символа можно передать любой символ программе. После обработки символ LNEXT уничтожается, но следующий за ним символ передается процессу.

NL Символ перевода строки, который служит разделителем строк. Этот символ распознается в каноническом режиме ввода (ICANON). Передается процессу, выполняющему чтение.

QUIT Символ завершения. Этот символ распознается при вводе, если установлен флаг ISIG. Символ QUIT приводит к генерации сигнала SIGQUIT, который посылается всем процессам из группы процессов переднего плана (рис. 9.7). После обработки этот символ уничтожается (т. е. не передается процессу).
В табл. 10.1 указано, что различие между символами INTR и QUIT заключается в том, что при вводе символа QUIT по умолчанию процесс не просто завершается, а создает при этом файл с дампом памяти (core).

REPRINT Символ перепечатки. Этот символ распознается в расширенном каноническом режиме ввода (установлены оба флага, IEXTEN и ICANON) и заставляет терминал вывести все символы из очереди ввода (повторный вывод). После обработки символ уничтожается (т. е. не передается процессу).

START Символ запуска. Этот символ распознается при вводе, если установлен флаг IXON, и автоматически отправляется на вывод, если установлен флаг IXOFF. Прием символа START при установленном флаге IXON возобновляет ввод, который был приостановлен введенным ранее символом STOP. В этом случае символ START уничтожается после обработки (т. е. не передается процессу).

При установленном флаге IXOFF драйвер терминала автоматически генерирует символ START, чтобы продолжить ввод, который ранее был приостановлен из-за переполнения очереди ввода.

STATUS Символ запроса состояния терминала в BSD-системах. Символ распознается в расширенном каноническом режиме ввода (установлены оба флага, IEXTEN и ICANON) и генерирует сигнал SIGINFO, который передается всем процессам в группе процессов переднего плана (рис. 9.7). Дополнительно, если не установлен флаг NOKERNINFO, информация о состоянии группы процессов переднего плана выводится на терминал. После обработки этот символ уничтожается (т. е. не передается процессу).

STOP Символ останова. Этот символ распознается при вводе, если установлен флаг IXON, и автоматически отправляется на вывод, если установлен флаг IXOFF. Прием символа STOP при установленном флаге IXON приостанавливает вывод данных. В этом случае после обработки символ STOP уничтожается (т. е. не передается процессу). Приостановленный вывод данных возобновляется после ввода символа START.

Если установлен флаг IXOFF, драйвер терминала автоматически генерирует символ STOP, когда возникает угроза переполнения очереди ввода.

SUSP Символ приостановки выполнения задания. Он распознается при вводе, если поддерживается управление заданиями и установлен флаг ISIG. Символ SUSP приводит к генерации сигнала SIGTSTP, который передается всем процессам из группы процессов переднего плана (рис. 9.7). Символ уничтожается в процессе обработки (т. е. не передается процессу).

WERASE Символ удаления слова. Этот символ распознается в расширенном каноническом режиме ввода (установлены оба флага, IEXTEN и ICANON) и приводит к удалению предыдущего слова. Сначала стираются любые предшествующие пробельные символы (пробелы или символы табуляции), затем символы предшествующей лексемы. Курсор ввода останавливается на месте первого символа стертой лексемы. Обычно границами лексем служат пробельные символы. Однако этот порядок распознавания границ лексем можно изменить, установив флаг ALTWERASE. Тогда границами лексем будут считаться любые не алфавитно-цифровые символы. Этот символ уничтожается в процессе обработки (т. е. не передается процессу).

Еще один «символ», который мы должны определить, – это символ BREAK (прерывание передачи связи). В действительности BREAK не является символом, это скорее состояние, которое возникает в процессе асинхронной последовательной передачи данных. Драйвер терминала может быть извещен о наступлении состояния BREAK различными способами, в зависимости от типа последовательного интерфейса.

Большинство старых терминалов имели специальную клавишу с надписью BREAK, нажатие которой порождало состояние BREAK. По этой причине многие думают, что BREAK – это символ. На клавиатурах современных терминалов эта клавиша отсутствует. На клавиатурах персональных компьютеров клавиша BREAK несет совсем другую смысловую нагрузку. Например, с помощью комбинации клавиш Control-BREAK в ОС Windows можно прервать работу командного интерпретатора.

При использовании асинхронного режима последовательной передачи данных BREAK представляет собой последовательность нулевых битов, которые продолжают передаваться в течение большего времени, чем требуется для передачи одного байта. Вся последовательность нулевых битов рассматривается как один «символ» BREAK. В разделе 18.8 мы узнаем, как можно передать «символ» BREAK с помощью функции tcsendbreak.

18.4. Получение и изменение характеристик терминала

Чтобы получить и установить структуру termios, можно воспользоваться двумя функциями: tcgetattr и tcsetattr. С их помощью можно проверить и изменить различные характеристики терминала и специальные символы, чтобы терминал действовал так, как нам требуется.

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termptr);

int tcsetattr(int filedes, int opt, const struct termios *termptr);
```

Обе возвращают 0 в случае успеха, −1 в случае ошибки

Обе функции принимают указатель на структуру termios и либо возвращают текущие характеристики терминала, либо изменяют их. Поскольку обе функции могут работать только с терминальными устройствами, то в случае, когда дескриптор *filedes* не является терминалом, они возвращают признак ошибки с кодом ENOTTY в переменной errno.

Аргумент *opt* функции tcsetattr позволяет определить, когда новые характеристики терминала должны вступить в силу. В этом аргументе можно передать одну из следующих констант.

TCSANOW Изменения вступают в силу немедленно.

TCSADRAIN Изменения вступят в силу после того, как будут отправлены все данные, находящиеся в очереди вывода. Эта константа используется в том случае, если мы изменяем характеристики вывода.

TCSAFLUSH Изменения вступят в силу после того, как будут отправлены все данные, находящиеся в очереди вывода. Кроме того, когда изменения вступят в силу, все непрочитанные данные в очереди ввода уничтожаются (сбрасываются).

Возвращаемое значение функции tcsetattr может ввести в заблуждение. Дело в том, что она возвращает признак успешного завершения в том случае, если ей удалось выполнить изменение хотя бы одной характеристики, а не всех. Поэтому, если функция tcsetattr возвращает признак успешного выполнения, мы должны убедиться в том, что были выполнены все запрошенные изменения. Это означает, что после вызова функции tcsetattr следует

вызвать функцию tcgetattr и сравнить фактические характеристики терминала с желаемыми.

18.5. Флаги режимов терминала

В этом разделе мы подробно рассмотрим все флаги режимов терминала, которые были перечислены в табл. 18.1–18.4. Этот список содержит описания флагов, расположенных в алфавитном порядке. Для каждого флага указывается, в каком из четырех полей он передается. (Обычно из названия флага трудно определить, для какого поля он предназначен.) Кроме того, для каждого флага указано, определен ли он в стандарте Single UNIX Specification, и перечисляются платформы, которые его поддерживают.

Каждому из перечисленных флагов соответствует один или более бит, если только флаг не является *маской*. Флаг-маска определяет набор сгруппированных битов, которые можно установить или сбросить. Мы перечислим имена всех масок и имена всех значений для каждой из них. Например, чтобы изменить размер символа, прежде всего нужно сбросить биты, используя для этого маску CSIZE, и затем установить одно из значений CS5, CS6, CS7 или CS8.

Шесть значений задержек, которые поддерживаются ОС Linux и Solaris, также являются масками: BSDLY, CRDLY, FFDLY, NLDLY, TABDLY и VTDLY. Значение каждой из них вы найдете на странице справочного руководства termio(7I) в ОС Solaris. В любом случае, нулевое значение маски означает отсутствие задержки. Если задержка определена, то флаги OFILL и OFDEL определяют, должен ли драйвер терминала действительно выполнять задержку или вместо этого он должен посылать символы-заполнители.

Пример

Программа из листинга 18.2 демонстрирует получение и изменение значений с помощью маски.

Листинг 18.2. Пример использования функций tcgetattr и tcsetattr

```
#include "apue.h"
#include <termios.h>

int
main(void)
{
    struct termios term;

    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("ошибка вызова функции tcgetattr");

    switch (term.c_cflag & CSIZE) {
    case CS5:
        printf("5 бит на байт\n");
        break;
    case CS6:
        printf("6 бит на байт \n");
```

```
            break;
        case CS7:
            printf("7 бит на байт \n");
            break;
        case CS8:
            printf("8 бит на байт \n");
            break;
        default:
            printf("неизвестное количество бит на байт\n");
        }

        term.c_cflag &=  CSIZE; /* обнулить биты */
        term.c_cflag |= CS8;    /* установить 8 бит на байт */

        if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
            err_sys("ошибка вызова функции tcsetattr");

        exit(0);
    }
```

А теперь опишем каждый из флагов.

ALTWERASE (c_lflag, FreeBSD, Mac OS X) Если флаг установлен, используется альтернативный алгоритм стирания слова при вводе символа WERASE. Предыдущее слово стирается не до первого пробельного символа, а до первого символа, не являющегося алфавитно-цифровым.

BRKINT (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, а IGNBRK – нет, то при появлении символа BREAK производится сброс очередей ввода и вывода и генерируется сигнал SIGINT. Этот сигнал посылается группе процессов переднего плана, если терминальное устройство является управляющим терминалом.
Если оба флага, BRKINT и IGNBRK, сброшены, то символ BREAK будет прочитан как символ \0, если сброшен флаг PARMRK, или как последовательность символов \377, \0, \0, если флаг PARMRK установлен.

BSDLY (c_oflag, Solaris) Это маска задержки символа забоя. Маска может иметь два значения: BS0 и BS1.

CBAUDEXT (c_cflag, Solaris) Расширенный диапазон скоростей передачи. Применяется для того, чтобы использовать скорости выше, чем B38400. (Скорость передачи мы рассмотрим в разделе 18.7.)

CCAR_OFLOW (c_cflag, FreeBSD, Mac OS X) Разрешает аппаратное управление выходным потоком данных с использованием сигнала модема RS-232 DCD (Data-Carrier-Detect – обнаружение несущего сигнала). То же самое, что устаревший флаг MDMBUF.

CCTS_OFLOW (c_cflag, FreeBSD, Mac OS X, Solaris) Разрешает аппаратное управление выходным потоком данных с использованием сигнала RS-232 CTS (Clear-To-Send – разрешение на передачу).

CDSR_OFLOW (c_cflag, FreeBSD, Mac OS X) Разрешает аппаратное управление выходным потоком данных с использованием сигнала RS-232 DSR (Data-Send-Ready – готовность к передаче).

CDTR_IFLOW (c_cflag, FreeBSD, MacOS X) Разрешает аппаратное управление выходным потоком данных с использованием сигнала RS-232 DTR (Data-Terminal-Ready – готовность терминала).

CIBAUDEXT (c_cflag, Solaris) Расширенный диапазон скоростей приема. Применяется для того, чтобы использовать скорости приема данных выше, чем B38400. (Скорость передачи мы рассмотрим в разделе 18.7.)

CIGNORE (c_cflag, FreeBSD, Mac OS X) Игнорировать флаги режима управления.

CLOCAL (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, то строки состояния модема игнорируются. Обычно это означает, что терминальное устройство подключено непосредственно к компьютеру. Если флаг не установлен, операция открытия терминального устройства блокируется до тех пор, пока, например, удаленный модем не ответит на звонок и не установит соединение.

CMSPAR (c_oflag, Linux) Выбор режима контроля четности по схеме MARK или SPACE. Если установлен флаг PARODD, то бит паритета всегда будет равен 1 (схема MARK). В противном случае бит паритета всегда будет равен 0 (схема SPACE).

CRDLY (c_oflag, XSI, Linux, Solaris) Маска задержки символа CR. Возможные значения маски: CR0, CR1, CR2 и CR3.

CREAD (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Установка этого флага разрешает прием данных.

CRTSCTS (c_cflag, FreeBSD, Linux, Mac OS X, Solaris) Назначение это флага зависит от платформы. В ОС Solaris он разрешает аппаратное управление исходящим потоком данных. На остальных трех платформах он разрешает аппаратное управление, как исходящим, так и входящим потоком данных (эквивалент CCTS_OFLOW|CRTS_OFLOW).

CRTS_IFLOW (c_cflag, FreeBSD, Mac OS X, Solaris) Разрешает аппаратное управление выходным потоком данных с использованием сигнала RS-232 RTS (Request-To-Send – запрос на передачу).

CRTSXOFF (c_cflag, Solaris) Разрешает аппаратное управление входным потоком данных. Проверяется состояние управляющего сигнала RS-232 RTS.

CSIZE (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Этот флаг является маской, которая определяет количество бит на символ при приеме/передаче. Этот размер не включает в себя бит паритета. Возможные значения маски: CS5, CS6, CS7 и CS8, которые соответствуют 5, 6, 7 и 8 битам на символ соответственно.

CSTOPB (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, используются два стоповых бита, в противном случае – один.

ECHO (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если установлен, то производится эхо-вывод введенных символов. Эхо-вывод может работать как в каноническом, так и в неканоническом режиме.

ECHOCTL (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлен флаг ECHO, управляющие символы ASCII (символы с восьмеричными кодами от 0 до 37 включительно), за исключением символов ASCII TAB, ASCII NL, START и STOP, выводятся в форме ^X, где X – символ, сформированный из кода управляющего символа путем добавления к нему восьмеричного числа 100. Это означает, например, что управляющий символ Control-A (восьмеричный код 1) будет выведен как ^A. Кроме того, символ ASCII DELETE (восьмеричный код 177) будет выводиться как ^?. Если флаг не установлен, управляющие символы ASCII выводятся как есть. Как и в случае с флагом ECHO, этот флаг воздействует на вывод управляющих символов как в каноническом, так и в неканоническом режиме.

Следует отметить, что в некоторых системах символ EOF выводится несколько иначе, т. к. обычное его значение – Control-D. (Control-D – это ASCII-символ EOT, который на некоторых терминалах вызывает разрыв связи.) Подробнее см. в справочном руководстве.

ECHOE (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлен флаг ICANON, при вводе символа ERASE производится стирание последнего символа в текущей строке на дисплее. Обычно это осуществляется драйвером терминала путем записи последовательности трех символов: шаг назад (backspace), пробел (space), шаг назад (backspace).

Если драйвер терминала поддерживает символ WERASE, установка флага ECHOE приводит к тому, что стирание последнего слова также выполняется за счет записи одной или более последовательностей этих же трех символов.

Если поддерживается флаг ECHOPRT, то данное описание ECHOE подразумевает, что ECHOPRT не установлен.

ECHOK (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлен флаг ICANON, символ KILL стирает текущую строку на дисплее или выводит символ NL (чтобы показать, что строка была стерта).

Если поддерживается флаг ECHOKE, то данное описание ECHOK подразумевает, что ECHOKE не установлен.

ECHOKE (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлен флаг ICANON, символ KILL стирает каждый символ в текущей строке на дисплее. Способ, которым это достигается, зависит от установки флагов ECHOE и ECHOPRT.

ECHONL (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлен флаг ICANON, эхо-вывод символа NL производится даже тогда, когда флаг ECHO не установлен.

ECHOPRT (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлены флаги ICANON и ECHO, то ввод символа ERASE (и символа WERASE, если он поддерживается) приводит к тому, что все удаляемые символы будут выводиться на печать. Это бывает удобно при работе с печатающими терминалами, так как позволяет точно увидеть, какие символы были стерты.

EXTPROC (c_oflag, XSI, Linux, Solaris) Если флаг установлен, каноническая обработка символов выполняется независимо от ОС. Например, устройство связи может само производить некоторую обработку данных, связанную с дисциплиной обслуживания линии связи. Аналогичная обработка производится при работе с псевдотерминалами (глава 19).

FFDLY (c_oflag, XSI, Linux, Solaris) Маска задержки символа FF (перевод страницы). Возможные значения маски: FF0 и FF1.

FLUSHO (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) При установке этого флага производится сброс очереди вывода. Этот флаг устанавливается при вводе символа DISCARD и сбрасывается при повторном вводе этого символа. Кроме того, можно установить этот флаг напрямую.

HUPCL (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, то после того как последний процесс закроет терминальное устройство, модемное соединение будет разорвано.

ICANON (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, вступает в силу канонический режим (раздел 18.10), в котором разрешена обработка символов EOF, EOL, EOL2, ERASE, KILL, REPRINT, STATUS и WERASE. Вводимые символы собираются в строки. В каноническом режиме запрос на чтение из очереди ввода не может быть удовлетворен немедленно, если не получено хотя бы MIN байт или не истек срок тайм-аута TIME после приема последнего байта. Подробности см. в разделе 18.11.

ICRNL (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом сброшен флаг IGNCR, принимаемые символы CR преобразуются в символы NL.

IEXTEN (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, распознаются и обрабатываются дополнительные специальные символы, определяемые реализацией.

IGNBRK (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, входной псевдосимвол BREAK игнорируется. Из описания флага BRKINT вы узнаете, когда псевдосимвол BREAK генерирует сигнал SIGINT, а когда может быть прочитан как обычные данные.

IGNCR (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, входной символ CR игнорируется. Если этот флаг сброшен, то становится возможным прием символа CR или его преобразование в символ NL при установленном флаге ICRNL.

IGNPAR (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, входной байт, принятый с ошибкой кадровой синхронизации (за исключением псевдосимвола BREAK) или с ошибкой контроля четности, игнорируется.

IMAXBEL (c_iflag, FreeBSD, Linux, Mac OS X, Solaris) Если установлен, при переполнении очереди ввода выдается звуковой сигнал.

INLCR (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, принимаемые символы NL преобразуются в символы CR.

INPCK (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, разрешается проверка бита паритета при вводе. Если сброшен, проверка бита паритета при вводе не производится.
«Контроль четности» и «проверка бита паритета» – это разные понятия. За контроль четности отвечает флаг PARENB. Установка этого флага обычно приводит к тому, что драйвер последовательного интерфейса генерирует биты четности для исходящих символов и проверяет для входящих. Флаг PARODD определяет схему контроля четности – ODD (чет) или EVEN (нечет). Если входящий символ поступает с неверным значением бита паритета, то проверяется состояние флага INPCK. Если он установлен, проверяется состояние флага IGNPAR (чтобы определить, следует ли игнорировать символ, поступивший с ошибкой контроля паритета). Если байт не должен игнорироваться, то проверяется состояние флага PARMRK, чтобы узнать, следует ли передавать процессу символы, принятые с ошибкой.

ISIG (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, при обработке входящих символов выполняется проверка необходимости генерации сигнала (символы INTR, QUIT, SUSP и DSUSP). Если был принят один из этих символов, будет сгенерирован соответствующий сигнал.

ISTRIP (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, в принятых байтах сбрасывается 8-й бит. Если флаг сброшен, обрабатываются все 8 бит.

IUCLC (c_iflag, Linux, Solaris) Если флаг установлен, символы верхнего регистра при вводе преобразуются в символы нижнего регистра.

IXANY (c_iflag, XSI, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, разрешается возобновление вывода по любому символу.

IXOFF (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, разрешено управление входным потоком с помощью символов START/STOP. Когда возникает угроза переполнения очереди ввода, драйвер терминала отправляет символ STOP. Этот символ должен распознаваться устройством, отправляющим данные, и вызывать приостановку передачи. Позднее, когда очередь ввода освободится, драйвер терминала отправит символ START, и передающее устройство сможет продолжить передачу данных.

IXON (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, разрешено управление выходным потоком с помощью символов START/STOP. Когда драйвер терминала получает символ STOP, он приостанавливает вывод данных. Когда драйвер терминала получит символ START, он возобновит вывод данных. Если этот флаг сброшен, символы START и STOP будут передаваться читающему процессу.

MDMBUF (c_cflag, FreeBSD, Mac OS X) Разрешает аппаратное управление потоком данных с использованием сигнала DCD модема. Это устаревшее название флага CCAR_OFLOW.

NLDLY (c_oflag, XSI, Linux, Solaris) Маска задержки символа NL. Возможные значения маски: NL0 и NL1.

NOFLSH (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) По умолчанию, когда драйвер терминала генерирует сигнал SIGINT или SIGQUIT, обе очереди (ввода и вывода) сбрасываются. Кроме того, когда генерируется сигнал SIGSUSP, сбрасывается очередь ввода. Если установлен флаг NOFLSH, то при генерации сигналов содержимое очередей не сбрасывается.

NOKERNINFO (c_oflag, XSI, FreeBSD, Mac OS X) Установка флага предотвращает вывод информации о группе процессов переднего плана при вводе символа STATUS. Независимо от состояния флага символ STATUS вызывает генерацию сигнала SIGINFO, который посылается группе процессов переднего плана.

OCRNL (c_oflag, XSI, FreeBSD, Linux, Solaris) Если флаг установлен, символы CR при выводе преобразуются в символы NL.

OFDEL (c_oflag, XSI, Linux, Solaris) Если флаг установлен, в качестве символа-заполнителя выводится символ ASCII DEL, в противном случае – ASCII NUL. Подробности см. в описании флага OFILL.

OFILL (c_oflag, XSI, Linux, Solaris) Если флаг установлен, вместо временной задержки будут передаваться символы-заполнители (ASCII DEL либо ASCII NUL). Существует шесть масок задержки: BSDLY, CRDLY, FFDLY, NLDLY, TABDLY и VTDLY.

OLCUC (c_oflag, Linux, Solaris) Если флаг установлен, символы нижнего регистра при выводе преобразуются в символы верхнего регистра.

ONLCR (c_oflag, XSI, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, то на выходе символы NL преобразуются в последовательности символов CR-NL.

ONLRET (c_oflag, XSI, FreeBSD, Linux, Solaris) Если флаг установлен, предполагается, что на выходе символ NL должен выполнять функцию символа возврата каретки.

ONOCR (c_oflag, XSI, FreeBSD, Linux, Solaris) Если флаг установлен, символ CR, находящийся в начале строки, не выводится.

ONOEOT (c_oflag, FreeBSD, Mac OS X) Если флаг установлен, символ EOT (^D) при выводе уничтожается. Это может потребоваться при работе с некоторыми терминалами, которые интерпретируют символ Control-D как разрыв соединения.

OPOST (c_oflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, выполняется дополнительная обработка выводимых данных, зависящая от реализации. В табл. 18.4 перечисляются различные флаги, определяемые отдельными реализациями.

OXTABS (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, при выводе символы табуляции заменяются пробелами. При использовании этого флага возникает тот же эффект, что и при установке маски задержки символа горизонтальной табуляции (TABDLY) в значение XTABS или TAB3.

PARENB (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Разрешает генерацию бита паритета для исходящих символов и его проверку для входящих символов. Если установлен флаг PARODD, контроль ведется по четности, в противном случае – по нечетности. Дополнительные сведения по этой теме см. в описаниях флагов INPCK, IGNPAR и PARMRK.

PAREXT (c_cflag, Solaris) Выбор схемы контроля четности MARK/SPACE. Если установлен флаг PARODD, бит паритета всегда будет равен 1 (MARK). В противном случае – 0 (SPACE).

PARMRK (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом сброшен флаг IGNPAR, входной байт, принятый с ошибкой в кадровой синхронизации (за исключением псевдосимвола BREAK) или с ошибкой контроля четности, передается процессу в виде последовательности из трех символов: \377, \0, *X*, где *X* – байт, принятый с ошибкой. Если флаг ISTRIP сброшен, то обычный символ \377 передается процессу в виде двух символов: \377, \377. Если флаги IGNPAR и PARMRK не установлены, входной байт, принятый с ошибкой в кадровой синхронизации (за исключением псевдосимвола BREAK) или с ошибкой контроля четности, передается процессу в виде одного байта \0.

PARODD (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, выбирается схема контроля бита паритета по четности исходящих и входящих данных. В противном случае бит паритета проверяется на нечетность. Обратите внимание: управление контролем четности производится с помощью флага PARENB.

Кроме того, флаг PARODD используется для выбора контроля бита паритета по схеме MARK или SPACE, если установлен флаг CMSPAR либо PAREXT.

PENDIN (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, при вводе очередного символа будут напечатаны все символы, которые еще не были прочитаны из очереди ввода. Действие этого флага аналогично тому, что происходит при нажатии клавиши REPRINT.

TABDLY (c_oflag, XSI, Linux, Solaris) Маска задержки символа горизонтальной табуляции. Возможные значения маски: TAB0, TAB1, TAB2 и TAB3.

Значение маски XTABS эквивалентно TAB3. Это значение заставляет систему заменять символы табуляции пробелами. При этом предполагается, что расстояние между соседними позициями табуляции составляет восемь пробелов, но его можно изменить.

TOSTOP (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом реализация поддерживает управление заданиями, то при попытке вывода на управляющий терминал группе фоновых процессов посылается сигнал SIGTTOU. По умолчанию этот сигнал приостанавливает работу процессов в группе. Этот сигнал не генерируется драйвером, если фоновый процесс, который произвел попытку записи в управляющий терминал, либо заблокировал сигнал, либо игнорирует его.

VTDLY (c_oflag, XSI, Linux, Solaris) Маска задержки символа вертикальной табуляции. Возможные значения маски: VT0 и VT1.

XCASE (c_lflag, Linux, Solaris) Если флаг установлен и при этом так же установлен флаг ICANON, все исходящие символы преобразуются в верхний регистр, а входящие – в нижний. В этом случае ввод символа верхнего регистра необходимо предварять символом обратного слэша. Аналогично при выводе символов верхнего регистра система также предваряет их символом обратного слэша. (Этот флаг считается устаревшим, поскольку сейчас терминалы, которые могут отображать только символы верхнего регистра, практически не используются.)

18.6. Команда stty

Состояние всех флагов, описанных в предыдущем разделе, может быть проверено и изменено из программы с помощью функций tcgetattr и tcsetattr (раздел 18.4) и из командной строки (или из сценариев командной оболочки) с помощью команды stty(1). Эта команда представляет собой упрощенный интерфейс к первым шести функциям из табл. 18.5. Если запустить эту команду с ключом -a, она выведет все характеристики терминала:

```
$ stty -a
speed 9600 baud; 25 rows; 80 columns;
lflags: icanon isig iexten echo echoe -echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -flusho pendin -nokerninfo
        -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -ocrnl -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crtscts
        -dsrflow -dtrflow -mdmbuf
cchars: discard = ЋO; dsusp = ЋY; eof = ЋD; eol = <undef>;
        eol2 = <undef>; erase = ЋH; erase2 = Ћ?; intr = ЋC; kill = ЋU;
        lnext = ЋV; min = 1; quit = Ћ\; reprint = ЋR; start = ЋQ;
        status = ЋT; stop = ЋS; susp = ЋZ; time = 0; werase = ЋW;
```

Дефис, предшествующий имени флага, означает, что флаг сброшен. В последних четырех строках выводятся текущие значения специальных символов (раздел 18.3). В первой строке выводится количество строк и символов в строке для текущего терминала – более подробно мы обсудим эти величины в разделе 18.12.

Для получения и изменения характеристик терминала команда stty использует стандартное устройство ввода. Некоторые старые версии команды использовали для этих целей стандартное устройство вывода, однако стандарт POSIX.1 явно требует, чтобы использовалось стандартное устройство ввода. Все четыре реализации, обсуждаемые в этой книге, предоставляют версию stty, которая работает со стандартным устройством ввода. Это означает, что если нас интересуют характеристики терминала tty1a, то можно ввести следующую команду:

```
stty -a </dev/tty1a
```

18.7. Функции для работы со скоростью передачи

Традиционно скорость передачи измеряется в *бодах*, что в наши дни можно трактовать как «биты в секунду». Хотя большинство терминалов используют одно и то же значение скорости как для ввода, так и для вывода, тем не менее возможно производить ввод и вывод на разных скоростях, если аппаратура это позволяет

```
#include <termios.h>
speed_t cfgetispeed(const struct termios *termptr);
speed_t cfgetospeed(const struct termios *termptr);
```

Обе возвращают значение скорости в бодах

```
int cfsetispeed(struct termios *termptr, speed_t speed);
int cfsetospeed(struct termios *termptr, speed_t speed);
```

Обе возвращают 0 в случае успеха, −1 в случае ошибки

Значение скорости, возвращаемое функциями `cfget` и передаваемое в виде аргументов функциям `cfset`, представляет собой одну из следующих констант: `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200` или `B38400`. Константа `B0` обозначает «разрыв соединения». Если с помощью функции `tcsetattr` устанавливается скорость вывода `B0`, то линии управления модемом не задействуются.

Большинство систем определяют две дополнительные константы: `B57600` и `B115200`.

При использовании этих функций необходимо четко понимать, что скорости ввода и вывода хранятся в структуре `termios`, как показано на рис. 18.3. Прежде чем вызвать какую-либо из функций `cfget`, необходимо сначала получить содержимое структуры `termios` устройства с помощью функции `tcgetattr`. Аналогично, после установки значения скорости в структуре `termios` функциями `cfset` необходимо сохранить эту структуру с помощью функции `tcsetattr`. Если было установлено ошибочное значение скорости, то мы не узнаем об этом, пока не вызовем функцию `tcsetattr`.

Четыре функции, предназначенные для работы со значениями скорости, скрывают от прикладных программ различные способы представления скорости в разных реализациях. Так, например, системы, производные от BSD, сохраняют значения скорости в числовом виде (то есть скорость 9600 бод хранится как число 9600), тогда как Linux и производные от System V представляют скорость в виде битовой маски. Функции `cfget` возвращают, а функции `cfset` принимают значения скорости в том виде, в каком они хранятся в структуре `termios`.

18.8. Функции управления линией связи

Следующие четыре функции предоставляют возможность управлять процессом обмена между терминалами. Все четыре требуют, чтобы аргумент *filedes* представлял собой дескриптор терминального устройства, в противном случае они будут возвращать управление с признаком ошибки и кодом ENOTTY в переменной errno.

```
#include <termios.h>
int tcdrain(int filedes);
int tcflow(int filedes, int action);
int tcflush(int filedes, int queue);
int tcsendbreak(int filedes, int duration);
```

Все четыре возвращают 0 в случае успеха, −1 в случае ошибки

Функция tcdrain ожидает, пока не будут отправлены все выходные данные. Функция tcflow дает возможность управлять входным и выходным потоками данных. В аргументе *action* допускается передавать одно из следующих значений:

TCOOFF Приостановить вывод.

TCOON Возобновить ранее приостановленный вывод.

TCIOFF Система отправляет символ STOP, который должен заставить терминал приостановить передачу.

TCION Система отправляет символ START, который должен заставить терминал возобновить передачу.

Функция tcflush позволяет либо сбросить (удалить) данные из очереди ввода (которые были приняты драйвером терминала, но еще не были прочитаны процессом), либо немедленно отправить данные из очереди вывода (которые были записаны процессом, но еще не отправлены). В аргументе *queue* допускается передавать одно из следующих значений:

TCIFLUSH Сбросить данные из очереди ввода.

TCOFLUSH Сбросить данные из очереди вывода.

TCIOFLUSH Сбросить данные из обеих очередей.

Функция tcsendbreak отправляет последовательность нулевых битов в течение заданного времени. Если в аргументе *duration* передается значение 0, продолжительность передачи будет находиться в диапазоне от 0,25 до 0,5 секунды. Стандарт POSIX.1 указывает, что продолжительность передачи при ненулевом значении аргумента *duration* определяется самой реализацией.

18.9. Идентификация терминала

Традиционно управляющий терминал в большинстве версий UNIX соответствует устройству /dev/tty. Стандарт POSIX.1 определяет функции, которые могут использоваться для получения имени управляющего терминала во время выполнения.

```
#include <stdio.h>

char *ctermid(char *ptr);
```

Возвращает указатель на строку с именем управляющего терминала в случае успеха, указатель на пустую строку – в случае ошибки

Если в аргументе *ptr* передается непустой указатель, то предполагается, что он указывает на буфер длиной не менее L_ctermid байт. В этом буфере будет сохранено имя управляющего терминала вызывающего процесса. Константа L_ctermid определена в файле <stdio.h>. Если в аргументе *ptr* был передан пустой указатель, функция выделит место для буфера (обычно в статической области памяти) и сохранит строку с именем управляющего терминала вызывающего процесса в этом буфере.

В обоих случаях функция передает адрес буфера вызывающему процессу в виде возвращаемого значения. Поскольку большинство версий UNIX используют в качестве имени управляющего терминала /dev/tty, эта функция предназначена для обеспечения переносимости приложений на другие операционные системы.

> На всех четырех платформах, описываемых в данной книге, функция ctermid возвращает имя /dev/tty.

Пример – функция ctermid

В листинге 18.3 приводится реализация функции ctermid стандарта POSIX.1.

Листинг 18.3. Функция ctermid стандарта POSIX.1

```
#include <stdio.h>
#include <string.h>

static char ctermid_name[L_ctermid];

char *
ctermid(char *str)
{
    if (str == NULL)
        str = ctermid_name;
    return(strcpy(str, "/dev/tty")); /* функция strcpy() вернет str */
}
```

Обратите внимание: мы никак не защищены от переполнения буфера, предоставляемого вызывающим процессом, поскольку у нас нет возможности определить его размер.

Для приложений UNIX больший интерес представляют другие две функции. Это функция isatty, которая возвращает значение «истина», если дескриптор является дескриптором терминального устройства, и ttyname, которая возвращает полное имя файла устройства терминала.

```
#include <unistd.h>

int isatty(int filedes);
```

Возвращает 1 (истина), если filedes представляет терминальное устройство, 0 (ложь) – в противном случае

```
char *ttyname(int filedes);
```

Возвращает указатель на строку с полным именем специального файла устройства, соответствующего терминалу, или NULL в случае ошибки

Пример – функция isatty

Функция isatty тривиальна в реализации, что хорошо видно из листинга 18.4. Она просто пытается вызвать одну из терминальных функций (которая в случае успеха ничего особенного не делает) и проверяет возвращаемое значение.

Листинг 18.4. Функция isatty стандарта POSIX.1

```
#include <termios.h>
int

isatty(int fd)
{
    struct termios ts;

    return(tcgetattr(fd, &ts) != -1); /* истина, если нет ошибки (это терминал) */
}
```

Протестируем работу нашей функции isatty с помощью программы из листинга 18.5.

Листинг 18.5. Тест функции isatty

```
#include "apue.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? "tty" : "не tty");
    printf("fd 1: %s\n", isatty(1) ? "tty" : "не tty");
    printf("fd 2: %s\n", isatty(2) ? "tty" : "не tty");
    exit(0);
}
```

После запуска этой программы мы получили следующие результаты:

```
$ ./a.out
fd 0: tty
```

```
fd 1: tty
fd 2: tty
$ ./a.out </etc/passwd 2>/dev/null
fd 0: не tty
fd 1: tty
fd 2: не tty
```

Пример – функция ttyname

Функция `ttyname` (листинг 18.6) гораздо сложнее, так как она должна просмотреть весь список устройств и отыскать совпадение.

Листинг 18.6. Функция ttyname стандарта POSIX.1

```
#include <sys/stat.h>
#include <dirent.h>
#include <limits.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
#include <stdlib.h>

struct devdir {
    struct devdir *d_next;
    char          *d_name;
};

static struct devdir *head;
static struct devdir *tail;
static char          pathname[_POSIX_PATH_MAX + 1];

static void
add(char *dirname)
{
    struct devdir *ddp;
    int len;
    len = strlen(dirname);
    /*
     * Пропустить каталоги ., .. и /dev/fd.
     */
    if ((dirname[len-1] == '.') && (dirname[len-2] == '/' ||
       (dirname[len-2] == '.' && dirname[len-3] == '/')))
        return;
    if (strcmp(dirname, "/dev/fd") == 0)
        return;
    ddp = malloc(sizeof(struct devdir));
    if (ddp == NULL)
        return;

    ddp->d_name = strdup(dirname);
    if (ddp->d_name == NULL) {
        free(ddp);
        return;
    }
```

```
        ddp->d_next = NULL;
        if (tail == NULL) {
            head = ddp;
            tail = ddp;
        } else {
            tail->d_next = ddp;
            tail = ddp;
        }
    }

    static void
    cleanup(void)
    {
        struct devdir *ddp, *nddp;

        ddp = head;
        while (ddp != NULL) {
            nddp = ddp->d_next;
            free(ddp->d_name);
            free(ddp);
            ddp = nddp;
        }
        head = NULL;
        tail = NULL;
    }

    static char *
    searchdir(char *dirname, struct stat *fdstatp)
    {
        struct stat devstat;
        DIR *dp;
        int devlen;
        struct dirent *dirp;

        strcpy(pathname, dirname);
        if ((dp = opendir(dirname)) == NULL)
            return(NULL);
        strcat(pathname, "/");
        devlen = strlen(pathname);
        while ((dirp = readdir(dp)) != NULL) {
            strncpy(pathname + devlen, dirp->d_name,
            _POSIX_PATH_MAX - devlen);

            /*
             * Пропустить псевдонимы.
             */
            if (strcmp(pathname, "/dev/stdin") == 0 ||
              strcmp(pathname, "/dev/stdout") == 0 ||
              strcmp(pathname, "/dev/stderr") == 0)
                continue;
            if (stat(pathname, &devstat) < 0)
                continue;
            if (S_ISDIR(devstat.st_mode)) {
```

```
                add(pathname);
                continue;
            }
            if (devstat.st_ino == fdstatp->st_ino &&
              devstat.st_dev == fdstatp->st_dev) {    /* совпадение найдено */
                closedir(dp);
                return(pathname);
            }
        }

        closedir(dp);
        return(NULL);
    }

    char *
    ttyname(int fd)
    {
        struct stat fdstat;
        struct devdir *ddp;
        char *rval;

        if (isatty(fd) == 0)
            return(NULL);
        if (fstat(fd, &fdstat) < 0)
            return(NULL);
        if (S_ISCHR(fdstat.st_mode) == 0)
            return(NULL);

        rval = searchdir("/dev", &fdstat);
        if (rval == NULL) {
            for (ddp = head; ddp != NULL; ddp = ddp->d_next)
                if ((rval = searchdir(ddp->d_name, &fdstat)) != NULL)
                    break;
        }

        cleanup();
        return(rval);
    }
```

Функция просматривает каталог `/dev` и отыскивает запись с указанным номером устройства и индексным узлом. Мы уже говорили в разделе 4.23, что каждая файловая система обладает уникальным номером устройства (поле `st_dev` структуры `stat`, раздел 4.2), а каждая запись в каталоге – уникальным номером индексного узла (поле `st_ino` структуры `stat`). Предполагается, что когда функция обнаружит запись с соответствующим номером устройства и номером индексного узла, можно сделать вывод, что было найдено требуемое устройство. Функция также могла бы проверить, совпадает ли содержимое поля `st_rdev` с заданными старшим и младшим номерами устройства и является ли найденный файл специальным файлом символьного устройства. Но поскольку она уже убедилась, что переданный ей дескриптор является терминальным устройством и специальным файлом символьного устройства (а номер устройства и номер индексного узла в системе являются уникальными), то отсутствует и необходимость выполнения дополнительных проверок.

Специальный файл устройства терминала может находиться в одном из подкаталогов каталога /dev. Поэтому необходимо обойти все дерево подкаталогов каталога /dev. Мы пропускаем специальные каталоги /dev/., /dev/.. и /dev/fd. Мы также не рассматриваем псевдонимы /dev/stdin, /dev/stdout и /dev/stderr, поскольку они являются символическими ссылками, ведущими в каталог /dev/fd.

Работоспособность нашей функции ttyname можно проверить с помощью программы, приведенной в листинге 18.7.

Листинг 18.7. Проверка функции ttyname

```
#include "apue.h"

int
main(void)
{
    char *name;

    if (isatty(0)) {
        name = ttyname(0);
        if (name == NULL)
            name = "не определено";
    } else {
        name = "не tty";
    }
    printf("fd 0: %s\n", name);
    if (isatty(1)) {
        name = ttyname(1);
        if (name == NULL)
            name = "не определено";
    } else {
        name = "не tty";
    }
    printf("fd 1: %s\n", name);
    if (isatty(2)) {
        name = ttyname(2);
        if (name == NULL)
            name = "не определено";
    } else {
        name = "не tty";
    }
    printf("fd 2: %s\n", name);
    exit(0);
}
```

После запуска программы из листинга 18.7 мы получили следующие результаты:

```
$ ./a.out < /dev/console 2> /dev/null
fd 0: /dev/console
fd 1: /dev/ttyp3
fd 2: не tty
```

18.10. Канонический режим

Канонический режим очень прост: мы запускаем операцию чтения, а драйвер терминала возвращает нам строку, когда она будет введена. Операция чтения завершается в следующих ситуациях:

- Когда прочитано запрошенное количество байт. Строка при этом может быть прочитана не до конца. Если прочитана только часть строки, то оставшаяся ее часть не будет потеряна; она может быть прочитана следующей операцией чтения.
- Когда достигнут разделитель строк. В разделе 18.3 мы уже говорили, что в каноническом режиме разделителями строк служат символы NL, EOL, EOL2 и EOF. Кроме того, в разделе 18.5 говорилось о том, что символ CR также рассматривается как разделитель строк, если установлен флаг `ICRNL`, а флаг `IGNCR` сброшен.

  Помните, что из этих пяти разделителей только один (EOF) уничтожается драйвером терминала в процессе обработки. Остальные четыре передаются читающему процессу в качестве последнего символа строки.
- Операция чтения также может завершиться, если был перехвачен сигнал и системный вызов не перезапускается автоматически (раздел 10.5).

Пример – функция getpass

Теперь продемонстрируем реализацию функции `getpass`, которая считывает пароль, вводимый пользователем с терминала. Эта функция вызывается программами `login`(1) и `crypt`(1). Чтобы прочитать пароль, функция должна отключить эхо-вывод, но оставить терминал в каноническом режиме, поскольку пароль представляет собой полноценную строку. В листинге 18.8 приводится типичная реализация этой функции в UNIX.

Несколько замечаний к данному примеру:

- Вместо того чтобы жестко «зашивать» в программу имя управляющего терминала (`/dev/tty`), мы воспользуемся функцией `ctermid`.
- Управляющий терминал необходим для выполнения операций записи/чтения, поэтому функция будет возвращать признак ошибки, если ей не удастся открыть соответствующее устройство для чтения и записи. Функция `getpass` в версии для BSD читает данные со стандартного ввода и выводит сообщение на стандартный вывод сообщений об ошибках, если ей не удалось открыть терминал для чтения и записи. В версии для System V вывод производится только на стандартный вывод сообщений об ошибках, а ввод – только из управляющего терминала.

Листинг 18.8. Реализация функции getpass

```
#include <signal.h>
#include <stdio.h>
#include <termios.h>

#define MAX_PASS_LEN 8 /* максимальное количество символов в пароле */
```

```
char *
getpass(const char *prompt)
{
    static char buf[MAX_PASS_LEN + 1]; /* нулевой байт в конце */
    char *ptr;
    sigset_t sig, osig;
    struct termios ts, ots;
    FILE *fp;
    int c;

    if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
        return(NULL);
    setbuf(fp, NULL);

    sigemptyset(&sig);
    sigaddset(&sig, SIGINT);            /* заблокировать SIGINT */
    sigaddset(&sig, SIGTSTP);           /* заблокировать SIGTSTP */
    sigprocmask(SIG_BLOCK, &sig, &osig); /* сохранить маску */

    tcgetattr(fileno(fp), &ts);         /* сохранить состояние терминала */
    ots = ts;                           /* скопировать структуру */
    ts.c_lflag &=  (ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fp), TCSAFLUSH, &ts);
    fputs(prompt, fp);

    ptr = buf;
    while ((c = getc(fp)) != EOF && c != '\n')
        if (ptr < &buf[MAX_PASS_LEN])
            *ptr++ = c;
    *ptr = 0;                           /* завершающий нулевой символ */
    putc('\n', fp);                     /* вывести символ перевода строки */

    tcsetattr(fileno(fp), TCSAFLUSH, &ots); /* восстановить состояние терминала */
    sigprocmask(SIG_SETMASK, &osig, NULL);  /* восстановить маску */
    fclose(fp);                             /* завершить работу с /dev/tty */
    return(buf);
}
```

- Функция блокирует сигналы SIGINT и SIGTSTP. Если этого не сделать, ввод символа INTR может завершить работу программы и оставить терминал в состоянии запрещенного эхо-вывода. Аналогично, ввод символа SUSP может приостановить работу программы и вернуть управление командной оболочке при запрещенном эхо-выводе. Сигналы остаются заблокированными до того момента, пока не будет восстановлено прежнее состояние терминала. Если эти сигналы будут сгенерированы во время чтения пароля, они останутся в состоянии ожидания обработки, пока функция не вернет управление. Существуют и другие способы обработки этих сигналов. Некоторые версии просто игнорируют сигнал SIGINT (сохранив его предыдущую диспозицию) во время работы функции getpass, восстанавливая диспозицию сигнала в исходное состояние перед выходом из функции. Другие версии перехватывают сигнал SIGINT (сохранив его предыдущую диспозицию) и после восстановления состояния терминала и диспо-

зиции сигнала посылают его себе с помощью функции kill. Но ни одна версия функции getpass не игнорирует, не блокирует и не перехватывает сигнал SIGQUIT – таким образом, ввод символа QUIT может прервать работу программы и, скорее всего, оставить терминал в состоянии отключенного эхо-вывода.

- Следует знать, что некоторые командные оболочки (в первую очередь Korn shell) включают эхо-вывод, когда они ожидают интерактивного взаимодействия с пользователем. Эти командные оболочки предоставляют возможность редактирования командной строки и поэтому корректируют состояние терминала всякий раз, когда вводится очередная команда. Таким образом, если запустить эту программу в одной из таких командных оболочек и затем прервать ее выполнение вводом символа QUIT, то режим эхо-вывода будет восстановлен. Другие командные оболочки, такие как Bourne shell, при аварийном завершении программы не восстанавливают состояние терминала и оставляют его с отключенным эхо-выводом. В этом случае можно восстановить эхо-вывод с помощью команды stty.
- Наша версия функции getpass для работы с управляющим терминалом использует функции стандартной библиотеки ввода-вывода. Мы специально назначаем небуферизованный режим работы потока – в противном случае могут возникнуть взаимовлияния между операциями чтения и записи, производимыми над потоком (нам потребовалось бы добавить несколько вызовов функции fflush). Можно было бы использовать функции небуферизованного ввода-вывода (глава 3), но тогда пришлось бы эмулировать поведение функции getc через функцию read.
- Мы считываем только первые восемь символов пароля. Любые последующие символы просто игнорируются.

Программа, приведенная в листинге 18.9, вызывает функцию getpass и выводит то, что было введено, позволяя убедиться, что символы ERASE и KILL обрабатываются должным образом (как и следует ожидать при работе в каноническом режиме).

Листинг 18.9. Вызов функции getpass

```
#include "apue.h"

char *getpass(const char *);

int
main(void)
{
    char *ptr;

    if ((ptr = getpass("Введите пароль:")) == NULL)
        err_sys("ошибка вызова функции getpass");
    printf("пароль: %s\n", ptr);

    /* здесь можно работать с паролем (например, зашифровать его) ... */

    while (*ptr != 0)
        *ptr++ = 0;      /* забить нулями, когда он стал больше не нужен */
```

```
    exit(0);
}
```

Всякий раз, когда программа завершает работу с паролем в виде открытого текста, она должна забить соответствующую область памяти нулями – просто для безопасности. Если программа завершится аварийно с созданием файла `core`, доступного на чтение для всех, или если другой процесс сможет просмотреть содержимое памяти нашего процесса, то пароль может быть прочитан. (Под «паролем в виде открытого текста» мы подразумеваем строку, которая вводится с клавиатуры в ответ на запрос функции `getpass`. В большинстве случаев, получив пароль, программы UNIX тут же шифруют его. Так, например, поле `pw_passwd` в файле паролей хранит пароль в зашифрованном, а не в открытом виде.)

18.11. Неканонический режим

Переход в неканонический режим осуществляется сбросом флага `ICANON` в поле `c_lflag` структуры `termios`. В неканоническом режиме принимаемые символы не собираются в строки, а следующие служебные символы ERASE, KILL, EOF, NL, EOL, EOL2, CR, REPRINT, STATUS и WERASE не обрабатываются.

Как мы уже говорили, канонический режим очень прост в использовании: система возвращает одну строку символов за раз. Но как узнать, когда система сможет вернуть нам данные при использовании неканонического режима? Если считывать данные по одному байту, это повлечет за собой непроизводительное расходование системных ресурсов. (Вспомните табл. 3.2, где приводились экспериментальные данные, которые наглядно показывают, что при удвоении объема считываемых данных в два раза снижаются накладные расходы.) Не всегда можно заранее сказать, какое количество данных находится в очереди ввода.

Решение состоит в том, чтобы сообщить системе, когда она должна возвращать управление – по прочтении заданного объема данных или по прошествии определенного времени. Для этих целей в массиве `c_cc` структуры `termios` предусмотрены два элемента MIN и TIME, с индексами `VMIN` и `VTIME`.

Элемент MIN определяет минимальное количество байт, по прочтении которого функция `read` должна возвращать управление. Элемент TIME задает количество десятых долей секунды, в течение которых следует ожидать поступления данных. Таким образом, существует четыре возможных случая.

Случай A: MIN > 0, TIME > 0

> Элемент TIME определяет время таймера, который запускается только после приема первого байта. Если MIN байт будет принято раньше, чем истечет время таймера, функция `read` вернет MIN байт. Если время таймера истечет до того, как будет принято MIN байт, функция `read` вернет столько байт, сколько было принято. (Будет возвращен по меньшей мере один байт, поскольку таймер запускается только после приема первого байта.) В этом случае вызывающий процесс блокируется до тех пор, пока

не будет принят первый байт. Если во время вызова функции `read` в очереди уже имеются данные, считается, что эти данные были приняты сразу же после входа в функцию `read`.

Случай Б: MIN > 0, TIME == 0

Функция `read` не вернет управление до тех пор, пока не будет прочитано MIN байт. Это может привести к тому, что процесс окажется заблокированным на неопределенное время.

Случай В: MIN == 0, TIME > 0

Элемент TIME задает время таймера чтения, который запускается в момент вызова функции `read`. (Сравните со случаем А, когда таймер запускается только после того, как будет принят первый байт.) Функция `read` вернет управление после приема первого байта или по истечении времени таймера. Если время таймера истечет до того, как будет принят хотя бы один байт, функция `read` вернет значение 0.

Случай Г: MIN == 0, TIME == 0

Если в очереди имеются какие-либо данные, функция `read` вернет либо запрошенное количество байт, либо столько, сколько доступно в очереди. Если очередь пуста, функция `read` сразу же вернет 0.

Важно понимать, что значение MIN – это всего лишь минимальный объем данных. Если программа запрашивает большее количество байт, то она вполне может получить объем данных вплоть до запрошенного количества. То же самое относится и к случаям В и Г, когда значение MIN равно нулю.

Таблица 18.7 обобщает все четыре случая неканонического ввода. В этой таблице число *nbytes* соответствует третьему аргументу функции `read` (максимальное количество байт, которое она может вернуть).

Таблица 18.7. Четыре случая неканонического ввода

	MIN > 0	**MIN == 0**
TIME > 0	**А:** `read` возвращает [MIN, *nbytes*] до того, как истечет время таймера; `read` возвращает [1, MIN] по истечении времени таймера. (Вызывающий процесс может оказаться заблокированным на неопределенное время.)	**В:** `read` возвращает [1, *nbytes*] до того, как истечет время таймера; `read` возвращает 0 по истечении времени таймера.
TIME == 0	**Б:** `read` возвращает [MIN, *nbytes*], если в очереди имеются данные. (Вызывающий процесс может оказаться заблокированным на неопределенное время.)	**Г:** `read` возвращает [0, *nbytes*] немедленно.

Следует знать, что стандарт POSIX.1 допускает, чтобы индексы `VMIN` и `VTIME` совпадали с индексами `VEOF` и `VEOL` соответственно. И действительно, в ОС Solaris это делается для обеспечения обратной совместимости с устаревшими версиями System V. Однако это порождает проблему переносимости. При переходе из неканонического в канониче-

ский режим мы вынуждены восстанавливать значения элементов с индексами VEOF и VEOL. Так, если VMIN совпадает с VEOF и мы при переходе из неканонического режима в канонический не восстановим значение этого элемента, который в случае VMIN обычно равен 1, то признаком конца файла станет символ Control-A. Самый простой способ решения этой проблемы – сохранять все содержимое структуры termios при переходе в неканонический режим и восстанавливать ее при возврате к каноническому режиму.

Пример

Программа из листинга 18.10 определяет функции tty_cbreak и tty_raw, которые служат для перевода терминала в режимы посимвольного (*cbreak*) и прозрачного (*raw*) ввода. (Термины *cbreak* и *raw* пришли из драйвера терминала Version 7.) Вернуть терминал в первоначальное состояние (предшествовавшее вызову любой из этих двух функций) можно с помощью функции tty_reset.

После вызова функции tty_cbreak нужно обратиться к функции tty_reset, прежде чем вызывать функцию tty_raw. То же самое относится к вызову функции tty_cbreak после вызова tty_raw. Это повышает вероятность того, что терминал останется в состоянии, пригодном к работе, если мы столкнемся с непредвиденными ошибками.

Также представлены две дополнительные функции: tty_atexit, которая может использоваться в качестве обработчика выхода, обеспечивая возврат терминала в первоначальное состояние при вызове функции exit, и tty_termios, которая возвращает указатель на оригинальную структуру termios, соответствующую каноническому режиму терминала.

Листинг 18.10. Установка режимов прозрачного и посимвольного ввода

```
#include "apue.h"
#include <termios.h>
#include <errno.h>

static struct termios save_termios;
static int ttysavefd = -1;
static enum { RESET, RAW, CBREAK } ttystate = RESET;

int
tty_cbreak(int fd) /* перевести терминал в режим посимвольного ввода */
{
    int err;
    struct termios buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* копия структуры */

    /*
     * Отключить эхо-вывод и выйти из канонического режима.
     */
```

```
    buf.c_lflag &=  (ECHO | ICANON);

    /*
     * Случай Б: минимум 1 байт, время ожидания не ограничено.
     */
    buf.c_cc[VMIN] = 1;
    buf.c_cc[VTIME] = 0;
    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);

    /*
     * Убедиться, что были произведены все изменения. Функция tcsetattr может
     * вернуть 0, даже если выполнена лишь часть изменений.
     */
    if (tcgetattr(fd, &buf) < 0) {
        err = errno;
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = err;
        return(-1);
    }
    if ((buf.c_lflag & (ECHO | ICANON)) || buf.c_cc[VMIN] != 1 ||
      buf.c_cc[VTIME] != 0) {

        /*
         * Были произведены лишь некоторые изменения.
         * Восстановить начальные настройки.
         */
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = EINVAL;
        return(-1);
    }
    ttystate = CBREAK;
    ttysavefd = fd;
    return(0);
}

int
tty_raw(int fd)   /* перевести терминал в режим прозрачного ввода (raw) */
{
    int err;
    struct termios buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* копия структуры */

    /*
     * Отключить эхо-вывод, выйти из канонического режима, отключить
     * расширенную обработку ввода, отключить обработку символов,
     * генерирующих сигналы.
     */
```

```
buf.c_lflag &=  (ECHO | ICANON | IEXTEN | ISIG);

/*
 * Не выдавать сигнал SIGINT по псевдосимволу BREAK, отключить
 * преобразование CR->NL, отключить проверку паритета ввода,
 * не сбрасывать 8-й бит, отключить управление выводом.
 */
buf.c_iflag &=  (BRKINT | ICRNL | INPCK | ISTRIP | IXON);

/*
 * Сбросить маску управления размером, отключить контроль четности.
 */
buf.c_cflag &=  (CSIZE | PARENB);

/*
 * Установить размер символа 8 бит/символ.
 */
buf.c_cflag |= CS8;

/*
 * Отключить обработку вывода.
 */
buf.c_oflag &=  (OPOST);

/*
 * Случай Б: минимум 1 байт, время ожидания не ограничено.
 */
buf.c_cc[VMIN] = 1;
buf.c_cc[VTIME] = 0;
if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
    return(-1);

/*
 * Убедиться, что были произведены все изменения. Функция tcsetattr может
 * вернуть 0, даже если выполнена лишь часть изменений.
 */
if (tcgetattr(fd, &buf) < 0) {
    err = errno;
    tcsetattr(fd, TCSAFLUSH, &save_termios);
    errno = err;
    return(-1);
}
if ((buf.c_lflag & (ECHO | ICANON | IEXTEN | ISIG)) ||
  (buf.c_iflag & (BRKINT | ICRNL | INPCK | ISTRIP | IXON)) ||
  (buf.c_cflag & (CSIZE | PARENB | CS8)) != CS8 ||
  (buf.c_oflag & OPOST) || buf.c_cc[VMIN] != 1 ||
  buf.c_cc[VTIME] != 0) {

    /*
     * Были произведены лишь некоторые изменения.
     * Восстановить начальные настройки.
     */
    tcsetattr(fd, TCSAFLUSH, &save_termios);
    errno = EINVAL;
```

```
        return(-1);
    }
    ttystate = RAW;
    ttysavefd = fd;
    return(0);
}

int
tty_reset(int fd) /* восстановить состояние терминала */
{
    if (ttystate == RESET)
        return(0);
    if (tcsetattr(fd, TCSAFLUSH, &save_termios) < 0)
        return(-1);
    ttystate = RESET;
    return(0);
}

void
tty_atexit(void) /* может быть установлена вызовом atexit(tty_atexit) */
{
    if (ttysavefd >= 0)
        tty_reset(ttysavefd);
}

struct termios *
tty_termios(void)   /* позволить вызывающему процессу */
{                   /* узнать начальное состояние терминала */
    return(&save_termios);
}
```

Мы определили режим посимвольного (cbreak) ввода следующим образом:

- Неканонический режим. Как уже упоминалось в начале главы, в этом режиме отключена обработка некоторых служебных символов при вводе. Генерация сигналов не запрещена, поэтому пользователь всегда сможет послать сигнал посредством ввода соотвествующих символов. Необходимо понимать, что вызывающий процесс должен предусмотреть их обработку, в противном случае есть вероятность, что сигнал приведет к завершению процесса и терминал останется в режиме посимвольного ввода.

  Как правило, при написании программ, изменяющих состояние терминала, нужно предусматривать обработку большинства сигналов. Это позволяет восстановить состояние терминала перед завершением приложения.
- Эхо-вывод отключен.
- За один раз читается как минимум один байт. Для этого мы записываем в элемент MIN значение 1, а в элемент TIME – значение 0. Это случай Б из табл. 18.7. Функция `read` не вернет управление до тех пор, пока не будет доступен для чтения хотя бы 1 байт.

Мы определили режим прозрачного (raw) ввода следующим образом:

- Неканонический режим. Мы также отключаем обработку символов, генерирующих сигналы (ISIG), и расширенную обработку символов при вводе (IEXTEN). Дополнительно мы запрещаем генерацию сигнала SIGINT при получении псевдосимвола BREAK, выключив флаг BRKINT.
- Эхо-вывод отключен.
- Запрещены преобразование CR->NL при вводе (ICRNL), проверка паритета (INPCK), сброс восьмого бита (ISTRIP) при вводе и управление выходным потоком (IXON).
- Размер символа 8 бит (CS8), запрещен контроль четности (PARENB).
- Запрещена обработка вывода (OPOST).
- За один раз читается как минимум один байт (MIN = 1, TIME = 0).

Программа из листинга 18.11 тестирует режимы прозрачного и посимвольного ввода.

Листинг 18.11. Тест режимов raw и cbreak

```
#include "apue.h"

static void
sig_catch(int signo)
{
    printf("перехвачен сигнал\n");
    tty_reset(STDIN_FILENO);
    exit(0);
}

int
main(void)
{
    int i;
    char c;

    if (signal(SIGINT, sig_catch) == SIG_ERR) /* предусматриваем обработку сигналов */
        err_sys("ошибка вызова функции signal(SIGINT)");
    if (signal(SIGQUIT, sig_catch) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGQUIT)");
    if (signal(SIGTERM, sig_catch) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGTERM)");

    if (tty_raw(STDIN_FILENO) < 0)
        err_sys("ошибка вызова функции tty_raw");
    printf("Переход в режим raw, выход из режима по нажатию DELETE\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        if ((c &= 255) == 0177) /* 0177 = ASCII DELETE */
            break;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("ошибка вызова функции tty_reset");
```

```
    if (i <= 0)
        err_sys("ошибка чтения");
    if (tty_cbreak(STDIN_FILENO) < 0)
        err_sys("ошибка вызова функции tty_cbreak");
    printf("\nПереход в режим cbreak, выход из режима по сигналу SIGINT\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        c &= 255;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("ошибка вызова функции tty_reset");
    if (i <= 0)
        err_sys("ошибка чтения");
    exit(0);
}
```

Запустив программу из листинга 18.11, мы сможем наблюдать за поведением терминала в этих двух режимах:

```
$ ./a.out
Переход в режим raw, выход из режима по нажатию DELETE
                                                    4
                                                    33
                                                      133
                                                         61
                                                           70
                                                             176
                                нажата клавиша DELETE
Переход в режим cbreak, выход из режима по сигналу SIGINT
1                               нажата клавиша Control-A
10                              нажата клавиша backspace
перехвачен сигнал               нажата клавиша прерывания
```

В режиме прозрачного ввода (raw) были нажаты клавиши Control-D (04) и функциональная клавиша F7. На данном терминале эта функциональная клавиша генерирует пять символов: *ESC* (033), *[* (0133), *1* (061), *8* (070) и ~ (0176). Обратите внимание: когда отключена обработка вывода (~OPOST), возврат каретки после ввода каждого символа не производится. Обратите также внимание на то, что в режиме посимвольного ввода (cbreak) запрещена обработка некоторых служебных символов (таких как символ конца файла (Control-D) и символ забоя (backspace)), тогда как символы, генерирующие сигналы, по-прежнему обрабатываются.

18.12. Размер окна терминала

Большинство версий UNIX предоставляют возможность определить размер окна терминала и сообщить процессам из группы процессов переднего плана об изменении размеров. Каждому терминалу и псевдотерминалу ядро ставит в соответствие структуру winsize:

```
struct winsize {
    unsigned short ws_row;    /* количество строк */
    unsigned short ws_col;    /* количество символов в строке */
    unsigned short ws_xpixel; /* горизонтальный размер в пикселях */
                              /* (не используется) */
    unsigned short ws_ypixel; /* вертикальный размер в пикселях (не используется) */
};
```

Правила работы со структурой:

- Текущее содержимое структуры можно получить с помощью команды TIOCGWINSZ функции ioctl (раздел 3.15).
- Можно записать новое содержимое структуры в ядро, используя команду TIOCSWINSZ функции ioctl. Если новые размеры окна отличаются от текущих, группе процессов переднего плана будет послан сигнал SIGWINCH. (Обратите внимание, что согласно табл. 10.1 по умолчанию этот сигнал игнорируется.)
- Кроме хранения текущих значений и посылки сигнала при их изменении, ядро больше ничего не делает с этой структурой. Интерпретация структуры полностью возлагается на прикладные программы.

Основное назначение этой функциональной возможности – извещать приложения (такие как редактор vi) об изменении окна терминала. Когда сигнал будет доставлен, приложение сможет узнать новые размеры окна и перерисовать экран.

Пример

В листинге 18.12 приводится исходный текст программы, которая выводит текущие размеры окна и приостанавливается. Каждый раз, когда изменяется размер окна, программа перехватывает сигнал SIGWINCH и выводит новые значения размеров. Чтобы завершить работу программы, мы должны сгенерировать сигнал.

Листинг 18.12. Вывод информации о размерах окна

```
#include "apue.h"
#include <termios.h>

#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

static void
pr_winsize(int fd)
{
    struct winsize size;

    if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("ошибка выполнения команды TIOCGWINSZ");
    printf("%d строк, %d символов в строке\n", size.ws_row, size.ws_col);
}
```

```
static void
sig_winch(int signo)
{
    printf("доставлен сигнал SIGWINCH\n");
    pr_winsize(STDIN_FILENO);
}

int
main(void)
{
    if (isatty(STDIN_FILENO) == 0)
        exit(1);
    if (signal(SIGWINCH, sig_winch) == SIG_ERR)
        err_sys("ошибка вызова функции signal");
    pr_winsize(STDIN_FILENO); /* вывести начальные размеры окна */
    for ( ; ; )               /* и приостановиться */
        pause();
}
```

Запустив эту программу на терминале с изменяемым размером окна, мы получили следующие результаты:

```
$ ./a.out
35 строк, 80 символов в строке      начальный размер окна
доставлен сигнал SIGWINCH           изменен размер окна: перехвачен сигнал
40 строк, 123 символов в строке
доставлен сигнал SIGWINCH           и еще раз
42 строк, 33 символов в строке
Б? $                                нажата клавиша прерывания,
                                    чтобы завершить программу
```

18.13. termcap, terminfo и curses

Схема хранения информации о терминалах под названием `termcap` (от «terminal capability» – возможности терминала) была разработана в Беркли для поддержки текстового редактора `vi`. Она включает текстовый файл `/etc/termcap` и набор процедур для работы с ним. Файл `termcap` содержит характеристики различных терминалов: какие возможности поддерживаются терминалом (количество строк и символов в строке, поддержка символа забоя и т. п.) и как заставить терминал выполнять определенные операции (очистку экрана, перемещение курсора в заданную позицию и прочие). Убрав эту информацию из кода программы и поместив ее в обычный текстовый файл, который легко можно отредактировать, разработчики сделали возможным использование редактора `vi` на самых разных терминалах.

Процедуры поддержки `termcap` также были извлечены из редактора `vi` и размещены в отдельной библиотеке под названием `curses`. В эту библиотеку было добавлено много новых функций, что сделало ее пригодной для работы в составе любой программы, которая должна управлять выводом информации на экран.

Но у схемы termcap были недостатки. Все больше описаний терминалов добавлялось в файл termcap, и все больше времени требовалось программам всякий раз, когда им необходимо было отыскать в нем описание какого-либо терминала. Кроме того, для обозначения различных характеристик терминалов использовались двухсимвольные имена. Эти недостатки привели к появлению новой схемы terminfo и связанной с ней библиотеки curses. Описания терминалов в схеме terminfo хранятся в скомпилированном виде, что значительно ускоряет поиск нужной информации во время выполнения программы. Впервые схема terminfo появилась в SVR2 и с тех пор используется во всех версиях System V.

> Системы, основанные на System V, традиционно используют схему terminfo, а BSD-системы – termcap, но современные системы обычно поддерживают обе схемы. Однако Mac OS X поддерживает только terminfo.

Описание terminfo и библиотеки curses можно найти в [Goodheart 1991], но весь тираж этой книги уже распродан. В книге [Strang 1986] описывается версия библиотеки curses из Беркли. В книге [Strang, Mui, and O'Reilly 1988] содержится описание termcap и terminfo.

> Библиотеку ncurses, которая представляет собой свободно распространяемую версию, совместимую с интерфейсом curses SVR4, вы найдете по адресу: *http://invisible-island.net/ncurses/ncurses.html*.

Ни termcap, ни terminfo сами по себе не имеют отношения к задачам, которые мы рассматривали в этой главе (изменение режима терминала, изменение значений управляющих символов, обслуживание размеров окна и тому подобное). На самом деле они предоставляют средства выполнения типичных операций (очистка экрана, перемещение курсора) для различных терминалов. С другой стороны, библиотека curses действительно помогает при решении некоторых задач, которых мы касались в этой главе. Она предоставляет функции для перевода терминала в режим посимвольного и прозрачного ввода, включения и отключения эхо-вывода и т. п. Но изначально библиотека curses была разработана для простых алфавитно-цифровых терминалов, которые сегодня в большинстве своем заменены графическими терминалами.

18.14. Подведение итогов

Терминалы обладают множеством свойств и возможностей, большинство из которых можно контролировать и подстраивать под свои нужды. В этой главе мы описали большое количество функций, которые изменяют характеристики терминалов – флаги режимов и значения служебных символов. Мы подробно рассмотрели все специальные символы и флаги, которые могут быть сброшены или установлены.

Терминалы могут работать в двух режимах ввода – каноническом (построчный ввод) и неканоническом. Мы продемонстрировали примеры обоих режимов и показали функции для переключения терминала в устаревшие ре-

жимы прозрачного (raw) и посимвольного (cbreak) ввода. Также мы рассказали, как получить и изменить размеры окна терминала.

Упражнения

18.1. Напишите программу, которая вызывала бы функцию `tty_raw` и завершала работу (без восстановления канонического режима терминала). Если ваша система предоставляет команду `reset`(1) (она доступна на всех четырех платформах, обсуждаемых в этой книге), попробуйте с ее помощью восстановить режим ввода терминала.

18.2. Схему контроля четности – ODD или EVEN – можно задать с помощью флага `PARODD` в поле `c_cflag`. Программа `tip` в BSD, кроме того, позволяет задать значение бита паритета 0 или 1. Как она делает это?

18.3. Если в вашей системе команда `stty`(1) поддерживает элементы MIN и TIME, выполните следующее упражнение. Войдите в систему с двух терминалов и запустите редактор `vi` на одном из них. С помощью команды `stty` с другого терминала определите, какие значения MIN и TIME устанавливает редактор `vi` (так как этот редактор переводит терминал в неканонический режим). (Если ваш терминал работает под управлением многооконной системы, то же самое можно сделать, открыв два терминала в отдельных окнах.)

19

Псевдотерминалы

19.1. Введение

В главе 9 мы видели, что вход в систему осуществляется через терминальное устройство, которое автоматически реализует семантику терминала. Управление взаимодействием запускаемых программ с терминалом осуществляется модулем дисциплины обслуживания линии связи (рис. 18.2), что позволяет нам назначить специальные символы терминала (символы забоя, стирания строки, прерывания и пр.) и изменить другие его характеристики. Однако при входе в систему через сетевое соединение модуль дисциплины обслуживания линии связи между сетевым соединением и оболочкой входа не предоставляется автоматически. На рис. 9.5 показано, что семантика терминала в этом случае реализуется драйвером *псевдотерминала*.

Помимо входа в систему через сетевое соединение псевдотерминалы используются и в других случаях, которые мы будем рассматривать в этой главе. Обсуждение псевдотерминалов мы начнем с краткого обзора их применения, который завершится описанием некоторых особых случаев. После этого мы рассмотрим функции создания псевдотерминалов, предоставляемые различными платформами, и воспользуемся ими при написании программы, которую мы назвали `pty`. Мы покажем различные способы использования этой программы: создание журнала ввода-вывода терминала (программа `script`(1)) и запуск сопроцессов, не подверженных проблемам с буферизацией, с которыми мы столкнулись в программе из листинга 15.10.

19.2. Обзор

Термином *псевдотерминал* обозначается программное устройство, которое похоже на терминал, но не является им. На рис. 19.1 показана типичная схема использования псевдотерминала процессами. Вот ключевые моменты, на которые следует обратить особое внимание:

- Обычно процесс открывает ведущий (master) псевдотерминал, затем вызывается функция `fork`. Дочерний процесс создает новую сессию, открывает

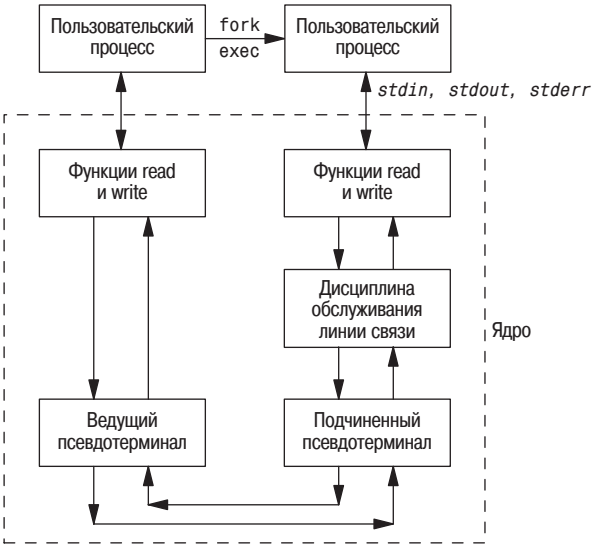


***Рис. 19.1.** Типичная схема взаимоотношения процессов, использующих псевдотерминал*

соответствующий подчиненный (slave) псевдотерминал, создает дубликаты дескрипторов стандартного ввода, стандартного вывода и стандартного вывода сообщений об ошибках и вызывает функцию exec. Подчиненный псевдотерминал становится управляющим терминалом дочернего процесса.

- Пользовательский процесс, расположенный на рисунке над подчиненным терминалом, считает, что его стандартный ввод, стандартный вывод и стандартный вывод сообщений связаны с терминальным устройством. Процесс может использовать любые функции из главы 18, предназначенные для работы с терминалом. Но поскольку подчиненный терминал не является настоящим терминальным устройством, функции, которые не будут иметь смысла (изменение скорости передачи, отправка псевдосимвола BREAK, проверка бита паритета и подобные), просто игнорируются.
- Все, что будет записано в ведущий псевдотерминал, появится на входе подчиненного псевдотерминала, и наоборот. То есть вывод процесса, владеющего ведущим псевдотерминалом, передается на вход процесса, владеющего подчиненным псевдотерминалом. Это очень напоминает двунаправленный канал, но благодаря наличию промежуточного модуля, реализующего дисциплину обслуживания линии связи, мы получаем дополнительные преимущества перед обычными каналами.

На рис. 19.1 показано, как реализованы псевдотерминалы в ОС FreeBSD, Mac OS X и Linux. В разделах 19.3.2 и 19.3.3 обсуждается, как открывать эти устройства.

В ОС Solaris псевдотерминалы построены на базе механизма STREAMS (раздел 14.4). На рис. 19.2 показано строение псевдотерминала на основе моду-

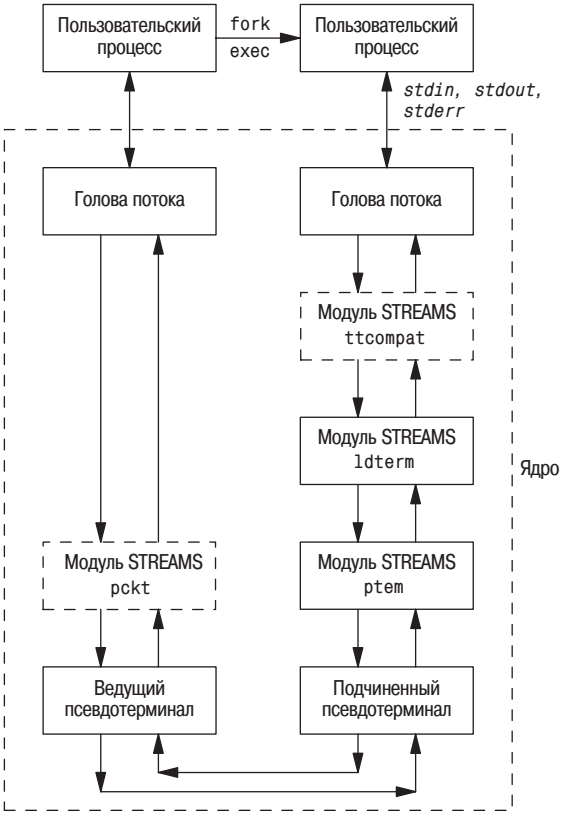


Рис. 19.2. *Схема реализации псевдотерминалов в ОС Solaris*

лей STREAMS в Solaris. Два модуля, которые изображены в виде пунктирных прямоугольников, являются необязательными. Модули `pckt` и `ptem` обеспечивают семантику псевдотерминала. Другие два модуля (`ldterm` и `ttcompat`) реализуют дисциплину обслуживания потока данных.

Обратите внимание, что между подчиненным псевдотерминалом и дочерним процессом расположены те же самые три модуля, что мы видели в выводе программы из листинга 14.9, запущенной после входа в систему через сетевое соединение. В разделе 19.3.1 мы покажем, как построить такую схему расположения модулей STREAMS.

Чтобы упростить последующие рисунки, мы не будем показывать на них «функции read и write» (рис. 19.1) и «голову потока» (рис. 19.2). Кроме того, псевдотерминал мы будем обозначать аббревиатурой PTY, а все модули, расположенные выше подчиненного PTY на рис. 19.2, будем объединять в один блок с названием «дисциплина обслуживания терминала», как на рис. 19.1.

А теперь рассмотрим типичные области применения псевдотерминалов.

Серверы сетевого входа в систему

Псевдотерминалы встроены в серверы, обеспечивающие возможность сетевого входа в систему. Примерами таких серверов являются `telnetd` и `rlogind`. Детальное описание службы `rlogin` вы найдете в главе 15 [Stevens 1990]. После запуска оболочки входа на удаленной машине мы получим схему, которая изображена на рис. 19.3. Аналогичные результаты будут получены при использовании сервера `telnetd`.

Между оболочкой входа и сервером `rlogind` показаны два вызова функции `exec`, потому что для идентификации пользователя обычно вызывается программа `login`.

Ключевым моментом в этой схеме является то, что программа, управляющая ведущим терминалом, параллельно производит чтение и запись в другой поток ввода-вывода. В данном примере этот поток ввода-вывода показан как блок TCP/IP. Это означает, что процесс должен использовать ту или иную форму мультиплексирования ввода-вывода (раздел 14.5), например `select` или `poll`, или разделиться на два процесса или потока.

Программа script

В большинстве версий UNIX имеется программа `script`(1), которая копирует входные и выходные данные терминала в файл. Достигается это за счет того, что программа размещает себя между терминалом и вызовом новой командной оболочки. На рис. 19.4 подробно показаны все взаимодействия между процессами при запуске программы `script`. В частности, рисунок показывает, что программа `script` обычно запускается из оболочки входа, которая затем просто ожидает ее завершения.

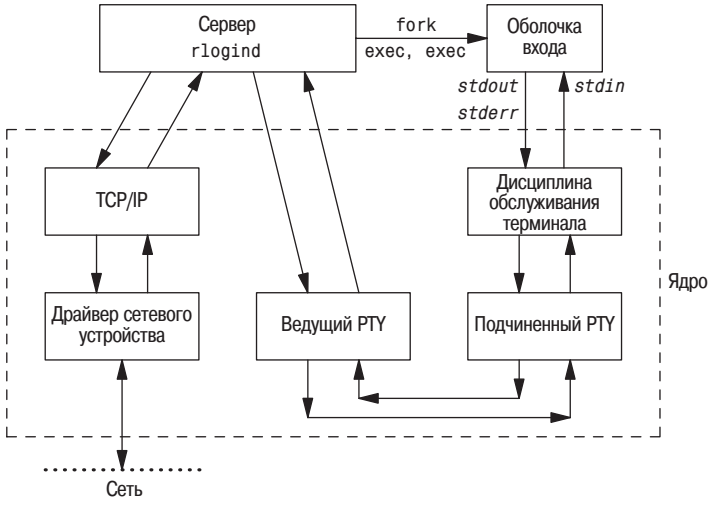


***Рис. 19.3.** Типичная схема взаимоотношения процессов при использовании службы rlogind*

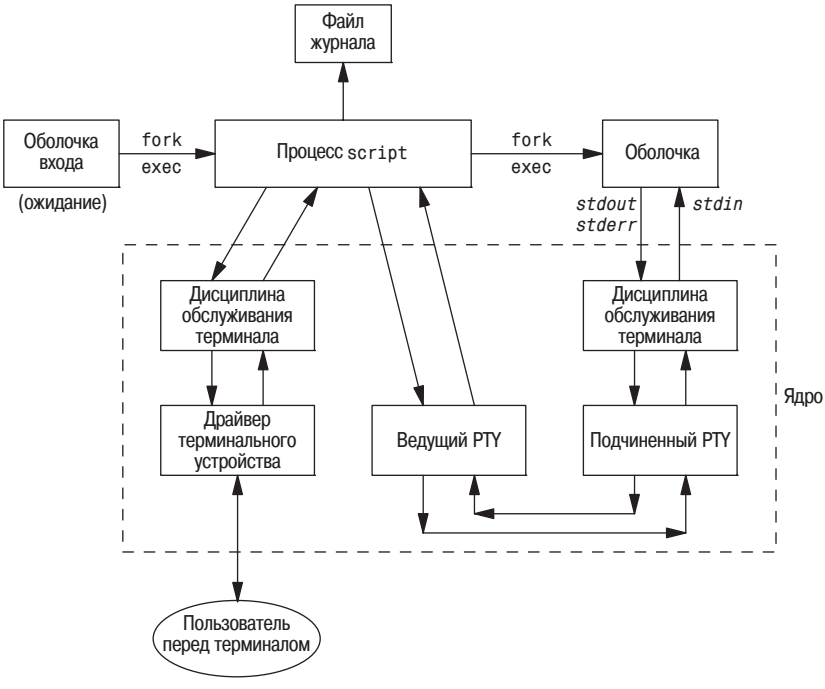


***Рис. 19.4.** Программа script*

Во время работы программы `script` весь вывод с терминала, который идет от модуля дисциплины обслуживания терминала, расположенного выше подчиненного PTY, копируется в файл журнала (который обычно называется `typescript`). Поскольку весь ввод с клавиатуры обычно выводится модулем дисциплины обслуживания, то в файл журнала попадает и все, что было введено с клавиатуры. Пароли, вводимые с клавиатуры, не могут попасть в файл журнала, поскольку во время ввода пароля эхо-вывод отключен.

> При работе над первым изданием этой книги Ричард Стивенс использовал программу `script` для захвата вывода программ-примеров, чтобы избежать опечаток, которые наверняка возникли бы при ручном наборе. Недостаток такого использования программы `script` заключался в том, что приходилось разбираться с представлением управляющих символов в файле журнала.

В разделе 19.5 мы разработаем универсальную программу `pty` и увидим, что из нее легко можно сделать версию программы `script` с помощью простенького сценария на языке командной оболочки.

Программа expect

Псевдотерминалы могут использоваться для управления интерактивными программами в неинтерактивном режиме. Множество программ требуют взаимодействия с терминалом. Одна из них – программа `passwd`(1), которая ожидает ввода пароля в ответ на приглашение.

Вместо того чтобы добавлять во все интерактивные программы поддержку работы в пакетном режиме, удобнее управлять ими из сценариев. Такую возможность предоставляет программа `expect` (см. [Libes 1990, 1991, 1994]). Она использует псевдотерминалы, подобные программе `pty` из раздела 19.5, для запуска других программ. Кроме того, программа `expect` предоставляет в наше распоряжение язык программирования для проверки вывода запущенных программ, что позволяет на основе анализа выводимых данных принимать решение о том, какие данные программа ожидает получить. Когда интерактивная программа запускается из сценария, мы не можем просто скопировать данные из сценария в программу и обратно. Вместо этого мы должны отправить программе некоторые данные, проанализировать полученный вывод и принять решение о том, что ввести в следующий раз.

Запуск сопроцессов

В примере сопроцесса из листинга 15.10 мы не могли использовать для взаимодействия с сопроцессом функции стандартной библиотеки ввода-вывода, потому что при работе с неименованными каналами стандартная библиотека ввода-вывода устанавливает режим полной буферизации для стандартных потоков ввода и вывода, что приводит к тупиковой ситуации. Если сопроцесс представляет собой скомпилированную программу, исходный код которой недоступен, мы не сможем добавить дополнительные вызовы функции `fflush`, чтобы решить эту проблему. Рисунок 15.8 иллюстрирует управление сопроцессом. Все, что нам необходимо, – это поместить псевдотерминал между двумя процессами, как показано на рис. 19.5, чтобы «обмануть» сопроцесс и заставить его думать, что он взаимодействует с терминалом, а не с другим процессом.

Теперь стандартный ввод и стандартный вывод сопроцесса с его точки зрения выглядят так, как будто они связаны с терминальным устройством, поэтому стандартная библиотека ввода-вывода установит для них построчный режим буферизации.

Родительский процесс может вставить псевдотерминал между собой и сопроцессом двумя способами. (В этом случае родительским процессом может быть программа из листинга 15.9, которая использует для взаимодействия с сопроцессом два неименованных канала, или программа из листинга 17.1, которая использует один канал STREAMS.) Первый способ заключается в том, чтобы запустить дочерний процесс функцией `pty_fork` (раздел 19.4). Второй способ – запустить с помощью функции `exec` программу `pty` (раздел 19.5)

Рис. 19.5. Управление сопроцессом с помощью псевдотерминала

и передать ей имя программы сопроцесса в качестве аргумента. Мы продемонстрируем оба способа после того, как рассмотрим программу `pty`.

Отслеживание вывода программ, работающих продолжительное время

Если программа должна работать продолжительное время, мы можем просто запустить ее в фоновом режиме средствами любой стандартной командной оболочки. Но если стандартный вывод программы перенаправлен в файл, а объем генерируемых программой данных невелик, то отслеживать ход выполнения программы будет очень непросто, потому что стандартная библиотека ввода-вывода назначит режим полной буферизации для потока стандартного вывода, причем размер буфера может превышать 8192 байта.

Если доступен исходный код программы, мы можем вставить дополнительные вызовы функции `fflush`. В качестве альтернативы можно запустить программу под управлением программы `pty`, которая заставит стандарнтую библиотеку ввода-вывода думать, что стандартный вывод связан с терминалом. На рис. 19.6 приводится схема взаимодействия процессов, соответствующая такому случаю. Здесь `slowout` – это имя программы, которая выводит данные редко и в небольшом объеме. Стрелка, соответствующая запуску программы

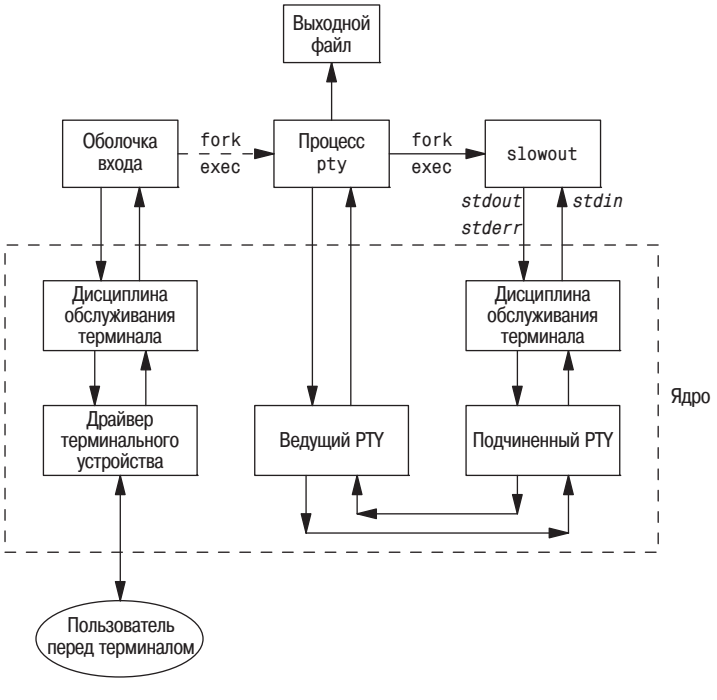


Рис. 19.6. *Запуск программы, которая выводит данные редко и в небольшом объеме с помощью псевдотерминала*

pty, нарисована пунктиром, чтобы подчеркнуть, что она запускается как фоновое задание.

19.3. Открытие устройств псевдотерминалов

На разных платформах открытие устройств псевдотерминалов осуществляется различными способами. Чтобы упорядочить их, стандарт Single UNIX Specification определяет ряд функций в качестве расширений XSI. Эти расширения основаны на функциях, которые изначально были предназначены для управления псевдотерминалами STREAMS в System V Release 4.

Функция posix_openpt предоставляет переносимый способ открытия устройства ведущего псевдотерминала.

```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int oflag);
```

Возвращает дескриптор следующего доступного ведущего PTY в случае успеха, −1 в случае ошибки

Аргумент *oflag* похож на аналогичный аргумент функции open(2) и представляет собой битовую маску, которая определяет режим открытия устройства. Однако эта функция поддерживает не все флаги режимов открытия. С функцией posix_openpt можно использовать флаг O_RDWR, чтобы открыть устройство для чтения и записи, и O_NOCTTY, чтобы открываемое устройство не стало управляющим терминалом для вызывающего процесса. Поведение остальных флагов не определено.

Прежде чем использовать подчиненный псевдотерминал, необходимо установить права доступа к нему таким образом, чтобы он стал доступен для приложений. Сделать это можно с помощью функции grantpt. Идентификатор пользователя подчиненного устройства она устанавливает равным реальному идентификатору пользователя вызывающего процесса, а идентификатор группы устройства – в неопределенное состояние; обычно это идентификатор некоторой группы, которая имеет право доступа к терминальным устройствам. Биты прав доступа устанавливаются так, чтобы разрешить доступ на чтение и на запись владельцу и право на запись группе владельца (0620).

```
#include <stdlib.h>

int grantpt(int filedes);

int unlockpt(int filedes);
```

Обе возвращают 0 в случае успеха, −1 в случае ошибки

Чтобы изменить права доступа к подчиненному устройству, функции grantpt может потребоваться запустить программу с установленным битом set-user-ID (например, /usr/lib/pt_chmod в Solaris). Таким образом, поведение функ-

ции grantpt окажется непредсказуемым, если процесс предусматривает перехват сигнала SIGCHLD.

Функция unlockpt применяется для того, чтобы разблокировать доступ к подчиненному псевдотерминалу, что позволит другим приложениям открывать устройство. Пока устройство заблокировано, приложению-владельцу предоставляется удобная возможность инициализировать и должным образом настроить ведущее и подчиненное устройства прежде, чем они будут использованы.

Обратите внимание, что обе функции принимают в качестве аргумента файловый дескриптор, связанный с ведущим псевдотерминалом.

Функция ptsname возвращает полное имя подчиненного устройства псевдотерминала по заданному дескриптору ведущего псевдотерминала. Это позволяет приложениям идентифицировать подчиненное устройство независимо от соглашений, принятых на той или иной платформе. Заметьте, что возвращаемая строка может храниться статически, так что она может быть затерта в результате следующего вызова ptsname.

```
#include <stdlib.h>

char *ptsname(int filedes);
```

Возвращает указатель на строку с именем подчиненного PTY в случа успеха, NULL – в случае ошибки

В табл. 19.1 перечислены функции, определяемые стандартом Single UNIX Specification для работы с псевдотерминалами, и указано, какие из них поддерживаются четырьмя платформами, обсуждаемыми в этой книге.

> В ОС FreeBSD функция unlockpt не выполняет никаких действий, а флаг O_NOCTTY определен только для совместимости с программами, которые вызывают функцию posix_openpt. Эта операционная система не назначает открываемое устройство управляющим терминалом, поэтому флаг O_NOCTTY просто игнорируется.

Таблица 19.1. Функции XSI для работы с псевдотерминалами

Функция	Описание	XSI	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
grantpt	Изменяет права доступа к подчиненному PTY	•	•	•		•
posix_openpt	Открывает ведущее устройство PTY	•	•			
ptsname	Возвращает имя подчиненного устройства PTY	•	•	•		•
unlockpt	Разрешает открытие подчиненного устройства PTY	•	•	•		•

Несмотря на то, что стандарт Single UNIX Specification пытается таким образом повысить переносимость приложений, реализации еще не полностью

поддерживают указанные функции, о чем свидетельствует табл. 19.1. Поэтому мы разработали две функции, которые предназначены для открытия псевдотерминалов: ptym_open открывает следующее доступное ведущее устройство PTY, а ptys_open – соответствующее подчиненное устройство.

```
#include "apue.h"

int ptym_open(char *pts_name, int pts_namesz);

                                    Возвращает дескриптор ведущего
                                устройства PTY в случае успеха, –1 в случае ошибки

int ptys_open(char *pts_name);

                                    Возвращает дескриптор ведомого
                                устройства PTY в случае успеха, –1 в случае ошибки
```

Обычно не требуется напрямую обращаться к этим функциям; это будет делать функция pty_fork (раздел 19.4), которая одновременно запускает дочерний процесс.

Функция ptym_open определяет следующий доступный ведущий PTY и открывает его. Вызывающий процесс должен разместить в памяти буфер, в котором будет храниться имя ведущего или подчиненного устройства; в случае успеха в этом буфере (аргумент *pts_name*) будет возвращено имя подчиненного PTY. Это имя затем передается функции ptys_open, которая открывает подчиненное устройство. Размер буфера в байтах передается в аргументе *pts_namesz*, чтобы функция ptym_open не создавала копию строки длиннее, чем размер буфера.

Причина, по которой мы предоставляем две разные функции для открытия двух типов устройств, станет понятна после того, как мы рассмотрим функцию pty_fork. Обычно процесс вызывает функцию ptym_open для того, чтобы открыть ведущее устройство и получить имя подчиненного. Затем процесс вызывает функцию fork и дочерний процесс с помощью ptys_open открывает подчиненное устройство, после чего открывает новую сессию обращением к функции setsid. Благодаря этому подчиненный псевдотерминал становится управляющим терминалом дочернего процесса.

19.3.1. Псевдотерминалы на основе STREAMS

Подробное описание реализации псевдотерминалов на основе механизма STREAMS в ОС Solaris вы найдете в приложении С к [Sun Microsystems 2002]. Следующее доступное ведущее устройство PTY может быть получено с помощью *устройства клонирования* STREAMS. Устройство клонирования – это специальное устройство, которое при открытии возвращает неиспользуемое устройство. (Как происходит открытие устройства клонирования STREAMS, подробно рассказывается в [Rago 1993].)

Устройство клонирования, создающее ведущий PTY, называется /dev/ptmx. Когда мы открываем его, процедура открытия автоматически определяет

первое неиспользуемое ведущее устройство PTY и открывает его. (В следующем разделе мы узнаем, что в BSD-системах нам сначала придется самостоятельно отыскать первое неиспользуемое ведущее устройство PTY.)

Листинг 19.1. Функции открытия псевдотерминала, реализованного на основе STREAMS

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <stropts.h>

int
ptym_open(char *pts_name, int pts_namesz)
{
    char *ptr;
    int fdm;

    /*
     * Возвращает имя ведущего устройства так, чтобы в случае ошибки
     * вызывающий процесс мог вывести сообщение об ошибке.
     * Завершить нулевым символом, чтобы обработать ситуацию, когда
     * strlen("/dev/ptmx") > pts_namesz.
     */
    strncpy(pts_name, "/dev/ptmx", pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    if ((fdm = open(pts_name, O_RDWR)) < 0)
        return(-1);
    if (grantpt(fdm) < 0) { /* разрешить доступ к подчиненному */
        close(fdm);
        return(-2);
    }
    if (unlockpt(fdm) < 0) { /* сбросить флаг блокировки подчиненного */
        close(fdm);
        return(-3);
    }
    if ((ptr = ptsname(fdm)) == NULL) { /* получить имя подчиненного */
        close(fdm);
        return(-4);
    }

    /*
     * Вернуть имя подчиненного устройства.
     * Завершить нулевым символом, чтобы обработать ситуацию, когда
     * strlen(ptr) > pts_namesz.
     */
    strncpy(pts_name, ptr, pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    return(fdm);                        /* вернуть дескриптор ведущего */
}

int
ptys_open(char *pts_name)
```

```
{
    int fds, setup;

    /*
     * Следующая операция открытия должна разместить управляющий терминал.
     */
    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-5);

    /*
     * Проверить - возможно, поток уже настроен должным образом
     * благодаря автоматической вставке модулей.
     */
    if ((setup = ioctl(fds, I_FIND, "ldterm")) < 0) {
        close(fds);
        return(-6);
    }
    if (setup == 0) {
        if (ioctl(fds, I_PUSH, "ptem") < 0) {
            close(fds);
            return(-7);
        }
        if (ioctl(fds, I_PUSH, "ldterm") < 0) {
            close(fds);
            return(-8);
        }
        if (ioctl(fds, I_PUSH, "ttcompat") < 0) {
            close(fds);
            return(-9);
        }
    }
    return(fds);
}
```

Прежде всего мы открываем устройство клонирования `/dev/ptmx`, чтобы получить дескриптор ведущего PTY. При открытии ведущего устройства автоматически блокируется соответствующее ему подчиненное устройство.

Затем вызывается функция `grantpt`, которая изменяет права доступа подчиненного устройства. В Solaris владелец подчиненного устройства определяется в соответствии с реальным идентификатором пользователя, в качестве группы владельца устанавливается группа `tty`, а права доступа назначаются таким образом, чтобы позволить чтение и запись для владельца и запись для группы. В качестве группы владельца выбрана группа `tty`, потому что у программ `wall`(1) и `write`(1) установлен бит set-group-ID, и они принадлежат группе `tty`. Функция `grantpt` вызывает программу `/usr/lib/pt_chmod`, у которой установлен бит set-user-ID, а владельцем файла программы является пользователь `root`, поэтому она может беспрепятственно изменять принадлежность и права доступа подчиненного устройства.

Чтобы сбросить внутреннюю блокировку подчиненного устройства, вызывается функция `unlockpt`. Это нужно делать перед открытием подчиненного

устройства. Кроме того, мы должны получить имя подчиненного устройства обращением к функции ptsname. Имя устройства возвращается в виде /dev/pts/NNN.

Следующая функция – ptys_open, которая открывает подчиненное устройство. ОС Solaris следует исторически сложившемуся в System V правилу: если вызывающий процесс является лидером сессии, который еще не имеет управляющего терминала, то вызов функции open назначает открываемый подчиненный PTY управляющим терминалом процесса. Если нужно избежать этого, следует передать функции open флаг O_NOCTTY.

После открытия может понадобиться добавить в поток STREAMS подчиненного устройства модули промежуточной обработки данных. Модуль эмуляции псевдотерминала (ptem) и модуль дисциплины обслуживания терминала (ldterm) совместно работают как реальный терминал. Модуль ttcompat обеспечивает совместимость с устаревшими версиями системного вызова ioctl в V7, 4BSD и Xenix. Этот модуль относится к разряду необязательных, но поскольку он автоматически добавляется при входе в систему с консоли или через сетевое соединение (смотрите вывод программы из листинга 14.9), мы также помещаем его в поток подчиненного устройства.

Эти три модуля могут размещаться в потоке автоматически. Система STREAMS поддерживает функциональную возможность, известную как *autopush* (автоматическое добавление). Она позволяет администратору создать список модулей, которые должны автоматически размещаться в потоке того или иного устройства при его открытии (более подробное описание вы найдете в [Rago 1993]). С помощью команды I_FIND функции ioctl мы проверяем наличие модуля ldterm в потоке. Если модуль найден, следовательно, поток уже был сконфигурирован механизмом автоматического добавления модулей и нам не нужно вставлять его повторно.

В результате вызова функций ptym_open и ptys_open в вызывающем процессе открываются два файловых дескриптора: один для ведущего устройства, другой – для подчиненного.

19.3.2. Псевдотерминалы в BSD

Для операционных систем BSD и Linux мы реализуем свои версии функций XSI, которые подключаются в зависимости от того, какие функции поддерживаются той или иной платформой.

В нашей версии posix_openpt мы должны отыскать первое доступное ведущее устройство PTY. Для этого мы пытаемся открывать все устройства подряд, начиная с /dev/ptyp0, пока не сможем открыть одно из них или пока не обнаружим, что все устройства уже заняты. При обращении к функции open можно получить две ошибки: EIO означает, что устройство уже занято, а ENOENT означает, что данное устройство не существует. В последнем случае мы завершаем поиск, так как все псевдотерминалы уже заняты. После открытия ведущего PTY – пусть это будет /dev/pty*MN* – именем соответствующего ему подчиненного устройства будет /dev/tty*MN*. В ОС Linux, если именем ведуще-

го устройства является строка /dev/pty/m*XX*, то именем соответствующего ему подчиненного устройства будет строка /dev/pty/s*XX*.

Листинг 19.2. Функции открытия псевдотерминала для BSD и Linux

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <grp.h>

#ifndef _HAS_OPENPT
int
posix_openpt(int oflag)
{
    int fdm;
    char *ptr1, *ptr2;
    char ptm_name[16];
    strcpy(ptm_name, "/dev/ptyXY");

    /* индексы в массиве: 0123456789 (потребуется в дальнейшем) */
    for (ptr1 = "pqrstuvwxyzPQRST"; *ptr1 != 0; ptr1++) {
        ptm_name[8] = *ptr1;
        for (ptr2 = "0123456789abcdef"; *ptr2 != 0; ptr2++) {
            ptm_name[9] = *ptr2;

            /*
             * Попытаться открыть ведущий PTY.
             */
            if ((fdm = open(ptm_name, oflag)) < 0) {
                if (errno == ENOENT) /* отличается от EIO */
                    return(-1);      /* доступных устройств pty нет */
                else
                    continue;        /* попробовать следующий pty */
            }
            return(fdm);             /* открыто, вернуть fd ведущего pty */
        }
    }
    errno = EAGAIN;
    return(-1);                      /* доступных устройств pty нет */
}
#endif

#ifndef _HAS_PTSNAME
char *
ptsname(int fdm)
{
    static char pts_name[16];
    char *ptm_name;

    ptm_name = ttyname(fdm);
    if (ptm_name == NULL)
        return(NULL);
    strncpy(pts_name, ptm_name, sizeof(pts_name));
    pts_name[sizeof(pts_name) - 1] = '\0';
```

```
    if (strncmp(pts_name, "/dev/pty/", 9) == 0)
        pts_name[9] = 's'; /* заменить /dev/pty/mXX на /dev/pty/sXX */
    else
        pts_name[5] = 't'; /* заменить "pty" на "tty" */
    return(pts_name);
}
#endif

#ifndef _HAS_GRANTPT
int
grantpt(int fdm)
{
    struct group *grptr;
    int gid;
    char *pts_name;

    pts_name = ptsname(fdm);
    if ((grptr = getgrnam("tty")) != NULL)
        gid = grptr->gr_gid;
    else
        gid = -1;          /* группа tty отсутствует в файле groups */

    /*
     * Следующие два вызова не будут работать без привилегий суперпользователя.
     */
    if (chown(pts_name, getuid(), gid) < 0)
        return(-1);
    return(chmod(pts_name, S_IRUSR | S_IWUSR | S_IWGRP));
}
#endif

#ifndef _HAS_UNLOCKPT
int
unlockpt(int fdm)
{
    return(0); /* ничего делать не надо */
}
#endif

int
ptym_open(char *pts_name, int pts_namesz)
{
    char *ptr;
    int fdm;

    /*
     * Возвращает имя ведущего устройства так, чтобы в случае ошибки
     * вызывающий процесс мог вывести сообщение об ошибке.
     * Завершить нулевым символом, чтобы обработать ситуацию, когда
     * длина строки с шаблоном больше, чем pts_namesz.
     */
    strncpy(pts_name, "/dev/ptyXX", pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    if ((fdm = posix_openpt(O_RDWR)) < 0)
```

```
        return(-1);
    if (grantpt(fdm) < 0) { /* выдать права на доступ к подчиненному устройству */
        close(fdm);
        return(-2);
    }
    if (unlockpt(fdm) < 0) { /* сбросить флаг блокировки */
        close(fdm);
        return(-3);
    }
    if ((ptr = ptsname(fdm)) == NULL) { /* получить имя подчиненного устройства */
        close(fdm);
        return(-4);
    }

    /*
     * Вернуть имя подчиненного устройства.
     * Завершить нулевым символом, чтобы обработать ситуацию, когда
     * strlen(ptr) > pts_namesz.
     */
    strncpy(pts_name, ptr, pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    return(fdm);          /* вернуть дескриптор ведущего устройства */
}

int
ptys_open(char *pts_name)
{
    int fds;

    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-5);
    return(fds);
}
```

В нашей версии `grantpt` мы вызываем `chown` и `chmod`, но необходимо понимать, что эти две функции не будут работать, если вызывающий процесс не обладает привилегиями суперпользователя. Если возможность менять владельца и права доступа важна, обращения к этим двум функциям должны быть размещены в исполняемом файле с установленным битом set-user-ID, принадлежащем пользователю `root`, как это сделано в ОС Solaris.

Функция `ptys_open` из листинга 19.2 просто открывает подчиненное устройство. Она не выполняет никаких других действий по инициализации. Функция `open` при открытии подчиненного PTY в BSD-системах не назначает открываемое устройство управляющим терминалом. В разделе 19.4 мы покажем, как назначается управляющий терминал в BSD-системах.

> Наша версия `posix_openpt` пробует открыть ведущее устройство, расположенное в одной из 16 групп по 16 устройств в каждой, от `/dev/ptyp0` до `/dev/ptyTf`. Фактическое количество доступных устройств в системе зависит от двух факторов: (а) от количества, заданного в ядре, и (б) от количества специальных файлов устройств, созданных в каталоге `/dev`. Общее количество устройств, доступных программам, всегда будет равно наименьшему из (а) и (б).

19.3.3. Псевдотерминалы в Linux

ОС Linux поддерживает метод доступа к псевдотерминалам, используемый в BSD, поэтому функции из листинга 19.2 будут работать и в Linux. Однако Linux, кроме того, поддерживает интерфейс доступа к псевдотерминалам через устройство клонирования /dev/ptmx (но в данном случае оно не является устройством STREAMS). Использование интерфейса клонирования требует от нас дополнительных действий по идентификации и разблокированию подчиненного устройства. Функции, реализующие доступ к псевдотерминалам с использованием этого устройства в Linux, приводятся в листинге 19.3.

Листинг 19.3. Функции открытия псевдотерминала для Linux

```
#include "apue.h"
#include <fcntl.h>

#ifndef _HAS_OPENPT
int
posix_openpt(int oflag)
{
    int fdm;

    fdm = open("/dev/ptmx", oflag);
    return(fdm);
}
#endif

#ifndef _HAS_PTSNAME
char *
ptsname(int fdm)
{
    int sminor;
    static char pts_name[16];

    if (ioctl(fdm, TIOCGPTN, &sminor) < 0)
        return(NULL);
    snprintf(pts_name, sizeof(pts_name), "/dev/pts/%d", sminor);
    return(pts_name);
}
#endif

#ifndef _HAS_GRANTPT
int
grantpt(int fdm)
{
    char *pts_name;

    pts_name = ptsname(fdm);
    return(chmod(pts_name, S_IRUSR | S_IWUSR | S_IWGRP));
}
#endif

#ifndef _HAS_UNLOCKPT
int
unlockpt(int fdm)
```

```
{
    int lock = 0;
    return(ioctl(fdm, TIOCSPTLCK, &lock));
}
#endif

int
ptym_open(char *pts_name, int pts_namesz)
{
    char *ptr;
    int fdm;

    /*
     * Возвращает имя ведущего устройства так, чтобы в случае ошибки
     * вызывающий процесс мог вывести сообщение об ошибке.
     * Завершить нулевым символом, чтобы обработать ситуацию, когда
     * длина строки с шаблоном больше, чем pts_namesz.
     */
    strncpy(pts_name, "/dev/ptmx", pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    fdm = posix_openpt(O_RDWR);
    if (fdm < 0)
        return(-1);
    if (grantpt(fdm) < 0) { /* выдать права на доступ к подчиненному устройству */
        close(fdm);
        return(-2);
    }
    if (unlockpt(fdm) < 0) { /* сбросить флаг блокировки подчиненного PTY */
        close(fdm);
        return(-3);
    }
    if ((ptr = ptsname(fdm)) == NULL) { /* получить имя подчиненного PTY */
        close(fdm);
        return(-4);
    }

    /*
     * Вернуть имя подчиненного устройства.
     * Завершить нулевым символом, чтобы обработать ситуацию, когда
     * strlen(ptr) > pts_namesz.
     */
    strncpy(pts_name, ptr, pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    return(fdm); /* вернуть дескриптор ведущего PTY */
}

int
ptys_open(char *pts_name)
{
    int fds;

    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-5);
```

```
    return(fds);
}
```

В Linux подчиненное устройство уже принадлежит группе `tty`, поэтому все, что нам нужно сделать в функции `grantpt`, – это обеспечить корректную установку прав доступа.

19.4. Функция pty_fork

Теперь, используя функции открытия псевдотерминала из предыдущего раздела, `ptym_open` и `ptys_open`, напишем новую функцию `pty_fork`. Эта новая функция будет совмещать открытие ведущего и подчиненного устройств, запуск дочернего процесса и назначение его лидером сессии со своим управляющим терминалом.

```
#include "apue.h"
#include <termios.h>
#include <sys/ioctl.h> /* find struct winsize on BSD systems */

pid_t pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
               const struct termios *slave_termios,
               const struct winsize *slave_winsize);
```

Возвращает 0 в дочернем процессе, идентификатор дочернего процесса в родительском процессе, –1 в случае ошибки

Дескриптор ведущего PTY возвращается в переменной по адресу *ptrfdm*.

Если в аргументе *slave_name* передается непустой указатель, то по заданному адресу сохраняется имя подчиненного устройства. Разумеется, вызывающий процесс должен выделить память для хранения строки, на которую указывает этот аргумент.

Если в аргументе *slave_termios* передается непустой указатель, то система использует структуру, на которую ссылается этот аргумент, для инициализации подчиненного терминала. Если в этом аргументе передается значение `NULL`, то система инициализирует структуру `termios` подчиненного устройства значениями по умолчанию, зависящими от реализации. Аналогичным образом инициализируется структура с размером окна подчиненного устройства, если в аргументе *slave_winsize* передается непустой указатель. Если в этом аргументе передается значение `NULL`, то, как правило, структура `winsize` инициализируется нулями.

В листинге 19.4 приводится исходный код этой функции. Она будет работать на всех четырех платформах, рассматриваемых в этой книге, обращаясь к соответствующим функциям `ptym_open` и `ptys_open`.

После открытия ведущего PTY вызывается функция `fork`. Как уже говорилось ранее, функция `ptys_open` должна вызываться только после того, как дочерний процесс откроет новую сессию вызовом функции `setsid`. К моменту вызова функции `setsid` дочерний процесс не является лидером группы, поэто-

му выполняются три действия, описанные в разделе 9.5: (а) открывается новая сессия, в которой дочерний процесс выступает в роли лидера, (б) создается новая группа процессов для дочернего процесса и (в) дочерний процесс теряет любую установленную связь с предыдущим управляющим терминалом. В ОС Linux и Solaris при вызове ptys_open подчиненное устройство становится управляющим терминалом этой новой сессии. В ОС FreeBSD и Mac OS X мы должны назначить управляющий терминал с помощью команды TIOCSCTTY функции ioctl. (Linux также поддерживает команду TIOCSCTTY функции ioctl.) После этого в дочернем процессе инициализируются структуры termios и winsize. В заключение дочерний процесс дублирует дескриптор подчиненного PTY на стандартный ввод, стандартный вывод и стандартный вывод сообщений об ошибках. Это означает, что в любом процессе, запускаемом из дочернего процесса с помощью функции exec, эти три дескриптора останутся связаны с подчиненным PTY (с его управляющим терминалом).

После вызова функции fork родительскому процессу возвращается дескриптор ведущего PTY и идентификатор дочернего процесса. В следующем разделе мы будем использовать функцию pty_fork при разработке программы pty.

Листинг 19.4. Функция pty_fork

```
#include "apue.h"
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

pid_t
pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
         const struct termios *slave_termios,
         const struct winsize *slave_winsize)
{
    int fdm, fds;
    pid_t pid;
    char pts_name[20];

    if ((fdm = ptym_open(pts_name, sizeof(pts_name))) < 0)
        err_sys("невозможно открыть ведущий pty: %s, ошибка %d",
                pts_name, fdm);
    if (slave_name != NULL) {

        /*
         * Вернуть имя подчиненного устройства.
         * Завершить нулевым символом, чтобы обработать ситуацию, когда
         * strlen(ptr) > pts_namesz.
         */
        strncpy(slave_name, pts_name, slave_namesz);
        slave_name[slave_namesz - 1] = '\0';
    }
    if ((pid = fork()) < 0) {
        return(-1);
    } else if (pid == 0) { /* дочерний процесс */
        if (setsid() < 0)
```

```
        err_sys("ошибка вызова функции setsid");

        /*
         * System V автоматически назначает управляющий терминал при открытии.
         */
        if ((fds = ptys_open(pts_name)) < 0)
            err_sys("невозможно открыть подчиненный pty");
        close(fdm); /* работа с ведущим pty в дочернем процессе завершена */

#if defined(TIOCSCTTY)
        /*
         * Команда TIOCSCTTY - способ назначения управляющего терминала в BSD.
         */
        if (ioctl(fds, TIOCSCTTY, (char *)0) < 0)
            err_sys("ошибка выполнения команды TIOCSCTTY");
#endif
        /*
         * Инициализировать структуры termios и winsize подчиненного pty.
         */
        if (slave_termios != NULL) {
            if (tcsetattr(fds, TCSANOW, slave_termios) < 0)
                err_sys("ошибка вызова функции tcsetattr для подчиненного pty");
        }
        if (slave_winsize != NULL) {
            if (ioctl(fds, TIOCSWINSZ, slave_winsize) < 0)
                err_sys("ошибка выполнения TIOCSWINSZ для подчиненного pty");
        }

        /*
         * Связать stdin/stdout/stderr с терминалом в дочернем процессе.
         */
        if (dup2(fds, STDIN_FILENO) != STDIN_FILENO)
            err_sys("ошибка вызова функции dup2 для stdin");
        if (dup2(fds, STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("ошибка вызова функции dup2 для stdout");
        if (dup2(fds, STDERR_FILENO) != STDERR_FILENO)
            err_sys("ошибка вызова функции dup2 для stderr");
        if (fds != STDIN_FILENO && fds != STDOUT_FILENO &&
          fds != STDERR_FILENO)
            close(fds);
        return(0);   /* вернуть 0 дочернему процессу, как это делает fork() */
    } else {         /* родительский процесс */
        *ptrfdm = fdm; /* вернуть fd ведущего pty */
        return(pid);   /* вернуть pid дочернего процесса родителю */
    }
}
```

19.5. Программа pty

Смысл программы `pty` в том, что она предоставляет возможность давать команды в виде

```
pty prog arg1 arg2
```

вместо

```
prog arg1 arg2
```

Когда для запуска программы используется pty, эта программа работает в рамках своей собственной сессии, связанной с псевдотерминалом.

Рассмотрим исходный код программы pty. Первый файл содержит функцию main (листинг 19.5). Она обращается к функции pty_fork, которая была описана в предыдущем разделе.

Листинг 19.5. Функция main программы pty

```
#include "apue.h"
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>       /* для struct winsize */
#endif

#ifdef LINUX
#define OPTSTR "+d:einv"
#else
#define OPTSTR "d:einv"
#endif

static void set_noecho(int); /* реализация находится в конце этого файла */
void do_driver(char *);      /* в файле driver.c */
void loop(int, int);         /* в файле loop.c */

int
main(int argc, char *argv[])
{
    int fdm, c, ignoreeof, interactive, noecho, verbose;
    pid_t pid;
    char *driver;
    char slave_name[20];
    struct termios orig_termios;
    struct winsize size;

    interactive = isatty(STDIN_FILENO);
    ignoreeof = 0;
    noecho = 0;
    verbose = 0;
    driver = NULL;

    opterr = 0;   /* нежелательно, чтобы getopt() выводила на stderr */
    while ((c = getopt(argc, argv, OPTSTR)) != EOF) {
        switch (c) {
        case 'd': /* драйвер для stdin/stdout */
            driver = optarg;
            break;
        case 'e': /* отключить эхо-вывод для подчиненного pty */
            noecho = 1;
            break;
        case 'i': /* игнорировать символ EOF для стандартного ввода */
```

```
            ignoreeof = 1;
            break;
        case 'n': /* неинтерактивный режим */
            interactive = 0;
            break;
        case 'v': /* вывод подробных сообщений */
            verbose = 1;
            break;
        case '?':
            err_quit("недопустимая опция: -%c", optopt);
        }
    }
    if (optind >= argc)
        err_quit("Использование: "
                 "pty [ -d driver -einv ] program [ arg ... ]");
    if (interactive) { /* получить текущие termios и winsize */
        if (tcgetattr(STDIN_FILENO, &orig_termios) < 0)
            err_sys("ошибка вызова функции tcgetattr для stdin");
        if (ioctl(STDIN_FILENO, TIOCGWINSZ, (char *) &size) < 0)
            err_sys("ошибка выполнения команды TIOCGWINSZ");
        pid = pty_fork(&fdm, slave_name, sizeof(slave_name),
                       &orig_termios, &size);
    } else {
        pid = pty_fork(&fdm, slave_name, sizeof(slave_name),
                       NULL, NULL);
    }
    if (pid < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) { /* дочерний процесс */
        if (noecho)
            set_noecho(STDIN_FILENO); /* stdin - подчиненный pty */
        if (execvp(argv[optind], &argv[optind]) < 0)
            err_sys("невозможно запустить: %s", argv[optind]);
    }
    if (verbose) {
        fprintf(stderr, "имя подчиненного = %s\n", slave_name);
        if (driver != NULL)
            fprintf(stderr, "драйвер = %s\n", driver);
    }
    if (interactive && driver == NULL) {
        if (tty_raw(STDIN_FILENO) < 0) /* перевести tty в прозрачный режим */
            err_sys("ошибка вызова функции tty_raw");
        if (atexit(tty_atexit) < 0) /* восстановление настроек tty при выходе */
            err_sys("ошибка вызова функции atexit");
    }
    if (driver)
        do_driver(driver);    /* изменить наши stdin/stdout */
        loop(fdm, ignoreeof); /* копировать stdin -> ptym, ptym -> stdout */
        exit(0);
    }
```

```
static void
set_noecho(int fd) /* отключить эхо-вывод (для подчиненного pty) */
{
    struct termios stermios;

    if (tcgetattr(fd, &stermios) < 0)
        err_sys("ошибка вызова функции tcgetattr");
    stermios.c_lflag &=  (ECHO | ECHOE | ECHOK | ECHONL);

    /*
     * Кроме того, отключить преобразование NL в CR/NL при выводе.
     */
    stermios.c_oflag &=  (ONLCR);
    if (tcsetattr(fd, TCSANOW, &stermios) < 0)
        err_sys("ошибка вызова функции tcsetattr");
}
```

Мы рассмотрим различные опции командной строки в следующем разделе, когда будем экспериментировать с программой `pty`. Функция `getopt` помогает разобрать аргументы командной строки. Более подробно эта функция рассматривается в главе 21.

Перед обращением к функции `pty_fork` мы получаем текущие значения структур `termios` и `winsize` и передаем их функции `pty_fork` в виде аргументов. Таким образом, подчиненный PTY будет иметь точно такие же настройки, что и текущий терминал.

После возврата из `pty_fork` дочерний процесс отключает эхо-вывод для подчиненного PTY (если задана соответствующая опция) и затем с помощью `execvp` запускает программу, указанную в командной строке. Все остальные аргументы командной строки передаются этой программе.

Родительский процесс переводит пользовательский терминал в прозрачный (raw) режим (если выбрана соответствующая опция) и при необходимости устанавливает обработчик выхода, который восстановит настройки терминала, когда будет вызвана функция `exit`. Функция `do_driver` будет описана в следующем разделе.

После этого родительский процесс вызывает функцию `loop` (листинг 19.6), которая копирует все, что будет принято со стандартного ввода и стандартного вывода ведущего PTY. Для разнообразия мы предусмотрели выполнение этих операций двумя процессами, хотя в этой ситуации вполне можно было бы решить эту задачу с помощью функций мультиплексирования ввода-вывода `select` и `poll` или с помощью нескольких потоков.

Листинг 19.6. Функция loop

```
#include "apue.h"

#define BUFFSIZE 512

static void sig_term(int);
static volatile sig_atomic_t sigcaught; /* изменяется обработчиком сигнала */

void
```

```
loop(int ptym, int ignoreeof)
{
    pid_t child;
    int nread;
    char buf[BUFFSIZE];

    if ((child = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (child == 0) { /* дочерний процесс копирует stdin в ptym */
        for ( ; ; ) {
            if ((nread = read(STDIN_FILENO, buf, BUFFSIZE)) < 0)
                err_sys("ошибка чтения из stdin");
            else if (nread == 0)
                break; /* EOF в stdin означает конец ввода */
            if (writen(ptym, buf, nread) != nread)
                err_sys("ошибка записи в ведущий pty");
        }

        /*
         * Мы всегда завершаем работу, когда обнаруживаем EOF в stdin,
         * но извещаем родителя только тогда, когда ignoreeof == 0.
         */
        if (ignoreeof == 0)
            kill(getppid(), SIGTERM); /* известить родительский процесс */
        exit(0);                      /* и завершить работу; дочерний процесс */
                                      /* не может вернуть управление */
    }

    /*
     * Родительский процесс копирует ptym в stdout.
     */
    if (signal_intr(SIGTERM, sig_term) == SIG_ERR)
        err_sys("ошибка вызова функции signal_intr для SIGTERM");

    for ( ; ; ) {
        if ((nread = read(ptym, buf, BUFFSIZE)) <= 0)
            break; /* перехвачен сигнал, ошибка или получен EOF */
        if (writen(STDOUT_FILENO, buf, nread) != nread)
            err_sys("ошибка записи в stdout");
    }

    /*
     * В этой точке мы оказываемся в трех случаях: функция sig_term()
     * (ниже) перехватила сигнал SIGTERM от дочернего процесса,
     * был прочитан символ EOF из ведущего pty (это означает,
     * что мы должны известить об этом потомка) или в случае ошибки.
     */
    if (sigcaught == 0)       /* послать сигнал потомку, */
        kill(child, SIGTERM); /* если от него не был получен сигнал */

    /*
     * Родительский процесс возвращает управление вызывающему.
     */
}
```

```
/*
 * Потомок посылает сигнал SIGTERM, когда получает EOF из подчиненного pty или
 * когда функция read() терпит неудачу. Вероятно, было прервано чтение из ptym.
 */
static void
sig_term(int signo)
{
    sigcaught = 1; /* просто установить флаг и вернуться */
}
```

Обратите внимание: в случае с двумя процессами, когда один завершает работу, он сообщает об этом другому с помощью сигнала SIGTERM.

19.6. Использование программы pty

Теперь рассмотрим некоторые примеры использования программы pty и попутно объясним назначение различных опций.

Если в качестве командной оболочки используется Korn shell, то можно запустить команду

```
pty ksh
```

и получить совершенно новый экземпляр командной оболочки, работающей под управлением псевдотерминала.

Если предположить, что программа из листинга 18.7 называется ttyname, то мы можем запустить ее следующим образом:

```
$ who
sar :0 Oct 5 18:07
sar pts/0 Oct 5 18:07
sar pts/1 Oct 5 18:07
sar pts/2 Oct 5 18:07
sar pts/3 Oct 5 18:07
sar pts/4 Oct 5 18:07 pts/4      наибольший номер PTY, используемый
                                 в настоящее время
$ pty ttyname                    запуск программы из листинга 18.7 из PTY
fd 0: /dev/pts/5                 pts/5 – следующий доступный PTY
fd 1: /dev/pts/5
fd 2: /dev/pts/5
```

Файл utmp

В разделе 6.8 мы рассматривали файл utmp, в котором хранятся сведения обо всех работающих в системе пользователях. Запуск пользовательской программы на псевдотерминале рассматривается как вход в систему. В случае удаленного входа в систему через telnetd или rlogin в файл utmp должна помещаться соответствующая запись о входе с псевдотерминала. Однако в случае запуска командной оболочки на псевдотерминале из оконной системы или из таких программ, как script, это соглашение соблюдается не всегда. Некоторые системы в этом случае добавляют записи в файл utmp, а некото-

рые – нет. Если система в таких случаях не производит запись в файл utmp, то команда who(1), как правило, не сообщает о том, что соответствующие псевдотерминалы используются.

Если для файла utmp не установлен бит права на запись для остальных (что, строго говоря, рассматривается как уязвимость в системе безопасности), то не все программы, использующие псевдотерминалы, смогут добавлять записи в этот файл.

Взаимодействие с механизмом управления заданиями

При запуске под управлением программы pty командной оболочки, поддерживающей управление заданиями, мы не заметим ничего необычного. Например, команда

```
pty ksh
```

запустит Korn shell под управлением pty. Мы можем запускать программы в этой новой командной оболочке и использовать механизм управления заданиями точно так же, как это делается в командной оболочке входа. Но когда под управлением pty запускается не командная оболочка, которая поддерживает управление заданиями, а любая другая интерактивная программа, как например

```
pty cat
```

то все будет работать прекрасно, пока мы не введем символ приостановки задания. В этом случае управляющий символ выводится как ^Z и игнорируется. В ранних версиях BSD это приводило к завершению процессов cat и pty и возвращению в первоначальную командную оболочку. Чтобы разобраться в этой ситуации, нам необходимо исследовать все задействованные процессы, их группы процессов и сессии. На рис. 19.7 приводится схема состояния процессов, которая соответствует запуску команды pty cat.

Когда мы вводим символ приостановки задания (Control-Z), он распознается модулем дисциплины обслуживания терминала (на рис. 19.7 расположен ниже процесса cat), поскольку pty переводит терминал (на рисунке – ниже родительского процесса pty) в прозрачный режим. Но ядро не будет приостанавливать процесс cat, поскольку он принадлежит осиротевшей группе процессов (раздел 9.10). Родительским процессом для cat является pty, а он принадлежит другой сессии.

Исторически разные реализации по-разному обрабатывали эту ситуацию. Стандарт POSIX.1 утверждает лишь, что сигнал SIGTSTP не может доставляться процессу. Системы, производные от 4.3BSD, вместо него доставляли сигнал SIGKILL, который не может быть перехвачен процессом. В 4.4BSD такое поведение системы было изменено в соответствии со стандартом POSIX.1. Вместо посылки сигнала SIGKILL ядро 4.4BSD просто уничтожает сигнал SIGTSTP, если для него выбрана диспозиция по умолчанию и получатель находится в осиротевшей группе процессов. Большинство современных реализаций придерживаются именно такого поведения.

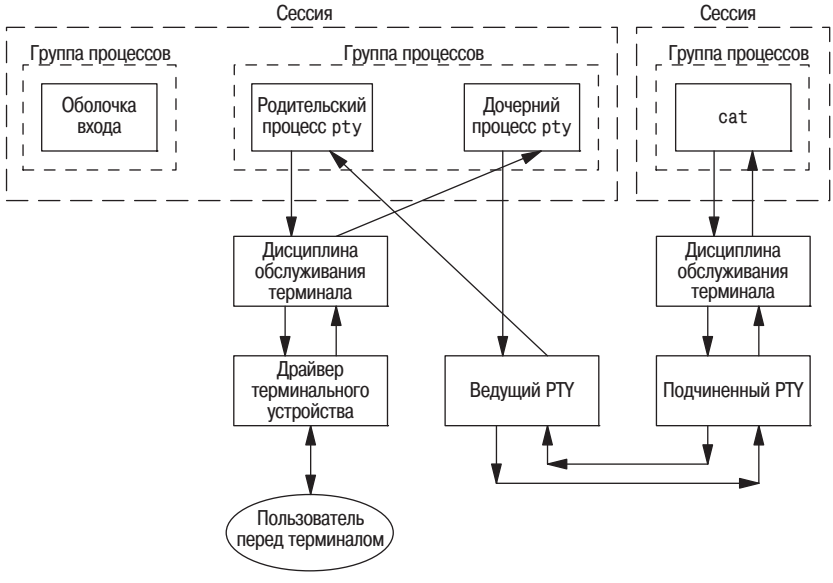


***Рис. 19.7.** Группы процессов и сессии, создаваемые командой pty cat*

Когда мы запускаем под управлением `pty` командную оболочку, поддерживающую управление заданиями, то задания, запускаемые из этой новой оболочки, уже не принадлежат осиротевшей группе процессов, поскольку командная оболочка сама принадлежит той же самой сессии. В этом случае символ Control-Z передается процессу, запущенному из оболочки, а не самой оболочке.

Единственный способ ликвидировать неспособность процесса, вызываемого программой `pty`, обрабатывать символы управления заданиями – добавить в программу `pty` еще одну опцию командной строки, которая будет заставлять ее самостоятельно распознавать символ приостановки задания (в дочернем процессе `pty`), вместо того чтобы передавать его на обработку другим модулям дисциплины обслуживания.

Отслеживание вывода программ, работающих продолжительное время

Другой пример взаимодействия с механизмом управления заданиями приводится на рис. 19.6. Если мы запустим программу, которая выводит данные достаточно редко и небольшими порциями, например

```
pty slowout > file.out &
```

то выполнение процесса `pty` будет приостановлено, как только дочерний процесс попытается прочитать данные со своего стандартного ввода (с терминала). Дело в том, что задание выполняется в фоновом режиме и будет приостановлено механизмом управления заданиями, как только попытается полу-

чить доступ к терминалу. Если перенаправить стандартный ввод так, чтобы pty не пыталась читать из терминала, например:

```
pty slowout < /dev/null > file.out &
```

то программа pty сразу же приостановится, потому что прочитает со стандартного ввода признак конца файла, и завершится. Решение этой проблемы заключается в передаче программе ключа -i, который сообщает о том, что программа должна игнорировать признак конца файла, полученный со стандартного ввода:

```
pty -i slowout < /dev/null > file.out &
```

Передача этого ключа приводит к тому, что дочерний процесс pty из листинга 19.6 завершается при получении признака конца файла, но не сообщает об этом родительскому процессу. Благодаря этому родительский процесс продолжает копировать вывод из подчиненного PTY на стандартный вывод (в данном примере – файл file.out).

Программа script

С помощью программы pty можно реализовать программу script(1) в виде простого сценария на языке командной оболочки:

```
#!/bin/sh
pty "${SHELL:-/bin/sh}" | tee typescript
```

После запуска этого сценария мы можем проследить взаимоотношения между процессами с помощью команды ps. Схема процессов, описывающая эти взаимоотношения, приводится на рис. 19.8.

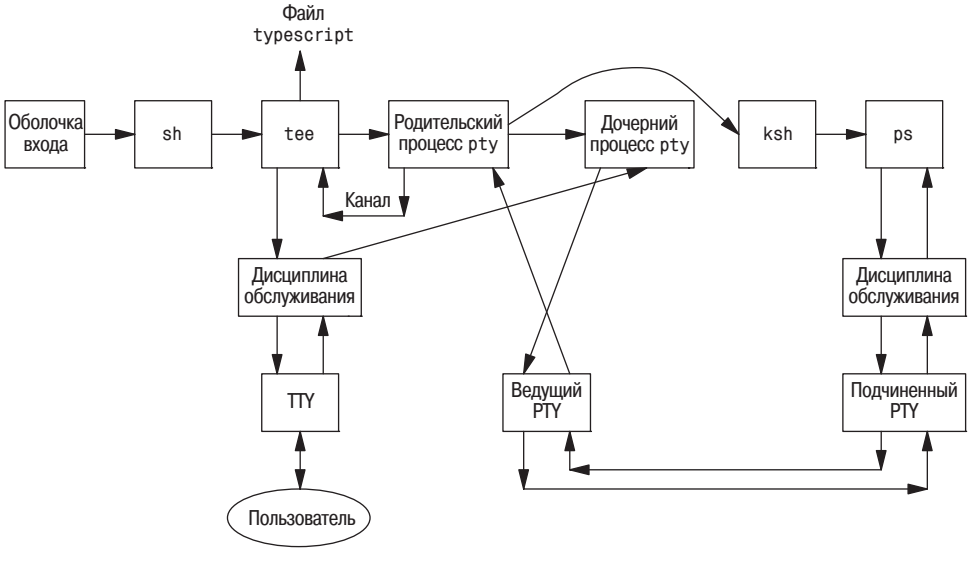


***Рис. 19.8.** Схема процессов, запущенных сценарием script*

В этом примере предполагается, что содержимое переменной SHELL соответствует командной оболочке Korn shell (скорее всего, /bin/ksh). Как уже говорилось, программа script копирует только то, что выводится новой оболочкой (и любыми процессами, которые из нее были запущены). Но благодаря тому, что модуль дисциплины обслуживания, расположенный на рисунке выше подчиненного PTY, обычно разрешает эхо-вывод, то большая часть ввода с клавиатуры также попадает в файл typescript.

Запуск сопроцессов

Сопроцессы из листинга 15.9 не могли использовать функции стандартной библиотеки ввода-вывода, потому что для стандартных потоков ввода и вывода, не связанных с терминалом, выбирается режим полной буферизации. Если запустить сопроцесс под управлением программы pty, заменив строку

```
if (execl("./add2", "add2", (char *)0) < 0)
```

на

```
if (execl("./pty", "pty", "-e", "add2", (char *)0) < 0)
```

то программа будет работать, даже если сопроцесс использует функции стандартной библиотеки ввода-вывода.

На рис. 19.9 приводится схема состояния процессов при запуске сопроцесса с вводом-выводом через псевдотерминал. Он представляет собой расширенную версию рис. 19.5 и показывает все связанные процессы и потоки движения данных. Под блоком «управляющая программа» подразумевается программа из листинга 15.9, для которой мы изменили вызов функции execl.

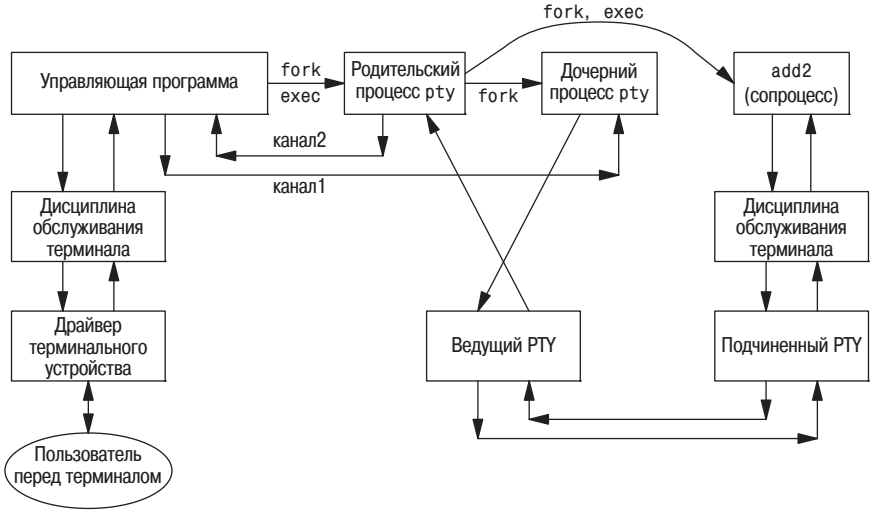


***Рис. 19.9.** Запуск сопроцесса с вводом-выводом через псевдотерминал*

Из этого примера видно, что мы должны запускать программу pty с ключом -e (запрет эхо-вывода). Программа pty запускается в неинтерактивном режиме, потому что ее стандартный ввод не связан с терминалом. В листинге 19.5 флаг interactive будет по умолчанию сброшен, потому что функция isatty вернет значение «ложь». Это означает, что дисциплина обслуживания терминала, расположенная на рисунке выше реального терминала, останется в каноническом режиме с разрешенным эхо-выводом. Ключом -e мы отключаем эхо-вывод в модуле дисциплины обслуживания, расположенном на рисунке выше подчиненного PTY. Если этого не сделать, весь ввод с клавиатуры будет выведен дважды – обоими модулями дисциплины обслуживания.

Кроме того, ключ -e сбрасывает флаг ONLCR в структуре termios, что предотвращает преобразование символов перевода строки, выводимых сопроцессом, в последовательности символов CR-NL.

Тестирование этого примера на разных платформах выявило еще одну проблему, о которой мы упоминали в разделе 14.8, когда описывали функции readn и writen. Объем данных, возвращаемых функцией read при работе с дескриптором, соответствующим файлу, отличному от обычного дискового файла, может различаться в разных реализациях. В программе из листинга 15.9 запуск сопроцесса с использованием pty дал неожиданные результаты при использовании функции read с неименованными каналами, когда read возвращала неполную строку. Решение проблемы заключается в том, чтобы вместо программы из листинга 15.9 использовать программу из упражнения 15.5, которая использует функции стандартной библиотеки ввода-вывода для работы с каналами, для которых устанавливается режим построчной буферизации. Благодаря этому функция fgets будет вызывать функцию read столько раз, сколько потребуется для получения полной строки. Цикл while в листинге 15.9 построен в предположении, что каждой строке, переданной сопроцессу, будет соответствовать одна строка, полученная от него.

Неинтерактивное управление интерактивными программами

Хотя все вышеизложенное заставляет думать, что посредством программы pty можно запускать любые сопроцессы, в действительности pty не может взаимодействовать с интерактивными сопроцессами. Проблема заключается в том, что pty просто копирует данные, полученные со своего стандартного ввода, в PTY и выводит на свой стандартный вывод все, что поступает от PTY, не анализируя при этом, что было получено и что следует отправить.

Например, мы можем запустить команду telnet под управлением pty, передав ей адрес удаленного хоста:

```
pty telnet 192.168.1.3
```

Такая последовательность команд не дает никаких преимуществ перед простой командой telnet 192.168.1.3, но нам может потребоваться запускать команду telnet из сценария, чтобы с ее помощью проверить состояние удален-

ного хоста. Представим себе, что у нас есть файл `telnet.cmd`, который содержит четыре строки:

```
sar
   passwd
uptime
exit
```

где первая строка – имя пользователя, вторая строка – пароль, третья – команда, запускаемая на удаленной машине, и четвертая – команда закрытия сессии. Но если запустить сценарий как

```
pty -i < telnet.cmd telnet 192.168.1.3
```

то мы не получим ожидаемого результата. Дело в том, что содержимое файла `telnet.cmd` будет отправлено удаленной стороне еще до того, как будет выполнен запрос на ввод имени пользователя и пароля. Для отключения эховывода при вводе пароля программа `login` вызывает функцию `tcsetattr`, которая сбрасывает все данные, находившиеся в очереди. Таким образом, отправленные нами данные будут просто потеряны.

Когда команда `telnet` запускается в интерактивном режиме, мы ждем приглашения от удаленной стороны, прежде чем вводить пароль, но программа `pty` ничего не знает об этом. По этой причине для управления интерактивными программами из сценария необходимо использовать программу более сложную, чем `pty`, например `expect`.

Даже запуск `pty` из программы, приведенной в листинге 15.9, нам не поможет, потому что эта программа построена на предположении, что каждой строке, записываемой в один канал, соответствует только одна строка, получаемая из другого канала. При работе с интерактивными программами нередки случаи, когда ввод одной строки приводит к выводу нескольких строк. Кроме того, программа из листинга 15.9 всегда отправляет строку сопроцессу, прежде чем прочитать ответ от него. Это не подходит для случая, когда мы должны получить какие-либо данные от сопроцесса, прежде чем отправить ему что-нибудь.

Существует несколько способов организовать взаимодействие с интерактивной программой из сценария. Можно было бы придумать для `pty` язык команд и его интерпретатор, но такой «довесок» наверняка будет в десятки раз превышать размер самой программы `pty`. Другой вариант – взять существующий командный интерпретатор, который мог бы управлять интерактивной программой, запуская ее с помощью функции `pty_fork`. Именно таким образом работает программа `expect`.

Мы выберем иной путь и просто добавим возможность соединить ввод и вывод программы `pty` с управляющим процессом с помощью ключа `-d`. Стандартный вывод управляющего процесса (драйвера) соединяется со стандартным вводом программы `pty` и наоборот. Это очень похоже на работу с сопроцессом. Результат напоминает рис. 19.9, но в данной ситуации процесс-драйвер запускается программой `pty`. Кроме того, вместо двух полудуплекс-

ных каналов мы используем для взаимодействия pty и драйвера один двунаправленный канал.

В листинге 19.7 приводится исходный код функции do_driver, которая вызывается из функции main (листинг 19.5), если указан ключ -d.

Листинг 19.7. Функция do_driver для программы pty

```
#include "apue.h"

void
do_driver(char *driver)
{
    pid_t child;
    int pipe[2];

    /*
     * Создать канал для взаимодействия с драйвером.
     */
    if (s_pipe(pipe) < 0)
        err_sys("невозможно создать канал");
    if ((child = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (child == 0) { /* дочерний процесс */
        close(pipe[1]);

        /* stdin драйвера */
        if (dup2(pipe[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("ошибка вызова функции dup2 для stdin");

        /* stdout драйвера */
        if (dup2(pipe[0], STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("ошибка вызова функции dup2 для stdout");
        if (pipe[0] != STDIN_FILENO && pipe[0] != STDOUT_FILENO)
            close(pipe[0]);

        /*оставить stderr драйвера неизменным */
        execlp(driver, driver, (char *)0);
        err_sys("ошибка вызова функции execlp для: %s", driver);
    }
    close(pipe[0]); /* родительский процесс */
    if (dup2(pipe[1], STDIN_FILENO) != STDIN_FILENO)
        err_sys("ошибка вызова функции dup2 для stdin");
    if (dup2(pipe[1], STDOUT_FILENO) != STDOUT_FILENO)
        err_sys("ошибка вызова функции dup2 для stdout");
    if (pipe[1] != STDIN_FILENO && pipe[1] != STDOUT_FILENO)
        close(pipe[1]);

    /*
     * Родительский процесс возвращает управление, но stdin и stdout
     * остаются связанными с драйвером.
     */
}
```

Написав свой драйвер, который будет вызываться программой pty, мы сможем управлять интерактивными программами по своему желанию. Даже несмотря на то, что стандартный ввод и стандартный вывод драйвера связаны с программой pty, он по-прежнему может взаимодействовать с пользователем, обращаясь к устройству /dev/tty. Наше решение не такое универсальное, как программа expect, но зато мы дополнили программу pty очень полезной возможностью, добавив всего 50 строк кода.

19.7. Дополнительные возможности

Псевдотерминалы обладают рядом дополнительных возможностей, о которых мы коротко расскажем в этом разделе. Эти возможности описываются в [Sun Microsystems 2002] и на страницах справочного руководства BSD к программе pty(4).

Пакетный режим

Пакетный режим позволяет ведущему PTY узнавать об изменении состояния подчиненного PTY. В Solaris этот режим устанавливается размещением модуля pckt в потоке STREAMS со стороны ведущего PTY. Мы показали этот модуль на рис. 19.2. В FreeBSD, Linux и Mac OS X этот режим устанавливается командой TIOCPKT функции ioctl.

Внутренняя реализация пакетного режима в Solaris отличается от реализации на других платформах. В ОС Solaris, процесс, получающий данные из ведущего PTY, должен вызывать функцию getmsg, чтобы получать сообщения из головы потока, так как модуль pckt преобразует отдельные события в сообщения STREAMS, не содержащие данных. На других платформах операция чтения из ведущего PTY возвращает байт статуса, сопровождаемый необязательными данными.

Вне зависимости от внутренней реализации, назначение пакетного режима заключается в том, чтобы информировать процесс, читающий данные из ведущего PTY, о наступлении определенных событий, которые происходят в модуле дисциплины обслуживания терминала, расположенного выше подчиненного PTY: сбрасывается очередь чтения, сбрасывается очередь записи, приостанавливается вывод (например, по Control-S), возобновляется вывод, разрешается управление потоком данных XON/XOFF после того, как оно было запрещено, запрещается управление потоком данных XON/XOFF после того, как оно было разрешено. Эти события используются, например, клиентом rlogin и сервером rlogind.

Дистанционный режим

Ведущий PTY может перевести подчиненный PTY в дистанционный режим, запустив функцию ioctl с командой TIOCREMOTE. Хотя FreeBSD 5.2.1, Mac OS X 10.3 и Solaris 9 используют для входа и выхода из этого режима одну и ту же команду, в Solaris в третьем аргументе функции ioctl передается целое чис-

ло, в то время как в FreeBSD и Mac OS X – указатель на целое число. (ОС Linux 2.4.22 не поддерживает эту команду.)

Установкой этого режима ведущий PTY сообщает модулю дисциплины обслуживания подчиненного PTY о том, что он не должен производить какую-либо обработку данных, поступающих от ведущего PTY, вне зависимости от состояния флага канонического режима в структуре `termios` подчиненного PTY. Дистанционный режим предназначен для таких приложений, как оконный менеджер, который реализует свою процедуру обработки строк.

Изменение размеров окна

Процесс, расположенный над ведущим PTY, может с помощью команды `TIOCWINSZ` функции `ioctl` изменить размер окна подчиненного PTY. Если новый размер окна отличается от текущего размера, то группе процессов переднего плана подчиненного PTY будет послан сигнал `SIGWINCH`.

Генерация сигналов

Процесс, имеющий доступ к ведущему PTY, может посылать сигналы группе процессов подчиненного PTY. В Solaris 9 это можно сделать с помощью команды `TIOCSIGNAL` функции `ioctl`, в третьем аргументе которой передается номер сигнала. В FreeBSD 5.2.1 и Mac OS X 10.3 для этих целей используется команда `TIOCSIG` функции `ioctl`, в третьем аргументе которой передается указатель на целочисленную переменную, содержащую номер сигнала. (ОС Linux 2.4.22 не поддерживает ни одну из этих команд.)

19.8. Подведение итогов

Эту главу мы начали с краткого обзора области применения псевдотерминалов и рассмотрели ряд примеров их использования. После этого мы исследовали код открытия псевдотерминала под управлением каждой из четырех платформ, обсуждаемых в этой книге. Затем мы использовали этот код для реализации универсальной функции `pty_fork`, которая может применяться различными приложениями. Эта функция легла в основу небольшой программы `pty`, с помощью которой мы смогли исследовать многочисленные свойства псевдотерминалов.

Псевдотерминалы широко применяются в большинстве современных версий UNIX для организации сетевого входа в систему. Мы рассмотрели и другие способы использования псевдотерминалов – от простой программы `script` до управления интерактивными программами из сценариев пакетной обработки данных.

Упражнения

19.1. В процессе удаленного входа в BSD-систему посредством программы `telnet` либо `rlogin` идентификатор владельца и права доступа для под-

чиненного PTY устанавливаются согласно правилам, описанным в разделе 19.3.2. Как это происходит?

19.2. Измените функцию `grantpt` из листинга 19.2 таким образом, чтобы для изменения владельца и прав доступа к подчиненному PTY в BSD-системах она вызывала программу с установленным битом set-user-ID (как это делается в ОС Solaris).

19.3. С помощью программы `pty` определите, какими значениями инициализируются структуры `termios` и `winsize` подчиненного PTY в вашей системе.

19.4. Реализуйте функцию `loop` (листинг 19.6) в виде отдельного процесса так, чтобы она использовала функцию `select` или `poll`.

19.5. В дочернем процессе после возврата из функции `pty_fork` стандартные устройства ввода, вывода и сообщений об ошибках открыты для чтения и записи. Можно ли изменить права доступа к ним так, чтобы стандартное устройство ввода было доступно только для чтения, а остальные два устройства – только для записи?

19.6. На рис. 19.7 попытайтесь определить, какая группа процессов выполняется на переднем плане, а какая в фоновом режиме. Определите лидеров сессий.

19.7. В каком порядке завершатся процессы из рис. 19.7, если ввести символ конца файла? Если возможно, проверьте это с помощью механизма учета ресурсов, потребляемых процессами.

19.8. Обычно программа `script`(1) добавляет в начало выходного файла строку, содержащую время начала работы, а в конец файла – время окончания. Добавьте эту возможность в сценарий на языке командной оболочки, который мы продемонстрировали.

19.9. Объясните, почему содержимое файла `data` в следующем примере выводится на терминал, если программа `ttyname` только выводит данные и никогда не читает их?

```
$ cat data                      файл с двумя строками текста
hello,
world
$ pty -i < data ttyname         ключ -i говорит о том, что символ EOF
                                должен игнорироваться
hello,                          откуда появились эти две строки?
world
fd 0: /dev/ttyp5                эти три строки мы ожидали
                                получить от ttyname
fd 1: /dev/ttyp5
fd 2: /dev/ttyp5
```

19.10. Напишите программу, которая вызывала бы функцию `pty_fork`, и дочерний процесс запускал бы другую программу, которую вы также должны написать. Новая программа, запускаемая из дочернего процесса, должна перехватывать сигналы `SIGTERM` и `SIGWINCH`. При получе-

нии сигналов программа должна выводить соответствующие сообщения, причем при получении сигнала SIGWINCH она должна дополнительно выводить размеры окна терминала. После запуска программы родительский процесс должен послать сигнал группе процессов подчиненного PTY с помощью функции ioctl, как это было описано в разделе 19.7, прочитать строку с сообщением от подчиненного PTY и убедиться, что дочерний процесс получил сигнал. Затем родительский процесс должен изменить размер окна подчиненного PTY и опять прочитать строку с сообщением от подчиненного PTY. Далее завершите родительский процесс и посмотрите, завершился ли дочерний. Если это произошло, то объясните почему.

20

Библиотека базы данных

20.1. Введение

В начале 80-х годов ОС UNIX считалась недружественной средой для многопользовательских систем управления базами данных. (См. [Stonebraker 1981] и [Weinberger 1982].) Для ранних версий UNIX, таких как Version 7, подобные утверждения действительно представлялись обоснованными, поскольку какие-либо разновидности механизмов межпроцессного взаимодействия (исключая полудуплексные каналы) отсутствовали, и механизм блокировок отдельных записей в файлах еще не был реализован. Однако со временем большинство из этих недостатков было устранено. К концу 80-х годов UNIX достигла такого уровня развития, который позволил обеспечить подходящую среду для работы надежных многопользовательских систем управления базами данных. С тех пор коммерческими фирмами было разработаны самые различные системы баз данных.

В этой главе мы создадим простую библиотеку функций на языке C, которая может быть использована любыми программами для получения и хранения записей в базе данных. Подобные библиотеки функций являются лишь частью полной системы управления базами данных. Мы не будем заниматься разработкой других ее компонентов, таких как язык запросов, оставляя освещение этих тем многочисленным учебникам по базам данных. Основной интерес для нас будут представлять интерфейсы UNIX, которые требуются для реализации библиотеки, и как эти интерфейсы связаны с уже рассмотренными темами (такими как блокировка записей в файлах, раздел 14.3).

20.2. Предыстория

Одной из популярных библиотек функций, предназначенных для работы с базами данных в UNIX, является библиотека `dbm`(3). Эта библиотека, разработанная Кеном Томпсоном (Ken Thompson), использует схему динамического хеширования. Изначально она распространялась вместе с Version 7, затем

была перенесена во все выпуски BSD, а также поставлялась вместе с SVR4 для обеспечения совместимости с BSD [AT&T 1990c]. Разработчики BSD расширили библиотеку dbm и назвали ее ndbm. Библиотека ndbm вошла в состав как BSD, так и SVR4. Функции ndbm стандартизированы в виде расширений XSI стандарта Single UNIX Specification.

Подробная история развития алгоритма динамического хеширования, используемого библиотекой dbm и последующими ее реализациями, включая gdbm (GNU-версия библиотеки dbm), приводится в книге [Seltzer and Yigit 1991]. К сожалению, основное ограничение всех этих реализаций заключается в том, что ни одна из них не допускает одновременное обновление данных из нескольких процессов. Эти реализации не предусматривают никаких средств управления одновременным доступом (например, блокировку записей).

В 4.4BSD была реализована новая библиотека db(3), которая поддерживала три формы выборки: (а) ориентированную на записи, (б) хеширование и (в) двоичные деревья (B-tree). Но она также не предоставляла возможности одновременного доступа (о чем прямо было заявлено в разделе BUGS страницы справочного руководства db(3)).

> Фирма Sleepycat Software (*http://www.sleepycat.com*) разработала версии библиотеки db, которые поддерживают одновременную работу нескольких пользователей, а также механизмы блокировки записей и транзакций.

Большинство коммерческих библиотек баз данных реализуют механизмы управления одновременным доступом к данным из нескольких процессов. Для этого они обычно используют рекомендательные блокировки, описанные в разделе 14.3, но нередко реализуют собственные примитивы, чтобы избежать накладных расходов, которые неизбежно возрастают, когда системные вызовы не могут установить уже захваченную блокировку. Чаще всего эти коммерческие системы реализуют базы данных на основе сбалансированных двоичных деревьев (B+ tree) [Comer 1979] или алгоритмов динамического хеширования – например, линейного хеширования [Litwin1980] или расширяемого хеширования [Faginet al. 1979].

В табл. 20.1 приводится список библиотек, которые обычно поставляются в составе четырех платформ, рассматриваемых в этой книге. Обратите внимание, что в ОС Linux поддержка функций dbm и ndbm предоставляется библиотекой gdbm.

Таблица 20.1. Библиотеки баз данных, поддерживаемые различными платформами

Библиотека	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
dbm			gdbm		•
ndbm	XSI	•	gdbm	•	•
db		•	•	•	•

20.3. Библиотека

Библиотека, которую мы разработаем в этой главе, будет похожа на библиотеку ndbm, но мы добавим к ней механизм одновременного доступа к данным из нескольких процессов. В первую очередь мы рассмотрим интерфейс к библиотеке на языке С, а в следующем разделе перейдем к фактической реализации.

При открытии базы данных мы получаем некоторый дескриптор (обычный указатель), который представляет эту базу данных. Этот дескриптор мы передаем всем функциям, работающим с базой данных.

```
#include "apue_db.h"

DBHANDLE db_open(const char *pathname, int oflag, ... /* int mode */);

                                        Возвращает дескриптор базы данных
                                        в случае успеха, NULL – в случае ошибки

void db_close(DBHANDLE db);
```

Если вызов функции db_open завершается успехом, она создает два файла: индексный файл *pathname.idx* и файл с данными *pathname.dat*. Аргумент *oflag* используется точно так же, как второй аргумент функции open (раздел 3.3) – он определяет режим открытия файлов (только для чтения, для записи и для чтения, создание файла, если он не существует, и т. д.). Аргумент *mode* используется при создании файлов базы данных подобно третьему аргументу функции open (он определяет права доступа).

По завершении работы с базой данных ее следует закрыть вызовом функции db_close. Эта функция закрывает индексный файл и файл с данными и освобождает память, которая была выделена под внутренние буферы.

При добавлении в базу новой записи необходимо указать ключ записи и данные, связанные с этим ключом. Так, если база данных хранит данные о сотрудниках, в качестве ключа может использоваться идентификатор сотрудника, а в качестве данных – имя сотрудника, его домашний адрес, номер телефона, дата приема на работу и т. п. Наша реализация требует, чтобы ключ для каждой из записей имел уникальное значение. (Это означает, что мы не сможем, например, создать две записи с одинаковыми идентификаторами.)

```
#include "apue_db.h"

int db_store(DBHANDLE db, const char *key, const char *data, int flag);

                                        Возвращает 0 в случае успеха,
                                        ненулевое значение – в случае ошибки (см. ниже)
```

Аргументы *key* и *data* – это строки, завершающиеся нулевым символом. Единственное ограничение, связанное с этими строками, состоит в том, что они не могут содержать нулевые символы в середине, но зато они могут содержать, например, символы перевода строки.

Аргумент *flag* может принимать значения DB_INSERT (при добавлении новой записи), DB_REPLACE (при изменении существующей записи) или DB_STORE (при добавлении новой или изменении существующей записи, в зависимости от наличия записи в базе). Эти три константы определены в заголовочном файле apue_db.h. Если указан флаг DB_INSERT или DB_STORE и запись не существует в базе, то будет добавлена новая запись. Если указан флаг DB_REPLACE или DB_STORE и запись уже существует в базе, то существующая запись будет замещена новой записью. Если указан флаг DB_REPLACE, а искомой записи в базе не окажется, то функция вернет значение –1 и код ошибки ENOENT в переменной errno, при этом новая запись добавлена не будет. Если указан флаг DB_INSERT и запись уже существует в базе, то добавление записи в базу не производится. В этом случае возвращается значение 1, чтобы можно было отличить его от обычного завершения с ошибкой (–1).

Мы можем извлечь запись из базы данных, указав ее ключ.

```
#include "apue_db.h"

char *db_fetch(DBHANDLE db, const char *key);
```

Возвращает указатель на данные в случае успеха, NULL, если запись не была найдена

Если запись была найдена, то функция возвращает указатель на данные, которые были сохранены с ключом *key*. Мы можем также удалить запись из базы данных, указав ее ключ.

```
#include "apue_db.h"

int db_delete(DBHANDLE db, const char *key);
```

Возвращает 0 в случае успеха, –1, если запись не была найдена

Кроме извлечения отдельных записей по заданным ключам, можно выполнить обход всей базы данных, считывая записи по очереди. Для этого нужно сначала вызвать функцию db_rewind, чтобы переместиться на первую запись, и затем в цикле вызывать db_nextrec, читая записи одну за другой.

```
#include "apue_db.h"

void db_rewind(DBHANDLE db);

char *db_nextrec(DBHANDLE db, char *key);
```

Возвращает указатель на данные в случае успеха, NULL по достижении конца файла

Если в аргументе *key* передается непустой указатель, то функция db_nextrec будет возвращать по этому адресу значение ключа очередной записи.

Порядок следования записей, возвращаемых db_nextrec, заранее не определен. Единственное, что можно гарантировать, – это то, что каждая запись

будет возвращена всего один раз. Так, если в базе хранятся три записи с ключами A, B и C, то нельзя заранее предсказать, в каком порядке они будут возвращены функцией db_nextrec. Она может вернуть сначала B, потом A, а потом C или в каком-либо другом (совершенно случайном) порядке. Фактический порядок следования записей зависит от реализации базы данных.

Эти семь функций составляют интерфейс библиотеки базы данных. А теперь перейдем к описанию фактической реализации, выбранной нами.

20.4. Обзор реализации

Обычно библиотеки, реализующие доступ к базе данных, для хранения данных используют два файла: индексный файл и файл с данными. Индексный файл содержит значения индексов (ключей) и указатели на соответствующие им записи в файле с данными. Для организации хранения индексов используется множество методик, которые ускоряют поиск конкретного ключа; хеширование и сбалансированные двоичные деревья являются наиболее популярными. Мы выбрали методику на основе хеш-таблицы фиксированного размера с объединением в цепочки записей, имеющих одинаковые значения хешей. При описании функции db_open мы уже упоминали, что она создает два файла: один с расширением .idx, а другой – с расширением .dat.

Мы будем хранить индексы и ключи в виде строк, завершающихся нулевым символом; они не допускают возможности хранения произвольных двоичных данных. Некоторые системы управления базами данных хранят числовые данные в двоичном формате (например, 1, 2 или 4 байта для хранения целых чисел), чтобы сэкономить место. Это приводит к усложнению функций и требует дополнительных усилий для обеспечения переносимости файлов базы данных между разными платформами. Например, если в сети имеются две системы, которые используют разные форматы представления целых чисел в двоичном формате, то придется предусмотреть обработку ситуации, когда необходим доступ к базе данных из обеих систем. (Сегодня нет ничего необычного в том, что посредством сети файлы совместно используются системами с различной архитектурой.) Хранение всех записей, и ключей и данных, в виде строк символов упрощает реализацию. Такой подход ведет к увеличению занимаемого дискового пространства, но это небольшая плата за переносимость.

Функция db_store допускает хранение только одной записи для каждого ключа. Некоторые системы управления базами данных позволяют хранить несколько записей с одинаковыми ключами и предоставляют возможность получить доступ ко всем записям, ассоциированным с заданным ключом. Кроме того, в нашем распоряжении будет всего один индексный файл – это означает, что каждой записи с данными может быть поставлен в соответствие только один ключ (мы не предусматриваем поддержку вторичных ключей).

Базы данных, которые допускают поставить в соответствие одной записи несколько ключей, очень часто создают по одному индексному файлу для каждого ключа. Каждый раз при удалении или добавлении записи все индексные файлы должны соответствующим образом обновляться. (Например, для

файла с данными о сотрудниках мы могли бы определить несколько индексов: один – идентификатор сотрудника, а другой – номер карточки социального обеспечения. Если в качестве индекса использовать имена сотрудников, это может породить определенные проблемы, так как имена могут не быть уникальными.)

На рис. 20.1 показана общая схема строения базы данных.

Индексный файл состоит из трех частей: указателя на список свободных записей, таблицы хешей и индексных записей. На рис. 20.1 все поля, которые названы указателями, представляют собой смещение от начала файла и хранятся в виде чисел в формате ASCII.

Чтобы отыскать в базе данных запись по заданному ключу, функция `db_fetch` рассчитывает значение хеша ключа, по которому отыскивается требуемая цепочка в таблице хешей. (Поле *указатель на цепочку* может содержать 0; это говорит о том, что цепочка пуста.) Затем осуществляется переход к най-

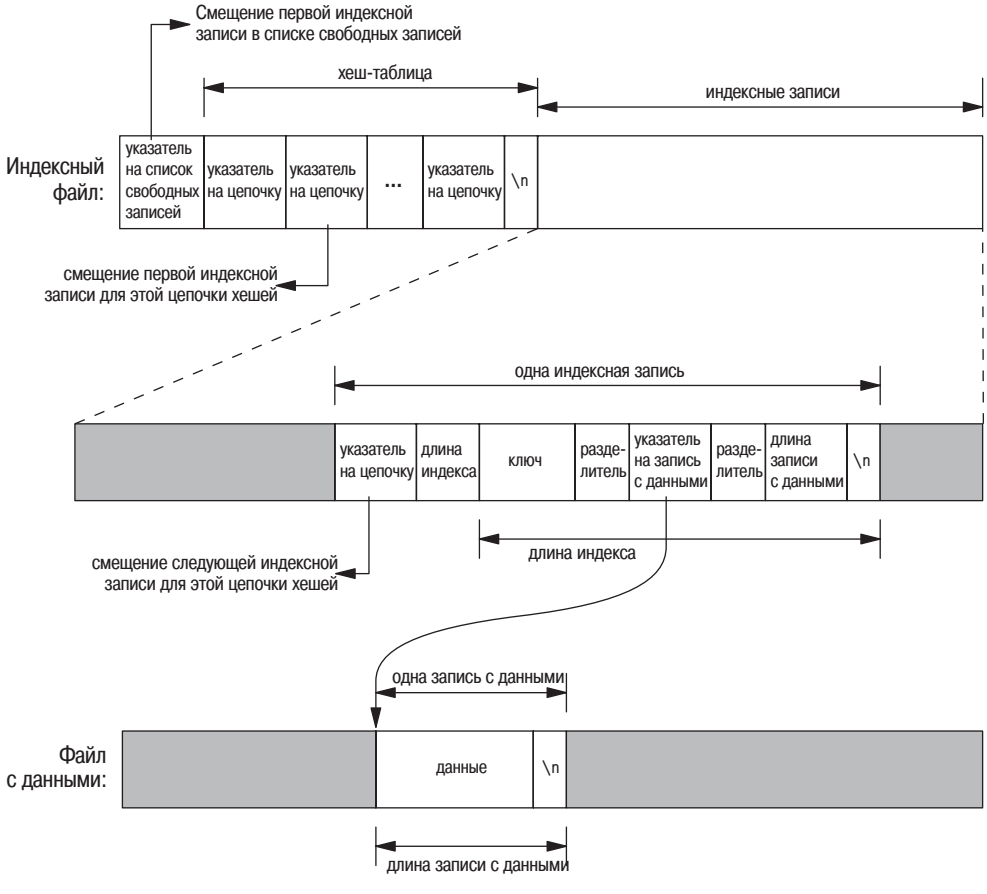


***Рис. 20.1.** Схема строения индексного файла и файла с данными*

денной цепочке, которая представляет собой связанный список из всех индексных записей, которые имеют то же самое значение хеша. Когда функция обнаруживает в поле *указатель на цепочку* значение 0, это означает, что достигнут конец цепочки.

Давайте рассмотрим фактическое содержимое файлов базы данных. Программа из листинга 20.1 создает новую базу данных и записывает в нее три записи. Поскольку все поля в базе данных хранятся в виде символов ASCII, мы можем просмотреть содержимое файлов, используя любые стандартные средства UNIX:

```
$ ls -l db4.*
-rw-r--r-- 1 sar         28 Oct 19 21:33 db4.dat
-rw-r--r-- 1 sar         72 Oct 19 21:33 db4.idx
$ cat db4.idx
   0   53   35    0
   0   10Alpha:0:6
   0   10beta:6:14
  17   11gamma:20:8
$ cat db4.dat
data1
Data for beta
record3
```

Чтобы уменьшить объем этого примера, мы ограничили размер поля *указатель* четырьмя символами ASCII, а количество цепочек в таблице хешей – тремя. Так как размер каждого поля *указатель*, являющегося смещением от начала файла, ограничен четырьмя символами, общий размер индексного файла и файла с данными не может превышать 9999 байт. В разделе 20.9, измеряя производительность базы данных, мы установим размер каждого поля *указатель* равным шести символам (что позволит увеличить размер файлов до 1 миллиона байт), а количество цепочек в таблице хешей будет превышать 100.

Первая строка в индексном файле

```
0   53   35    0
```

содержит указатель на первую запись в списке свободных записей (0 – список пуст) и три указателя на цепочки для каждого из хешей: 53, 35 и 0. Следующая строка

```
0   10Alpha:0:6
```

демонстрирует формат каждой индексной записи. Первое поле (0) – это четырехсимвольный указатель на следующую запись в цепочке. Данная запись является последней в цепочке. Следующее поле (10) – это длина индексной записи в 4-символьном формате. Каждая индексная запись считывается в два приема: первая операция чтения возвращает два поля фиксированной длины (*указатель на следующую запись* и *длина индексной записи*), а вторая операция чтения – остальную часть записи (переменной длины). Оставшиеся три поля в индексной записи – *ключ*, *указатель на запись с данными* и *длина записи с данными* – отделяются друг от друга символом-разделителем (в дан-

ном случае двоеточием). Символ-разделитель необходим, потому что каждое из полей имеет переменную длину. Вследствие этого символ-разделитель не может входить в состав ключа. Завершает индексную запись символ перевода строки. Строго говоря, символ перевода строки не требуется, поскольку поле *длина индексной записи* содержит длину записи. Но мы вставляем символ перевода строки для отделения одной индексной записи от другой, благодаря чему мы можем просматривать содержимое индексного файла стандартными утилитами, такими как `cat` или `more`. Поле *ключ* содержит значение, которое мы задали при добавлении записи в базу данных. Поля *указатель на запись с данными* (0) и *длина записи с данными* (6) относятся к файлу с данными. Эта информация говорит о том, что запись с данными расположена в самом начале файла и имеет длину 6 байт. (Как и в случае с индексами, мы автоматически добавляем символ перевода строки в конец каждой записи с данными, чтобы файл данных можно было просматривать с помощью стандартных утилит. Этот заключительный символ перевода строки не будет возвращаться функцией `db_fetch` вызывающему процессу.)

Листинг 20.1. Создает базу данных и записывает в нее три записи

```
#include "apue.h"
#include "apue_db.h"
#include <fcntl.h>

int
main(void)
{
    DBHANDLE db;

    if ((db = db_open("db4", O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) == NULL)
        err_sys("ошибка вызова функции db_open");

    if (db_store(db, "Alpha", "data1", DB_INSERT) != 0)
        err_quit("ошибка функции db_store при добавлении первой записи");
    if (db_store(db, "beta", "Data for beta", DB_INSERT) != 0)
        err_quit("ошибка функции db_store при добавлении второй записи ");
    if (db_store(db, "gamma", "record3", DB_INSERT) != 0)
        err_quit("ошибка функции db_store при добавлении третьей записи ");

    db_close(db);
    exit(0);
}
```

Если мы пройдемся по трем цепочкам хешей в данном примере, то увидим, что первая запись в первой цепочке имеет смещение 53 (`gamma`). Следующая запись в этой цепочке начинается со смещения 17 (`Alpha`) и является последней записью в цепочке. Первая запись во второй цепочке начинается со смещения 35 (`beta`) и является последней записью в цепочке. Третья цепочка пустая.

Обратите внимание, что порядок ключей в индексном файле и порядок соответствующих им записей в файле с данными – такие же, что и порядок вызовов функции `db_store` из листинга 20.3. Так как при вызове функции `db_open` был указан флаг `O_TRUNC`, размеры индексного файла и файла с данными будут

усечены, и база данных будет заново инициализирована. В такой ситуации db_store просто добавляет новые индексные записи и записи с данными в конец соответствующего файла. Позже мы увидим, как db_store может повторно использовать участки в файлах, соответствующие удаленным записям.

Выбор алгоритма поиска на основе хеш-таблицы фиксированного размера представляет собой компромиссное решение. Этот алгоритм дает высокую скорость поиска, если размеры цепочек невелики. Нам необходима высокая скорость поиска, но мы не хотим усложнять структуры данных, используя алгоритмы поиска на основе двоичных деревьев или динамического хеширования. Динамическое хеширование дает возможность отыскать любую запись всего за дваобращения к диску (за подробностями обращайтесь к [Litwin 1980] или [Fagin et al. 1979]). Двоичные деревья предоставляют возможность обхода базы данных в (отсортированном) порядке следования ключей (что невозможно сделать в функции db_nextrec, используя таблицу хешей).

20.5. Централизация или децентрализация?

Учитывая, что к одной и той же базе данных могут обращаться сразу несколько процессов, мы можем реализовать функции двумя способами:

1. Централизованный. Доступ к базе данных осуществляется посредством выделенного процесса – менеджера базы данных, и только этот процесс напрямую обращается к базе данных. Взаимодействие с центральным процессом осуществляется с помощью одного из механизмов IPC.
2. Децентрализованный. Каждая функция доступа к базе данных должна сначала применить необходимые средства управления одновременным доступом (захват блокировки) и затем производить операции ввода-вывода.

Системы управления базами данных могут быть построены по любой из этих схем. При использовании эффективных алгоритмов наложения блокировок децентрализованные реализации, как правило, дают более высокую производительность, так как в них не задействованы механизмы IPC. На рис. 20.2 показана схема реализации базы данных на основе централизованного подхода.

Мы намеренно поместили блок, отображающий механизмы межпроцессного взаимодействия, в ядро, так как в большинстве случаев передача сообщений в UNIX производится именно таким образом. (Механизм разделяемой памяти, описанный в разделе 15.9, позволяет избежать копирования данных в пространство ядра.) Как видите, при использовании централизованной схемы запись читается центральным процессом и затем передается запрашивающему процессу через механизм IPC. Это основной недостаток централизованной схемы. Обратите внимание: фактический доступ к файлам базы данных осуществляет только центральный процесс.

Но централизованный подход имеет и преимущества – он позволяет более точно настраивать порядок взаимодействия с клиентскими процессами. Например, при использовании централизованной схемы процессам могут быть назначены разные приоритеты. Они могут учитываться при планировании операций ввода-вывода центральным процессом. При использовании децен-

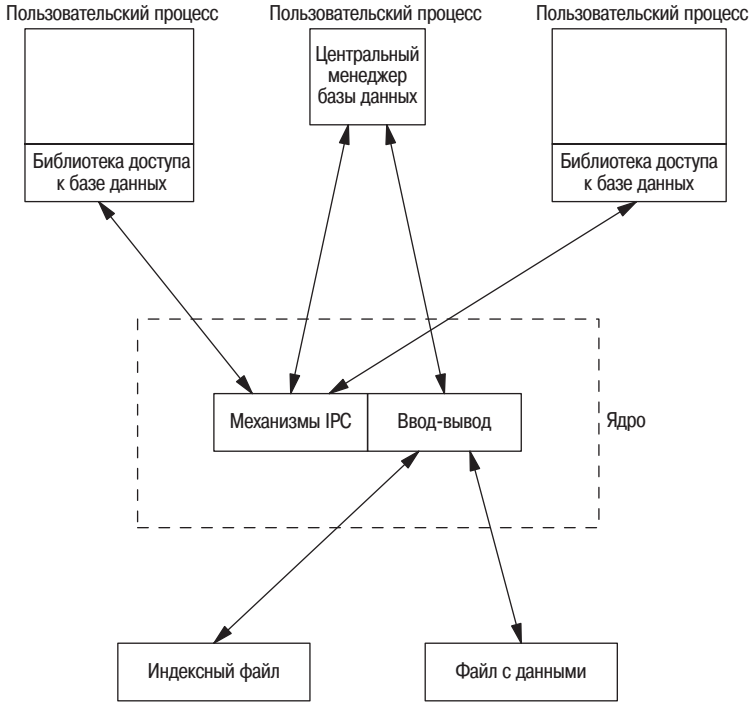


Рис. 20.2. *Централизованный доступ к базе данных*

трализованной схемы сделать это будет намного сложнее. В этом случае все обычно зависит от того, как ядро планирует дисковые операции ввода-вывода – если, например, три процесса ожидают снятия блокировки, какой из них первым сможет установить свою блокировку?

Централизованный подход имеет еще одно преимущество, которое заключается в том, что восстановление после ошибок производится проще, чем при использовании децентрализованной схемы. При централизованном подходе вся информация о состоянии базы данных находится в одном месте, поэтому если процесс базы данных завершится аварийно, то нам легко будет найти информацию о незавершенных транзакциях, которая необходима, чтобы привести базу данных в непротиворечивое состояние.

На рис. 20.3 показана схема реализации базы данных на основе децентрализованного подхода. Именно эту схему мы реализуем в данной главе.

Пользовательские процессы, вызывающие функции из библиотеки базы данных для выполнения операций ввода-вывода, рассматриваются как сотрудничающие процессы, так как они используют механизм блокировки записей в файле для обеспечения одновременного доступа.

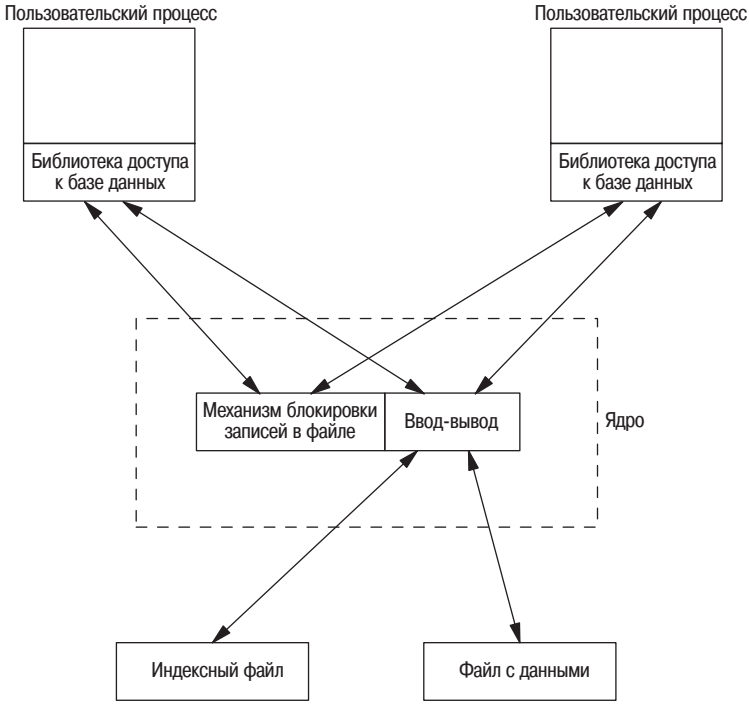


Рис. 20.3. *Децентрализованный доступ к базе данных*

20.6. Одновременный доступ

Мы специально выбрали реализацию базы данных на основе двух файлов (индексный файл и файл с данными), потому что это наиболее общий подход. Он требует наложения блокировок на оба файла. Но способов организовать такое наложение блокировок существует множество.

Крупноблочная блокировка

Самый простой вариант – использовать в качестве блокировки один из файлов базы данных и требовать, чтобы вызывающий процесс получил эту блокировку перед выполнением операций над базой данных. Мы назвали этот способ *крупноблочной блокировкой* (*coarse-grained locking*). Например, можно сказать, что процесс, получивший блокировку для чтения нулевого байта индексного файла, обладает правом на чтение всей базы данных. Процесс, получивший блокировку для записи в нулевой байт индексного файла, обладает правом на запись для всей базы данных. Мы можем использовать обычную семантику блокировок диапазонов байтов в UNIX, чтобы разрешить одновременное чтение данных сразу нескольким процессам, но запись – только одному процессу (табл. 14.2). При таком подходе функции `db_fetch` и `db_nextrec`

должны приобретать блокировку для чтения, а функции db_delete, db_store и db_open – блокировку для записи. (Блокировка для записи в функции db_open нужна потому, что при создании индексного файла в него должен быть записан список пустых записей и хеш-таблица.)

Проблема с крупноблочной блокировкой заключается в том, что она не дает большого выигрыша в производительности. Если один процесс добавляет запись в какую-либо цепочку хеш-таблицы, другой процесс должен иметь возможность читать записи из другой цепочки.

Мелкоблочная блокировка

Чтобы добиться более высокой производительности, мы будем использовать блокировку, которую мы назвали *мелкоблочной блокировкой* (*fine-grained locking*). Прежде всего, читающий или пишущий процесс должен получить блокировку для чтения или для записи на цепочку, в которой находится заданная запись. Мы допускаем возможность одновременного чтения цепочки несколькими процессами, но только один процесс может выполнять запись в цепочку. Кроме того, процесс, которому требуется получить доступ для записи к списку свободных записей (функции db_delete и db_store), должен получить блокировку на список свободных записей. И наконец, всякий раз, когда производится добавление новой записи в конец индексного файла или в конец файла с данными, функция db_store должна получить блокировку для записи на этот участок файла.

Мы полагаем, что мелкоблочная блокировка даст выигрыш в производительности по сравнению с крупноблочной блокировкой. В разделе 20.9 будут приведены результаты измерения производительности. В разделе 20.8 мы продемонстрируем исходный код нашей реализации библиотеки на основе мелкоблочной блокировки и обсудим подробности реализации блокировок. (Крупноблочная блокировка по сути является упрощенным ее вариантом.)

В исходном коде библиотеки вместо функций стандартной библиотеки ввода-вывода мы будем использовать функции read, write, readv и writev. Хотя и существует возможность использования блокировок диапазона байтов совместно со стандартной библиотекой ввода-вывода, но при этом необходимо учитывать особенности буферизации. Нам наверняка не нужно, чтобы fgets возвращала данные десятиминутной давности, если 5 минут назад запись была обновлена другим процессом.

Наше обсуждение проблемы одновременного доступа основывается на упрощенных требованиях библиотеки базы данных. Коммерческие системы зачастую предъявляют дополнительные требования. Подробнее вопросы организации одновременного доступа обсуждаются в [Date 2004].

20.7. Сборка библиотеки

Библиотека базы данных состоит из двух файлов: общедоступного заголовочного файла и файла с исходными текстами на C. Собрать статическую версию библиотеки можно командами:

```
gcc -I../include -Wall -c db.c
ar rsv libapue_db.a db.o
```

Приложение, которое будет связано с библиотекой libapue_db.a, также должно быть связано с библиотекой libapue.a, поскольку мы использовали некоторые функции из нее в библиотеке базы данных.

С другой стороны, если нам нужна динамическая версия библиотеки базы данных, то можно использовать следующие команды:

```
gcc -I../include -Wall -fPIC -c db.c
gcc -shared -Wl,-soname,libapue_db.so.1 -o libapue_db.so.1 \
    -L../lib -lapue -lc db.o
```

Полученный файл библиотеки libapue_db.so.1 должен быть помещен в каталог, где динамический загрузчик/редактор связей сможет отыскать его. Как вариант, можно поместить файл библиотеки в произвольный каталог и изменить содержимое переменной окружения LD_LIBRARY_PATH таким образом, чтобы она включала в себя путь к этой библиотеке.

> Команды компиляции и сборки динамических библиотек могут различаться на разных платформах. Здесь мы привели команды для ОС Linux, оснащенной компилятором GNU C compiler.

20.8. Исходный код

Рассмотрение исходного кода библиотеки мы начнем с заголовочного файла apue_db.h. Этот файл подключается библиотекой и всеми приложениями, которые к ней обращаются.

В этом обсуждении мы отступим от правил, которым следовали в предыдущих разделах. Во-первых, поскольку объем исходного кода значительно больше, чем обычно, мы будем нумеровать строки. Это позволит привязать обсуждение к конкретным участкам исходного кода. Во-вторых, мы будем помещать описание сразу же вслед за фрагментами кода, к которым оно относится, на той же самой странице.

> Такой стиль был использован Джоном Лайонсом (John Lions) в его книге, описывающей исходный код ОС UNIX Version 6 [Lions 1977, 1996]. Это упрощает исследование больших объемов кода.

Обратите внимание: пустые строки не пронумерованы. Хотя это правило и не соответствует поведению таких утилит, как pr(1), но нам нечего сказать о пустых строках.

```
1   #ifndef _APUE_DB_H
2   #define _APUE_DB_H

3   typedef void * DBHANDLE;

4   DBHANDLE db_open(const char *, int, ...);
5   void db_close(DBHANDLE);
6   char *db_fetch(DBHANDLE, const char *);
```

```
 7    int db_store(DBHANDLE, const char *, const char *, int);
 8    int db_delete(DBHANDLE, const char *);
 9    void db_rewind(DBHANDLE);
10    char *db_nextrec(DBHANDLE, char *);

11    /*
12     * Флаги для функции db_store().
13     */
14    #define DB_INSERT  1      /* вставить новую запись */
15    #define DB_REPLACE 2      /* заменить существующую запись */
16    #define DB_STORE   3      /* заменить или вставить */

17    /*
18     * Ограничения реализации.
19     */
20    #define IDXLEN_MIN    6  /* ключ, разделитель, смещение, */
                               /* разделитель, длина, \n */
21    #define IDXLEN_MAX 1024  /* выбрано произвольно */
22    #define DATLEN_MIN    2  /* байт данных, перевод строки */
23    #define DATLEN_MAX 1024  /* выбрано произвольно */

24    #endif                   /* _APUE_DB_H */
```

[1–3] Использование символа `_APUE_DB_H` гарантирует, что содержимое данного заголовочного файла будет подключено только один раз. Тип `DBHANDLE` представляет активную ссылку на базу данных и используется для сокрытия от приложений внутренних особенностей реализации базы данных. Сравните это с использованием структуры `FILE` при работе со стандартной библиотекой ввода-вывода.

[4–10] Здесь объявляются прототипы общедоступных библиотечных функций. Поскольку этот файл подключается из приложений, которые желают использовать библиотеку, мы не объявляем здесь частные функции библиотеки.

[11–24] Здесь объявлены флаги, которые могут передаваться функции `db_store`. Вслед за флагами объявляются фундаментальные ограничения реализации. Значения этих ограничений могут быть изменены, если необходима поддержка баз данных большего объема.

Минимальная длина индексной записи определяется константой `IDXLEN_MIN`. Значение этой константы складывается из 1 байта ключа, 1 байта символа-разделителя, 1 байта смещения, еще 1 байта символа-разделителя, 1 байта длины и завершающего символа перевода строки. (Формат индексной записи был показан на рис. 20.1.) Обычно индексные записи будут превышать `IDXLEN_MIN` байт, но в данном случае мы определяем минимально возможный размер.

Следующий файл – `db.c`, исходный код библиотеки на языке C. Для простоты мы включили все функции библиотеки в один файл. Такая организация имеет свои преимущества, поскольку мы можем скрыть частные функции, объявив их со спецификатором `static`.

```
1    #include "apue.h"
2    #include "apue_db.h"
3    #include <fcntl.h> /* флаги для функций open и db_open */
```

```
 4  #include <stdarg.h>
 5  #include <errno.h>
 6  #include <sys/uio.h> /* struct iovec */
 7  /*
 8   * Внутренние константы, имеющие отношение к индексному файлу.
 9   * Они используются при создании записей в индексном
10   * файле и в файле с данными.
11   */
12  #define IDXLEN_SZ  4  /* размер поля длины индексной записи */
                          /* (в символах ASCII) */
13  #define SEP      ':'  /* символ-разделитель полей в индексной записи */
14  #define SPACE    ' '  /* символ пробела */
15  #define NEWLINE '\n'  /* символ перевода строки */
16  /*
17   * Следующие определения необходимы для работы с цепочками
18   * в хеш-таблице и в списке свободных записей в индексном файле.
19   */
20  #define PTR_SZ        6   /* размер поля-указателя в цепочке */
21  #define PTR_MAX  999999   /* максимальное смещение */
                              /* в файле = 10**PTR_SZ - 1 */
22  #define NHASH_DEF   137   /* размер хеш-таблицы по умолчанию */
23  #define FREE_OFF      0   /* начало списка свободных записей */
                              /* в индексном файле */
24  #define HASH_OFF PTR_SZ   /* начало хеш-таблицы в индексном файле */
25  typedef unsigned long DBHASH; /* значения хешей */
26  typedef unsigned long  COUNT; /* беззнаковый счетчик */
```

[1–6] **Здесь мы подключили** `apue.h`**, потому что библиотека базы данных использует некоторые функции из нашей частной библиотеки. В свою очередь,** `apue.h` **подключает ряд стандартных заголовочных файлов, среди которых** `<stdio.h>` **и** `<unistd.h>`**. Заголовочный файл** `<stdarg.h>` **необходим потому, что в нем находятся определения функций для работы со списками аргументов переменной длины, которые используются в функции** `db_open`**.**

[7–26] **Размер поля, в котором хранится длина индексной записи, определяется константой** `IDXLEN_SZ`**. Далее следуют определения некоторых символов, таких как двоеточие и перевод строки, которые будут использоваться в качестве символов-разделителей. Символ пробела используется для «затирания» удаленных записей.**

Некоторые значения, определенные как константы, могут быть оформлены в виде переменных – при небольшом усложнении реализации. Например, размер хеш-таблицы мы ограничили 137 записями. Наверное, лучше было бы позволить вызывающему процессу задавать это значение в виде аргумента функции `db_open`**, основываясь на предполагаемом размере базы данных. Это значение затем можно было сохранять в начале индексного файла.**

```
27  /*
28   * Внутреннее представление базы данных в библиотеке.
29   */
30  typedef struct {
31      int idxfd;    /* дескриптор индексного файла */
```

```
32      int datfd;    /* дескриптор файла с данными */
33      char *idxbuf; /* адрес буфера для индексной записи */
34      char *datbuf; /* адрес буфера для записи с данными */
35      char *name;   /* имя базы данных, под которым она была открыта */
36      off_t idxoff; /* смещение индексной записи в индексном файле */
37      /* ключ располагается с позиции (idxoff + PTR_SZ + IDXLEN_SZ) */
38      size_t idxlen; /* длина индексной записи */
39      /* исключая IDXLEN_SZ байт, находящихся в начале записи */
40      /* включая символ перевода строки в конце записи */
41      off_t datoff;  /* смещение записи с данными в файле данных */
42      size_t datlen; /* длина записи с данными */
43      /* Включая символ перевода строки в конце записи */
44      off_t ptrval;   /* содержимое указателя на цепочку в индексной записи */
45      off_t ptroff;   /* смещение указателя, содержащего адрес этой записи */
46      off_t chainoff; /* смещение цепочки для этой индексной записи */
47      off_t hashoff;  /* смещение хеш-таблицы в индексном файле */
48      DBHASH nhash;   /* текущий размер хеш-таблицы */
49      COUNT cnt_delok;    /* счетчик удачных операций удаления */
50      COUNT cnt_delerr;   /* счетчик ошибочных операций удаления */
51      COUNT cnt_fetchok;  /* счетчик удачных операций извлечения данных */
52      COUNT cnt_fetcherr; /* счетчик ошибочных операций извлечения данных */
53      COUNT cnt_nextrec;  /* nextrec */
54      COUNT cnt_stor1;   /* store: DB_INSERT, нет пустых записей, добавить */
55      COUNT cnt_stor2;   /* store: DB_INSERT, есть пустые записи, */
                           /* использовать их */
56      COUNT cnt_stor3;   /* store: DB_REPLACE, другая длина, добавить */
57      COUNT cnt_stor4;   /* store: DB_REPLACE, та же длина, перезаписать */
58      COUNT cnt_storerr; /* счетчик ошибок добавления записи */
59  } DB;
```

[27–48] В структуре `DB` мы будем хранить всю информацию о каждой из открытых баз данных. Значение `DBHANDLE`, которое возвращается функцией `db_open` и используется всеми остальными функциями, в действительности представляет собой указатель на одну из этих структур, но это обстоятельство скрыто от вызывающего процесса.

Поскольку все указатели и размеры в базе данных хранятся в виде строк символов ASCII, мы будем преобразовывать их в числовые значения и сохранять в структуре. Кроме того, в структуре будет сохраняться размер хеш-таблицы, даже несмотря на то, что она имеет фиксированный размер. Сделано это на тот случай, если мы задумаем модернизировать библиотеку и позволить вызывающему процессу определять размер хеш-таблицы на этапе создания базы данных (упражнение 20.7).

[49–59] Последние десять полей структуры DB – это счетчики количества операций, завершившихся успехом или неудачей. Если понадобится проанализировать производительность базы данных, мы сможем написать функцию, которая будет возвращать эти статистические характеристики, но пока мы только позаботимся о существовании самих счетчиков и их заполнении.

```
60  /*
61   * Внутренние функции.
62   */
```

```
63  static DB       *_db_alloc (int);
64  static void     _db_dodelete(DB *);
65  static int      _db_find_and_lock(DB *, const char *, int);
66  static int      _db_findfree(DB *, int, int);
67  static void     _db_free(DB *);
68  static DBHASH   _db_hash(DB *, const char *);
69  static char   *_db_readdat(DB *);
70  static off_t    _db_readidx(DB *, off_t);
71  static off_t    _db_readptr(DB *, off_t);
72  static void     _db_writedat(DB *, const char *, off_t, int);
73  static void     _db_writeidx(DB *, const char *, off_t, int, off_t);
74  static void     _db_writeptr(DB *, off_t, off_t);

75  /*
76   * Открыть или создать базу данных. Аргументы аналогичны функции open(2).
77   */
78  DBHANDLE
79  db_open(const char *pathname, int oflag, ...)
80  {
81      DB     *db;
82      int    len, mode;
83      size_t i;
84      char   asciiptr[PTR_SZ + 1],
85             hash[(NHASH_DEF + 1) * PTR_SZ + 2];
86                 /* +2 для символа перевода строки и нулевого символа */
87      struct stat statbuff;

88      /*
89       * Разместить в памяти структуру DB и все необходимые буферы.
90       */
91      len = strlen(pathname);
92      if ((db = _db_alloc(len)) == NULL)
93          err_dump("db_open: ошибка_размещения структуры DB");
```

[60–74] Мы выбрали следующий порядок именования функций: имена всех общедоступных функций будут начинаться с префикса `db_`, а всех частных функций – с префикса `_db_`. Прототипы общедоступных функций объявлены в заголовочном файле библиотеки `apue_db.h`. Все частные функции объявлены со спецификатором `static`, благодаря чему они будут доступны только тем функциям, которые размещены в том же самом файле (содержащем реализацию библиотеки).

[75–93] Функция `db_open` принимает те же аргументы, что и функция `open`(2). Если вызывающий процесс предполагает создание новых файлов базы данных, в третьем аргументе он должен определить права доступа к создаваемым файлам. Функция `db_open` открывает индексный файл и файл с данными и в случае необходимости инициализирует содержимое индексного файла. Начинается эта функция с вызова функции `_db_alloc`, которая размещает в памяти и инициализирует структуру `DB`.

```
94      db->nhash = NHASH_DEF;  /* размер таблицы хешей */
95      db->hashoff = HASH_OFF; /* начало хеш-таблицы в индексном файле */
96      strcpy(db->name, pathname);
```

```
 97      strcat(db->name, ".idx");
 98      if (oflag & O_CREAT) {
 99          va_list ap;
100          va_start(ap, oflag);
101          mode = va_arg(ap, int);
102          va_end(ap);
103          /*
104           * Создать индексный файл и файл с данными.
105           */
106          db->idxfd = open(db->name, oflag, mode);
107          strcpy(db->name + len, ".dat");
108          db->datfd = open(db->name, oflag, mode);
109      } else {
110          /*
111           * Открыть индексный файл и файл с данными.
112           */
113          db->idxfd = open(db->name, oflag);
114          strcpy(db->name + len, ".dat");
115          db->datfd = open(db->name, oflag);
116      }
117      if (db->idxfd < 0 || db->datfd < 0) {
118          _db_free(db);
119          return(NULL);
120      }
```

[94–97] Продолжение инициализации структуры `DB`. Имя базы данных, которое передает вызывающий процесс, используется как префикс для имен файлов базы данных. Чтобы получить имя индексного файла, мы добавляем к полученному префиксу расширение `.idx`.

[98–108] Если вызывающий процесс желает создать новые файлы базы данных, то для того, чтобы получить значение третьего аргумента, мы используем функции для работы со списками аргументов переменной длины из заголовочного файла `<stdarg.h>`. После этого мы создаем и открываем файлы базы данных с помощью функции `open`. Обратите внимание: имя файла с данными начинается с того же самого префикса, что и имя индексного файла, но в отличие от последнего имеет расширение `.dat`.

[109–116] Если вызывающий процесс не указал флаг `O_CREAT`, то мы открываем существующие файлы базы данных. В этом случае мы просто вызываем `open` с двумя аргументами.

[117–120] Если в процессе открытия файлов возникла ошибка, мы вызываем функцию `_db_free`, чтобы освободить память, занимаемую структурой `DB`, и возвращаем значение `NULL` вызывающему процессу. Если случилось так, что один файл был благополучно открыт, а второй нет, то функция `_db_free` позаботится о закрытии открытого дескриптора файла, в чем мы вскоре убедимся.

```
121      if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
122          /*
```

```
123          * Если была создана новая база данных, мы должны инициализировать
124          * ее. Блокировка для записи всего файла обеспечит атомарность
125          * операции получения его характеристик и инициализации.
126          */
127         if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
128             err_dump("db_open: ошибка вызова функции writew_lock");
129         if (fstat(db->idxfd, &statbuff) < 0)
130             err_sys("db_open: ошибка вызова функции fstat");
131         if (statbuff.st_size == 0) {
132             /*
133              * Мы должны создать список из (NHASH_DEF + 1) указателей
134              * на цепочки с нулевыми значениями. В данном случае +1 -
135              * это место для указателя на список свободных
136              * записей перед таблицей.
137              */
138             sprintf(asciiptr, "%*d", PTR_SZ, 0);
```

[121–130] При создании базы данных мы сталкиваемся с необходимостью наложения блокировки. Рассмотрим два процесса, которые пытаются создать базу данных с одним и тем же именем примерно в одно и то же время. Пусть первый процесс вызвал функцию `fstat` и был приостановлен ядром после того, как `fstat` вернула управление. Второй процесс также вызвал `db_open`, обнаружил, что индексный файл имеет нулевую длину, и инициализировал цепочки хешей и список свободных записей. Затем второй процесс записал одну запись в базу. В этот момент ядро приостановило второй процесс и передало управление первому процессу, который продолжил выполнение функции `db_open` сразу после вызова функции `fstat`. Первый процесс обнаруживает, что индексный файл имеет нулевой размер (поскольку вызов `fstat` произошел еще до того, как второй процесс инициализировал индексный файл), и повторно инициализирует цепочки хешей и список свободных записей, стирая то, что записал в базу данных второй процесс. Чтобы предотвратить подобное развитие событий, необходимо применять блокировки. Здесь мы будем использовать макросы `readw_lock`, `writew_lock` и `un_lock` из раздела 14.3.

[131–137] Если индексный файл имеет нулевой размер, это значит, что он был только что создан и необходимо инициализировать цепочки хешей и список свободных записей. Обратите внимание, что для преобразования числа, представляющего значение указателя, в строку ASCII мы используем формат `%*d`. (Аналогичную строку формата мы будем использовать в функциях `_db_writeidx` и `_db_writeptr`.) Этот формат сообщает функции `sprintf`, что она должна использовать аргумент `PTR_SZ` в качестве минимального размера ширины поля для вывода следующего аргумента, который в данном случае представлен числом 0 (мы инициализируем нулями все указатели, поскольку создается новая база данных). Это приводит к тому, что создается строка, содержащая по меньшей мере `PTR_SZ` символов (дополненная слева пробелами). В функциях `_db_writeidx` и `_db_writeptr` вместо нуля мы будем передавать фактические значения указателей и обязательно будем сравнивать эти значения с константой `PTR_MAX`, чтобы гарантировать, что каждый указатель, записываемый в базу данных, имеет размер точно `PTR_SZ` (6) символов.

```
138             hash[0] = 0;
139             for (i = 0; i < NHASH_DEF + 1; i++)
140                 strcat(hash, asciiptr);
141             strcat(hash, "\n");
142             i = strlen(hash);
143             if (write(db->idxfd, hash, i) != i)
144                err_dump("db_open: ошибка инициализации индексного файла");
145         }
146         if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
147             err_dump("db_open: ошибка вызова функции un_lock");
148     }
149     db_rewind(db);
150     return(db);
151 }
152 /*
153  * Размещает в памяти и инициализирует структуру DB и ее буферы.
154  */
155 static DB *
156 _db_alloc(int namelen)
157 {
158     DB *db;
159     /*
160      * Функция calloc выделяет память и забивает ее нулями.
161      */
162     if ((db = calloc(1, sizeof(DB))) == NULL)
163         err_dump("_db_alloc: ошибка размещения структуры DB");
164     db->idxfd = db->datfd = -1; /* дескрипторы */
165     /*
166      * Выделить место для имени.
167      * +5 для ".idx" или ".dat" и нулевого байта в конце.
168      */
169     if ((db->name = malloc(namelen + 5)) == NULL)
170         err_dump("_db_alloc: ошибка выделения памяти для имени");
```

[138–151] Продолжается инициализация вновь созданной базы данных. Здесь производится построение таблицы хешей и запись ее в индексный файл. После этого снимается блокировка с индексного файла, производится переход к началу базы данных и вызывающему процессу возвращается указатель на структуру `DB` как своего рода дескриптор, который будет использоваться при вызове всех остальных функций библиотеки.

[152–164] Для размещения в памяти структуры `DB`, индексного буфера и буфера с данными функция `db_open` вызывает `_db_alloc`. Для выделения памяти под структуру `DB` мы используем функцию `calloc`, которая очищает выделенную память, забивая ее нулями. При этом в качестве побочного эффекта мы получаем дескрипторы файлов базы данных со значениями 0, поэтому нам необходимо записать в них число –1, чтобы указать, что дескрипторы еще не открыты.

[165–170] Далее выделяется пространство для хранения имени файла базы данных. Этот буфер будет использоваться для конструирования имен обоих файлов путем изменения расширения, как мы это видели в функции `db_open`.

```
171     /*
172      * Выделить память для индексного буфера и для буфера данных.
173      * +2 для символа перевода строки и нулевого символа в конце.
174      */
175     if ((db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
176         err_dump("_db_alloc: ошибка распределения индексного буфера");
177     if ((db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
178         err_dump("_db_alloc: ошибка распределения буфера данных");
179     return(db);
180   }

181   /*
182    * Закрыть доступ к базе данных.
183    */
184   void
185   db_close(DBHANDLE h)
186   {
187     _db_free((DB *)h); /* закрывает дескрипторы файлов, освобождает память */
188   }

189   /*
190    * Освободить память, занимаемую структурой DB и буферами.
191    * А также закрыть дескрипторы файлов, которые могут быть открыты.
192    */
193   static void
194   _db_free(DB *db)
195   {
196     if (db->idxfd >= 0)
197         close(db->idxfd);
198     if (db->datfd >= 0)
199         close(db->datfd);
```

[171–180] Мы выделяем память для буферов, в которых будет храниться информация из индексного файла и файла с данными. Размеры буферов определены в заголовочном файле `apue_db.h`. При дальнейшем усовершенствовании библиотеки мы сможем предусмотреть увеличение размеров этих буферов по мере необходимости. При этом мы должны будем отслеживать их текущие размеры и вызывать функцию `realloc`, когда возникает необходимость в буферах большего размера. В завершение мы возвращаем указатель на структуру `DB`, которую только что распределили.

[181–188] Функция `db_close` представляет собой функцию-обертку, которая приводит дескриптор базы данных к типу `DB*` и передает его функции `_db_free`, чтобы освободить все занимаемые ресурсы.

[189–199] Функция `_db_free` вызывается из `db_open`, если в процессе открытия базы данных возникли ошибки, а также из `db_close`, когда приложение прекращает работу с базой данных. Если дескриптор индексного файла открыт, мы закрываем его. То же самое происходит и с дескриптором файла данных. (Мы уже говорили ранее, что функция `_db_alloc` инициализирует дескрипторы файлов значениями −1. Если мы не сможем открыть какой-либо из файлов базы данных, соответствующий ему дескриптор будет иметь значение −1 и мы не будем даже пытаться закрыть его.)

```
200     if (db->idxbuf != NULL)
201         free(db->idxbuf);
202     if (db->datbuf != NULL)
203         free(db->datbuf);
204     if (db->name != NULL)
205         free(db->name);
206     free(db);
207   }
208   /*
209    * Извлечь одну запись. Возвращает указатель на строку с данными.
210    */
211   char *
212   db_fetch(DBHANDLE h, const char *key)
213   {
214     DB *db = h;
215     char *ptr;
216     if (_db_find_and_lock(db, key, 0) < 0) {
217         ptr = NULL;     /* ошибка, запись не найдена */
218         db->cnt_fetcherr++;
219     } else {
220         ptr = _db_readdat(db); /* вернуть указатель на строку с данными */
221         db->cnt_fetchok++;
222     }
223     /*
224      * Снять блокировку с цепочки, установленную в _db_find_and_lock.
225      */
226     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
227         err_dump("db_fetch: ошибка вызова функции un_lock");
228     return(ptr);
229   }
```

[200–207] Далее мы освобождаем память, занимаемую буферами, распределенными динамически. Мы можем без опаски передавать пустой указатель функции `free`, поэтому нет необходимости выполнять дополнительные проверки значений каждого из указателей, но мы делаем такие проверки, потому что считаем, что лучше освобождать только те объекты, которые действительно были размещены (не все функции освобождения динамической памяти так дружелюбны, как функция `free`). В заключение мы освобождаем память, занимаемую структурой `DB`.

[208–218] Функция `db_fetch` используется для извлечения записи по заданному ключу. Прежде всего мы пытаемся с помощью функции `_db_find_and_lock` найти требуемую запись. Если запись не найдена, мы записываем `NULL` в возвращаемое значение и увеличиваем счетчик неудачных обращений. Поскольку `_db_find_and_lock` устанавливает блокировку на индексный файл, мы не можем вернуть управление, пока не снимем ее.

[219–229] Если запись была найдена, вызывается функция `_db_readdat`, которая читает данные и увеличивает счетчик удачных обращений. Перед возвратом управления мы снимаем блокировку с индексного файла вызовом функции `un_lock`. После этого мы возвращаем указатель на найденную запись (или `NULL`, если запись не была найдена).

```
230  /*
231   * Отыскать заданную запись. Вызывается из db_delete, db_fetch
232   * и db_store. Устанавливает блокировку на цепочку из хеш-таблицы.
233   */
234  static int
235  _db_find_and_lock(DB *db, const char *key, int writelock)
236  {
237    off_t offset, nextoffset;

238    /*
239     * Рассчитать значение хеша для данного ключа и найти
240     * смещение соответствующей цепочки в хеш-таблице.
241     * С этого места начинается поиск. Прежде всего мы должны
242     * рассчитать смещение в хеш-таблице для данного ключа.
243     */
244    db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
245    db->ptroff = db->chainoff;

246    /*
247     * Здесь устанавливается блокировка. Вызывающая функция должна снять
248     * ее. Внимание: блокировка устанавливается только на первый байт.
249     */
250    if (writelock) {
251        if (writew_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
252            err_dump("_db_find_and_lock: ошибка вызова writew_lock");
253    } else {
254        if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
255            err_dump("_db_find_and_lock: ошибка вызова readw_lock");
256    }

257    /*
258     * Получить смещение первой записи в данной цепочке от начала
259     * индексного файла (может быть 0).
260     */
261    offset = _db_readptr(db, db->ptroff);
```

[230–237] Функция `_db_find_and_lock` используется библиотекой для поиска записи по заданному ключу. В аргументе `writelock` передается ненулевое значение, если на время поиска необходимо установить блокировку для записи. Чтобы на время поиска установить блокировку для чтения, в аргументе `writelock` передается значение 0.

[238–256] В функции `_db_find_and_lock` производится подготовка к обходу цепочки. Ключ преобразуется в значение хеша, которое используется для вычисления смещения цепочки от начала файла (`chainoff`). Прежде чем приступить к поиску по цепочке, мы ожидаем, пока не будет установлена блокировка. Обратите внимание: блокировка устанавливается только на первый байт цепочки. Это позволяет нескольким процессам одновременно производить поиск по разным цепочкам.

[257–261] Чтобы получить первый указатель из цепочки, мы вызываем функцию `_db_readptr`. Если она возвращает 0, это значит, что цепочка пуста.

```
262    while (offset != 0) {
263        nextoffset = _db_readidx(db, offset);
```

```
264         if (strcmp(db->idxbuf, key) == 0)
265             break;              /* найдено совпадение */
266         db->ptroff = offset;  /* смещение данной записи  */
267         offset = nextoffset;  /* переход к следующей записи */
268     }
269     /*
270      * offset == 0 означает ошибку (запись не найдена).
271      */
272     return(offset == 0 ? -1 : 0);
273   }
274   /*
275    * Вычислить значение хеша по ключу.
276    */
277   static DBHASH
278   _db_hash(DB *db, const char *key)
279   {
280     DBHASH hval = 0;
281     char c;
282     int i;
283     for (i = 1; (c = *key++) != 0; i++)
284         hval += c * i; /* произведение ASCII-кода символа и его индекса */
285     return(hval % db->nhash);
286 }
```

[262–268] В цикле `while` производится обход всех индексных записей в цепочке и выполняется сравнение ключей. Чтение записей выполняется функцией `_db_readidx`. Она заполняет буфер `idxbuf` строкой ключа из текущей записи. Если функция `_db_readidx` возвращает 0, то мы достигли конца цепочки.

[269–273] Если после выхода из цикла в переменной `offset` содержится значение 0, это значит, что мы добрались до конца цепочки, но искомую запись не нашли, поэтому возвращается значение –1. В противном случае совпадение было найдено (выполнение цикла `while` было прервано оператором `break`), и возвращается признак успеха – значение 0. В этом случае поле `ptroff` будет содержать адрес предыдущей индексной записи, `dataoff` – адрес записи с данными, а `datalen` – размер записи с данными. Так как при обходе цепочки мы сохраняем адрес предыдущей индексной записи, которая ссылается на текущую, мы будем использовать это обстоятельство при удалении записи, поскольку в этом случае надо будет изменить указатель в предыдущей записи, чтобы удалить текущую.

[274–286] Функция `_db_hash` вычисляет значение хеша по заданному ключу. Она находит сумму произведений ASCII-кодов символов на их индексы (начиная с 1) и делит результат на количество записей в хеш-таблице. Согласно [Knuth 1998], элементарные хеш-функции обычно дают более равномерные характеристики распределения.

```
287   /*
288    * Прочитать поле указателя на цепочку из индексного файла:
289    * указатель на список свободных записей, на цепочку из хеш-таблицы
```

```
290     * или на индексную запись в цепочке.
291     */
292    static off_t
293    _db_readptr(DB *db, off_t offset)
294    {
295      char asciiptr[PTR_SZ + 1];

296      if (lseek(db->idxfd, offset, SEEK_SET) == -1)
297          err_dump("_db_readptr: ошибка перемещения на поле с указателем");
298      if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
299          err_dump("_db_readptr: ошибка чтения поля с указателем");
300      asciiptr[PTR_SZ] = 0; /* завершающий нулевой символ */
301      return(atol(asciiptr));
302    }

303    /*
304     * Прочитать следующую индексную запись, начиная с указанного смещения
305     * в индексном файле. Индексная запись считывается в буфер db->idxbuf,
306     * а символы-разделители замещаются нулевыми байтами. Если все в порядке,
307     * мы записываем в db->datoff и db->datlen смещение и длину
308     * соответствующей записи из файла с данными.
309     */
310    static off_t
311    _db_readidx(DB *db, off_t offset)
312    {
313      ssize_t i;
314      char *ptr1, *ptr2;
315      char asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
316      struct iovec iov[2];
```

[287–302] **Функция** `_db_readptr` **считывает один из трех возможных указателей: (а) указатель на первую запись из списка свободных индексных записей, (б) указатель в хеш-таблице, который указывает на первую запись в цепочке и (в) указатель, который хранится в начале каждой индексной записи (неважно, является ли эта запись частью цепочки или частью списка свободных записей). Перед возвратом мы преобразуем значение указателя из ASCII-представления в длинное целое. Функция** `_db_readptr` **не устанавливает никаких блокировок – это должно выполняться в вызывающей функции.**

[303–316] **Функция** `_db_readidx` **используется для чтения индексной записи с заданным смещением из индексного файла. В случае успеха функция возвращает смещение очередной записи в списке и заполняет некоторые поля структуры** `DB`**:** `idxoff` **– смещение текущей записи в индексном файле,** `ptrval` **– смещение следующей записи в списке,** `idxlen` **– длина текущей индексной записи,** `idxbuf` **– сама индексная запись,** `datoff` **– смещение записи в файле с данными и** `datalen` **– длина записи с данными.**

```
317     /*
318      * Позиция в файле и смещение записи. db_nextrec вызывает
319      * эту функцию с offset==0, что означает чтение из текущей позиции.
320      * Мы все равно должны вызвать lseek, чтобы прочитать запись.
321      */
```

```
322     if ((db->idxoff = lseek(db->idxfd, offset,
323       offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
324         err_dump("_db_readidx: ошибка вызова функции lseek");

325     /*
326      * Прочитать длину записи и указатель на следующую запись
327      * в начале текущей индексной записи. Это позволит нам
328      * прочитать оставшуюся часть индексной записи.
329      */
330     iov[0].iov_base = asciiptr;
331     iov[0].iov_len = PTR_SZ;
332     iov[1].iov_base = asciilen;
333     iov[1].iov_len = IDXLEN_SZ;
334     if ((i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
335         if (i == 0 && offset == 0)
336             return(-1); /* признак конца файла для db_nextrec */
337         err_dump("_db_readidx: ошибка readv при чтении индексной записи");
338     }

339     /*
340      * Это возвращаемое значение, всегда >= 0.
341      */
342     asciiptr[PTR_SZ] = 0;        /* завершающий нулевой символ */
343     db->ptrval = atol(asciiptr); /* смещение следующей записи в цепочке */

344     asciilen[IDXLEN_SZ] = 0;     /* завершающий нулевой символ */
345     if ((db->idxlen = atoi(asciilen)) < IDXLEN_MIN ||
346       db->idxlen > IDXLEN_MAX)
347         err_dump("_db_readidx: неверная длина записи");
```

[317–324] Мы начинаем с установки позиции в индексном файле, полученной от вызывающей функции. Смещение записывается в структуру `DB`, поэтому, даже если вызывающая функция предполагает чтение из текущей позиции файла (передавая в аргументе `offset` значение 0), мы все равно должны вызвать `lseek`, чтобы определить эту позицию. Поскольку ни одна индексная запись не хранится со смещением 0, мы можем определить для этого значения специальный смысл – «прочитать запись из текущей позиции».

[325–338] С помощью функции `readv` из начала текущей записи производится чтение двух полей фиксированной длины: указателя на следующую запись в цепочке и размера текущей индексной записи.

[339–347] Мы преобразуем смещение следующей записи в целое число и запоминаем его в поле `ptrval` (оно будет использовано как возвращаемое значение этой функции). Затем аналогичным образом выполняется преобразование значения длины текущей записи в целое число, которое сохраняется в поле `idxlen`.

```
348     /*
349      * Теперь будет прочитана сама запись. Мы прочитаем ее в индексный
350      * буфер, который был распределен при открытии базы данных.
351      */
```

```
352     if ((i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
353         err_dump("_db_readidx: ошибка чтения индексной записи");
354     if (db->idxbuf[db->idxlen-1] != NEWLINE) /* проверка целостности */
355         err_dump("_db_readidx: отсутствует символ перевода строки");
356     db->idxbuf[db->idxlen-1] = 0; /* заменить NL нулевым символом */

357     /*
358      * Найти символы-разделители в индексной записи.
359      */
360     if ((ptr1 = strchr(db->idxbuf, SEP)) == NULL)
361         err_dump("_db_readidx: отсутствует первый разделитель");
362     *ptr1++ = 0; /* заменить SEP нулевым символом */

363     if ((ptr2 = strchr(ptr1, SEP)) == NULL)
364         err_dump("_db_readidx: отсутствует второй разделитель");
365     *ptr2++ = 0; /* заменить SEP нулевым символом */

366     if (strchr(ptr2, SEP) != NULL)
367         err_dump("_db_readidx: слишком много символов-разделителей");

368     /*
369      * Получить смещение и длину записи с данными.
370      */
371     if ((db->datoff = atol(ptr1)) < 0)
372         err_dump("_db_readidx: смещение записи с данными < 0");
373     if ((db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
374         err_dump("_db_readidx: неверная длина записи с данными");
375     return(db->ptrval); /* вернуть позицию следующей индексной записи */
376 }
```

[348–356] **Мы читаем индексную запись в поле `idxbuf` структуры `DB`. Запись должна заканчиваться символом перевода строки, который мы заменяем нулевым символом. Если индексный файл поврежден, мы завершаем работу процесса с созданием файла `core`, вызвав функцию `err_dump`.**

[357–367] **Мы делим индексную запись на три поля: ключ, позиция соответствующей записи с данными и длина записи с данными. Функция `strchr` отыскивает первое вхождение заданного символа в заданной строке. В данной ситуации нас интересуют символы-разделители (константа `SEP`, которая определена как символ двоеточия).**

[368–376] **Позиция записи с данными и ее длина преобразуются в числовое представление и запоминаются в структуре `DB`. Затем мы возвращаем позицию следующей индексной записи в цепочке. Обратите внимание: мы не считываем запись с данными. Это будет сделано в вызывающей функции – например, в `db_fetch`. Мы не читаем записи с данными до тех пор, пока функция `_db_find_and_lock` не прочитает индексную запись, совпадающую с искомой.**

```
377 /*
378  * Прочитать текущую запись с данными в буфер.
379  * Вернуть указатель на буфер со строкой, завершающейся нулевым символом.
380  */
381 static char *
```

```
382  _db_readdat(DB *db)
383  {
384    if (lseek(db->datfd, db->datoff, SEEK_SET) == -1)
385        err_dump("_db_readdat: ошибка вызова функции lseek");
386    if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
387        err_dump("_db_readdat: ошибка вызова функции read");
388    if (db->datbuf[db->datlen-1] != NEWLINE) /* проверка целостности */
389        err_dump("_db_readdat: отсутствует символ перевода строки");
390    db->datbuf[db->datlen-1] = 0; /* заменить NL нулевым символом */
391    return(db->datbuf); /* вернуть указатель на запись с данными */
392  }

393  /*
394   * Удалить заданную запись.
395   */
396  int
397  db_delete(DBHANDLE h, const char *key)
398  {
399    DB *db = h;
400    int rc = 0; /* предполагается, что запись будет найдена */

401    if (_db_find_and_lock(db, key, 1) == 0) {
402 _        db_dodelete(db);
403        db->cnt_delok++;
404    } else {
405        rc = -1; /* не найдена */
406        db->cnt_delerr++;
407    }
408    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
409        err_dump("db_delete: ошибка вызова функции un_lock");
410    return(rc);
411  }
```

[377–392] **Функция** `_db_readdat` **заполняет поле** `databuf` **структуры** `DB` **содержимым записи с данными, предполагая, что в поля** `dataoff` **и** `datalen` **предварительно были записаны корректные значения.**

[393–411] **Функция** `db_delete` **используется для удаления записи по заданному ключу. С помощью функции** `_db_find_and_lock` **мы выполняем поиск требуемой записи в базе данных, и если таковая была найдена, вызываем функцию** `_db_dodelete`**, которая выполняет все необходимые действия. Третий аргумент функции** `_db_find_and_lock` **определяет характер блокировки – для чтения или для записи. Здесь мы запрашиваем установку блокировки для записи, поскольку вероятнее всего нам потребуется внести изменения в список. Так как** `_db_find_and_lock` **возвращает управление с установленной блокировкой, необходимо снять ее независимо от того, была найдена запись или нет.**

```
412  /*
413   * Удалить текущую запись, заданную в структуре DB.
414   * Эта функция вызывается из db_delete и db_store после того,
415   * как запись будет найдена функцией _db_find_and_lock.
416   */
```

```
417  static void
418  _db_dodelete(DB *db)
419  {
420    int i;
421    char *ptr;
422    off_t freeptr, saveptr;
423    /*
424     * Очистить индексный буфер и буфер с данными, забив их пробелами.
425     */
426    for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
427        *ptr++ = SPACE;
428    *ptr = 0; /* завершающий нулевой символ для _db_writedat */
429    ptr = db->idxbuf;
430    while (*ptr)
431        *ptr++ = SPACE;
432    /*
433     * Мы должны заблокировать список свободных записей.
434     */
435    if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
436        err_dump("_db_dodelete: ошибка вызова функции writew_lock");
437    /*
438     * Записать очищенную запись с данными.
439     */
440    _db_writedat(db, db->datbuf, db->datoff, SEEK_SET);
```

[412–431] Функция `_db_dodelete` выполняет все необходимые действия по удалению записи из базы данных. (Эта функция также вызывается из `db_store`.) По сути, эта функция лишь обновляет два связанных списка: список свободных записей и цепочку из хеш-таблицы, в которой находился заданный ключ. При удалении записи ключ и запись с данными заполняются пробелами. Это обстоятельство будет использовано функцией `db_nextrec`, которую мы исследуем в конце этого раздела.

[432–440] Чтобы установить блокировку для записи на список свободных записей, мы вызываем функцию `writew_lock`. Это предотвратит возможность взаимовлияния различных процессов при одновременном удалении записей из различных цепочек. Так как удаление записи сопряжено с изменением списка свободных записей, в каждый момент времени только один процесс должен делать это.

Заполненная пробелами запись с данными записывается функцией `_db_writedat`. Обратите внимание: в этом случае нам не нужно устанавливать блокировку на файл с данными. Так как `db_delete` установила блокировку для записи на цепочку из хеш-таблицы, мы точно знаем, что никакой другой процесс не сможет прочитать или изменить запись с данными.

```
441    /*
442     * Прочитать указатель на первую запись в списке свободных записей.
443     * На его место будет записан указатель на удаляемую запись.
444     * Это означает, что удаляемая запись вставляется в начало списка.
445     */
```

```
446     freeptr = _db_readptr(db, FREE_OFF);
447     /*
448      * Сохранить указатель на запись, следующую за удаляемой,
449      * прежде чем он будет затерт функцией_db_writeidx.
450      */
451     saveptr = db->ptrval;
452     /*
453      * Переписать индексную запись. В результате также будут переписаны
454      * значения длины индексной записи, позиции и длины записи с данными,
455      * ни одно из которых не было изменено, но так и должно быть.
456      */
457     _db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);
458     /*
459      * Записать новый указатель на начало списка свободных записей.
460      */
461     _db_writeptr(db, FREE_OFF, db->idxoff);
462     /*
463      * Изменить указатель, который указывает на удаляемую запись.
464      * Мы уже упоминали, что _db_find_and_lock записывает в db->ptroff
465      * адрес этого указателя. Мы запишем в этот указатель адрес записи,
466      * которая следует за удаляемой, то есть saveptr.
467      */
468     _db_writeptr(db, db->ptroff, saveptr);
469     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
470         err_dump("_db_dodelete: ошибка вызова функции un_lock");
471 }
```

[441–461] Мы читаем указатель на первую запись в списке свободных записей и затем обновляем удаляемую индексную запись таким образом, чтобы она указывала на первую запись в списке свободных записей. (Если до этого список был пуст, указатель на следующую запись будет содержать 0.) Ключ у нас уже забит пробелами. Затем мы обновляем указатель на первую запись в списке свободных записей так, чтобы он указывал на удаляемую запись. Это означает, что список свободных записей обслуживается по принципу «последний пришел, первый ушел», то есть удаляемая запись вставляется в начало списка (хотя удалять записи из этого списка мы будем по принципу «первого подходящего»).

Мы не предусматриваем отдельные списки свободных записей для каждого из файлов. Когда мы добавляем удаляемую индексную запись в список свободных записей, она по-прежнему ссылается на соответствующую удаленную ей запись с данными. Существуют более удобные способы удаления записей, но они требуют усложнения функции.

[462–471] Мы обновляем предыдущую запись в цепочке таким образом, чтобы она указывала на запись, следующую за удаляемой. В результате удаляемая запись исключается из цепочки. В заключение мы снимаем блокировку со списка свободных записей.

```
472 /*
473  * Записать запись с данными. Вызывается из _db_dodelete (чтобы записать
```

```
474     * запись, заполненную пробелами) и из db_store.
475     */
476    static void
477    _db_writedat(DB *db, const char *data, off_t offset, int whence)
478    {
479      struct iovec iov[2];
480      static char newline = NEWLINE;

481      /*
482       * Если мы добавляем запись в конец файла, необходимо предварительно
483       * установить блокировку, чтобы выполнить lseek и write атомарно.
484       * Если перезаписывается существующая запись, блокировка не нужна.
485       */
486      if (whence == SEEK_END) /* добавить в конец, заблокировать весь файл */
487          if (writew_lock(db->datfd, 0, SEEK_SET, 0) < 0)
488              err_dump("_db_writedat: ошибка вызова функции writew_lock");

489      if ((db->datoff = lseek(db->datfd, offset, whence)) == -1)
490          err_dump("_db_writedat: ошибка вызова функции lseek");
491      db->datlen = strlen(data) + 1; /* в datlen включен символ NL */

492      iov[0].iov_base = (char *) data;
493      iov[0].iov_len = db->datlen - 1;
494      iov[1].iov_base = &newline;
495      iov[1].iov_len = 1;
496      if (writev(db->datfd, &iov[0], 2) != db->datlen)
497          err_dump("_db_writedat: ошибка вывода записи с данными");

498      if (whence == SEEK_END)
499          if (un_lock(db->datfd, 0, SEEK_SET, 0) < 0)
500              err_dump("_db_writedat: ошибка вызова функции un_lock");
501    }
```

[472–491] Мы вызываем `_db_writedat`, чтобы вывести в файл запись с данными. Когда удаляется запись, мы с помощью `_db_writedat` перезаписываем запись, заполненную пробелами. При вызове `_db_writedat` не требуется устанавливать блокировку на файл с данными, поскольку `db_delete` уже установила блокировку для записи на цепочку хешей, в которой находится эта запись. Таким образом, никакой другой процесс не сможет ни прочитать, ни перезаписать эту запись. Когда далее в этом разделе мы перейдем к обсуждению функции `db_store`, то столкнемся с ситуацией, когда `_db_writedat` производит добавление записи в конец файла и должна установить блокировку на весь файл.

Мы перемещаемся в позицию, куда необходимо выполнить запись. Объем записываемых данных равен размеру записи плюс 1 байт для завершающего нулевого символа.

[492–501] Мы заполняем поля структуры `iovec` и вызываем `writev`, чтобы записать данные и символ перевода строки. Мы не можем полагаться на то, что в буфере, полученном от вызывающей функции, достаточно места для того, чтобы добавить символ перевода строки, поэтому мы берем его из отдельного буфера. После вывода записи в файл мы снимаем блокировку, которую установили ранее.

```
502  /*
503   * Записать индексную запись. Перед этой функцией вызывается _db_writedat,
504   * которая устанавливает значения полей datoff и datlen в структуре DB,
505   * необходимые для создания индексной записи.
506   */
507  static void
508  _db_writeidx(DB *db, const char *key,
509               off_t offset, int whence, off_t ptrval)
510  {
511    struct iovec iov[2];
512    char asciiptrlen[PTR_SZ + IDXLEN_SZ +1];
513    int len;
514    char *fmt;

515    if ((db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
516        err_quit("_db_writeidx: неверный указатель: %d", ptrval);
517    if (sizeof(off_t) == sizeof(long long))
518        fmt = "%s%c%lld%c%d\n";
519    else
520        fmt = "%s%c%ld%c%d\n";
521    sprintf(db->idxbuf, fmt, key, SEP, db->datoff, SEP, db->datlen);
522    if ((len = strlen(db->idxbuf)) < IDXLEN_MIN || len > IDXLEN_MAX)
523        err_dump("_db_writeidx: неверная длина ");
524    sprintf(asciiptrlen, "%*ld%*d", PTR_SZ, ptrval, IDXLEN_SZ, len);

525    /*
526     * Если запись добавляется в конец файла, необходимо предварительно
527     * установить блокировку, чтобы выполнить lseek и write атомарно.
528     * Если перезаписывается существующая запись, блокировка не нужна.
529     */
530    if (whence == SEEK_END) /* добавление в конец файла */
531        if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
532          SEEK_SET, 0) < 0)
533            err_dump("_db_writeidx: ошибка вызова функции writew_lock");
```

[502–524] **Функция** `_db_writeidx` **вызывается для того, чтобы вывести в файл индексную запись. После проверки указателя на следующую запись в цепочке мы создаем индексную запись и сохраняем ее вторую половину в буфере** `idxbuf`**. Нам потребуется размер этой части, чтобы создать первую половину индексной записи, которую мы сохраняем в переменной** `asciiptrlen`**.**

Обратите внимание: строка формата, передаваемая функции `sprintf`**, выбирается исходя из размера типа данных** `off_t`**. Даже 32-битные системы могут предоставлять 64-битные значения смещения в файле, поэтому мы не можем делать какие-либо предположения о размере типа** `off_t`**.**

[525–533] **Как и** `_db_writedat`**, эта функция устанавливает блокировку только в том случае, если новая индексная запись добавляется в конец индексного файла. Когда эта функция вызывается из** `_db_dodelete`**, мы перезаписываем существующую индексную запись. В этой ситуации вызывающая функция устанавливает блокировку для записи на цепочку из хеш-таблицы, и поэтому установка дополнительной блокировки не требуется.**

```
534     /*
535      * Позиция в индексном файле и смещение записи.
536      */
537     if ((db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
538         err_dump("_db_writeidx: ошибка вызова функции lseek");
539     iov[0].iov_base = asciiptrlen;
540     iov[0].iov_len = PTR_SZ + IDXLEN_SZ;
541     iov[1].iov_base = db->idxbuf;
542     iov[1].iov_len = len;
543     if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
544         err_dump("_db_writeidx: ошибка вывода в файл индексной записи");
545     if (whence == SEEK_END)
546         if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
547           SEEK_SET, 0) < 0)
548             err_dump("_db_writeidx: ошибка вызова функции un_lock");
549   }
550   /*
551    * Записать значение указателя куда-либо в индексный файл:
552    * в список свободных записей, хеш-таблицу или индексную запись.
553    */
554   static void
555   _db_writeptr(DB *db, off_t offset, off_t ptrval)
556   {
557     char asciiptr[PTR_SZ + 1];
558     if (ptrval < 0 || ptrval > PTR_MAX)
559         err_quit("_db_writeptr: неверный указатель: %d", ptrval);
560     sprintf(asciiptr, "%*ld", PTR_SZ, ptrval);
561     if (lseek(db->idxfd, offset, SEEK_SET) == -1)
562         err_dump("_db_writeptr: ошибка перемещения на поле с указателем");
563     if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
564         err_dump("_db_writeptr: ошибка записи в поле с указателем");
565   }
```

[534–549] **Мы перемещаемся в позицию, куда должна быть записана индексная запись, и сохраняем это смещение в поле `idxoff` структуры `DB`. Поскольку индексная запись собрана в двух отдельных буферах, для ее сохранения в индексном файле мы используем функцию `writev`. Если производилось добавление новой записи в конец файла, мы снимаем блокировку, которую установили перед изменением текущей позиции файла. Это позволяет производить операции изменения текущей позиции файла и записи атомарно для процессов, работающих параллельно и добавляющих новые записи в ту же самую базу данных.**

[550–565] **Функция `_db_writeptr` используется для записи в индексный файл указателя на очередную запись. Этот указатель проверяется на превышение допустимых пределов и преобразуется в строку символов ASCII. Мы переходим в заданную позицию в индексном файле и записываем указатель.**

```
566   /*
567    * Сохранить запись в базе данных. Вернуть 0 в случае успеха; 1, если
```

```
568     * запись существует и установлен флаг DB_INSERT; -1 в случае ошибки.
569     */
570    int
571    db_store(DBHANDLE h, const char *key, const char *data, int flag)
572    {
573      DB *db = h;
574      int rc, keylen, datlen;
575      off_t ptrval;
576      if (flag != DB_INSERT && flag != DB_REPLACE &&
577        flag != DB_STORE) {
578          errno = EINVAL;
579          return(-1);
580      }
581      keylen = strlen(key);
582      datlen = strlen(data) + 1; /* +1 для символа перевода строки */
583      if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
584          err_dump("db_store: неверная длина записи");
585      /*
586       * _db_find_and_lock вычисляет, в какую хеш-таблицу должна
587       * быть добавлена новая запись (db->chainoff), независимо от того,
588       * существует она или нет. Следующий вызов _db_writeptr изменит
589       * запись в хеш-таблице, записав в нее указатель на новую запись.
590       * Новая запись вставляется в начало цепочки.
591       */
592      if (_db_find_and_lock(db, key, 1) < 0) { /* запись не найдена */
593          if (flag == DB_REPLACE) {
594              rc = -1;
595              db->cnt_storerr++;
596              errno = ENOENT; /* ошибка, запись не найдена */
597              goto doreturn;
598          }
```

[566–584] Функция `db_store` используется для добавления новых записей в базу данных. Прежде всего мы проверяем значения флагов, которые были переданы функции. Затем мы проверяем длину записи. Если размер записи выходит за допустимые пределы, мы создаем файл `core` и завершаем работу процесса. Такое поведение допустимо для библиотеки-примера, но если мы собираемся создавать библиотеку для использования в реальных приложениях, необходимо вместо завершения процесса возвращать признак ошибки, чтобы позволить приложению исправить ее.

[585–598] Мы вызываем `_db_find_and_lock`, чтобы убедиться в существовании записи. Ситуации, когда запись не найдена и установлен флаг `DB_INSERT` или `DB_STORE`, или когда запись существует и установлен флаг `DB_REPLACE` или `DB_STORE`, не считаются ошибочными. Если мы замещаем существующую запись, это означает, что ключи записей идентичны, но сами данные могут отличаться. Обратите внимание, что последний аргумент функции `_db_find_and_lock` указывает на то, что на цепочку в хеш-таблице должна быть установлена блокировка для записи, поскольку она, скорее всего, будет подвергнута изменениям.

```
599         /*
600          * _db_find_and_lock уже заблокировала цепочку в хеш-таблице;
601          * прочитать указатель на первую индексную запись в цепочке.
602          */
603         ptrval = _db_readptr(db, db->chainoff);

604         if (_db_findfree(db, keylen, datlen) < 0) {
605             /*
606              * Не найдена пустая запись достаточного размера. Добавить
607              * новые записи в конец индексного файла и файла с данными.
608              */
609             _db_writedat(db, data, 0, SEEK_END);
610             _db_writeidx(db, key, 0, SEEK_END, ptrval);

611             /*
612              * Значение db->idxoff было установлено в _db_writeidx.
613              * Новая запись добавляется в начало цепочки хеш-таблицы.
614              */
615             _db_writeptr(db, db->chainoff, db->idxoff);
616             db->cnt_stor1++;
617         } else {
618             /*
619              * Использовать повторно пустую запись. _db_findfree удалит
620              * ее из списка свободных записей и установит значения полей
621              * db->datoff и db->idxoff. Запись добавляется в начало списка.
622              */
623             _db_writedat(db, data, db->datoff, SEEK_SET);
624             _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
625             _db_writeptr(db, db->chainoff, db->idxoff);
626             db->cnt_stor2++;
627         }
```

[599–603] После вызова `_db_find_and_lock` возможны четыре сценария дальнейшего развития событий. В первых двух, когда запись не была найдена, необходимо добавить новую запись. Мы читаем указатель на первую запись в цепочке хеш-таблицы.

[604–616] Случай 1: с помощью функции `_db_findfree` мы пытаемся отыскать в списке свободных записей ранее удаленную запись с тем же размером ключа и объемом данных. Если таковая не найдена, мы добавляем новые записи в конец индексного файла и файла с данными. Для записи данных вызывается функция `_db_writedat`, для записи индекса – функция `_db_writeidx`, а для вставки новой индексной записи в начало цепочки хеш-таблицы – функция `_db_writeptr`. Затем мы увеличиваем счетчик (`cnt_stor1`) ситуаций, развивающихся по этому сценарию, что позволит нам в дальнейшем проанализировать поведение базы данных.

[617–627] Случай 2: функция `_db_findfree` нашла пустую запись требуемого размера и исключила ее из списка свободных записей (вскоре мы рассмотрим реализацию функции `_db_findfree`). Мы записываем индексную запись и запись с данными и добавляем адрес записи в начало цепочки хеш-таблицы, как и в первом случае. Затем мы увеличиваем счетчик (`cnt_stor2`) ситуаций, развивающихся по данному сценарию.

```
628         } else { /* запись найдена */
629             if (flag == DB_INSERT) {
630                 rc = 1;     /* ошибка, запись уже имеется в базе данных */
631                 db->cnt_storerr++;
632                 goto doreturn;
633             }
634             /*
635              * Производится замена существующей записи. Мы знаем, что
636              * новый ключ равен существующему, но нам нужно проверить
637              * равенство размеров записей с данными.
638              */
639             if (datlen != db->datlen) {
640 _                db_dodelete(db); /* удалить существующую запись */
641                 /*
642                  * Перечитать указатель из хеш-таблицы
643                  * (он мог измениться в процессе удаления).
644                  */
645                 ptrval = _db_readptr(db, db->chainoff);
646                 /*
647                  * Добавить новые записи в конец файлов.
648                  */
649                 _db_writedat(db, data, 0, SEEK_END);
650                 _db_writeidx(db, key, 0, SEEK_END, ptrval);
651                 /*
652                  * Вставить указатель на запись в начало цепочки.
653                  */
654                 _db_writeptr(db, db->chainoff, db->idxoff);
655                 db->cnt_stor3++;
656             } else {
```

[628–633] Теперь мы перешли к двум возможным ситуациям, когда запись с тем же самым ключом уже существует в базе данных. Если вызывающий процесс не указал, что запись должна быть замещена, мы записываем в возвращаемое значение код, который свидетельствует о том, что запись уже существует, увеличиваем счетчик ошибок операций записи и переходим в конец функции, где реализован алгоритм выхода.

[634–656] Случай 3: существующая запись должна быть замещена, но длина записи с данными отличается от длины существующей записи с данными. Мы вызываем функцию `_do_delete`, которая удалит существующую запись. Как вы помните, при этом она вставит удаленную запись в начало списка свободных записей. Затем мы добавляем новые записи в конец индексного файла и в конец файла с данными с помощью функций `_db_writeidx` и `_db_writedat`. (Существуют и другие способы обработки этой ситуации. Можно, например, попытаться отыскать свободную запись подходящего размера.) Новая запись добавляется в начало цепочки хеш-таблицы вызовом функции `_db_writeptr`. В счетчик `cnt_stor3` структуры `DB` записывается количество ситуаций, развивающихся по этому сценарию.

```
657             /*
658              * Размеры данных совпадают, просто заменить запись.
```

```
659              */
660             _db_writedat(db, data, db->datoff, SEEK_SET);
661             db->cnt_stor4++;
662         }
663     }
664     rc = 0; /* OK */
665   doreturn:      /* снять блокировку, установленную в _db_find_and_lock */
666     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
667         err_dump("db_store: ошибка вызова функции un_lock");
668     return(rc);
669   }
670   /*
671    * Попытаться отыскать свободную индексную запись с данными
672    * нужного размера. Эта функция вызывается только из db_store.
673    */
674   static int
675   _db_findfree(DB *db, int keylen, int datlen)
676   {
677     int rc;
678     off_t offset, nextoffset, saveoffset;
679     /*
680      * Заблокировать указатель на список свободных записей.
681      */
682     if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
683         err_dump("_db_findfree: ошибка вызова функции writew_lock");
684     /*
685      * Прочитать указатель на первую запись в списке.
686      */
687     saveoffset = FREE_OFF;
688     offset = _db_readptr(db, saveoffset);
```

[657–663] **Случай 4: существующая запись должна быть замещена, и размер новой записи с данными совпадает с размером существующей записи с данными. Это самый простой случай – нужно лишь записать новые данные в файл и увеличить счетчик (**`cnt_stor4`**) аналогичных ситуаций.**

[664–669] **Если все в порядке, мы записываем в возвращаемое значение признак успешного завершения и переходим к выполнению алгоритма выхода. Здесь мы снимаем с цепочки в хеш-таблице блокировку, установленную функцией** `_db_find_and_lock`**, и возвращаем управление вызывающему процессу.**

[670–688] **Функция** `_db_findfree` **пытается отыскать свободную индексную запись и связанную с ней запись с данными заданных размеров. Чтобы избежать взаимовлияния с другими процессами, необходимо установить блокировку для записи на список свободных записей. Когда блокировка установлена, мы считываем адрес первой записи в списке.**

```
689     while (offset != 0) {
690         nextoffset = _db_readidx(db, offset);
```

```
691         if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
692             break; /* совпадение найдено */
693         saveoffset = offset;
694         offset = nextoffset;
695     }
696     if (offset == 0) {
697         rc = -1;   /* совпадений не найдено */
698     } else {
699         /*
700          * Найдена запись требуемого размера.
701          * Индексная запись была прочитана ранее в _db_readidx, которая
702          * установила значение db->ptrval. Кроме того, saveoffset
703          * указывает на запись в списке свободных записей, соответствующую
704          * найденной записи. Мы записываем в нее значение db->ptrval,
705          * исключая тем самым найденную запись из списка свободных записей.
706          */
707         _db_writeptr(db, saveoffset, db->ptrval);
708         rc = 0;
709         /*
710          * Обратите внимание: _db_readidx записывает значения в db->idxoff
711          * и в db->datoff. Это обстоятельство используется вызывающей
712          * функцией db_store для вывода новых записей в файлы.
713          */
714     }
715     /*
716      * Снять блокировку со списка свободных записей.
717      */
718     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
719         err_dump("_db_findfree: ошибка вызова функции un_lock");
720     return(rc);
721 }
```

[689–695] **В цикле `while` производится обход списка свободных записей в поисках записи с соответствующими размерами ключа и данных. В этой простой реализации мы повторно используем удаленные записи только в том случае, если размеры ключа и данных совпадают с размерами ключа и данных вставляемой записи. Существуют более эффективные алгоритмы использования свободного пространства, но они требуют усложнения реализации.**

[696–714] **Если запись с требуемыми размерами ключа и данных не была найдена, мы записываем в возвращаемое значение код, который свидетельствует о неудаче. В противном случае мы записываем в указатель предыдущей записи адрес записи, следующей за найденной. Таким способом мы исключаем найденную запись из списка свободных записей.**

[715–721] **По окончании операций со списком свободных записей мы снимаем блокировку и возвращаем код завершения операции вызывающей функции.**

```
722 /*
723  * Переход к первой записи для функции db_nextrec.
```

```
724    * Автоматически вызывается из db_open.
725    * Должна вызываться перед первым обращением к db_nextrec.
726    */
727   void
728   db_rewind(DBHANDLE h)
729   {
730     DB *db = h;
731     off_t offset;
732     offset = (db->nhash + 1) * PTR_SZ; /* +1 для списка свободных записей */
733     /*
734      * Мы просто устанавливаем текущую позицию файла для данного
735      * процесса на первую индексную запись - блокировка не требуется.
736      * +1, чтобы перешагнуть символ перевода строки в конце хеш-таблицы.
737      */
738     if ((db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
739         err_dump("db_rewind: ошибка вызова функции lseek");
740   }
741   /*
742    * Вернуть следующую запись.
743    * Мы просто двигаемся по индексному файлу, игнорируя удаленные записи.
744    * Перед первым обращением к этой функции должна быть вызвана
745    * функция db_rewind.
746    */
747   char *
748   db_nextrec(DBHANDLE h, char *key)
749   {
750     DB *db = h;
751     char c;
752     char *ptr;
```

[722–740] Функция `db_rewind` используется для перехода к «началу» базы данных – она устанавливает текущую позицию индексного файла на начало первой записи (которая находится сразу же за хеш-таблицей). (Вспомните структуру индексного файла, которая была показана на рис. 20.1.)

[741–752] Функция `db_nextrec` возвращает следующую запись из базы данных. Вызывающему процессу возвращается указатель на буфер с данными. Если в аргументе `key` передается непустой указатель, то по заданному адресу будет возвращен ключ, который соответствует записи с данными. Вся ответственность за выделение буфера достаточного размера для хранения ключа возлагается на вызывающий процесс. Буфер с размером `IDXLEN_MAX` сможет вместить в себя любой ключ.

Записи возвращаются в порядке, в котором они были записаны в базу данных. Таким образом, записи не сортируются по ключу. Кроме того, поскольку мы не принимаем во внимание цепочки из хеш-таблицы, в процессе обхода базы данных могут обнаружиться удаленные записи, но они не должны возвращаться вызывающему процессу.

```
753     /*
754      * На список свободных записей устанавливается блокировка для чтения,
```

```
755      * чтобы в процессе чтения невозможно было удалить запись.
756      */
757     if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
758         err_dump("db_nextrec: ошибка вызова функции readw_lock");
759     do {
760         /*
761          * Прочитать очередную запись.
762          */
763         if (_db_readidx(db, 0) < 0) {
764             ptr = NULL; /* конец индексного файла */
765             goto doreturn;
766         }
767         /*
768          * Проверить, не заполнен ли ключ пробелами (пустая запись).
769          */
770         ptr = db->idxbuf;
771         while ((c = *ptr++) != 0 && c == SPACE)
772             ;          /* перейти к первому символу, отличному от пробела */
773     } while (c == 0); /* повторять, пока не встретится непустой ключ */
774     if (key != NULL)
775         strcpy(key, db->idxbuf); /* вернуть ключ */
776     ptr = _db_readdat(db);       /* вернуть указатель на буфер */
777     db->cnt_nextrec++;
778   doreturn:
779     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
780         err_dump("db_nextrec: ошибка вызова функции un_lock");
781     return(ptr);
782   }
```

[753–758] Прежде всего необходимо установить блокировку для чтения, чтобы никакой другой процесс не смог удалить запись во время ее чтения.

[759–773] Для чтения очередной записи вызывается `_db_readidx`. Мы передаем ей в качестве смещения значение 0, чтобы указать, что чтение должно производиться с текущей позиции. Поскольку мы последовательно считываем все записи из индексного файла, мы можем обнаружить удаленные записи. Но так как должны возвращаться только нормальные записи, необходимо пропускать записи, ключи которых заполнены пробелами (функция `_db_dodelete` очищает строку ключа, заполняя ее пробелами).

[774–782] Встретив нормальный ключ, мы копируем его в буфер вызывающего процесса, если он был предоставлен. Затем мы считываем запись с данными и записываем в возвращаемое значение указатель на внутренний буфер, содержащий запись с данными. После этого мы увеличиваем счетчик обращений к функции `db_nextrec`, снимаем блокировку списка свободных записей и возвращаем указатель на буфер с данными.

Как правило, функции `db_rewind` и `db_nextrec` используются в цикле, например

```
db_rewind(db);
while ((ptr = db_nextrec(db, key)) != NULL) {
```

```
        /* обработка полученной записи */
    }
```

Как мы уже предупреждали ранее, записи возвращаются не по порядку – они не сортируются по значению ключа.

Если в процессе извлечения записей в цикле с помощью функции db_nextrec база данных будет изменяться, записи, возвращаемые db_nextrec, будут представлять собой просто срезы изменяющейся базы данных в некоторый момент времени. Функция db_nextrec всегда возвращает запись, которая была действительна на момент вызова функции, то есть она не возвращает записи, которые были удалены. Но вполне возможно, что запись будет удалена сразу же после возврата из функции db_nextrec. Аналогично, если пустая запись была заполнена уже после того, как db_nextrec перешагнула через нее, мы не сможем увидеть новую запись, пока не вернемся к началу базы данных и не повторим цикл обхода. Если важно получить точный «замороженный» срез базы данных с помощью db_nextrec, в системе не должно быть процессов, которые могли бы вставить новые или удалить существующие записи во время получения среза.

Взгляните на то, как db_nextrec использует механизм блокировок. Мы не учитываем цепочки в хеш-таблице и не можем определить, какой цепочке принадлежит та или иная запись. Таким образом, вполне возможна ситуация, когда индексная запись будет находиться в процессе удаления, в то время как db_nextrec читает ее. Чтобы предотвратить это, db_nextrec устанавливает блокировку для чтения на список свободных записей, благодаря чему исключается возможность взаимовлияния с функциями _db_dodelete и _db_findfree.

Прежде чем завершить исследование файла db.c, мы должны описать принцип действия блокировки, которая устанавливается при добавлении новых записей в конец файлов. В случаях 1 и 3 функция db_store вызывает _db_writeidx и _db_writedat, передавая им в третьем аргументе значение 0, а в четвертом – SEEK_END. Этот четвертый аргумент является признаком того, что новая запись добавляется в конец файла. Функция _db_writeidx устанавливает блокировку для записи от конца цепочки хеш-таблицы до конца файла. Такой прием не повлияет на другие читающие или пишущие в базу данных процессы (так как они будут устанавливать блокировку на цепочку хеш-таблицы) и при этом не даст возможности другим процессам в то же самое время добавлять записи в конец файла. Функция _db_writedat устанавливает блокировку для записи на весь файл с данными. Опять же это не повлияет на другие читающие или пишущие в базу процессы (так как они даже не будут пытаться установить блокировку на файл с данными) и в то же время не даст возможности другим процессам добавлять записи в конец файла (упражнение 20.3).

20.9. Производительность

Чтобы протестировать библиотеку базы данных и получить некоторые временные характеристики производительности, была написана тестовая программа. Эта программа принимает два аргумента командной строки: коли-

чество создаваемых дочерних процессов и количество записей (*nrec*), которые каждый из процессов должен записать в базу данных. Программа создает пустую базу данных (вызовом функции db_open), порождает заданное число дочерних процессов и ожидает их завершения. Каждый из дочерних процессов выполняет следующие действия.

1. Записывает *nrec* записей в базу данных.
2. Считывает *nrec* записей по заданному ключу.
3. Выполняет следующий цикл *nrec* × 5 раз.
   a. Считывает случайную запись.
   b. Через каждые 37 циклов удаляет случайную запись.
   c. Через каждые 11 циклов вставляет новую запись и считывает ее обратно.
   d. Через каждые 17 циклов замещает случайную запись новой записью. Новая запись имеет либо тот же размер строки с данными, либо больший – через раз.
4. Удаляет все созданные им записи. Каждый раз при удалении записи выполняется поиск десяти случайных записей.

Количество операций, произведенных над базой данных, сохраняется в счетчиках cnt_xxx структуры DB. Количество операций, выполняемых каждым из процессов, различно, поскольку для выборки записей используется генератор случайных чисел, инициализированный идентификатором дочернего процесса. Типичные значения счетчиков операций, производимых каждым дочерним процессом при *nrec*, равном 500, приводятся в табл. 20.2.

Таблица 20.2. Типичные значения счетчиков операций, выполняемых каждым из процессов при nrec = 500

Операция	Количество
db_store, DB_INSERT, подходящая пустая запись не найдена, добавление в конец файла	678
db_store, DB_INSERT, используется пустая запись	164
db_store, DB_REPLACE, новая запись имеет другой размер, добавление в конец файла	97
db_store, DB_REPLACE, новая запись имеет тот же размер, добавление в конец файла	109
db_store, запись не найдена	19
db_fetch, запись найдена	8114
db_fetch, запись не найдена	732
db_delete, запись найдена	842
db_delete, запись не найдена	110

Количество операций по извлечению записей превышает количество операций по удалению или добавлению новых записей примерно в десять раз, что типично для большинства приложений баз данных.

Каждый дочерний процесс выполняет все операции (извлечение, удаление и сохранение) только над теми записями, которые были записаны самим дочерним процессом. В процессе тестирования активно использовались средства управления одновременным доступом, поскольку все дочерние процессы работали с одной и той же базой данных (хотя и с разными записями). Общее количество записей в базе данных возрастает пропорционально количеству дочерних процессов. (Один дочерний процесс изначально записывает в базу данных *nrec* записей, два дочерних процесса – *nrec* × 2 записей и т. д.)

Чтобы получить и сравнить временные характеристики при использовании крупноблочной и мелкоблочной блокировок, а также выполнить сравнение трех типов блокировок (отсутствие блокировок, рекомендательные блокировки, принудительные блокировки), мы запускали три версии программы. Первая версия программы (исходный код которой приведен в разделе 20.8) использует мелкоблочную блокировку. Вторая версия программы использует крупноблочную блокировку, как это было описано в разделе 20.6. Из третьей версии были удалены все функции установки блокировок, что дало возможность определить накладные расходы на использование механизма блокировок. Первая и вторая версия программы (мелкоблочные блокировки и крупноблочные блокировки) могли использовать как рекомендательные, так и принудительные блокировки, для этого достаточно было изменить права доступа к файлам базы данных. (Во всех отчетах, приводимых в данном разделе, при использовании принудительных блокировок измерения производились только для версии с мелкоблочными блокировками.)

Результаты для единственного процесса

В табл. 20.3 приводятся результаты измерения временных характеристик для случая, когда работал один процесс с *nrec*, равным 500, 1000 и 2000.

Таблица 20.3. Один процесс, различные значения nrec, различные типы блокировок

nrec	Нет блокировок			Рекомендательные блокировки						Принудительные блокировки		
				Крупноблочные			Мелкоблочные			Мелкоблочные		
	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время
500	0,42	0,89	1,31	0,42	1,17	1,59	0,41	1,04	1,45	0,46	1,49	1,95
1000	1,51	3,89	5,41	1,64	4,13	5,78	1,63	4,12	5,76	1,73	6,34	8,07
2000	3,91	10,06	13,98	4,09	10,30	14,39	4,03	10,63	14,66	4,47	16,21	20,70

В табл. 20.3 приводятся результаты измерения в секундах. Во всех случаях сумма пользовательского и системного времени выполнения примерно равна общему времени. Это говорит о том, что в основном использовалась производительность центрального процессора, а не дисковой подсистемы.

В шести колонках, которые соответствуют рекомендательным блокировкам, значения времени практически одинаковы в каждой из строк. Это говорит о том, что в случае единственного процесса отсутствуют различия между крупноблочными и мелкоблочными блокировками.

Сравнение времени работы при использовании рекомендательных блокировок со временем работы версии, в которой блокировки вообще не использовались, показывает, что использование механизма блокировок добавляет от 2 до 31 процента к системному времени работы. Даже несмотря на то, что механизм блокировок фактически не использовался (поскольку работал только один процесс), обращения к системному вызову `fcntl` заняли определенное время. Обратите внимание: пользовательское время работы для всех четырех случаев практически одинаково. Это объясняется тем, что код, работающий в пространстве пользователя, практически не изменялся (за исключением нескольких вызовов функции `fcntl`).

И последнее замечание к результатам из табл. 20.3: использование принудительных блокировок увеличило системное время работы на 43–54 процента по сравнению с результатами, полученными при использовании рекомендательных блокировок. Так как количество наложений блокировок для версий с мелкоблочными принудительными и мелкоблочными рекомендательными блокировками одно и то же, можно утверждать, что дополнительные накладные расходы связаны с операциями чтения и записи.

В заключительном тесте была предпринята попытка запустить несколько дочерних процессов для версии, которая не использует механизм блокировок. Как и следовало ожидать, в результате мы получали случайные ошибки. Как правило, процессы не могли найти записи, которые были добавлены в базу данных, что приводило к аварийному завершению. Каждый раз при запуске программы мы получали разные типы ошибок. Это пример классического состояния гонки за ресурсами: множество процессов обновляют один и тот же файл, не используя никаких блокировок.

Результаты для нескольких процессов

Следующие ниже результаты демонстрируют главным образом различия между крупноблочными и мелкоблочными блокировками. Как уже говорилось, интуитивно мы ожидали, что мелкоблочные блокировки обеспечат дополнительную производительность, так как в этом случае блокируются небольшие участки базы данных. В табл. 20.4 приводятся результаты для *nrec* = 500 и количества дочерних процессов от 1 до 12.

Все результаты приводятся в секундах и представляют суммарное время для всех дочерних и родительского процессов. Полученные результаты позволяют сделать ряд выводов.

Таблица 20.4. Сравнение различных типов блокировок для nrec = 500

Количество процессов	Рекомендательные блокировки							Принудительные блокировки			
	Крупноблочные			Мелкоблочные			Δ	Мелкоблочные			Δ
	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время	Общее время	Пользовательское время	Системное время	Общее время	%
1	0,41	1,00	1,42	0,41	1,05	1,47	0,05	0,47	1,40	1,87	33
2	1,10	2,81	3,92	1,11	2,80	3,92	0,00	1,15	4,06	5,22	45
3	2,17	5,27	7,44	2,19	5,18	7,37	-0,07	2,31	7,67	9,99	48
4	3,36	8,55	11,91	3,26	8,67	11,94	0,03	3,51	12,69	16,20	46
5	4,72	13,08	17,80	4,99	12,64	17,64	-0,16	4,91	19,21	24,14	52
6	6,45	17,96	24,42	6,83	17,29	24,14	-0,28	7,03	26,59	33,66	54
7	8,46	23,12	31,62	8,67	22,96	31,65	0,03	9,25	35,47	44,74	54
8	10,83	29,68	40,55	11,00	29,39	40,41	-0,14	11,67	45,90	57,63	56
9	13,35	36,81	50,23	13,43	36,28	49,76	-0,47	14,45	58,02	72,49	60
10	16,35	45,28	61,66	16,09	44,10	60,23	-1,43	17,43	70,90	88,37	61
11	18,97	54,24	73,24	19,13	51,70	70,87	-2,37	20,62	84,98	105,69	64
12	22,92	63,54	86,51	22,94	61,28	84,29	-2,22	24,41	101,68	126,20	66

Восьмая колонка, отмеченная как «Δ, общее время», представляет различия в секундах между значениями общего времени при использовании рекомендательных крупноблочных и мелкоблочных блокировок. Это значение демонстрирует прирост производительности, который достигается при переходе от крупноблочных к мелкоблочным блокировкам. В системе, на которой проводились испытания, прирост производительности практически отсутствует, пока количество одновременно работающих процессов не превысит семь. Но даже при количестве одновременно работающих процессов, большем семи, прирост производительности, достигнутый в результате использования мелкоблочных блокировок, не так велик (меньше 3%), что заставляет задуматься, стоят ли дополнительные усилия, приложенные нами для реализации мелкоблочных блокировок, такого прироста производительности.

Мы предполагали, что при переходе от крупноблочных блокировок к мелкоблочным общее время выполнения будет уменьшаться (что в конечном итоге и происходит), но системное время выполнения при использовании мелкоблочных блокировок должно увеличиться независимо от количества одновременно работающих процессов. Причина этих ожиданий заключается в том, что при использовании мелкоблочных блокировок требуется больше обращений к функции `fcntl`, чем при использовании крупноблочных блокировок. Используя цифры из табл. 20.2, можно подсчитать, что в случае крупноблочных блокировок требуется в среднем 21 730 вызовов `fcntl`, а в случае мелко-

блочных блокировок – 25 292 вызова. (Чтобы получить эти числа, вспомните, что каждая операция из табл. 20.2 требует два обращения к `fcntl` в случае крупноблочных блокировок и что каждый из первых трех вызовов `db_store`, связанных с удалением записи (когда запись найдена), требует четыре обращения к функции `fcntl` в случае мелкоблочных блокировок.) Мы ожидали, что в случае мелкоблочных блокировок увеличение количества вызовов `fcntl` на 16% приведет к увеличению системного времени выполнения.

Таким образом, некоторое уменьшение системного времени выполнения при использовании мелкоблочных блокировок, когда количество одновременно работающих процессов больше семи, выглядит несколько загадочным.

Причина этого уменьшения кроется в том, что при использовании крупноблочных блокировок мы устанавливаем блокировки на более длительные периоды времени, что увеличивает вероятность простаивания других процессов в ожидании снятия блокировки. При использовании мелкоблочных блокировок они устанавливаются на менее продолжительные периоды времени, поэтому вероятность простаивания на блокировке уменьшается. Если мы проанализируем поведение системы при 12 работающих процессах, то увидим, что при использовании крупноблочных блокировок переключений между процессами производится в три раза больше, чем при использовании мелкоблочных блокировок. Это говорит о том, что при использовании мелкоблочных блокировок процессы блокируются реже.

В последней колонке, которая обозначена как «Δ %», приводится процент увеличения системного времени работы при переходе от рекомендательных мелкоблочных блокировок к принудительным мелкоблочным блокировкам. Эти значения еще раз подтверждают цифры, приводимые в табл. 20.3, которые говорят о существенном увеличении (от 33 до 66%) системного времени работы.

Поскольку код, выполняющийся в пространстве пользователя, практически идентичен для всех версий (если не учитывать некоторое увеличение количества обращений к функции `fcntl` при использовании мелкоблочных блокировок как в рекомендательном, так и в принудительном варианте), мы ожидали, что пользовательское время работы в каждой строке будет примерно одинаковым.

Значения из первой строки табл. 20.4 совпадают со значениями из табл. 20.3 для *nrec* = 500. Это вполне соответствует нашим ожиданиям.

На рис. 20.4. данные из табл. 20.4 для рекомендательных мелкоблочных блокировок представлены в виде графика. Мы построили график зависимости общего времени выполнения от количества процессов (1–12), а также графики, отображающие зависимость отношения пользовательского и системного времени выполнения к количеству процессов.

Обратите внимание: оба графика, которые соответствуют отношениям времени выполнения к количеству процессов, практически линейны, в то время как график общего времени – нелинейный. Вероятно, причина кроется в том, что при увеличении количества процессов операционной системе тре-

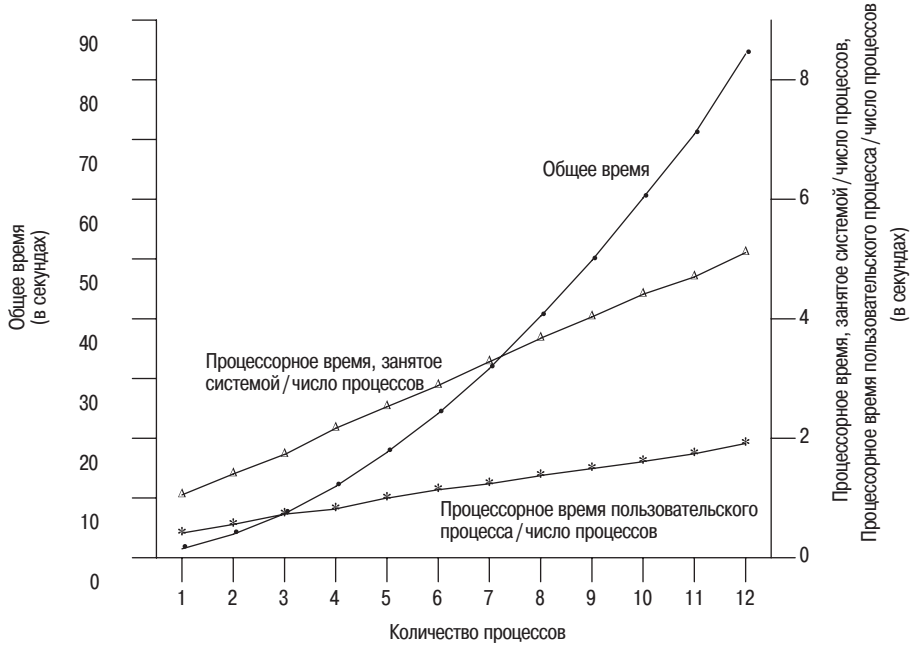


Рис. 20.4. *Значения из табл. 20.4 для рекомендательных мелкоблочных блокировок*

буется больше времени для переключения между ними. Накладные расходы, связанные с работой самой операционной системы, должны были проявиться в виде увеличения общего времени, но не должны сказываться на процессорном времени, затраченном каждым из процессов.

Причина роста пользовательского времени выполнения при увеличении количества процессов связана с увеличением количества записей в базе данных. Каждая цепочка в хеш-таблице становится длиннее, вследствие чего функции `_db_find_and_lock` приходится выполнять больший объем работы при поиске записей.

20.10. Подведение итогов

В этой главе мы детально разобрали архитектуру и реализацию библиотеки базы данных. Для наглядности мы старались сохранить небольшой размер и простоту библиотеки, но при этом она поддерживает механизм блокировок, который позволяет нескольким процессам одновременно работать с базой данных.

Мы также проанализировали производительность этой библиотеки при одновременной работе различного количества процессов в случаях отсутствия блокировок, рекомендательных блокировок (крупноблочных и мелкоблочных) и принудительных блокировок. Мы увидели, что использование рекомендательных блокировок увеличивает общее время работы менее чем на 10% по

сравнению с версией библиотеки, в которой механизм блокировок не используется, а применение принудительных блокировок увеличивает общее время работы на 33–66% по сравнению с версией, использующей рекомендательные блокировки.

Упражнения

20.1. Блокировка в функции `_db_dodelete` выполнена в несколько консервативном стиле. Мы, например, могли бы получить дополнительный прирост производительности при одновременной работе нескольких процессов, если бы устанавливали блокировку для записи на список свободных записей только тогда, когда это действительно необходимо – то есть мы могли бы вставить вызов функции `writew_lock` между вызовами `_db_writedat` и `_db_readptr`. Что произойдет, если мы сделаем это?

20.2. Представьте, что `db_nextrec` не устанавливает блокировку для чтения на список свободных записей и запись, которая была прочитана, одновременно была удалена другим процессом. Опишите, каким образом `db_nextrec` могла бы вернуть корректный ключ и запись с данными, заполненную пробелами (следовательно, неправильную). (Подсказка: загляните в функцию `_db_dodelete`.)

20.3. В конце раздела 20.8 мы описали принцип действия блокировок, устанавливаемых в `_db_writeidx` и `_db_writedat`. Мы утверждали, что эти блокировки не оказывают влияния на другие читающие или пишущие процессы, за исключением вызовов функции `db_store`. Будет ли истинным это утверждение при использовании принудительных блокировок?

20.4. Как бы вы интегрировали функцию `fsync` в эту библиотеку базы данных?

20.5. В функции `db_store` мы сначала записываем данные, а потом индекс. Что произойдет, если запись будет производиться в обратном порядке?

20.6. Создайте новую базу данных и добавьте в нее несколько записей. Напишите программу, которая просматривала бы все записи с помощью `db_nextrec` и вызывала бы `_db_hash`, чтобы вычислить хеш каждой записи. Программа должна выводить гистограмму, отражающую количество записей в каждой из цепочек хеш-таблицы. Ответьте на вопрос, насколько равномерное распределение дает хеш-функция, реализованная в `_db_hash`?

20.7. Измените библиотеку базы данных так, чтобы количество цепочек в хеш-таблице можно было указать в момент создания базы данных.

20.8. Сравните производительность библиотеки базы данных в случаях, когда (а) файлы базы данных находятся в локальной файловой системе и (б) в удаленной файловой системе, доступ к которой организован средствами NFS. Будет ли механизм блокировок работать во втором случае?

21

Взаимодействие с сетевым принтером

21.1. Введение

Сейчас мы разработаем программу, которая будет взаимодействовать с сетевым принтером. Подобные принтеры могут быть связаны сразу с несколькими компьютерами посредством Ethernet и зачастую поддерживают, наряду с простыми текстовыми файлами, печать файлов в формате PostScript. Для взаимодействия с такими принтерами приложения обычно используют протокол IPP (Internet Printing Protocol – протокол печати через Интернет), хотя некоторые принтеры поддерживают альтернативные протоколы.

Мы опишем две программы: демон спулера (диспетчер очереди) печати, который передает задания печати принтеру, и утилиту, с помощью которой задания для печати передаются демону спулера. Поскольку спулер печати выполняет массу разнообразных действий (взаимодействие с клиентом, взаимодействие с принтером, чтение файлов, сканирование каталогов и прочее), это позволит нам использовать функции, которые были описаны в предыдущих главах. Например, для упрощения архитектуры демона мы будем использовать потоки (главы 11 и 12), а для взаимодействия между спулером печати и программой, которая передает ему печатаемый файл, и между спулером печати и сетевым принтером – сокеты (глава 16).

21.2. Протокол печати через Интернет

Протокол печати через Интернет определяет правила построения сетевых систем печати. Благодаря наличию сервера IPP, встроенного в сетевую плату, принтер может обслуживать запросы от множества компьютерных систем. Однако совсем необязательно, чтобы эти компьютерные системы физически находились в той же самой сети, что и принтер. Протокол IPP работает поверх стандартных протоколов Интернета (IP), благодаря чему любой компьютер сможет создать TCP/IP-соединение с принтером и передать ему задание для печати.

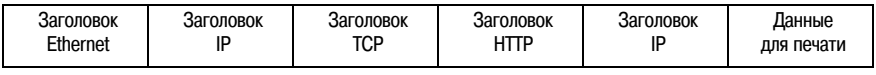


***Рис. 21.1.** Структура сообщения протокола IPP*

Если быть более точным, протокол IPP реализован поверх протокола HTTP (Hypertext Transfer Protocol – протокол передачи гипертекста, раздел 21.3). В свою очередь, протокол HTTP реализован поверх TCP/IP. Структура сообщения протокола IPP показана на рис. 21.1.

Протокол IPP построен по принципу запрос–ответ. Клиент передает сообщение-запрос серверу, а сервер возвращает сообщение-ответ. В заголовке IPP имеется поле, которое определяет запрашиваемую операцию. Возможные операции включают запуск печати задания, отмену печати задания, получение характеристик задания, получение характеристик принтера, приостановка и перезапуск принтера, приостановка печати задания, возобновление печати приостановленного задания.

На рис. 21.2 показана структура заголовка сообщения IPP. Первые 2 байта – это номер версии IPP. Для протокола версии 1.1 в каждом байте хранится число 1. Следующие 2 байта в случае запроса содержат значение, определяющее запрашиваемую операцию. В случае ответа эти 2 байта содержат код статуса.

Следующие 4 байта содержат целочисленный идентификатор запроса. Далее следуют необязательные атрибуты, завершающиеся признаком конца блока атрибутов. Сразу же за блоком атрибутов располагаются данные, которые могут быть связаны с запросом.

Целые числа в заголовке сохраняются со знаком в двоичном формате с обратным (сетевым, big-endian) порядком байтов. Атрибуты хранятся в виде групп. Каждая группа начинается с 1-байтного признака, идентифицирующего группу, за которым следуют 2 байта длины имени атрибута, имя атрибута, 2 байта длины значения атрибута и само значение. Значения атрибутов могут быть представлены в виде строк, целых чисел в двоичном формате или более сложных структур, таких как структуры представления даты и времени.

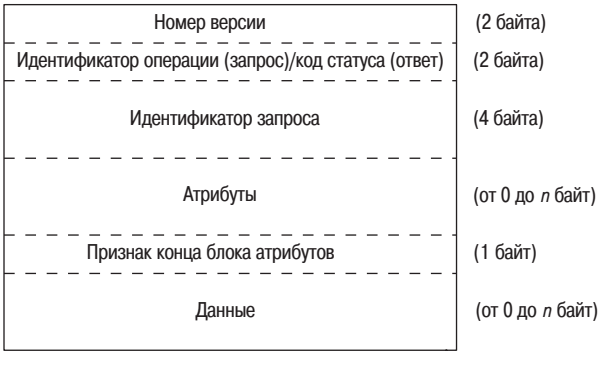


***Рис. 21.2.** Структура заголовка IPP*

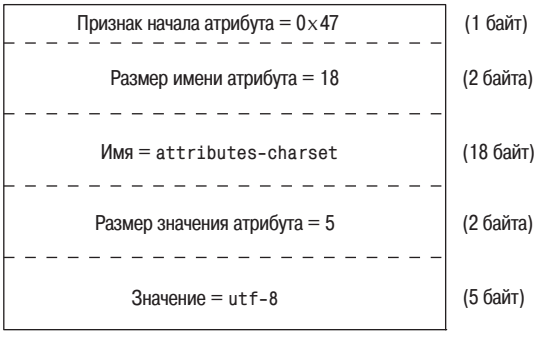


Рис. 21.3. *Пример представления атрибута в заголовке IPP*

На рис. 21.3 показано, как в заголовке IPP будет представлен атрибут `attributes-charset` со значением `utf-8`.

В зависимости от запрашиваемой операции некоторые атрибуты могут быть обязательными, а другие – необязательными. Например, в табл. 21.1 приводятся некоторые атрибуты, сопровождающие запрос на печать задания.

Таблица 21.1. Атрибуты запроса на печать задания

Атрибут	Статус	Описание
`attributes-charset`	Обязательный	Кодировка символов, используемая такими атрибутами, как `type` или `name`
`attributes-natural-language`	Обязательный	Естественный язык, используемый такими атрибутами, как `type` или `name`
`printer-uri`	Обязательный	Универсальный идентификатор ресурса принтера
`requesting-user-name`	Опциональный	Имя пользователя, отправившего задание печати (если поддерживается, используется для аутентификации пользователя)
`job-name`	Опциональный	Имя задания, используемое для идентификации различных заданий
`ipp-attribute-fidelity`	Опциональный	Когда имеет значение «истина», принтер должен отвергнуть задание, если получены не все атрибуты, в противном случае – принтер должен сделать все возможное, чтобы напечатать задание
`document-name`	Опциональный	Название документа (может потребоваться, например, при печати колонтитулов)
`document-format`	Опциональный	Формат документа (обычный текст, PostScript и пр.)
`document-natural-language`	Опциональный	Естественный язык документа

Атрибут	Статус	Описание
compression	Опциональный	Алгоритм сжатия документа
job-k-octets	Опциональный	Размер документа в блоках по 1024 октета
job-impressions	Опциональный	Количество отпечатков (фоновых изображений, встраиваемых в страницу), переданных вместе с заданием
job-media-sheets	Опциональный	Количество листов в задании

Заголовок IPP содержит как текстовые, так и двоичные данные. Имена атрибутов сохраняются в текстовом виде, а их размеры – в виде целых чисел в двоичном представлении. Это усложняет процесс сборки и анализа заголовка, поскольку необходимо постоянно помнить о сетевом порядке байтов и о том, может ли процессор размещать целые числа с произвольного адреса. Было бы лучше, если бы заголовок был разработан так, чтобы все данные в нем хранились только в текстовом представлении. Это упростило бы обработку, хотя и за счет некоторого увеличения размера сообщений.

Протокол IPP определяется целой серией документов (RFC, Requests For Comments – запросы на комментарии), которые вы найдете по адресу *http://www.pwg.org/ipp*. Основные документы перечислены в табл. 21.2, хотя существуют и другие документы, определяющие административные процедуры, атрибуты заданий и тому подобное.

Таблица 21.2. Основные документы RFC, определяющие протокол IPP

RFC	Заголовок
2567	Design Goals for an Internet Printing Protocol – Цели разработки протокола печати через Интернет
2568	Rationale for Structure of the Model and Protocol for the Internet Printing Protocol – Обоснование структурной модели протокола IPP
2911	Internet Printing Protocol/1.1:Model and Semantics – Протокол IPP/1.1:Модель и семантика
2910	Internet Printing Protocol/1.1:Encoding and Transport – Протокол IPP/1.1:Кодировка и передача данных
3196	Internet Printing Protocol/1.1:Implementator's Guide – Протокол IPP/1.1:Руководство разработчика

21.3. Протокол передачи гипертекста

Версия 1.1 протокола HTTP определяется в RFC 2616. Протокол HTTP также работает по принципу запрос-ответ. Сообщение-запрос содержит начальную строку, за которой следуют строки заголовка, пустая строка и необязательное тело запроса. В нашем случае тело запроса содержит заголовок IPP и данные.

Заголовки HTTP передаются в формате ASCII, где каждая строка завершается символами возврата каретки (\r) и перевода строки (\n). Начальная строка содержит метод выполнения запроса, универсальный адрес ресурса (URL – Uniform Resource Locator), который описывает сервер и протокол, и строку, определяющую версию протокола HTTP. Протокол IPP поддерживает только один метод HTTP для передачи данных серверу – это метод POST.

Строки заголовка определяют атрибуты, такие как формат и размер тела запроса. Каждая строка заголовка содержит имя атрибута, далее следуют двоеточие, необязательный пробел и значение атрибута. Завершается строка символами возврата каретки и перевода строки. Например, чтобы указать, что тело содержит сообщение IPP, нужно включить в заголовок строку

```
Content-Type: application/ipp
```

Начальная строка сообщения-ответа HTTP содержит версию протокола, за которой следуют код статуса и сообщение. Завершается начальная строка символами возврата каретки и перевода строки. Остальная часть сообщения-ответа имеет тот же формат, что и сообщение-запрос: строки заголовка, за которыми следуют пустая строка и необязательное тело сообщения.

Ниже приводится пример заголовка HTTP-запроса на печать, отправляемого принтеру автора:

```
POST /phaser860/ipp HTTP/1.1ЂM
Content-Length: 21931ЂM
Content-Type: application/ippЂM
Host: phaser860:ippЂM
ЂM
```

Символы ^M в конце каждой строки соответствуют символам возврата каретки, которые предшествуют символам перевода строки. Перевод строки не отображается как печатный символ. Обратите внимание на то, что последняя строка заголовка пустая – она содержит только символы возврата каретки и перевода строки.

21.4. Очередь печати

Программы, которые мы разработаем в этой главе, представляют собой основу простого спулера (диспетчера очереди) печати. С помощью специальной команды пользователь посылает файл спулеру принтера, спулер сохраняет его на диск, ставит запрос в очередь и в конечном счете отправляет файл принтеру.

Любая версия UNIX предоставляет по меньшей мере одну систему печати. Так, FreeBSD распространяется вместе с системой LPD (Line Printer Daemon – демон последовательной печати) (см. lpd(8) и главу 13 [Stevens 1990]). ОС Linux и Mac OS X включают в себя систему печати CUPS (Common UNIX Printing System – универсальная система печати в UNIX) (см. cupsd(8)). ОС Solaris распространяется со стандартным для System V спулером печати (см. lp(1) и lpsched(1M)). В данной главе основной интерес для нас представляют не са-

ми эти системы печати, а порядок взаимодействия с сетевым принтером. Нам необходимо разработать свою систему печати, которая будет способна организовать доступ нескольких пользователей к единственному ресурсу (принтеру).

Мы создадим простую утилиту, которая будет читать файл и передавать его демону спулера печати. Утилита будет иметь одну опцию – для печати файлов обычного текстового формата (по умолчанию предполагается, что файл имеет формат PostScript). Мы назвали эту утилиту `print`.

Демон спулера печати `printd` будет иметь многопоточную архитектуру, чтобы распределить между потоками работу, которая должна быть выполнена демоном.

- Один поток ожидает поступления через сокет новых запросов от клиентов, запустивших утилиту `print`.
- Для обслуживания каждого клиента порождается отдельный поток, который копирует файл в область очереди печати.
- Один поток взаимодействует с принтером, передавая ему задания из очереди.
- Один поток обслуживает сигналы.

На рис. 21.4 показано, как все эти компоненты связаны друг с другом.

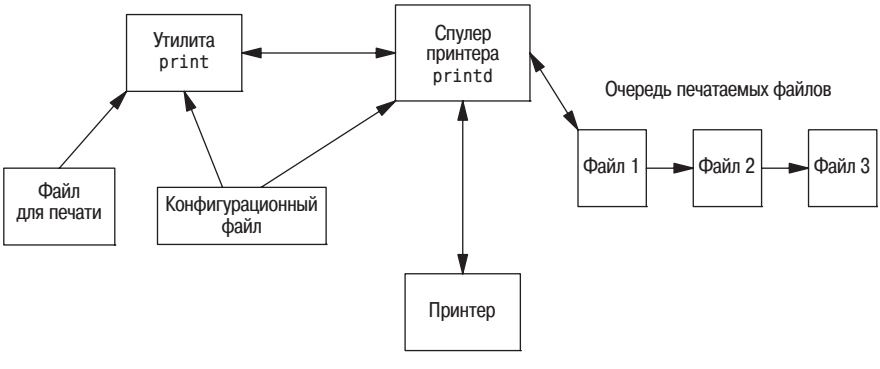


***Рис. 21.4.** Компоненты системы печати*

Конфигурационный файл системы печати называется `/etc/printer.conf`. Он определяет имя сервера, на котором запущен демон спулера печати, и сетевое имя принтера. Демон спулера идентифицируется строкой, начинающейся с ключевого слова `printserver`, за которым следуют пробельные символы и сетевое имя сервера. Принтер идентифицируется строкой, начинающейся с ключевого слова `printer`, за которым следуют пробельные символы и сетевое имя принтера.

Типичный конфигурационный файл может содержать следующие строки:

```
printserver    blade
printer        phaser860
```

где `blade` – это сетевое имя сервера, на котором запущен демон спулера печати, а `phaser860` – сетевое имя принтера.

Безопасность

Программы, которые работают с привилегиями суперпользователя, потенциально открывают систему для нападения. Сами по себе такие программы обычно не более уязвимы, чем любые другие, но в случае обнаружения уязвимостей они могут позволить атакующему получить неограниченный доступ к вашей системе.

Демон печати, который рассматривается в этой главе, запускается с привилегиями суперпользователя, чтобы назначить сокету привилегированный номер порта TCP. Чтобы сделать демон менее уязвимым, мы можем:

- Спроектировать демон так, чтобы он соответствовал принципам минимизации привилегий (раздел 8.11). После того, как сокету будет назначен привилегированный номер порта, можно изменить идентификаторы пользователя и группы демона на какие-либо другие, отличные от `root` (например, `lp`). Все файлы и каталоги, используемые для хранения заданий, поставленных в очередь печати, должны принадлежать этому непривилегированному пользователю. Благодаря этому обнаружение уязвимости даст атакующему доступ только к подсистеме печати. Это тоже неприятно, но гораздо менее серьезно, чем если бы атакующий получил неограниченный доступ ко всей системе.
- Проверить исходный код демона на наличие в нем всех известных потенциальных уязвимостей, таких как переполнение буфера.
- Журналировать случаи неожиданного или подозрительного поведения, чтобы впоследствии администратор мог обнаружить их и изучить.

21.5. Исходный код

Исходный код, рассматриваемый в этой главе, содержится в пяти файлах, за исключением некоторых библиотечных функций, которые мы использовали в предыдущих главах:

`ipp.h`	Заголовочный файл с определениями IPP
`print.h`	Заголовочный файл с константами общего назначения, определениями структур данных и объявлениями служебных процедур
`util.c`	Служебные процедуры, используемые обеими программами
`print.c`	Исходные тексты утилиты `print`, используемой для печати файлов
`printd.c`	Исходные тексты демона спулера печати

Мы будем исследовать эти файлы в указанном порядке.

Начнем с заголовочного файла `ipp.h`.

```
1  #ifndef _IPP_H
2  #define _IPP_H
```

```
 3  /*
 4   * Определения протокола IPP, касающиеся взаимодействия между
 5   * планировщиком и принтером. Основаны на RFC2911 и RFC2910.
 6   */

 7  /*
 8   * Классы кодов статуса.
 9   */
10  #define STATCLASS_OK(x)     ((x) >= 0x0000 && (x) <= 0x00ff)
11  #define STATCLASS_INFO(x)   ((x) >= 0x0100 && (x) <= 0x01ff)
12  #define STATCLASS_REDIR(x)  ((x) >= 0x0200 && (x) <= 0x02ff)
13  #define STATCLASS_CLIERR(x) ((x) >= 0x0400 && (x) <= 0x04ff)
14  #define STATCLASS_SRVERR(x) ((x) >= 0x0500 && (x) <= 0x05ff)

15  /*
16   * Коды статуса.
17   */
18  #define STAT_OK          0x0000 /* успех */
19  #define STAT_OK_ATTRIGN  0x0001  /* OK; некоторые атрибуты проигнорированы */
20  #define STAT_OK_ATTRCON  0x0002 /* OK; конфликты между */
                                    /* некоторыми атрибутами */

21  #define STAT_CLI_BADREQ  0x0400 /* неверный запрос клиента */
22  #define STAT_CLI_FORBID  0x0401 /* запрещенный запрос */
23  #define STAT_CLI_NOAUTH  0x0402 /* требуется аутентификация */
24  #define STAT_CLI_NOPERM  0x0403 /* клиент не авторизован */
25  #define STAT_CLI_NOTPOS  0x0404 /* невозможно выполнить запрос */
26  #define STAT_CLI_TIMOUT  0x0405 /* истекло время ожидания клиента */
27  #define STAT_CLI_NOTFND  0x0406 /* не найден объект по данному URI */
28  #define STAT_CLI_OBJGONE 0x0407 /* объект больше недоступен */
29  #define STAT_CLI_TOOBIG  0x0408 /* запрошенный объект слишком велик */
30  #define STAT_CLI_TOOLNG  0x0409 /* слишком большое значение атрибута */
31  #define STAT_CLI_BADFMT  0x040a /* неподдерживаемый формат документа */
32  #define STAT_CLI_NOTSUP  0x040b /* неподдерживаемые атрибуты */
33  #define STAT_CLI_NOSCHM  0x040c /* неподдерживаемая схема URI */
34  #define STAT_CLI_NOCHAR  0x040d /* неподдерживаемый набор символов */
35  #define STAT_CLI_ATTRCON 0x040e /* конфликтующие атрибуты */
36  #define STAT_CLI_NOCOMP  0x040f /* сжатие не поддерживается */
37  #define STAT_CLI_COMPERR 0x0410 /* данные не могут быть разжаты */
38  #define STAT_CLI_FMTERR  0x0411 /* ошибка в формате документа */
39  #define STAT_CLI_ACCERR  0x0412 /* ошибка доступа к данным */
```

[1–14] **Начинается загловочный файл со стандартного определения `#ifndef`, чтобы предотвратить возникновение ошибок, связанных с повторным подключением одного и того же заголовочного файла. Далее следуют определения классов кодов статуса IPP (раздел 13 RFC 2911).**

[15–39] **Мы определяем конкретные коды статуса на основе RFC 2911. Они не будут использоваться в нашей программе, но понадобятся в упражнении 21.1.**

```
40  #define STAT_SRV_INTERN  0x0500 /* неожиданная внутренняя ошибка */
41  #define STAT_SRV_NOTSUP  0x0501 /* операция не поддерживается */
42  #define STAT_SRV_UNAVAIL 0x0502 /* услуга недоступна */
```

```
43  #define STAT_SRV_BADVER  0x0503 /* неподдерживаемая версия */
44  #define STAT_SRV_DEVERR  0x0504 /* ошибка устройства */
45  #define STAT_SRV_TMPERR  0x0505 /* временная ошибка */
46  #define STAT_SRV_REJECT  0x0506 /* сервер не принял задание */
47  #define STAT_SRV_TOOBUSY 0x0507 /* сервер занят */
48  #define STAT_SRV_CANCEL  0x0508 /* задание было отменено */
49  #define STAT_SRV_NOMULTI 0x0509 /* задания из нескольких документов */
                                    /* не поддерживаются */
50  /*
51  * Идентификаторы операций.
52  */
53  #define OP_PRINT_JOB        0x02
54  #define OP_PRINT_URI        0x03
55  #define OP_VALIDATE_JOB     0x04
56  #define OP_CREATE_JOB       0x05
57  #define OP_SEND_DOC         0x06
58  #define OP_SEND_URI         0x07
59  #define OP_CANCEL_JOB       0x08
60  #define OP_GET_JOB_ATTR     0x09
61  #define OP_GET_JOBS         0x0a
62  #define OP_GET_PRINTER_ATTR 0x0b
63  #define OP_HOLD_JOB         0x0c
64  #define OP_RELEASE_JOB      0x0d
65  #define OP_RESTART_JOB      0x0e
66  #define OP_PAUSE_PRINTER    0x10
67  #define OP_RESUME_PRINTER   0x11
68  #define OP_PURGE_JOBS       0x12
69  /*
70  * Признаки атрибутов.
71  */
72  #define TAG_OPERATION_ATTR 0x01 /* признак атрибутов операции */
73  #define TAG_JOB_ATTR       0x02 /* признак атрибутов задания */
74  #define TAG_END_OF_ATTR    0x03 /* признак конца списка атрибутов */
75  #define TAG_PRINTER_ATTR   0x04 /* признак атрибутов принтера */
76  #define TAG_UNSUPP_ATTR    0x05 /* признак неподдерживаемых атрибутов */
```

[40–49] **Продолжение определений кодов статуса. Коды в диапазоне от `0x500` до `0x5ff` являются кодами ошибок сервера. Описания всех кодов вы найдете в разделах 13.1.1–13.1.5 RFC 2911.**

[50–68] **Далее мы определяем идентификаторы различных операций. Каждой исполняемой задаче, определяемой протоколом IPP, соответствует свой идентификатор (раздел 4.4.15 RFC 2911). В нашем примере мы будем использовать только операцию `OP_PRINT_JOB`.**

[69–76] **Признаки атрибутов, разделяющие группы атрибутов в сообщениях протокола IPP. Значения признаков определены в разделе 3.5.1 RFC 2910.**

```
77  /*
78  * Значения признаков.
79  */
```

```
 80  #define TAG_UNSUPPORTED 0x10 /* неподдерживаемое значение */
 81  #define TAG_UNKNOWN     0x12 /* неизвестное значение */
 82  #define TAG_NONE        0x13 /* нет значения */
 83  #define TAG_INTEGER     0x21 /* целое */
 84  #define TAG_BOOLEAN     0x22 /* булево */
 85  #define TAG_ENUM        0x23 /* перечисление */
 86  #define TAG_OCTSTR      0x30 /* строка октетов */
 87  #define TAG_DATETIME    0x31 /* дата и время */
 88  #define TAG_RESOLUTION  0x32 /* разрешающая способность */
 89  #define TAG_INTRANGE    0x33 /* диапазон целых чисел */
 90  #define TAG_TEXTWLANG   0x35 /* текст с признаком языка */
 91  #define TAG_NAMEWLANG   0x36 /* имя с признаком языка */
 92  #define TAG_TEXTWOLANG  0x41 /* текст */
 93  #define TAG_NAMEWOLANG  0x42 /* имя */
 94  #define TAG_KEYWORD     0x44 /* ключевое слово */
 95  #define TAG_URI         0x45 /* URI */
 96  #define TAG_URISCHEME   0x46 /* схема URI */
 97  #define TAG_CHARSET     0x47 /* кодировка символов */
 98  #define TAG_NATULANG    0x48 /* естественный язык */
 99  #define TAG_MIMETYPE    0x49 /* тип MIME */

100  struct ipp_hdr {
101    int8_t major_version; /* всегда 1 */
102    int8_t minor_version; /* всегда 1 */
103    union {
104        int16_t op;       /* идентификатор операции */
105        int16_t st;       /* статус */
106    } u;
107    int32_t request_id;   /* идентификатор запроса */
108    char attr_group[1];   /* начало группы опциональных атрибутов */
109    /* далее могут следовать дополнительные данные */
110  };

111  #define operation u.op
112  #define status u.st

113  #endif /* _IPP_H */
```

[77–99] Значения признаков определяют формат отдельных атрибутов и параметров. Они определены в разделе 3.5.2 RFC 2910.

[100–113] Определение структуры заголовка IPP. Сообщения-запросы и сообщения-ответы имеют одинаковую структуру заголовка, за исключением идентификатора операции, который в сообщении-ответе замещается кодом статуса.

В конце заголовочного файла находится закрывающий `#endif`, который соответствует директиве `#ifndef`, расположенной в начале заголовочного файла.

Далее следует заголовочный файл `print.h`.

```
1  #ifndef _PRINT_H
2  #define _PRINT_H
```

```
 3  /*
 4   * Заголовочный файл сервера печати.
 5   */
 6  #include <sys/socket.h>
 7  #include <arpa/inet.h>
 8  #if defined(BSD) || defined(MACOS)
 9  #include <netinet/in.h>
10  #endif
11  #include <netdb.h>
12  #include <errno.h>

13  #define CONFIG_FILE "/etc/printer.conf"
14  #define SPOOLDIR    "/var/spool/printer"
15  #define JOBFILE     "jobno"
16  #define DATADIR     "data"
17  #define REQDIR      "reqs"

18  #define FILENMSZ    64
19  #define FILEPERM   (S_IRUSR|S_IWUSR)
20  #define USERNM_MAX  64
21  #define JOBNM_MAX  256
22  #define MSGLEN_MAX 512

23  #ifndef HOST_NAME_MAX
24  #define HOST_NAME_MAX 256
25  #endif

26  #define IPP_PORT 631
27  #define QLEN      10
28  #define IBUFSZ   512 /* размер буфера для хранения заголовка IPP */
29  #define HBUFSZ   512 /* размер буфера для хранения заголовка HTTP */
30  #define IOBUFSZ 8192 /* размер буфера для хранения данных */
```

[1–12] Мы подключаем все заголовочные файлы, которые могут потребоваться приложению, подключающему этот заголовочный файл. Приложения могут просто подключать файл `print.h`, что облегчает отслеживание всех зависимостей заголовочных файлов.

[13–17] Мы определяем файлы и каталоги, используемые в данной реализации. Копии печатаемых файлов сохраняются в каталоге `/var/spool/printer/data`, управляющая информация по каждому из запросов – в каталоге `/var/spool/printer/reqs`. Файл, в котором хранится номер следующего задания печати, – `/var/spool/printer/jobno`.

[18–30] Далее следуют определения констант и пределов. При создании копий файлов, переданных для печати, им присваиваются права доступа `FILEPERM`. Права доступа к копиям файлов ограничены, потому что мы хотим предотвратить доступ других пользователей к этим файлам, пока они ожидают вывода на принтер. Порт 631 используется протоколом IPP по умолчанию. Константа `QLEN` определяет значение аргумента *backlog* функции `listen` (раздел 16.4).

```
31  #ifndef ETIME
32  #define ETIME ETIMEDOUT
33  #endif
```

```
34  extern int getaddrlist(const char *, const char *,
35    struct addrinfo **);
36  extern char *get_printserver(void);
37  extern struct addrinfo *get_printaddr(void);
38  extern ssize_t tread(int, void *, size_t, unsigned int);
39  extern ssize_t treadn(int, void *, size_t, unsigned int);
40  extern int connect_retry(int, const struct sockaddr *, socklen_t);
41  extern int initserver(int, struct sockaddr *, socklen_t, int);
42  /*
43   * Структура, описывающая запрос утилиты print.
44   */
45  struct printreq {
46    long size;               /* размер в байтах */
47    long flags;              /* см. ниже */
48    char usernm[USERNM_MAX]; /* имя пользователя */
49    char jobnm[JOBNM_MAX];   /* имя задания */
50  };
51  /*
52   * Флаги запроса.
53   */
54  #define PR_TEXT    0x01 /* интерпретировать файл как обычный текст */
55  /*
56   * Ответ демона на запрос утилиты print.
57   */
58  struct printresp {
59    long retcode;         /* 0=успех, !0=код ошибки */
60    long jobid;           /* идентификатор задания */
61    char msg[MSGLEN_MAX]; /* сообщение об ошибке */
62  };
63  #endif /* _PRINT_H */
```

[31–33] Некоторые платформы не определяют код ошибки `ETIME`, поэтому мы сами определяем эту константу для использования на таких платформах.

[34–41] Далее мы определяем все общедоступные функции, которые содержатся в файле `util.c` (он следует чуть ниже). Обратите внимание на то, что функции `connect_retry` из листинга 16.2 и `initserver` из листинга 16.9 не включены в файл `util.c`.

[42–63] Структуры `printreq` и `printresp` определяют протокол взаимодействия между утилитой `print` и демоном спулера печати. Утилита `print` отправляет структуру `printreq`, в которой определены имя пользователя, имя задания и размер файла. Демон отвечает структурой `printresp`, содержащей возвращаемый код, идентификатор задания и текст сообщения об ошибке в случае невозможности выполнить запрос.

Далее следует файл `util.c`, содержащий служебные функции.

```
1  #include "apue.h"
2  #include "print.h"
3  #include <ctype.h>
```

```
 4  #include <sys/select.h>

 5  #define MAXCFGLINE 512
 6  #define MAXKWLEN 16
 7  #define MAXFMTLEN 16

 8  /*
 9   * Получить перечень адресов для заданного хоста и службы и вернуть
10   * его в ailistpp. Возвращает 0 в случае успеха или ненулевое значение
11   * в случае ошибки (код ошибки). Обратите внимание: код ошибки
12   * не записывается в переменную errno.
13   *
14   * БЛОКИРОВКИ: отсутствуют.
15   */
16  int
17  getaddrlist(const char *host, const char *service,
18    struct addrinfo **ailistpp)
19  {
20    int             err;
21    struct addrinfo hint;

22    hint.ai_flags = AI_CANONNAME;
23    hint.ai_family = AF_INET;
24    hint.ai_socktype = SOCK_STREAM;
25    hint.ai_protocol = 0;
26    hint.ai_addrlen = 0;
27    hint.ai_canonname = NULL;
28    hint.ai_addr = NULL;
29    hint.ai_next = NULL;
30    err = getaddrinfo(host, service, &hint, ailistpp);
31    return(err);
32  }
```

[1–7] **Прежде всего мы устанавливаем пределы, необходимые для работы функций из этого файла. Константа** `MAXCFGLINE` **определяет максимальный размер строки конфигурационного файла,** `MAXKWLEN` **– максимальный размер ключевого слова в конфигурационном файле,** `MAXFMTLEN` **– максимальный размер строки формата, которая передается функции** `sscanf`**.**

[8–32] **Первая функция в файле** `-getaddrlist`**. Она представляет собой обертку вокруг** `getaddrinfo` **(раздел 16.3.3), так как мы всегда будем вызывать** `getaddrinfo` **с одними и теми же значениями полей структуры** `hint`**. Обратите внимание: использовать мьютексы в этой функции не требуется. Комментарий** `БЛОКИРОВКИ` **в начале каждой функции предназначен только для документирования используемых блокировок. В нем перечисляются предположения, касающиеся блокировок (если таковые имеются), и блокировки, которые должны быть установлены или сняты функцией, а также блокировки, которые должны быть установлены перед ее вызовом.**

```
33  /*
34   * Отыскать в конфигурационном файле заданное ключевое слово
35   * и вернуть строку, соответствующую этому ключевому слову.
36   *
```

```
37    * БЛОКИРОВКИ: отсутствуют.
38    */
39   static char *
40   scan_configfile(char *keyword)
41   {
42      int         n, match;
43      FILE        *fp;
44      char        keybuf[MAXKWLEN], pattern[MAXFMTLEN];
45      char        line[MAXCFGLINE];
46      static char valbuf[MAXCFGLINE];

47      if ((fp = fopen(CONFIG_FILE, "r")) == NULL)
48          log_sys("невозможно открыть %s", CONFIG_FILE);
49      sprintf(pattern, "%%%ds %%%ds", MAXKWLEN-1, MAXCFGLINE-1);
50      match = 0;
51      while (fgets(line, MAXCFGLINE, fp) != NULL) {
52          n = sscanf(line, pattern, keybuf, valbuf);
53          if (n == 2 && strcmp(keyword, keybuf) == 0) {
54             match = 1;
55             break;
56          }
57      }
58      fclose(fp);
59      if (match != 0)
60          return(valbuf);
61      else
62          return(NULL);
63   }
```

[33–46] Функция `scan_configfile` отыскивает в конфигурационном файле заданное ключевое слово.

[47–63] Мы открываем конфигурационный файл для чтения и строим строку формата, которая соответствует шаблону поиска. Нотация `%%%ds` создает спецификатор формата, который ограничивает размер строки, благодаря чему можно не опасаться ошибки переполнения буфера, размещаемого на стеке. Мы читаем из файла по одной строке за раз и выделяем из нее две подстроки, разделенные пробелами. Если они найдены, мы сравниваем первую подстроку с заданным ключевым словом. В случае совпадения или достижения конца файла цикл завершается, и мы закрываем файл. Если найдено совпадение с заданным ключевым словом, возвращается указатель на буфер, содержащий вторую подстроку, расположенную после ключевого слова, в противном случае возвращается `NULL`.

Возвращаемая подстрока сохраняется в статическом буфере (`valbuf`), который может перезаписываться при успешных вызовах функции. Таким образом, функцию `scan_configfile` нельзя использовать в многопоточных приложениях, если не позаботиться о том, чтобы ее вызов из нескольких потоков одновременно был невозможен.

```
64   /*
65   * Возвращает имя хоста, на котором работает демон печати, или NULL в случае ошибки
66   *
```

```
67    * БЛОКИРОВКИ: отсутствуют.
68    */
69   char *
70   get_printserver(void)
71   {
72       return(scan_configfile("printserver"));
73   }

74   /*
75    * Возвращает адрес сетевого принтера или NULL в случае ошибки.
76    *
77    * БЛОКИРОВКИ: отсутствуют.
78    */
79   struct addrinfo *
80   get_printaddr(void)
81   {
82      int             err;
83      char            *p;
84      struct addrinfo *ailist;

85      if ((p = scan_configfile("printer")) != NULL) {
86          if ((err = getaddrlist(p, "ipp", &ailist)) != 0) {
87              log_msg("нет сведений об адресе %s", p);
88              return(NULL);
89          }
90          return(ailist);
91      }
92      log_msg("не задан адрес принтера");
93      return(NULL);
94   }
```

[64–73] Функция `get_printserver` является просто функцией-оберткой, которая вызывает `scan_configfile`, чтобы отыскать имя системы, в которой работает демон печати.

[74–94] Мы используем функцию `get_printaddr`, чтобы получить адрес сетевого принтера. Она похожа на предыдущую функцию за исключением того, что после получения имени принтера из конфигурационного файла оно используется для получения сетевого адреса принтера.

Обе функции, `get_printserver` и `get_printaddr`, обращаются к функции `scan_configfile`. Если она не может открыть конфигурационный файл, то `scan_configfile` вызывает `log_sys`, чтобы вывести сообщение об ошибке, и завершается. Хотя функция `get_printserver` предназначена для использования утилитой `print`, а `get_printaddr` – демоном печати, вызов `log_sys` в обоих случаях можно считать вполне нормальным, поскольку мы можем простым изменением глобальной переменной заставить функции журналирования выводить сообщения не в файл журнала, а на стандартное устройство вывода сообщений об ошибках.

```
95   /*
96    * Ограниченная по времени операция чтения - тайм-аут задается в секундах
97    * (5-й аргумент функции select, который определяет предельное время
```

```
 98    * ожидания данных). Возвращает количество прочитанных байт или -1 (ошибка).
 99    *
100    * БЛОКИРОВКИ: отсутствуют.
101    */
102   ssize_t
103   tread(int fd, void *buf, size_t nbytes, unsigned int timout)
104   {
105     int            nfds;
106     fd_set         readfds;
107     struct timeval tv;

108     tv.tv_sec = timout;
109     tv.tv_usec = 0;
110     FD_ZERO(&readfds);
111     FD_SET(fd, &readfds);
112     nfds = select(fd+1, &readfds, NULL, NULL, &tv);
113     if (nfds <= 0) {
114         if (nfds == 0)
115             errno = ETIME;
116         return(-1);
117     }
118     return(read(fd, buf, nbytes));
119   }
```

[95–107] Мы предоставляем функцию `tread`, которая считывает заданное количество байт, но блокирует вызывающий процесс не более чем на *`timeout`* секунд. Эта функция удобна для чтения данных из сокета или неименованного канала. Если в течение времени тайм-аута данные так и не поступили, возвращается значение −1 и код ошибки `ETIME` в переменной `errno`. Если в течение заданного периода времени данные стали доступны, возвращается до *`nbytes`* байт данных, но мы можем прочитать меньшее количество байт, чем было запрошено, если не все данные пришли вовремя.

Мы будем использовать функцию `tread` для предотвращения атак типа «отказ в обслуживании» (denial-of-service – DOS) на демон печати. Злоумышленник мог бы непрерывно пытаться подключиться к демону, не передавая ему никаких данных, что лишило бы остальных пользователей возможности передать демону свои задания печати. Установив предел времени ожидания, мы исключаем возможность возникновения таких ситуаций. Сложность состоит в том, чтобы правильно подобрать значение этого предела, которое должно быть достаточно большим, чтобы предотвратить возможность преждевременной потери запросов при высокой нагрузке на систему, когда для выполнения задач требуется больше времени. Однако, выбрав слишком большое значение тайм-аута, мы рискуем подвергнуться атакам типа «отказ в обслуживании», позволяя демону захватить слишком много ресурсов для обслуживания ожидающих обработки запросов.

[108–119] Мы ожидаем, когда заданный дескриптор станет доступен для чтения, используя функцию `select`. Если время тайм-аута истечет раньше, чем появятся доступные для чтения данные, функция `select` вернет значение 0, в этом случае в переменную `errno` записывается значение `ETIME`. По истечении тайм-аута или в случае ошибки функции `select` возвращается значение −1. Иначе возвращаются данные, которые удалось прочитать.

```
120  /*
121   * Ограниченная по времени операция чтения - тайм-аут задается в секундах
122   * на каждый вызов read, функция пытается прочитать nbytes байт.
123   * Возвращает количество прочитанных байт или -1 в случае ошибки.
124   *
125   * БЛОКИРОВКИ: отсутствуют.
126   */
127  ssize_t
128  treadn(int fd, void *buf, size_t nbytes, unsigned int timout)
129  {
130    size_t nleft;
131    ssize_t nread;
132
133    nleft = nbytes;
134    while (nleft > 0) {
135        if ((nread = tread(fd, buf, nleft, timout)) < 0) {
136            if (nleft == nbytes)
137                return(-1); /* ошибка, вернуть -1 */
138            else
139                break;     /* ошибка, вернуть то, что удалось прочитать */
140        } else if (nread == 0) {
141            break;          /* конец файла */
142        }
143        nleft -= nread;
144        buf += nread;
145    }
146    return(nbytes - nleft); /* вернуть значение >= 0 */
147  }
```

[120–146] Мы реализовали еще одну версию функции `tread`, которую назвали `treadn`. Она пытается прочитать именно то количество байт, которое было запрошено. Она напоминает функцию `readn`, описанную в разделе 14.8, но в отличие от последней имеет дополнительный аргумент, в котором задается время тайм-аута.

Чтобы прочитать заданное количество байт, мы должны быть готовы произвести несколько обращений к функции `read`. Сложность заключается в использовании единого времени тайм-аута для всех вызовов `read`. Мы не хотели использовать таймер, поскольку обслуживать сигналы в многопоточных приложениях достаточно сложно. Кроме того, мы не можем полагаться на то, что система обновит содержимое структуры `timeval` при выходе из функции `select`, чтобы показать время, оставшееся до истечения тайм-аута, так как многие платформы не поддерживают эту возможность (раздел 14.5.1). Поэтому мы пошли на компромисс и определили значение тайм-аута для каждого отдельного вызова `read`. Вместо ограничения общего времени ожидания мы ограничили время ожидания в каждой итерации цикла. Максимальное возможное время ожидания ограничено значением *nbytes*×*timeout* секунд (в худшем случае мы будем получать не более 1 байта за раз).

Переменная `nleft` используется для хранения количества байт, которое осталось прочитать. Если функция `tread` терпит неудачу, но на предыдущих итерациях удалось прочитать некоторый объем данных, мы прерываем цикл `while` и возвращаем то, что удалось прочесть, иначе возвращается –1.

Далее следуют исходные тексты утилиты `print`, которая используется для передачи задания печати. Файл с исходным кодом на C называется `print.c`.

```
1  /*
2   * Утилита печати документов. Открывает файл и отправляет его демону печати.
3   * Использование:
4   * print [-t] filename
5   */
6  #include "apue.h"
7  #include "print.h"
8  #include <fcntl.h>
9  #include <pwd.h>
10 /*
11  * Необходимо для функций журналирования.
12  */
13 int log_to_stderr = 1;
14 void submit_file(int, int, const char *, size_t, int);
15 int
16 main(int argc, char *argv[])
17 {
18    int             fd, sockfd, err, text, c;
19    struct stat     sbuf;
20    char            *host;
21    struct addrinfo *ailist, *aip;
22    err = 0;
23    text = 0;
24    while ((c = getopt(argc, argv, "t")) != -1) {
25        switch (c) {
26        case 't':
27            text = 1;
28            break;
29        case '?':
30            err = 1;
31            break;
32        }
33   }
```

[1–14] Мы определяем целочисленную переменную `log_to_stderr`, чтобы иметь возможность использовать в нашей библиотеке функции журналирования. Если переменная имеет ненулевое значение, сообщения об ошибках будут выводиться на стандартное устройство вывода сообщений об ошибках, а не в файл журнала. Хотя в файле `print.c` не используются функции журналирования, но при сборке исполняемого файла `print` мы связываем `print.o` и `util.o`, а `util.c` содержит функции как для сервера, так и для клиента.

[15–33] Поддерживается единственная опция `-t`, с помощью которой мы указываем, что файл должен печататься как обычный текст (а не как PostScript, например). Для обработки параметров командной строки используется функция `getopt`(3).

```
34      if (err || (optind != argc - 1))
35          err_quit("Использование: print [-t] filename");
36      if ((fd = open(argv[optind], O_RDONLY)) < 0)
37          err_sys("print: невозможно открыть %s", argv[1]);
38      if (fstat(fd, &sbuf) < 0)
39          err_sys("print: невозможно получить сведения о %s", argv[1]);
40      if (!S_ISREG(sbuf.st_mode))
41          err_quit("print: %s должен быть обычным файлом\n", argv[1]);
42      /*
43       * Получить имя хоста, который выступает в роли сервера печати.
44       */
45      if ((host = get_printserver()) == NULL)
46          err_quit("print: сервер печати не определен");
47      if ((err = getaddrlist(host, "print", &ailist)) != 0)
48          err_quit("print: ошибка getaddrinfo: %s", gai_strerror(err));
49      for (aip = ailist; aip != NULL; aip = aip->ai_next) {
50          if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
51              err = errno;
52          } else if (connect_retry(sockfd, aip->ai_addr,
53            aip->ai_addrlen) < 0) {
54              err = errno;
```

[34–41] Когда функция `getopt` заканчивает обработку списка аргументов, она записывает в переменную `optind` индекс первого неопционального аргумента. Если это значение будет отличаться от индекса последнего аргумента, следовательно, программа получила неверное количество аргументов (поддерживается только один обязательный аргумент). Обработка ошибок включает проверку возможности открытия файла, отправляемого на печать, и проверку, является ли он обычным файлом (то есть не каталогом или файлом какого-либо другого типа).

[42–48] Мы получаем имя хоста, на котором работает демон печати, вызовом функции `get_printserver` из `util.c`, и затем преобразуем его в сетевой адрес вызовом функции `getaddrlist` (также из файла `util.c`).

Обратите внимание: мы определили имя службы как «print». При установке демона печати необходимо убедиться, что в `/etc/services` (или эквивалентной базе данных) имеется запись, соответствующая службе печати. При выборе номера порта для демона мы приняли правильное решение, взяв номер порта из привилегированного диапазона. Тем самым мы лишили потенциального злоумышленника возможности написать свою программу, имитирующую поведение демона печати, чтобы перехватывать копии файлов, отправляемых на печать. Это означает, что номер порта должен быть меньше 1024 (раздел 16.3.4) и что демон должен запускаться с привилегиями суперпользователя, чтобы иметь возможность связать сокет с привилегированным номером порта.

[49–54] Мы пытаемся соединиться с демоном, используя поочередно адреса из списка, полученного от `getaddrinfo`. Для передачи файла будет использоваться первый адрес, с которым нам удастся установить соединение.

```
55          } else {
56              submit_file(fd, sockfd, argv[1], sbuf.st_size, text);
```

```
57              exit(0);
58          }
59      }
60      errno = err;
61      err_ret("print: невозможно соединиться с %s", host);
62      exit(1);
63  }
64  /*
65   * Отправить файл демону печати.
66   */
67  void
68  submit_file(int fd, int sockfd, const char *fname, size_t nbytes,
69  int text)
70  {
71      int nr, nw, len;
72      struct passwd *pwd;
73      struct printreq req;
74      struct printresp res;
75      char buf[IOBUFSZ];
76      /*
77       * Сначала соберем заголовок.
78       */
79      if ((pwd = getpwuid(geteuid())) == NULL)
80          strcpy(req.usernm, "unknown");
81      else
82          strcpy(req.usernm, pwd->pw_name);
83      req.size = htonl(nbytes);
84      if (text)
85          req.flags = htonl(PR_TEXT);
86      else
87          req.flags = 0;
```

[55–63] Если удалось установить соединение, мы отправляем файл демону печати с помощью функции `submit_file`. Если установить соединение не удалось, мы выводим сообщение об ошибке и завершаем работу. Вместо одного вызова `err_sys` мы вызываем функции `err_ret` и `exit`, чтобы избежать предупреждений компилятора, потому что последняя строка функции `main` в первом случае не содержала бы оператора `return` или вызова функции `exit`.

[64–87] Функция `submit_file` отправляет запрос на печать демону и получает от него ответ. Для начала мы собираем заголовок запроса `printreq`. С помощью функции `getuid` мы получаем эффективный идентификатор пользователя, который затем передаем функции `getpwuid`, чтобы отыскать имя пользователя в файле паролей. Далее мы копируем полученное имя пользователя в заголовок запроса или, если идентифицировать пользователя не удалось, записываем в заголовок строку `unknown` (неизвестен). После этого мы записываем в заголовок размер отправляемого файла, попутно преобразуя его в значение с сетевым порядком байтов. То же самое мы делаем с флагом `PR_TEXT`, если файл должен печататься как простой текст.

```
88      if ((len = strlen(fname)) >= JOBNM_MAX) {
```

```
 89          /*
 90           * Усечь имя файла (с учетом 5 символов, отводимых под
 91           * четыре символа префикса и завершающий нулевой символ).
 92           */
 93          strcpy(req.jobnm, "... ");
 94          strncat(req.jobnm, &fname[len-JOBNM_MAX+5], JOBNM_MAX-5);
 95      } else {
 96          strcpy(req.jobnm, fname);
 97      }
 98      /*
 99       * Отправить заголовок серверу.
100       */
101      nw = writen(sockfd, &req, sizeof(struct printreq));
102      if (nw != sizeof(struct printreq)) {
103          if (nw < 0)
104              err_sys("невозможно передать запрос серверу");
105          else
106              err_quit("запрос серверу был передан не полностью (%d/%d)",
107                       nw, sizeof(struct printreq));
108      }
109      /*
110       * Теперь отправить файл.
111       */
112      while ((nr = read(fd, buf, IOBUFSZ)) != 0) {
113          nw = writen(sockfd, buf, nr);
114          if (nw != nr) {
115              if (nw < 0)
116                  err_sys("невозможно отправить файл серверу");
117              else
118                  err_quit("файл серверу был передан не полностью (%d/%d)",
119                           nw, nr);
120          }
121      }
```

[88–108] Из имени печатаемого файла мы собираем имя задания. Если имя файла длиннее, чем может вместить сообщение, мы усекаем его, а первые четыре символа замещаем многоточием, чтобы показать, что имя файла целиком не уместилось в поле структуры. После этого мы отправляем заголовок запроса демону с помощью функции `writen`. Если попытка записи не удалась или объем записанных данных оказался меньше размера заголовка, выводится сообщение об ошибке и работа программы завершается.

[109–121] После передачи заголовка мы отправляем демону файл, который должен быть напечатан. Мы читаем файл блоками по `IOBUFSZ` байт и отправляем их демону с помощью функции `writen`. Как и в случае с передачей заголовка, если какая-либо операция записи завершилась неудачей или объем записанных данных оказался меньше требуемого, мы выводим сообщение об ошибке и завершаем работу программы.

```
122      /*
123       * Прочитать ответ.
```

```
124      */
125     if ((nr = readn(sockfd, &res, sizeof(struct printresp))) !=
126       sizeof(struct printresp))
127         err_sys("невозможно прочитать ответ сервера");
128     if (res.retcode != 0) {
129         printf("запрос отвергнут: %s\n", res.msg);
130         exit(1);
131     } else {
132         printf("идентификатор задания %ld\n", ntohl(res.jobid));
133     }
134     exit(0);
135   }
```

[122–135] Отправив печатаемый файл, мы читаем ответ сервера. Если запрос был отвергнут, возвращаемый код (`retcode`) будет не равен нулю, в этом случае мы выводим текстовое сообщение об ошибке, включенное в ответ. Если запрос был благополучно принят сервером, мы выводим идентификатор задания на печать, чтобы пользователь знал, каким образом ссылаться на запрос. (В качестве упражнения мы предлагаем написать реализацию утилиты, с помощью которой можно отменить запрос на печать. В этом случае для идентификации задания, удаляемого из очереди печати, может использоваться его идентификатор.)

Обратите внимание: сообщение об успешном приеме задания сервером еще не означает, что принтер сможет напечатать файл. Это лишь означает, что демон благополучно добавил задание в очередь печати.

Большая часть из того, что мы увидели в `print.c`, уже обсуждалась в предыдущих главах. Единственное, о чем мы еще не говорили, – это функция `getopt`, хотя мы встречались с ней в программе `pty` в главе 19.

Очень важно, чтобы все команды в системе следовали одним и тем же соглашениям, потому что это облегчает их использование. Если программист хорошо знаком со способом передачи аргументов одной команде, это знание может стать источником ошибок, если другая команда следует иным соглашениям.

Эта проблема иногда проявляет себя, когда приходится иметь дело с пробелами в командной строке. Одни команды требуют, чтобы ключи отделялись от своих аргументов пробелами, а другие – чтобы аргумент ключа следовал сразу же за последним символом ключа без каких-либо пробелов. При отсутствии непротиворечивого набора правил пользователь вынужден либо запоминать синтаксис всех команд, либо использовать метод проб и ошибок при обращении к ним.

Стандарт Single UNIX Specification содержит ряд соглашений и рекомендаций, которые определяют непротиворечивый синтаксис командной строки. Они включают такие предписания, как «Имя каждой опции должно состоять из одного алфавитного символа» и «Все опции должны начинаться с символа -».

К счастью, существует функция `getopt`, которая помогает программистам обрабатывать параметры командной строки непротиворечивым образом.

```
#include <fcntl.h>

int getopt(int argc, const * const argv[], const char *options);

extern int optind, opterr, optopt;
extern char *optarg;
```

Возвращает символ следующей опции или –1, если были обработаны все опции

Аргументы *argc* и *argv* – те же самые, что передаются функции `main`. Аргумент *options* – это строка, содержащая символы поддерживаемых программой опций. Двоеточие, следующее за символом опции, говорит о том, что опция должна сопровождаться аргументом. В противном случае опция существует сама по себе. Например, если для команды `command` определен следующий синтаксис:

```
command [-i] [-u username] [-z] filename
```

то мы должны определить строку *options* как `"iu:z"`.

Обычно функция `getopt` используется в цикле, который завершается, когда `getopt` вернет значение –1. В каждой итерации `getopt` возвращает очередную опцию. Приложение само должно разобраться с возможными конфликтами между опциями, а `getopt` просто анализирует их и приводит к стандартному формату.

Когда функция `getopt` встречает недопустимую опцию, она вместо символа опции возвращает знак вопроса. Если отсутствует аргумент опции, `getopt` также возвращает знак вопроса, но если первым символом в строке *options* стоит двоеточие, то вместо знака вопроса будет возвращаться двоеточие. Специальная последовательность символов `--` заставляет `getopt` прервать обработку опций и вернуть значение –1. Это позволяет передавать командам аргументы, которые начинаются с символа `-`, но не являются опциями. Например, если имеется файл с именем `-bar`, вы не сможете удалить его командой

```
rm -bar
```

потому что `rm` будет пытаться интерпретировать `-bar` как набор опций. Таким образом, чтобы удалить этот файл, необходимо дать команду

```
rm -- -bar
```

Функция `getopt` поддерживает четыре глобальные переменные.

`optarg` Если опция сопровождается аргументом, `getopt` при обработке опции записывает в `optarg` указатель на строку с аргументом.

`opterr` Если при обработке опции возникнет ошибка, `getopt` по умолчанию выводит сообщение об ошибке. Чтобы запретить вывод таких сообщений, приложение может записать в переменную `opterr` значение 0.

`optind` Индекс в массиве `argv` строки, которая будет обработана следующей. Индексация начинается с 1 и увеличивается на 1 для каждого аргумента, обрабатываемого функцией `getopt`.

`optopt` **Если при обработке опции возникнет ошибка,** `getopt` **запишет в переменную** `optopt` **указатель на строку, в которой была обнаружена ошибка.**

Последний файл, который мы рассмотрим, содержит исходный код демона печати на языке C.

```
1  /*
2   * Демон сервера печати.
3   */
4  #include "apue.h"
5  #include "print.h"
6  #include "ipp.h"
7  #include <fcntl.h>
8  #include <dirent.h>
9  #include <ctype.h>
10 #include <pwd.h>
11 #include <pthread.h>
12 #include <strings.h>
13 #include <sys/select.h>
14 #include <sys/uio.h>

15 /*
16  * Ответы принтера по протоколу HTTP.
17  */
18 #define HTTP_INFO(x) ((x) >= 100 && (x) <= 199)
19 #define HTTP_SUCCESS(x) ((x) >= 200 && (x) <= 299)

20 /*
21  * Описание заданий для печати.
22  */
23 struct job {
24    struct job      *next; /* следующее задание в списке */
25    struct job      *prev; /* предыдущее задание в списке */
26    long            jobid; /* идентификатор задания */
27    struct printreq req;   /* копия запроса на печать */
28 };

29 /*
30  * Описание потока, обрабатывающего запрос от клиента.
31  */
32 struct worker_thread {
33    struct worker_thread *next;  /* следующее описание в списке */
34    struct worker_thread *prev;  /* предыдущее описание в списке */
35    pthread_t            tid;    /* идентификатор потока */
36    int                  sockfd; /* сокет */
37 };
```

[1–19] **Демон печати подключает описанный ранее заголовочный файл протокола IPP, так как он взаимодействует с принтером по этому протоколу. Макросы** `HTTP_INFO` **и** `HTTP_SUCCESS` **описывают коды статуса запроса HTTP (мы уже говорили, что протокол IPP реализован поверх протокола HTTP).**

[20–37] **Структуры** `job` **и** `worker_thread` **используются демоном для отслеживания заданий печати и потоков, принявших запросы на печать, соответственно.**

```
38  /*
39   * Для журналирования.
40   */
41  int log_to_stderr =  0;

42  /*
43   * Переменные, имеющие отношение к принтеру.
44   */
45  struct addrinfo      *printer;
46  char                 *printer_name;
47  pthread_mutex_t      configlock = PTHREAD_MUTEX_INITIALIZER;
48  int reread;

49  /*
50   * Переменные, имеющие отношение к потокам.
51   */
52  struct worker_thread *workers;
53  pthread_mutex_t      workerlock = PTHREAD_MUTEX_INITIALIZER;
54  sigset_t             mask;

55  /*
56   * Переменные, имеющие отношение к заданиям.
57   */
58  struct job           *jobhead, *jobtail;
59  int                  jobfd;
```

[38–41] Наши функции журналирования сообщений требуют определения переменной `log_to_stderr`. В эту переменную должно быть записано значение 0, чтобы сообщения выводились в системный журнал, а не на стандартное устройство вывода сообщений об ошибках. В файле `print.c` мы определяли переменную `log_to_stderr` и записывали в нее значение 1, хотя функции журналирования не использовались в утилите `print`. Можно было бы избежать этого, разделив служебные функции на два отдельных файла – один для сервера и один для клиентских приложений.

[42–48] В переменной `printer` хранится сетевой адрес принтера. Сетевое имя принтера хранится в переменной `printer_name`. Мьютекс `configlock` защищает доступ к переменной `reread`, которая указывает демону, что он должен перечитать конфигурационный файл – например, когда администратор изменил принтер или его сетевой адрес.

[49–54] Далее мы определяем переменные, имеющие отношение к потокам. Переменная `workers` хранит указатель на начало двусвязного списка потоков, принимающих файлы от клиентов. Доступ к этому списку осуществляется под защитой мьютекса `workerlock`. В переменной `mask` хранится маска сигналов, используемая потоками.

[55–59] В переменной `jobhead` хранится указатель на начало, а в переменной `jobtail` – на конец списка заданий, ожидающих обработки. Этот список также является двусвязным, но мы будем добавлять задания в конец списка, поэтому необходимо хранить указатель на конец списка. В случае с рабочими потоками порядок их расположения в списке не имеет значения, поэтому мы можем добавлять сведения о новых потоках в начало списка, и указатель на конец списка не нужен. Переменная `jobfd` – это дескриптор файла заданий.

```
60  long            nextjob;
61  pthread_mutex_t joblock = PTHREAD_MUTEX_INITIALIZER;
62  pthread_cond_t  jobwait = PTHREAD_COND_INITIALIZER;
63  /*
64   * Прототипы функций.
65   */
66  void    init_request(void);
67  void    init_printer(void);
68  void    update_jobno(void);
69  long    get_newjobno(void);
70  void    add_job(struct printreq *, long);
71  void    replace_job(struct job *);
72  void    remove_job(struct job *);
73  void    build_qonstart(void);
74  void    *client_thread(void *);
75  void    *printer_thread(void *);
76  void    *signal_thread(void *);
77  ssize_t readmore(int, char **, int, int *);
78  int     printer_status(int, struct job *);
79  void    add_worker(pthread_t, int);
80  void    kill_workers(void);
81  void    client_cleanup(void *);
82  /*
83   * Главный поток сервера печати. Принимает запросы на соединение
84   * от клиентов и запускает дополнительные потоки для обработки запросов.
85   *
86   * БЛОКИРОВКИ: отсутствуют.
87   */
88  int
89  main(int argc, char *argv[])
90  {
91      pthread_t        tid;
92      struct addrinfo  *ailist, *aip;
93      int              sockfd, err, i, n, maxfd;
94      char             *host;
95      fd_set           rendezvous, rset;
96      struct sigaction sa;
97      struct passwd    *pwdp;
```

[60–62] Переменная `nextjob` – это идентификатор следующего задания, которое будет принято. Мьютекс `joblock` служит для защиты доступа к связанному списку заданий и к состоянию, представленному переменной состояния `jobwait`.

[63–81] Объявления прототипов функций, которые будут использоваться в этом файле. Разместив прототипы в начале файла, мы можем больше не задумываться о том, в каком порядке они будут вызываться в файле.

[82–97] Функция `main` демона печати выполняет две задачи: инициализирует демон и принимает запросы на соединение от клиентов.

```
98      if (argc != 1)
99          err_quit("Использование: printd");
100     daemonize("printd");
```

```
101     sigemptyset(&sa.sa_mask);
102     sa.sa_flags = 0;
103     sa.sa_handler = SIG_IGN;
104     if (sigaction(SIGPIPE, &sa, NULL) < 0)
105         log_sys("ошибка вызова функции sigaction");
106     sigemptyset(&mask);
107     sigaddset(&mask, SIGHUP);
108     sigaddset(&mask, SIGTERM);
109     if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
110         log_sys("ошибка вызова функции pthread_sigmask");
111     init_request();
112     init_printer();
113   #ifdef _SC_HOST_NAME_MAX
114     n = sysconf(_SC_HOST_NAME_MAX);
115     if (n < 0)  /* лучшее, что можно предположить */
116   #endif
117         n = HOST_NAME_MAX;
118     if ((host = malloc(n)) == NULL)
119         log_sys("ошибка вызова функции malloc");
120     if (gethostname(host, n) < 0)
121         log_sys("ошибка вызова функции gethostname");
```

[98–100] Демон не принимает аргументов командной строки, поэтому, если значение `argc` не равно 1, мы вызываем `err_quit`, которая выводит сообщение об ошибке и завершает работу приложения. Далее вызывается функция `daemonize` из листинга 13.1, которая переводит процесс в режим демона. С этого момента мы уже не можем выводить сообщения на стандартное устройство вывода сообщений об ошибках – вместо этого мы должны выводить их в журнал.

[101–112] Мы будем игнорировать сигнал `SIGPIPE`. Запись будет выполняться в дескриптор сокета, и нам совершенно не нужно, чтобы ошибка записи приводила к генерации сигнала `SIGPIPE`, действие по умолчанию для которого заключается в завершении процесса. Далее мы включаем сигналы `SIGHUP` и `SIGTERM` в маску сигналов потока. Все потоки, которые будут запущены впоследствии, унаследуют эту маску сигналов. Мы будем использовать сигнал `SIGHUP`, чтобы сообщить демону о необходимости перечитать конфигурационный файл, а `SIGTERM` – чтобы сообщить ему о том, что он должен корректно завершить свою работу. Функция `init_request` вызывается, чтобы инициализировать прием запросов на выполнение заданий и убедиться, что запущена единственная копия демона, а затем вызывается функция `init_printer`, которая инициализирует информацию о принтере (вскоре мы рассмотрим обе функции).

[113–121] Если платформа определяет символ `_SC_HOST_NAME_MAX`, мы вызываем функцию `sysconf`, чтобы получить максимально возможный размер имени хоста. Если функция `sysconf` завершается неудачей или значение этого предела не определено, мы будем использовать константу `HOST_NAME_MAX` – это лучшее, что можно сделать в данной ситуации. На некоторых платформах эта константа может быть уже определена, но если это не так, мы будем использовать значение, указанное в заголовочном файле `print.h`.

Далее мы выделяем память для хранения имени хоста и вызываем функцию gethostname, чтобы получить его.

```
122     if ((err = getaddrlist(host, "print", &ailist)) != 0) {
123         log_quit("ошибка функции getaddrinfo: %s", gai_strerror(err));
124         exit(1);
125     }
126     FD_ZERO(&rendezvous);
127     maxfd = -1;
128     for (aip = ailist; aip != NULL; aip = aip->ai_next) {
129         if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
130           aip->ai_addrlen, QLEN)) >= 0) {
131             FD_SET(sockfd, &rendezvous);
132             if (sockfd > maxfd)
133                 maxfd = sockfd;
134         }
135     }
136     if (maxfd == -1)
137         log_quit("служба отключена");

138     pwdp = getpwnam("lp");
139     if (pwdp == NULL)
140         log_sys("отсутствует пользователь lp");
141     if (pwdp->pw_uid == 0)
142         log_quit("lp является привилегированным пользователем");
143     if (setuid(pwdp->pw_uid) < 0)
144         log_sys("невозможно сменить идентификатор на пользователя lp");
```

[122–135] Далее мы пытаемся найти сетевой адрес, который мог бы использоваться демоном для предоставления службы печати. Мы очищаем набор дескрипторов rendezvous, который будет передаваться функции select в ожидании поступления запросов на соединение. Мы инициализируем переменную максимального номера дескриптора значением –1, в результате первый же дескриптор, который нам удастся получить, наверняка будет больше, чем maxfd. Для каждого из полученных сетевых адресов мы вызываем функцию initserver (листинг 16.9), которая размещает и инициализирует сокет. В случае успешного завершения initserver мы добавляем полученный от нее дескриптор к набору rendevouz и, если он больше максимального, заносим в переменную maxfd номер этого дескриптора.

[136–137] Если после просмотра списка структур addrinfo значение переменной maxfd осталось равным -1, мы не можем запустить службу печати, поэтому мы выводим сообщение в журнал и завершаем работу.

[138–144] Чтобы назначить сокету привилегированный номер порта, демон должен обладать привилегиями суперпользователя. Теперь, когда сокеты уже созданы, мы можем понизить привилегии демона, заменив существующий идентификатор пользователя на идентификатор пользователя lp (вопросы безопасности обсуждались в разделе 21.4). Мы должны следовать принципам минимизации привилегий, чтобы избежать появления потенциальных уязвимостей в демоне. Мы вызываем функцию getpwnam, чтобы найти в файле паролей запись, связанную с пользователем lp. Если

такой пользователь не будет найден или если он существует, но его идентификатор совпадает с идентификатором суперпользователя, мы выводим сообщение в журнал и завершаем работу. В противном случае мы вызываем функцию setuid, чтобы заменить реальный и эффективный идентификаторы пользователя процесса на идентификатор пользователя lp. Чтобы не подвергать систему опасности, мы предпочитаем вообще не предоставлять услуги, если невозможно понизить привилегии.

```
145     pthread_create(&tid, NULL, printer_thread, NULL);
146     pthread_create(&tid, NULL, signal_thread, NULL);
147     build_qonstart();
148     log_msg("демон инициализирован");
149     for (;;) {
150         rset = rendezvous;
151         if (select(maxfd+1, &rset, NULL, NULL, NULL) < 0)
152             log_sys("ошибка вызова функции select");
153         for (i = 0; i <= maxfd; i++) {
154             if (FD_ISSET(i, &rset)) {
155                 /*
156                  * Принять соединение
157                  * и обработать запрос.
158                  */
159                 sockfd = accept(i, NULL, NULL);
160                 if (sockfd < 0)
161                     log_ret("ошибка вызова функции accept");
162                 pthread_create(&tid, NULL, client_thread,
163                                (void *)sockfd);
164             }
165         }
166     }
167     exit(1);
168 }
```

[145–148] Мы дважды вызываем функцию pthread_create, чтобы запустить поток для обработки сигналов и поток для взаимодействия с принтером. (Разместив все операции с принтером в одном потоке, мы можем упростить алгоритмы блокировки структур данных, связанных с принтером.) Затем мы вызываем build_qonstart, чтобы отыскать в каталоге /var/spool/printer подкаталоги, соответствующие заданиям, ожидающим обработки. Для каждого задания, найденного на диске, будет создана структура, что позволит потоку взаимодействия с принтером узнать, какие файлы должны быть отправлены принтеру. На этом мы заканчиваем инициализацию демона и выводим в журнал сообщение, которое говорит о том, что инициализация прошла успешно.

[149–168] Мы копируем набор дескрипторов rendevouz в переменную rset и вызываем select, чтобы дождаться момента, когда один из дескрипторов станет доступен для чтения. Мы используем копию rendevouz, потому что select модифицирует переданный ей набор дескрипторов и оставляет в наборе только те дескрипторы, которые соответствуют наступившему событию.

Поскольку сокеты инициализированы для использования сервером, доступность дескрипторов для чтения означает, что поступил запрос на соединение. После возврата из функции `select` мы проверяем, какие дескрипторы из набора `rset` доступны для чтения. Если такие дескрипторы будут найдены, они передаются функции `accept`, чтобы принять соединение. Если функция `accept` терпит неудачу, мы выводим в журнал сообщение и продолжаем проверку набора в поисках дескрипторов, доступных для чтения. Иначе запускается поток, который займется обслуживанием соединения с клиентом. Функция `main` входит в бесконечный цикл, принимая запросы и передавая их для обработки другим потокам, – она никогда не должна дойти до исполнения строки с вызовом функции `exit`.

```
169  /*
170   * Инициализировать файл с идентификатором задания. Установить блокировку
171   * для записи, чтобы предотвратить запуск других копий демона.
172   *
173   * БЛОКИРОВКИ: отсутствуют, за исключением блокировки на файл задания.
174   */
175  void
176  init_request(void)
177  {
178      int n;
179      char name[FILENMSZ];

180      sprintf(name, "%s/%s", SPOOLDIR, JOBFILE);
181      jobfd = open(name, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
182      if (write_lock(jobfd, 0, SEEK_SET, 0) < 0)
183          log_quit("демон уже запущен");

184     /*
185      * Повторно использовать буфер с именем файла для счетчика заданий.
186      */
187     if ((n = read(jobfd, name, FILENMSZ)) < 0)
188         log_sys("невозможно прочитать содержимое файла задания");
189     if (n == 0)
190         nextjob = 1;
191     else
192         nextjob = atol(name);
193  }
```

[169–183] Функция `init_request` выполняет два действия: устанавливает блокировку на файл задания `/var/spool/printer/jobno` и читает его содержимое, чтобы определить номер, который будет присвоен следующему заданию. Мы устанавливаем блокировку на весь файл, которая служит признаком того, что демон уже запущен. Если кто-либо попытается запустить еще одну копию демона печати, когда демон уже работает, эта дополнительная копия не сможет установить блокировку на файл и завершит работу. (Этот прием мы использовали в листинге 13.2, а макрос `write_lock` был описан в разделе 14.3.)

[184–193] Файл задания содержит номер следующего задания в виде строки ASCII. Если файл был только что создан и поэтому не содержит данных, мы записываем в переменную `nextjob` число 1. В противном случае с помощью

функции atol выполняется преобразование строки в целое число, которое будет использовано в качестве номера следующего задания. Мы оставляем дескриптор jobfd открытым, чтобы можно было обновлять номер в файле по мере поступления новых заданий. Мы не можем закрыть этот файл, потому что тогда блокировка, которую мы только что установили, будет снята автоматически.

В системах, в которых длинное целое представлено 64 битами, для представления наибольшего длинного целого числа в виде строки потребуется буфер длиною не менее 21 байта. Для этих целей вполне подходит буфер с именем файла, потому что константа FILENMSZ определена в заголовочном файле print.h со значением 64.

```
194  /*
195   * Инициализация информации о принтере.
196   *
197   * БЛОКИРОВКИ: отсутствуют.
198   */
199  void
200  init_printer(void)
201  {
202      printer = get_printaddr();
203      if (printer == NULL) {
204          log_msg("печатающее устройство не найдено");
205          exit(1);
206      }
207      printer_name = printer->ai_canonname;
208      if (printer_name == NULL)
209          printer_name = "printer";
210      log_msg("имя принтера: %s", printer_name);
211  }

212  /*
213   * Обновить идентификатор задания в файле.
214   *
215   * БЛОКИРОВКИ: отсутствуют.
216   */
217  void
218  update_jobno(void)
219  {
220      char buf[32];

221      lseek(jobfd, 0, SEEK_SET);
222      sprintf(buf, "%ld", nextjob);
223      if (write(jobfd, buf, strlen(buf)) < 0)
224          log_sys("невозможно обновить файл с номером задания");
225  }
```

[194–211] Функция `init_printer` используется для того, чтобы установить имя принтера и его сетевой адрес. Адрес принтера мы получаем с помощью `get_printaddr` (из `util.c`). Если вызов этой функции завершается неудачей, мы выводим сообщение в журнал и завершаем работу. Нельзя использовать здесь функцию `log_sys`, потому что `get_printaddr` может завершаться с ошибкой, не записывая код ошибки в переменную `errno`. Однако, когда `get_printaddr` терпит неудачу и при этом изменяет содержимое переменной `errno`, она выводит в журнал собственное сообщение об ошибке. Далее мы принимаем имя принтера из поля `ai_canonname` структуры `addrinfo`. Если это поле пустое, мы записываем имя принтера по умолчанию – `printer`. Обратите внимание: имя используемого принтера выводится в журнал, чтобы помочь администраторам в диагностике возможных проблем с системой печати.

[212–225] Функция `update_jobno` записывает номер следующего задания в файл `/var/spool/printer/jobno`. Прежде всего мы устанавливаем текущую позицию записи в начало файла. Затем преобразуем целочисленный номер задания в строку и записываем ее в файл. Если операция записи не увенчалась успехом, мы выводим сообщение об ошибке в журнал и завершаем работу.

```
226  /*
227   * Получить номер следующего задания.
228   *
229   * БЛОКИРОВКИ: запирает и отпирает joblock.
230   */
231  long
232  get_newjobno(void)
233  {
234    long jobid;

235    pthread_mutex_lock(&joblock);
236    jobid = nextjob++;
237    if (nextjob <= 0)
238        nextjob = 1;
239    pthread_mutex_unlock(&joblock);
240    return(jobid);
241  }

242  /*
243   * Добавляет в очередь новое задание. После этого посылает
244   * потоку принтера сигнал о том, что появилось новое задание.
245   *
246   * БЛОКИРОВКИ: захватывает и отпускает joblock.
247   */
248  void
249  add_job(struct printreq *reqp, long jobid)
250  {
251    struct job *jp;

252    if ((jp = malloc(sizeof(struct job))) == NULL)
253        log_sys("ошибка вызова функции malloc");
254    memcpy(&jp->req, reqp, sizeof(struct printreq));
```

[226–241] Функция `get_newjobno` используется для того, чтобы получить номер следующего задания. Сначала мы запираем мьютекс `joblock`. Увеличиваем значение переменной `nextjob` на 1 и обрабатываем ситуацию ее переполнения. Затем мы отпираем мьютекс и возвращаем значение, которое имела переменная `nextjob` до ее увеличения. Функция `get_newjobno` может вызываться из нескольких потоков одновременно, поэтому мы должны упорядочить доступ к номеру следующего задания, чтобы каждый из потоков получил свой, уникальный номер задания. (На рис. 11.4 мы уже показывали, что может произойти, если не организовать поочередный доступ к данным из нескольких потоков.)

[242–254] Функция `add_job` используется для добавления нового запроса в конец очереди заданий печати. Функция начинается с выделения памяти под структуру `job`. Если память не может быть выделена, мы выводим сообщение об ошибке в журнал и завершаем работу. К этому моменту запрос на печать уже сохранен на диске, поэтому он будет принят демоном после перезапуска. После выделения памяти для структуры мы копируем структуру запроса `printreq`, полученную от клиента, в структуру `job`. В файле `print.h` мы видели, что структура `job` состоит из пары указателей, идентификатора задания и копии структуры `printreq`, полученной от клиентской утилиты `print`.

```
255     jp->jobid = jobid;
256     jp->next = NULL;
257     pthread_mutex_lock(&joblock);
258     jp->prev = jobtail;
259     if (jobtail == NULL)
260         jobhead = jp;
261     else
262         jobtail->next = jp;
263     jobtail = jp;
264     pthread_mutex_unlock(&joblock);
265     pthread_cond_signal(&jobwait);
266   }
267   /*
268    * Вставить задание в начало списка.
269    *
270    * БЛОКИРОВКИ: запирает и отпирает joblock.
271    */
272   void
273   replace_job(struct job *jp)
274   {
275     pthread_mutex_lock(&joblock);
276     jp->prev = NULL;
277     jp->next = jobhead;
278     if (jobhead == NULL)
279         jobtail = jp;
280     else
281         jobhead->prev = jp;
282     jobhead = jp;
283     pthread_mutex_unlock(&joblock);
284   }
```

[255–266] Мы сохраняем идентификатор задания и запираем мьютекс, чтобы гарантировать исключительность доступа к списку заданий печати. Добавление новой структуры будет производиться в конец списка. Мы записываем в указатель на предыдущий элемент новой структуры адрес последнего задания в списке. Если список пуст, адрес новой структуры записывается в `jobhead`. В противном случае адрес новой структуры записывается в указатель на следующий элемент последней структуры в списке. После этого адрес новой структуры записывается в `jobtail`. В заключение мы отпираем мьютекс и посылаем сигнал потоку, обслуживающему принтер, чтобы известить его о прибытии нового задания.

[267–284] Функция `replace_job` используется для того, чтобы вставить задание в начало списка. Мы запираем мьютекс `joblock`, записываем значение `NULL` в указатель на предыдущий элемент списка в добавляемой структуре, а в указатель на следующий элемент записывается адрес начала списка. Если список пуст, адрес добавляемой структуры записывается в `jobtail`. В противном случае адрес добавляемой структуры записывается в указатель на предыдущий элемент первой структуры в списке. После этого адрес добавляемой структуры записывается в `jobhead`. В заключение мы отпираем мьютекс.

```
285  /*
286   * Удалить задание из очереди.
287   *
288   * БЛОКИРОВКИ: вызывающая функция должна запереть joblock.
289   */
290  void
291  remove_job(struct job *target)
292  {
293    if (target->next != NULL)
294        target->next->prev = target->prev;
295    else
296        jobtail = target->prev;
297    if (target->prev != NULL)
298        target->prev->next = target->next;
299    else
300        jobhead = target->next;
301  }

302  /*
303   * Проверить при запуске каталог очереди на наличие заданий.
304   *
305   * БЛОКИРОВКИ: отсутствуют.
306   */
307  void
308  build_qonstart(void)
309  {
310    int    fd, err, nr;
311    long   jobid;
312    DIR    *dirp;
313    struct dirent *entp;
314    struct printreq req;
```

```
315     char   dname[FILENMSZ], fname[FILENMSZ];
316     sprintf(dname, "%s/%s", SPOOLDIR, REQDIR);
317     if ((dirp = opendir(dname)) == NULL)
318         return;
```

[285–301] Функция remove_job удаляет из очереди задание, указатель на которое передается в функцию. Вызывающая функция должна запереть мьютекс joblock перед вызовом remove_job. Если указатель на следующий элемент очереди не является пустым, мы записываем в указатель на предыдущий элемент следующего элемента списка адрес элемента, предшествующего удаляемому. В противном случае задание является последним в очереди, поэтому адрес предыдущего элемента списка записывается в переменную jobtail. Если указатель на предыдущий элемент не пустой, мы записываем в указатель на следующий элемент предыдущего элемента списка адрес элемента, следующего за удаляемым. В противном случае задание является первым в списке, поэтому адрес следующего элемента записывается в переменную jobhead.

[302–318] При запуске демон вызывает функцию build_qonstart, которая собирает в памяти список заданий из файлов, сохраняемых в каталоге /var/spool/printer/reqs. Если невозможно открыть этот каталог, то задания печати отсутствуют, поэтому мы просто возвращаем управление.

```
319     while ((entp = readdir(dirp)) != NULL) {
320         /*
321          * Пропустить каталоги "." и ".."
322          */
323         if (strcmp(entp->d_name, ".") == 0 ||
324           strcmp(entp->d_name, "..") == 0)
325             continue;
326         /*
327          * Прочитать структуру запроса.
328          */
329         sprintf(fname, "%s/%s/%s", SPOOLDIR, REQDIR, entp->d_name);
330         if ((fd = open(fname, O_RDONLY)) < 0)
331             continue;
332         nr = read(fd, &req, sizeof(struct printreq));
333         if (nr != sizeof(struct printreq)) {
334             if (nr < 0)
335                 err = errno;
336             else
337                 err = EIO;
338             close(fd);
339             log_msg("build_qonstart: невозможно прочитать %s: %s",
340                     fname, strerror(err));
341             unlink(fname);
342             sprintf(fname, "%s/%s/%s", SPOOLDIR, DATADIR,
343                     entp->d_name);
344             unlink(fname);
345             continue;
346         }
```

```
347         jobid = atol(entp->d_name);
348         log_msg("в очередь добавлено задание %ld", jobid);
349         add_job(&req, jobid);
350     }
351     closedir(dirp);
352   }
```

[319–325] Мы читаем все записи из каталога, по одной за раз. При этом пропускаем каталоги «.» и «..».

[326–346] Для каждой записи собирается полный путь к файлу, который затем открывается для чтения. Если операция открытия терпит неудачу, мы просто пропускаем этот файл. В противном случае читаем содержимое структуры `printreq`, сохраненное в файле. Если прочитать структуру целиком не удается, мы закрываем файл, выводим в журнал сообщение об ошибке и удаляем файл. После этого мы собираем полное имя соответствующего файла с данными и также удаляем его.

[347–352] Если нам удалось прочитать структуру `printreq`, мы преобразуем имя файла в идентификатор задания (имя файла представляет собой идентификатор задания), выводим сообщение в журнал и добавляем запрос в очередь заданий печати. Когда все записи будут прочитаны и функция `readdir` вернет значение `NULL`, мы закрываем каталог и возвращаем управление.

```
353  /*
354   * Принять задание печати от клиента.
355   *
356   * БЛОКИРОВКИ: отсутствуют.
357   */
358  void *
359  client_thread(void *arg)
360  {
361    int       n, fd, sockfd, nr, nw, first;
362    long      jobid;
363    pthread_t tid;
364    struct    printreq req;
365    struct    printresp res;
366    char      name[FILENMSZ];
367    char      buf[IOBUFSZ];

368    tid = pthread_self();
369    pthread_cleanup_push(client_cleanup, (void *)tid);
370    sockfd = (int)arg;
371    add_worker(tid, sockfd);

372    /*
373     * Прочитать заголовок запроса.
374     */
375    if ((n = treadn(sockfd, &req, sizeof(struct printreq), 10)) !=
376      sizeof(struct printreq)) {
377        res.jobid = 0;
378        if (n < 0)
379            res.retcode = htonl(errno);
```

```
380         else
381             res.retcode = htonl(EIO);
382         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
383         writen(sockfd, &res, sizeof(struct printresp));
384         pthread_exit((void *)1);
385     }
```

[353–371] Функция `client_thread` – это функция запуска потока, порождаемого функцией `main` при получении запроса на соединение от клиента. Задача этого потока заключается в том, чтобы принять от клиента файл, который должен быть напечатан. Для обработки каждого запроса, полученного от клиента, запускается отдельный поток.

Прежде всего необходимо установить обработчик завершения потока (обработчики завершения обсуждались в разделе 11.5). В качестве обработчика завершения потока используется функция `client_cleanup`, которую мы рассмотрим несколько позже. Она получает единственный аргумент – идентификатор потока. Затем мы вызываем `add_worker`, чтобы создать структуру `worker_thread` и добавить ее в список активных клиентских потоков.

[372–385] На этом инициализация потока завершается, и мы переходим к чтению заголовка запроса клиента. Если от клиента было получено меньше данных, чем мы ожидаем, или в процессе чтения возникла какая-либо ошибка, мы отправляем клиенту сообщение-ответ, в котором указываем причину ошибки, и вызовом `pthread_exit` завершаем работу потока.

```
386     req.size = ntohl(req.size);
387     req.flags = ntohl(req.flags);

388     /*
389      * Создать файл данных.
390      */
391     jobid = get_newjobno();
392     sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jobid);
393     if ((fd = creat(name, FILEPERM)) < 0) {
394         res.jobid = 0;
395         if (n < 0)
396             res.retcode = htonl(errno);
397         else
398             res.retcode = htonl(EIO);
399         log_msg("client_thread: невозможно создать %s: %s", name,
400                 strerror(res.retcode));
401         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
402         writen(sockfd, &res, sizeof(struct printresp));
403         pthread_exit((void *)1);
404     }

405     /*
406      * Прочитать файл и сохранить его в каталоге очереди печати.
407      */
408     first = 1;
409     while ((nr = tread(sockfd, buf, IOBUFSZ, 20)) > 0) {
410         if (first) {
```

```
411             first = 0;
412             if (strncmp(buf, "%!PS", 4) != 0)
413                 req.flags|= PR_TEXT;
414         }
```

[386–404] Мы преобразуем целочисленные поля заголовка в значения с порядком байтов, соответствующим аппаратной архитектуре, и вызываем `get_newjobno`, чтобы зарезервировать очередной идентификатор для данного запроса. Мы создаем файл с данными для печати под именем `var/spool/printer/data/`*`jobid`*, где *`jobid`* – это идентификатор задания для данного запроса. При создании файла мы задаем такие права доступа, которые не позволят всем остальным прочитать файл (константа `FILEPERM` определена как `S_IRUSR|S_IWUSR` в заголовочном файле `print.h`). Если файл создать не удалось, мы выводим в журнал сообщение об ошибке, отправляем код ошибки клиенту и завершаем работу потока вызовом функции `pthread_exit`.

[405–414] Мы принимаем от клиента содержимое файла, которое записываем в свою копию файла. Но прежде чем записать что-либо, на первой итерации цикла нам необходимо проверить, является ли принимаемый файл файлом формата PostScript. Если содержимое файла не начинается с последовательности `%!PS`, мы предполагаем, что файл содержит обычный текст, и устанавливаем флаг `PR_TEXT` в заголовке запроса. (Мы уже говорили, что клиент тоже может установить этот флаг, запустив утилиту `print` с ключом `-t`.) Хотя файлы формата PostScript не обязательно должны начинаться с данной последовательности, тем не менее руководство по форматированию документов [Adobe Systems 1999] настоятельно это рекомендует.

```
415         nw = write(fd, buf, nr);
416         if (nw != nr) {
417             if (nw < 0)
418                 res.retcode = htonl(errno);
419             else
420                 res.retcode = htonl(EIO);
421             log_msg("client_thread: невозможно записать в %s: %s", name,
422                     strerror(res.retcode));
423             close(fd);
424             strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
425             writen(sockfd, &res, sizeof(struct printresp));
426             unlink(name);
427             pthread_exit((void *)1);
428         }
429     }
430     close(fd);

431     /*
432      * Создать управляющий файл.
433      */
434     sprintf(name, "%s/%s/%ld", SPOOLDIR, REQDIR, jobid);
435     fd = creat(name, FILEPERM);
436     if (fd < 0) {
```

```
437         res.jobid = 0;
438         if (n < 0)
439             res.retcode = htonl(errno);
440         else
441             res.retcode = htonl(EIO);
442         log_msg("client_thread: невозможно создать %s: %s", name,
443                 strerror(res.retcode));
444         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
445         writen(sockfd, &res, sizeof(struct printresp));
446         sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jobid);
447         unlink(name);
448         pthread_exit((void *)1);
449     }
```

[415–430] Мы записываем в файл данные, полученные от клиента. Если операция записи терпит неудачу, мы выводим в журнал сообщение, закрываем дескриптор файла данных, отправляем сообщение об ошибке клиенту, удаляем файл с данными и завершаем работу потока вызовом функции `pthread_exit`.

По окончании приема всех данных, которые должны быть напечатаны, мы закрываем дескриптор файла с данными.

[431–449] Затем мы создаем файл `/var/spool/printer/reqs/`*jobid*, в котором будет храниться содержимое запроса на печать. Если создать файл не удалось, мы выводим в журнал сообщение, отправляем сообщение об ошибке клиенту, удаляем файл с данными и завершаем работу потока.

```
450     nw = write(fd, &req, sizeof(struct printreq));
451     if (nw != sizeof(struct printreq)) {
452         res.jobid = 0;
453         if (nw < 0)
454             res.retcode = htonl(errno);
455         else
456             res.retcode = htonl(EIO);
457         log_msg("client_thread: невозможно записать в %s: %s", name,
458                 strerror(res.retcode));
459         close(fd);
460         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
461         writen(sockfd, &res, sizeof(struct printresp));
462         unlink(name);
463         sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jobid);
464         unlink(name);
465         pthread_exit((void *)1);
466     }
467     close(fd);
468     /*
469      * Отправить ответ клиенту.
470      */
471     res.retcode = 0;
472     res.jobid = htonl(jobid);
473     sprintf(res.msg, "request ID %ld", jobid);
```

```
474     writen(sockfd, &res, sizeof(struct printresp));
475     /*
476      * Оповестить поток обслуживания принтера и корректно выйти.
477      */
478     log_msg("в очередь добавлено задание %ld", jobid);
479     add_job(&req, jobid);
480     pthread_cleanup_pop(1);
481     return((void *)0);
482   }
```

[450–466] Мы записываем структуру `printreq` в управляющий файл. В случае ошибки выводим в журнал сообщение, закрываем дескриптор управляющего файла, отправляем клиенту сообщение об ошибке, удаляем файл с данными и управляющий файл и завершаем работу потока.

[467–474] Мы закрываем дескриптор управляющего файла и отправляем клиенту сообщение об успешной постановке задания в очередь вместе с идентификатором задания (`retcode = 0`).

[475–482] Мы вызываем `add_job`, чтобы добавить принятое задание в очередь, и `pthread_cleanup_pop`, чтобы освободить занятые ресурсы. Работа потока завершается оператором `return`.

Обратите внимание: перед завершением потока мы должны закрыть все дескрипторы файлов, которые более не понадобятся. В отличие от процедуры завершения процесса, при завершении потока дескрипторы файлов не закрываются автоматически, если в процессе остается еще хотя бы один поток. Поэтому, если мы не будем закрывать дескрипторы, то рано или поздно столкнемся с нехваткой ресурсов.

```
483   /*
484    * Добавить новый поток в список рабочих потоков.
485    *
486    * БЛОКИРОВКИ: запирает и отпирает workerlock.
487    */
488   void
489   add_worker(pthread_t tid, int sockfd)
490   {
491     struct worker_thread *wtp;
492     if ((wtp = malloc(sizeof(struct worker_thread))) == NULL) {
493         log_ret("add_worker: ошибка вызова функции malloc");
494         pthread_exit((void *)1);
495     }
496     wtp->tid = tid;
497     wtp->sockfd = sockfd;
498     pthread_mutex_lock(&workerlock);
499     wtp->prev = NULL;
500     wtp->next = workers;
501     if (workers == NULL)
502         workers = wtp;
503     else
504         workers->prev = wtp;
```

```
505     pthread_mutex_unlock(&workerlock);
506   }
507   /*
508    * Завершить все имеющиеся рабочие потоки.
509    *
510    * БЛОКИРОВКИ: запирает и отпирает workerlock.
511    */
512   void
513   kill_workers(void)
514   {
515     struct worker_thread *wtp;
516     pthread_mutex_lock(&workerlock);
517     for (wtp = workers; wtp != NULL; wtp = wtp->next)
518         pthread_cancel(wtp->tid);
519     pthread_mutex_unlock(&workerlock);
520   }
```

[483–506] **Функция** `add_worker` **добавляет новую структуру** `worker_thread` **в список активных потоков. Мы выделяем память для структуры, инициализируем ее, запираем мьютекс** `workerlock`**, добавляем структуру в начало списка и отпираем мьютекс.**

[507–520] **Функция** `kill_workers` **обходит список рабочих потоков и пытается завершить их один за другим. На время обхода списка мы запираем мьютекс** `workerlock`**. Мы уже говорили, что функция** `pthread_cancel` **просто посылает запрос на завершение потока, а фактическое завершение потока произойдет лишь тогда, когда он достигнет ближайшей точки выхода.**

```
521   /*
522    * Процедура выхода для рабочего потока.
523    *
524    * БЛОКИРОВКИ: запирает и отпирает workerlock.
525    */
526   void
527   client_cleanup(void *arg)
528   {
529     struct worker_thread *wtp;
530     pthread_t            tid;
531     tid = (pthread_t)arg;
532     pthread_mutex_lock(&workerlock);
533     for (wtp = workers; wtp != NULL; wtp = wtp->next) {
534         if (wtp->tid == tid) {
535             if (wtp->next != NULL)
536                 wtp->next->prev = wtp->prev;
537             if (wtp->prev != NULL)
538                 wtp->prev->next = wtp->next;
539             else
540                 workers = wtp->next;
541             break;
542         }
```

```
543     }
544     pthread_mutex_unlock(&workerlock);
545     if (wtp != NULL) {
546         close(wtp->sockfd);
547         free(wtp);
548     }
549 }
```

[521–543] Функция `client_cleanup` – это обработчик выхода для рабочих потоков, которые занимаются взаимодействием с клиентами. Она вызывается, когда поток вызывает функцию `pthread_exit` или `pthread_cleanup_pop` с ненулевым аргументом или откликается на запрос о принудительном завершении. В качестве аргумента ей передается идентификатор потока, завершающего работу.

Мы запираем мьютекс `workerlock` и обходим список в поисках потока с заданным идентификатором. Когда соответствующий поток будет найден, мы удаляем структуру из списка и прекращаем поиск.

[544–549] Мы отпираем мьютекс `workerlock`, закрываем дескриптор сокета, использовавшийся для взаимодействия с клиентом, и освобождаем память, занимаемую структурой `worker_thread`.

Поскольку мы пытаемся запереть мьютекс `workerlock`, если поток достигает точки выхода раньше, чем функция `kill_workers` успеет обойти весь список, то нам придется ждать, пока `kill_workers` не отопрет мьютекс, и лишь после этого мы сможем продолжить работу.

```
550 /*
551  * Обслуживание сигналов.
552  *
553  * БЛОКИРОВКИ: запирает и отпирает configlock.
554  */
555 void *
556 signal_thread(void *arg)
557 {
558   int err, signo;
559   for (;;) {
560       err = sigwait(&mask, &signo);
561       if (err != 0)
562           log_quit("ошибка вызова функции sigwait: %s", strerror(err));
563       switch (signo) {
564       case SIGHUP:
565           /*
566            * Запланировать чтение конфигурационного файла.
567            */
568           pthread_mutex_lock(&configlock);
569           reread = 1;
570           pthread_mutex_unlock(&configlock);
571           break;
572       case SIGTERM:
573           kill_workers();
574           log_msg("завершение по сигналу %s", strsignal(signo));
```

```
575             exit(0);
576         default:
577             kill_workers();
578             log_quit("принят неожиданный сигнал %d", signo);
579         }
580     }
581 }
```

[550–563] Функция `signal_thread` — это функция запуска потока, ответственного за обработку сигналов. В маску сигналов, которую мы инициализировали в функции `main`, были включены сигналы `SIGHUP` и `SIGTERM`. Здесь мы вызываем функцию `sigwait`, ожидая доставки одного из этих сигналов. Если она возвращает признак ошибки, мы выводим в журнал сообщение об ошибке и завершаем работу.

[564–571] Если был принят сигнал `SIGHUP`, мы запираем мьютекс `configlock`, записываем число 1 в переменную `reread` и отпираем мьютекс. Таким образом мы сообщаем демону о необходимости перечитать содержимое конфигурационного файла в ближайшей итерации главного цикла.

[572–575] Если был принят сигнал `SIGTERM`, мы завершаем все рабочие потоки вызовом функции `kill_workers`, выводим в журнал сообщение и вызываем функцию `exit`, которая завершает работу процесса.

[576–581] Если был принят сигнал, которого мы не ожидали, то все рабочие потоки завершаются и вызывается функция `log_quit`, которая выводит в журнал сообщение и завершает процесс.

```
582 /*
583  * Добавить атрибут в заголовок IPP.
584  *
585  * БЛОКИРОВКИ: отсутствуют.
586  */
587 char *
588 add_option(char *cp, int tag, char *optname, char *optval)
589 {
590     int n;
591     union {
592         int16_t s;
593         char c[2];
594     } u;
595     *cp++ = tag;
596     n = strlen(optname);
597     u.s = htons(n);
598     *cp++ = u.c[0];
599     *cp++ = u.c[1];
600     strcpy(cp, optname);
601     cp += n;
602     n = strlen(optval);
603     u.s = htons(n);
604     *cp++ = u.c[0];
```

```
605     *cp++ = u.c[1];
606     strcpy(cp, optval);
607     return(cp + n);
608   }
```

[582–594] Функция `add_option` используется для добавления атрибута в заголовок IPP, который будет передан принтеру. На рис. 21.3 было показано, что формат атрибута состоит из 1 байта признака, описывающего тип атрибута, за которым следуют 2-байтное целое в двоичном формате, представляющее длину имени атрибута, имя атрибута, размер значения атрибута и, наконец, само значение.

Протокол IPP не предполагает какого-либо выравнивания целых чисел в двоичном представлении, имеющихся в заголовке. Некоторые аппаратные архитектуры, такие как SPARC, не могут хранить целые числа, начиная с произвольного адреса. Это означает, что мы не можем сохранить целое число в заголовке простым приведением типа адреса в заголовке, куда должно быть записано число, к типу указателя на `int16_t`. Вместо этого мы должны скопировать число как строку, байт за байтом. По этой причине мы определили объединение (`union`), содержащее 16-битное целое и 2 байта.

[595–608] Мы сохраняем признак атрибута в заголовке и преобразуем значение длины имени атрибута в значение с сетевым порядком байтов. Далее мы побайтно копируем в заголовок длину имени и само имя атрибута. Мы повторяем тот же процесс для значения атрибута и возвращаем адрес в заголовке, с которого должен начинаться следующий раздел заголовка.

```
609   /*
610    * Единственный поток, который занимается взаимодействием с принтером.
611    *
612    * БЛОКИРОВКИ: запирает и отпирает joblock и configlock.
613    */
614   void *
615   printer_thread(void *arg)
616   {
617     struct job   *jp;
618     int          hlen, ilen, sockfd, fd, nr, nw;
619     char         *icp, *hcp;
620     struct       ipp_hdr *hp;
621     struct       stat sbuf;
622     struct iovec iov[2];
623     char         name[FILENMSZ];
624     char         hbuf[HBUFSZ];
625     char         ibuf[IBUFSZ];
626     char         buf[IOBUFSZ];
627     char         str[64];

628     for (;;) {
629         /*
630          * Получить задание печати.
631          */
632         pthread_mutex_lock(&joblock);
```

```
633         while (jobhead == NULL) {
634             log_msg("printer_thread: ожидание...");
635             pthread_cond_wait(&jobwait, &joblock);
636         }
637         remove_job(jp = jobhead);
638         log_msg("printer_thread: принято задание %ld", jp->jobid);
639         pthread_mutex_unlock(&joblock);

640         update_jobno();
```

[609–627] Функция `printer_thread` — это функция запуска потока, который обеспечивает взаимодействие с сетевым принтером. Переменные `icp` и `ibuf` будут использоваться для сборки заголовка IPP, а переменные `hcp` и `hbuf` — для сборки заголовка HTTP. Заголовки должны собираться в отдельных буферах. Заголовок HTTP включает в себя поле длины в формате ASCII, но пока не собран заголовок IPP, значение этого поля неизвестно. Мы будем использовать единственный вызов `writev` для записи обоих заголовков сразу.

[628–640] Поток взаимодействия с принтером входит в бесконечный цикл и ожидает заданий, которые должны быть переданы принтеру. Доступ к списку заданий осуществляется под защитой мьютекса `joblock`. Если очередь заданий пуста, мы вызываем функцию `pthread_cond_wait`, чтобы дождаться хотя бы одного задания. Когда задание появится в очереди, мы удаляем его из списка вызовом функции `remove_job`. В этот момент список все еще находится под защитой мьютекса, поэтому мы отпираем его и вызываем `update_jobno`, чтобы записать номер очередного задания в файл `/var/spool/printer/jobno`.

```
641         /*
642          * Проверить наличие изменений в конфигурационном файле.
643          */
644         pthread_mutex_lock(&configlock);
645         if (reread) {
646             freeaddrinfo(printer);
647             printer = NULL;
648             printer_name = NULL;
649             reread = 0;
650             pthread_mutex_unlock(&configlock);
651             init_printer();
652         } else {
653             pthread_mutex_unlock(&configlock);
654         }

655         /*
656          * Отправить задание принтеру.
657          */
658         sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jp->jobid);
659         if ((fd = open(name, O_RDONLY)) < 0) {
660             log_msg("задание %ld отменено - невозможно открыть %s: %s",
661                     jp->jobid, name, strerror(errno));
662             free(jp);
```

```
663             continue;
664         }
665         if (fstat(fd, &sbuf) < 0) {
666             log_msg("задание %ld отменено - ошибка fstat для %s: %s",
667                     jp->jobid, name, strerror(errno));
668             free(jp);
669             close(fd);
670             continue;
671         }
```

[641–654] Теперь, когда у нас имеется задание печати, мы поверяем наличие изменений в конфигурационном файле. Для этого мы запираем мьютекс `configlock` и проверяем значение переменной `reread`. Если она имеет ненулевое значение, мы освобождаем память, занимаемую списком `addrinfo` принтера, очищаем указатели, отпираем мьютекс и вызываем `init_printer`, чтобы провести повторную инициализацию информации о принтере. Так как после инициализации информации о принтере в функции `main` это единственное место в программе, где данная информация может быть изменена, нам не требуется какая-либо дополнительная синхронизация, кроме использования мьютекса `configlock` для доступа к переменной `reread`.

Обратите внимание: хотя в этой функции мы запираем и отпираем различные мьютексы, мы нигде не удерживаем их запертыми одновременно, поэтому нам не придется ломать голову над иерархией блокировок (раздел 11.6).

[655–671] Если невозможно открыть файл данных, в журнал выводится сообщение, освобождается память, занимаемая структурой `job`, и управление передается в начало цикла. После открытия файла мы вызываем функцию `fstat`, чтобы узнать размер файла. Если это не удается, мы выводим в журнал сообщение, освобождаем память, занимаемую структурой `job`, закрываем дескриптор файла и переходим в начало цикла.

```
672         if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
673             log_msg("задание %ld отложено - невозможно создать сокет: %s",
674                     jp->jobid, strerror(errno));
675             goto defer;
676         }
677         if (connect_retry(sockfd, printer->ai_addr,
678           printer->ai_addrlen) < 0) {
679           log_msg("задание %ld отложено - невозможно соединиться с принтером: %s",
680                   jp->jobid, strerror(errno));
681             goto defer;
682         }
683         /*
684          * Собрать заголовок IPP.
685          */
686         icp = ibuf;
687         hp = (struct ipp_hdr *)icp;
688         hp->major_version = 1;
689         hp->minor_version = 1;
690         hp->operation = htons(OP_PRINT_JOB);
```

```
691         hp->request_id = htonl(jp->jobid);
692         icp += offsetof(struct ipp_hdr, attr_group);
693         *icp++ = TAG_OPERATION_ATTR;
694         icp = add_option(icp, TAG_CHARSET, "attributes-charset",
695                          "utf-8");
696         icp = add_option(icp, TAG_NATULANG,
697                          "attributes-natural-language", "en-us");
698         sprintf(str, "http://%s:%d", printer_name, IPP_PORT);
699         icp = add_option(icp, TAG_URI, "printer-uri", str);
```

[672–682] Мы открываем сокет для взаимодействия с принтером. Если вызов функции `socket` терпит неудачу, мы переходим на метку `defer`, где освобождаем уже занятые ресурсы, и после задержки повторяем попытку. Если удалось создать сокет, мы вызываем функцию `connect_retry`, чтобы соединиться с принтером.

[683–699] Затем мы начинаем сборку заголовка IPP. В качестве операции назначается операция запроса на печать задания. С помощью функций `htons` и `htonl` мы преобразуем 2-байтный идентификатор операции 4-байтовый идентификатор задания из значений с порядком байтов, определяемым аппаратной архитектурой, в значения с сетевым порядком байтов. После начальной части заголовка мы вставляем признак начала блока атрибутов операции. С помощью функции `add_option` атрибуты добавляются в заголовок. В табл. 21.1 перечислены обязательные и опциональные атрибуты операции запроса на печать. Первые три из них являются обязательными. Мы определяем в качестве кодировки символов UTF-8, которая должна поддерживаться принтером. Естественный язык мы задаем как `en-us`, что соответствует американскому английскому. Еще один обязательный атрибут – универсальный идентификатор ресурса принтера (URI – Universal Resource Identifier). Мы определяем его как `http://`*`printer_name`*`:631`. (На самом деле следовало бы запросить у принтера список поддерживаемых им URI и выбрать один из этого списка, но это лишь усложнит данный пример, не добавляя много смысла.)

```
700         icp = add_option(icp, TAG_NAMEWOLANG,
701                          "requesting-user-name", jp->req.usernm);
702         icp = add_option(icp, TAG_NAMEWOLANG, "job-name",
703                          jp->req.jobnm);
704         if (jp->req.flags & PR_TEXT) {
705             icp = add_option(icp, TAG_MIMETYPE, "document-format",
706                              "text/plain");
707         } else {
708             icp = add_option(icp, TAG_MIMETYPE, "document-format",
709                              "application/postscript");
710         }
711         *icp++ = TAG_END_OF_ATTR;
712         ilen = icp - ibuf;
713         /*
714          * Собрать заголовок HTTP.
715          */
716         hcp = hbuf;
```

```
717         sprintf(hcp, "POST /%s/ipp HTTP/1.1\r\n", printer_name);
718         hcp += strlen(hcp);
719         sprintf(hcp, "Content-Length: %ld\r\n",
720                 (long)sbuf.st_size + ilen);
721         hcp += strlen(hcp);
722         strcpy(hcp, "Content-Type: application/ipp\r\n");
723         hcp += strlen(hcp);
724         sprintf(hcp, "Host: %s:%d\r\n", printer_name, IPP_PORT);
725         hcp += strlen(hcp);
726         *hcp++ = '\r';
727         *hcp++ = '\n';
728         hlen = hcp - hbuf;
```

[700–712] Атрибут `requesting-user-name` рекомендуется, но не является обязательным. Атрибут `job-name` также является опциональным. Мы уже говорили, что утилита `print` в качестве имени задания передает имя файла, который должен быть напечатан, что помогает пользователям разобраться в большом количестве заданий, ожидающих обработки. Последний атрибут, который мы добавляем, – это `document-format`. Если опустить его, принтер будет полагать, что формат документ соответствует формату принтера по умолчанию. Для принтеров PostScript это, скорее всего, будет формат PostScript, но некоторые принтеры могут определять формат автоматически и выбирать между PostScript и простым текстом или между PostScript и PCL (Printer Command Language – язык команд принтера компании Hewlett-Packard). Если установлен флаг `PR_TEXT`, формат документа определяется как `text/plain`, в противном случае – как `application/postscript`. Затем мы вставляем признак конца блока атрибутов и подсчитываем размер получившегося заголовка IPP.

[713–728] Теперь, когда размер заголовка IPP известен, мы можем приступить к сборке заголовка HTTP. Мы устанавливаем значение атрибута `Content-Length` равным сумме размеров заголовка IPPи печатаемого файла. В атрибут `Content-Type` записывается значение `application/ipp`. Конец заголовка HTTP отмечается символами возврата каретки и перевода строки.

```
729         /*
730          * Передать сначала заголовки, потом файл.
731          */
732         iov[0].iov_base = hbuf;
733         iov[0].iov_len = hlen;
734         iov[1].iov_base = ibuf;
735         iov[1].iov_len = ilen;
736         if ((nw = writev(sockfd, iov, 2)) != hlen + ilen) {
737             log_ret("невозможно передать данные принтеру");
738             goto defer;
739         }
740         while ((nr = read(fd, buf, IOBUFSZ)) > 0) {
741             if ((nw = write(sockfd, buf, nr)) != nr) {
742                 if (nw < 0)
743                     log_ret("невозможно передать данные принтеру");
744                 else
745                     log_msg("данные переданы частично (%d/%d)", nw, nr);
```

```
746                 goto defer;
747             }
748         }
749         if (nr < 0) {
750             log_ret("невозможно прочитать из %s", name);
751             goto defer;
752         }
753         /*
754          * Прочитать ответ принтера.
755          */
756         if (printer_status(sockfd, jp)) {
757             unlink(name);
758             sprintf(name, "%s/%s/%ld", SPOOLDIR, REQDIR, jp->jobid);
759             unlink(name);
760             free(jp);
761             jp = NULL;
762         }
```

[729–739] В первый элемент массива `iovec` записывается заголовок HTTP, а во второй – заголовок IPP. Затем с помощью функции `writev` оба заголовка отправляются принтеру. Если операция записи терпит неудачу, мы записываем в журнал сообщение и переходим на метку `defer`, где производится освобождение занятых ресурсов и выполняется задержка перед повторной попыткой.

[740–752] Затем мы отправляем принтеру файл с данными. Мы читаем содержимое файла порциями по `IOBUFSZ` байт и записываем в сокет, соединенный с принтером. Если какая-либо из операций чтения или записи терпит неудачу, мы записываем в журнал сообщение и переходим на метку `defer`.

[753–762] После передачи файла принтеру мы вызываем функцию `printer_status`, которая принимает ответ принтера на наш запрос. Если функция `printer_status` завершается успехом, она возвращает положительное значение, после чего мы удаляем файл с данными и управляющий файл. Затем мы освобождаем память, занимаемую структурой `job`, записываем в указатель на нее значение `NULL` и переходим к метке `defer`.

```
763 defer:
764         close(fd);
765         if (sockfd >= 0)
766             close(sockfd);
767         if (jp != NULL) {
768             replace_job(jp);
769             sleep(60);
770         }
771     }
772 }
773 /*
774  * Прочитать данные из принтера - возможно, увеличивая приемный буфер.
775  * Возвращает смещение конца данных в буфере или -1 в случае ошибки.
776  *
```

```
 777    * БЛОКИРОВКИ: отсутствуют.
 778    */
 779   ssize_t
 780   readmore(int sockfd, char **bpp, int off, int *bszp)
 781   {
 782     ssize_t nr;
 783     char    *bp = *bpp;
 784     int     bsz = *bszp;

 785     if (off >= bsz) {
 786         bsz += IOBUFSZ;
 787         if ((bp = realloc(*bpp, bsz)) == NULL)
 788             log_sys("readmore: невозможно увеличить размер буфера");
 789         *bszp = bsz;
 790         *bpp = bp;
 791     }
 792     if ((nr = tread(sockfd, &bp[off], bsz-off, 1)) > 0)
 793         return(off+nr);
 794     else
 795         return(-1);
 796   }
```

[763–772] На метке `defer` мы закрываем дескриптор файла с данными. Если дескриптор сокета открыт, мы закрываем его. В случае ошибки мы помещаем задание обратно в начало очереди заданий и выполняем задержку на 1 минуту. В случае успеха указатель `jp` будет содержать значение `NULL`, поэтому мы просто возвращаемся к началу цикла, чтобы обработать следующее задание печати.

[773–796] Функция `readmore` используется для того, чтобы прочитать часть сообщения, отправленного принтером. Если текущая позиция чтения находится в конце буфера, мы увеличиваем его размер и возвращаем адрес начала нового буфера и его размер через аргументы `bpp` и `bszp` соответственно. В любом случае мы пытаемся прочитать столько данных, сколько поместится в буфер, дописывая новые данные после данных, уже находящихся в буфере. Мы возвращаем новое значение смещения конца данных в буфере. Если операция чтения потерпела неудачу или истекло время тайм-аута, возвращается значение –1.

```
 797   /*
 798    * Прочитать и проанализировать ответ принтера. Вернуть 1, если ответ
 799    * свидетельствует об успехе, 0 - в противном случае.
 800    *
 801    * БЛОКИРОВКИ: отсутствуют.
 802    */
 803   int
 804   printer_status(int sockfd, struct job *jp)
 805   {
 806     int     i, success, code, len, found, bufsz;
 807     long    jobid;
 808     ssize_t nr;
 809     char    *statcode, *reason, *cp, *contentlen;
```

```
810     struct  ipp_hdr *hp;
811     char    *bp;

812     /*
813      * Прочитать заголовок HTTP и следующий за ним заголовок IPP.
814      * Для их получения может понадобиться несколько попыток чтения
815      * Для определения объема читаемых данных используется Content-Length.
816      */
817     success = 0;
818     bufsz = IOBUFSZ;
819     if ((bp = malloc(IOBUFSZ)) == NULL)
820         log_sys("printer_status: невозможно разместить буфер чтения");

821     while ((nr = tread(sockfd, bp, IOBUFSZ, 5)) > 0) {
822         /*
823          * Отыскать код статуса. Ответ начинается со строки "HTTP/x.y",
824          * поэтому нужно пропустить 8 символов.
825          */
826         cp = bp + 8;
827         while (isspace((int)*cp))
828             cp++;
829         statcode = cp;
830         while (isdigit((int)*cp))
831             cp++;
832         if (cp == statcode) {/* неверный формат, записать его в журнал */
833             log_msg(bp);
```

[797–811] **Функция** `printer_status` **читает ответ принтера на наш запрос. Нам не известно заранее, как ответит принтер – он может разделить ответ на несколько сообщений, отправить его в виде одного сообщения или использовать промежуточные сообщения HTTP** `100 Continue`**. Необходимо обработать все возможные ситуации.**

[812–833] **Мы размещаем в динамической памяти буфер и читаем данные из принтера, предполагая, что операция чтения займет не более 5 секунд. Мы пропускаем строку** `HTTP/1.1` **в начале сообщения и все последующие пробельные символы. Дальше должен располагаться числовой код статуса. Если это не так, мы выводим в журнал содержимое сообщения.**

```
834         } else {
835             *cp++ = '\0';
836             reason = cp;
837             while (*cp != '\r' && *cp != '\n')
838                 cp++;
839             *cp = '\0';
840             code = atoi(statcode);
841             if (HTTP_INFO(code))
842                 continue;
843             if (!HTTP_SUCCESS(code)) { /* возможная ошибка: записать ее */
844                 bp[nr] = '\0';
845                 log_msg("ошибка: %s", reason);
846                 break;
```

```
847             }
848             /*
849              * Заголовок HTTP в порядке, но нам нужно
850              * проверить статус IPP. Для начала найдем
851              * спецификатор Content-Length.
852              */
853             i = cp - bp;
854             for (;;) {
855                 while (*cp != 'C' && *cp != 'c' && i < nr) {
856                     cp++;
857                     i++;
858                 }
859                 if (i >= nr && /* продолжить чтение заголовка */
860                   ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
861                     goto out;
862                 cp = &bp[i];
```

[834–839] Если в ответе обнаружен числовой код статуса, необходимо первый нецифровой символ заменить нулевым символом. Далее должна следовать строка `reason` (текст сообщения). Мы находим завершающие символы возврата каретки и перевода строки и также вставляем в конец строки завершающий нулевой символ.

[840–847] Код преобразуется в число. Если это всего лишь информационное сообщение, мы игнорируем его и переходим к началу цикла, чтобы продолжить чтение. Мы ожидаем получить либо сообщение об успехе операции, либо сообщение об ошибке. Если получено сообщение об ошибке, мы выводим его в журнал и прерываем работу цикла.

[848–862] Если получено сообщение об успехе, нам необходимо проверить статус в заголовке IPP. Мы ищем в тексте сообщения строку `Content-Length`, которая может начинаться с символа `C` или `c`. Так как заголовки HTTP нечувствительны к регистру, приходится искать символы верхнего и нижнего регистров.

При выходе за пределы буфера, чтение данных продолжается. Поскольку функция `readmore` вызывает функцию `realloc`, это может привести к изменению адреса буфера, и поэтому нам необходимо переустановить указатель `cp` так, чтобы он указывал на нужное место в буфере.

```
863                 if (strncasecmp(cp, "Content-Length:", 15) == 0) {
864                     cp+= 15;
865                     while (isspace((int)*cp))
866                         cp++;
867                     contentlen = cp;
868                     while (isdigit((int)*cp))
869                         cp++;
870                     *cp++ = '\0';
871                     i= cp - bp;
872                     len = atoi(contentlen);
873                     break;
874                 }else {
```

```
875                 cp++;
876                 i++;
877             }
878         }
879         if (i >= nr && /* продолжить чтение заголовка */
880           ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
881             goto out;
882         cp = &bp[i];

883         found = 0;
884         while (!found) { /* поиск конца заголовка HTTP */
885             while (i < nr - 2) {
886                 if (*cp == '\n' && *(cp + 1) == '\r' &&
887                   *(cp + 2) =='\n') {
888                     found = 1;
889                     cp+= 3;
890                     i+= 3;
891                     break;
892             }
893             cp++;
894             i++;
895         }
896         if (i >= nr && /* продолжить чтение заголовка */
897           ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
898             goto out;
899             cp = &bp[i];
900         }
```

[863–882] **Когда найдена строка с именем атрибута `Content-Length`, нам нужно получить его значение. Мы преобразуем строку из цифровых символов в целое число, прерываем цикл `for` и продолжаем чтение данных от принтера, если мы вышли за пределы буфера. Если достигнут конец буфера, но строка `Content-Length` не найдена, мы продолжаем работу цикла и повторяем чтение данных от принтера.**

[883–900] **Получив длину сообщения из атрибута `Content-Length`, мы начинаем поиск конца заголовка HTTP – пустой строки. Обнаружив такую строку, мы устанавливаем флаг `found` и пропускаем ее.**

```
901         if (nr - i < len && /* продолжить чтение заголовка */
902           ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
903             goto out;
904         cp = &bp[i];

905         hp = (struct ipp_hdr *)cp;
906         i = ntohs(hp->status);
907         jobid = ntohl(hp->request_id);
908         if (jobid != jp->jobid) {
909             /*
910              * Другие задания. Игнорировать их.
911              */
912             log_msg("задание %ld, код статуса %d", jobid, i);
```

```
913                 break;
914             }
915             if (STATCLASS_OK(i))
916                 success = 1;
917             break;
918         }
919     }
920 out:
921     free(bp);
922     if (nr < 0) {
923         log_msg("задание %ld: ошибка чтения ответа от принтера: %s",
924                 jobid, strerror(errno));
925     }
926     return(success);
927 }
```

[901–904] Мы продолжаем поиск конца заголовка HTTP. Если мы вышли за пределы буфера, необходимо продолжить чтение заголовка. Когда конец заголовка HTTP будет найден, мы вычисляем количество байт, занимаемое заголовком HTTP. Если объем прочитанных данных за вычетом размера заголовка HTTP не совпадает с размером сообщения IPP (этот размер был получен из атрибута `Content-Length`), то нужно продолжить чтение.

[905–927] Мы извлекаем код статуса и идентификатор задания из заголовка IPP. Оба значения хранятся в виде целых чисел с сетевым порядком байтов, поэтому нужно с помощью функций `ntohs` и `ntohl` преобразовать их в значения с порядком байтов, принятым для данной аппаратной архитектуры. Если идентификатор задания не совпадает с ожидаемым, мы выводим в журнал сообщение и прерываем выполнение внешнего цикла `while`. Если статус IPP говорит об успехе, мы сохраняем возвращаемое значение и прерываем работу цикла. Если наш запрос был успешно выполнен, возвращается значение 1, если была обнаружена ошибка – 0.

На этом мы заканчиваем рассмотрение этого достаточно объемного примера. Программы из этой главы были протестированы с сетевым PostScript-принтером Xerox Phaser 860. К сожалению, этот принтер не распознает документы в формате `text/plain`, но он может автоматически различать документы в простом текстовом формате и в формате PostScript. Таким образом, с помощью этого принтера можно печатать как простые текстовые документы, так и документы в формате PostScript, но нельзя печатать файлы PostScript как простой текст, не используя при этом некоторые другие утилиты, такие как `a2ps`(1), чтобы инкапсулировать программы PostScript.

21.6. Подведение итогов

В этой главе были подробно рассмотрены две законченные программы: демон спулера печати, который посылает задание печати сетевому принтеру, и утилита, которая может использоваться для передачи задания печати демону. Это дало нам возможность увидеть, как в реальной программе могут ис-

пользоваться функциональные возможности, описанные в предыдущих главах: потоки, мультиплексирование ввода-вывода, операции файлового ввода-вывода, сокеты и сигналы.

Упражнения

21.1. Переведите значения кодов ошибок IPP, перечисленные в `ipp.h`, в текстовые сообщения. Затем измените демон печати таким образом, чтобы в конце функции `printer_status` в журнал выводилось текстовое сообщение об ошибке, если заголовок IPP свидетельствует о ее наличии.

21.2. Добавьте в утилиту `print` и демон `printd` поддержку двусторонней печати. Добавьте также возможность изменения ориентации бумаги.

21.3. Измените демон печати таким образом, чтобы при запуске он запрашивал у принтера перечень функциональных возможностей, которые им поддерживаются. Это необходимо для того, чтобы демон не указывал атрибуты, которые не поддерживаются принтером.

21.4. Напишите утилиту, с помощью которой можно было бы получать информацию о состоянии заданий, стоящих в очереди.

21.5. Напишите утилиту, с помощью которой можно было бы отменить печать задания, стоящего в очереди.

21.6. Добавьте в демон печати возможность одновременной работы с несколькими принтерами. Предусмотрите возможность переброски задания печати с одного принтера на другой.

A

Прототипы функций

Это приложение содержит прототипы функций, определяемых стандартами ISO C и POSIX, а также функций UNIX, описанных в этой книге. Часто необходимо узнать, какие аргументы принимает та или иная функция («В каком аргументе передается функции `fgets` указатель на структуру `FILE`?») или что она возвращает («Что возвращает функция `sprintf` — указатель или счетчик?»). В описаниях прототипов указаны заголовочные файлы, которые нужно подключить для получения определений всех специальных констант и прототипов функций ISO C, что поможет в диагностике ошибок времени компиляции.

Для каждой функции справа от первого заголовочного файла приводится номер страницы, на которой был приведен прототип этой функции. Там же следует искать дополнительную информацию о ней.

Некоторые функции поддерживаются не всеми платформами, описанными в этой книге. Кроме того, некоторые платформы поддерживают флаги функций, которые не поддерживаются другими платформами. Обычно мы будем перечислять платформы, которые поддерживают ту или иную функциональность. Однако в отдельных случаях будут перечислены платформы, на которых поддержка отсутствует.

```
void    abort(void);
                    <stdlib.h>                                  стр. 414
                    Эта функция никогда не возвращает управление
int     accept(int sockfd, struct sockaddr *restrict addr,
               socklen_t *restrict len);
                    <sys/socket.h>                              стр. 662
                    Возвращает дескриптор файла (сокета) в случае
                    успеха, –1 в случае ошибки

int     access(const char *pathname, int mode);
                    <unistd.h>                                  стр. 139
                    mode: R_OK, W_OK, X_OK, F_OK
                    Возвращает 0 в случае успеха, –1 в случае ошибки
```

```
unsigned
int     alarm(unsigned int seconds);
                    <unistd.h>                                          стр. 385
                    Возвращает 0 или количество секунд, оставшихся
                      до истечения установленного ранее интервала времени
char    *asctime(const struct tm *tmptr);
                    <time.h>                                            стр. 229
                    Возвращает указатель на строку, завершающуюся
                    нулевым символом

int     atexit(void (*func)(void));
                    <stdlib.h>                                          стр. 237
                    Возвращает 0 в случае успеха, ненулевое значение
                    в случае ошибки

int     bind(int sockfd, const struct sockaddr *addr, socklen_t len);
                    <sys/socket.h>                                      стр. 659
                    Возвращает 0 в случае успеха, –1 в случае ошибки

void    *calloc(size_t nobj, size_t size);
                    <stdlib.h>                                          стр. 245
                    Возвращает непустой указатель в случае успеха,
                    NULL в случае ошибки

speed_t cfgetispeed(const struct termios *termptr);
                    <termios.h>                                         стр. 758
                    Возвращает значение скорости в бодах

speed_t cfgetospeed(const struct termios *termptr);
                    <termios.h>                                         стр. 758
                    Возвращает значение скорости в бодах

int     cfsetispeed(struct termios *termptr, speed_t speed);
                    <termios.h>                                         стр. 758
                    Возвращает 0 в случае успеха, –1 в случае ошибки

int     cfsetospeed(struct termios *termptr, speed_t speed);
                    <termios.h>                                         стр. 758
                    Возвращает 0 в случае успеха, –1 в случае ошибки

int     chdir(const char *pathname);
                    <unistd.h>                                          стр. 172
                    Возвращает 0 в случае успеха, –1 в случае ошибки

int     chmod(const char *pathname, mode_t mode);
                    <sys/stat.h>                                        стр. 143
                    mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                    Возвращает 0 в случае успеха, –1 в случае ошибки

int     chown(const char *pathname, uid_t owner, gid_t group);
                    <unistd.h>                                          стр. 147
                    Возвращает 0 в случае успеха, -1 в случае ошибки

void    clearerr(FILE *fp);
                    <stdio.h>                                           стр. 190
```

```
int     close(int filedes);
```
<unistd.h> стр. 101
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     closedir(DIR *dp);
```
<dirent.h> стр. 167
Возвращает 0 в случае успеха, −1 в случае ошибки

```
void    closelog(void);
```
<syslog.h> стр. 511

```
unsigned
char    *CMSG_DATA(struct cmsghdr *cp);
```
<sys/socket.h> стр. 709
Возвращает указатель на данные, связанные со структурой cmsghdr

```
struct
cmsghdr *CMSG_FIRSTHDR(struct msghdr *mp);
```
<sys/socket.h> стр. 709
Возвращает указатель на первую структуру cmsghdr, связанную со структурой msghdr, или NULL, если таковой не существует

```
unsigned
int     CMSG_LEN(unsigned int nbytes);
```
<sys/socket.h> стр. 709
Возвращает объем памяти, который необходимо выделить для хранения объекта размером *nbytes*

```
struct
cmsghdr *CMSG_NXTHDR(struct msghdr *mp, struct cmsghdr *cp);
```
<sys/socket.h> стр. 709
Возвращает указатель на следующую структуру cmsghdr, связанную со структурой msghdr, которую представляет текущая структура cmsghdr, или NULL, если таковой не существует

```
int     connect(int sockfd, const struct sockaddr *addr, socklen_t len);
```
<sys/socket.h> стр. 660
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     creat(const char *pathname, mode_t mode);
```
<fcntl.h> стр. 100
mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
Возвращает дескриптор файла, открытый только для чтения, в случае успеха, −1 в случае ошибки

```
char    *ctermid(char *ptr);
```
<stdio.h> стр. 760
Возвращает указатель на имя управляющего терминала в случае успеха, указатель на пустую строку в случае ошибки

```
char    *ctime(const time_t *calptr);
```
<time.h> стр. 229
Возвращает указатель на строку, завершающуюся нулевым символом

```
int     dup(int filedes);
                <unistd.h>                                              стр. 115
```
Возвращает новый дескриптор файла в случае успеха, −1 в случае ошибки

```
int     dup2(int filedes, int filedes2);
                <unistd.h>                                              стр. 115
```
Возвращает новый дескриптор файла в случае успеха, −1 в случае ошибки

```
void    endgrent(void);
                <grp.h>                                                 стр. 220

void    endhostent(void);
                <netdb.h>                                               стр. 651

void    endnetent(void);
                <netdb.h>                                               стр. 652

void    endprotoent(void);
                <netdb.h>                                               стр. 653

void    endpwent(void);
                <pwd.h>                                                 стр. 216

void    endservent(void);
                <netdb.h>                                               стр. 653

void    endspent(void);
                <shadow.h>                                              стр. 219
```
Платформы: Linux 2.4.22, Solaris 9

```
int     execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);
                <unistd.h>                                              стр. 292
```
Возвращает −1 в случае ошибки, в случае успеха не возвращает управление

```
int     execle(const char *pathname, const char *arg0, ... /* (char *) 0,
            char *const envp[] */);
                <unistd.h>                                              стр. 292
```
Возвращает −1 в случае ошибки, в случае успеха не возвращает управление

```
int     execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);
                <unistd.h>                                              стр. 292
```
Возвращает −1 в случае ошибки, в случае успеха не возвращает управление

```
int     execv(const char *pathname, char *const argv[]);
                <unistd.h>                                              стр. 292
```
Возвращает −1 в случае ошибки, в случае успеха не возвращает управление

```
int     execve(const char *pathname, char *const argv[], char *const envp[]);
                <unistd.h>                                              стр. 292
```
Возвращает −1 в случае ошибки, в случае успеха не возвращает управление

```
int     execvp(const char *filename, char *const argv[]);
```
<unistd.h> стр. 292
Возвращает −1 в случае ошибки, в случае успеха не возвращает управление

```
void    _Exit(int status);
```
<stdlib.h> стр. 235
Эта функция никогда не возвращает управление

```
void    _exit(int status);
```
<unistd.h> стр. 235
Эта функция никогда не возвращает управление

```
void    exit(int status);
```
<stdlib.h> стр. 235
Эта функция никогда не возвращает управление

```
int     fattach(int filedes, const char *path);
```
<stropts.h> стр. 690
Возвращает 0 в случае успеха, −1 в случае ошибки
Платформы: Linux 2.4.22, Solaris 9

```
int     fchdir(int filedes);
```
<unistd.h> стр. 172
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     fchmod(int filedes, mode_t mode);
```
<sys/stat.h> стр. 143
mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     fchown(int filedes, uid_t owner, gid_t group);
```
<unistd.h> стр. 147
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     fclose(FILE *fp);
```
<stdio.h> стр. 189
Возвращает 0 в случае успеха, EOF в случае ошибки

```
int     fcntl(int filedes, int cmd, ... /* int arg */);
```
<fcntl.h> стр. 118
cmd: F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL, F_GETOWN, F_SETOWN, F_GETLK, F_SETLK, F_SETLKW
Возвращаемое значение зависит от аргумента *cmd* в случае успеха, −1 в случае ошибки

```
int     fdatasync(int filedes);
```
<unistd.h> стр. 117
Возвращает 0 в случае успеха, −1 в случае ошибки
Платформы: Linux 2.4.22, Solaris 9

```
void    FD_CLR(int fd, fd_set *fdset);
```
<sys/select.h> стр. 563

```
int     fdetach(const char *path);
                <stropts.h>                                          стр. 691
```
Возвращает 0 в случае успеха, -1 в случае ошибки.
Платформы: Linux 2.4.22, Solaris 9

```
int     FD_ISSET(int fd, fd_set *fdset);
                <sys/select.h>                                       стр. 563
```
Возвращает ненулевое значение, если *fd* имеется в наборе, 0 в противном случае

```
FILE    *fdopen(int filedes, const char *type);
                <stdio.h>                                            стр. 187
                type: "r", "w", "a", "r+", "w+", "a+",
```
Возвращает указатель на структуру FILE в случае успеха, NULL в случае ошибки

```
void    FD_SET(int fd, fd_set *fdset);
                <sys/select.h>                                       стр. 563
```

```
void    FD_ZERO(fd_set *fdset);
                <sys/select.h>                                       стр. 563
```

```
int     feof(FILE *fp);
                <stdio.h>                                            стр. 190
```
Возвращает ненулевое значение (истина), если в потоке достигнут конец файла, 0 (ложь) – в противном случае

```
int     ferror(FILE *fp);
                <stdio.h>                                            стр. 190
```
Возвращает ненулевое значение (истина), если при работе с потоком возникла ошибка, 0 (ложь) в противном случае

```
int     fflush(FILE *fp);
                <stdio.h>                                            стр. 186
```
Возвращает 0 в случае успеха, EOF в случае ошибки

```
int     fgetc(FILE *fp);
                <stdio.h>                                            стр. 190
```
Возвращает следующий символ в случае успеха, EOF по достижении конца файла или в случае ошибки

```
int     fgetpos(FILE *restrict fp, fpos_t *restrict pos);
                <stdio.h>                                            стр. 199
```
Возвращает 0 в случае успеха, ненулевое значение в случае ошибки

```
char    *fgets(char *restrict buf, int n, FILE *restrict fp);
                <stdio.h>                                            стр. 192
```
Возвращает указатель на *buf* в случае успеха, NULL по достижении конца файла или в случае ошибки

```
int     fileno(FILE *fp);
                <stdio.h>                                            стр. 205
```
Возвращает дескриптор файла, ассоциированный с потоком

```
void    flockfile(FILE *fp);
                <stdio.h>                                            стр. 482
```

```
FILE    *fopen(const char *restrict pathname, const char *restrict type);
                    <stdio.h>                                              стр. 187
                    type: "r", "w", "a", "r+", "w+", "a+",
                    Возвращает указатель на структуру FILE в случае успеха,
                    NULL в случае ошибки

pid_t   fork(void);
                    <unistd.h>                                             стр. 268
                    Возвращает 0 в дочернем процессе, идентификатор дочернего
                    процесса – в родительском процессе, –1 в случае ошибки

long    fpathconf(int filedes, int name);
                    <unistd.h>                                              стр. 76
                    name: _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED,
                          _PC_FILESIZEBITS, _PC_LINK_MAX, _PC_MAX_CANON,
                          _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC,
                          _PC_PATH_MAX, _PC_PIPE_BUF, _PC_PRIO_IO,
                          _PC_SYNC_IO, _PC_SYMLINK_MAX, _PC_VDISABLE
                    Возвращает соответствующее значение в случае успеха,
                    –1 в случае ошибки

int     fprintf(FILE *restrict fp, const char *restrict format, ...);
                    <stdio.h>                                              стр. 200
                    Возвращает количество выведенных символов в случае
                    успеха, отрицательное значение в случае ошибки

int     fputc(int c, FILE *fp);
                    <stdio.h>                                              стр. 192
                    Возвращает символ c в случае успеха, EOF в случае ошибки

int     fputs(const char *restrict str, FILE *restrict fp);
                    <stdio.h>                                              стр. 193
                    Возвращает неотрицательное значение в случае успеха,
                    EOF в случае ошибки

size_t  fread(void *restrict ptr, size_t size, size_t nobj,
              FILE *restrict fp);
                    <stdio.h>                                              стр. 196
                    Возвращает количество прочитанных блоков

void    free(void *ptr);
                    <stdlib.h>                                             стр. 245

void    freeaddrinfo(struct addrinfo *ai);
                    <sys/socket.h>                                         стр. 654
                    <netdb.h>

FILE    *freopen(const char *restrict pathname, const char *restrict type,
                 FILE *restrict fp);
                    <stdio.h>                                              стр. 187
                    type: "r", "w", "a", "r+", "w+", "a+",
                    Возвращает указатель на структуру FILE в случае
                    успеха, NULL в случае ошибки
```

```
int     fscanf(FILE *restrict fp, const char *restrict format, ...);
                <stdio.h>                                              стр. 203
```
Возвращает количество введенных элементов, EOF – по достижении конца файла или в случае ошибки перед выполнением преобразования

```
int     fseek(FILE *fp, long offset, int whence);
                <stdio.h>                                              стр. 198
                whence: SEEK_SET, SEEK_CUR, SEEK_END
```
Возвращает 0 в случае успеха, ненулевое значение в случае ошибки

```
int     fseeko(FILE *fp, off_t offset, int whence);
                <stdio.h>                                              стр. 199
                whence: SEEK_SET, SEEK_CUR, SEEK_END
```
Возвращает 0 в случае успеха, ненулевое значение в случае ошибки

```
int     fsetpos(FILE *fp, const fpos_t *pos);
                <stdio.h>                                              стр. 199
```
Возвращает 0 в случае успеха, ненулевое значение в случае ошибки

```
Int     fstat(int filedes, struct stat *buf);
                <sys/stat.h>                                           стр. 129
```
Возвращает 0 в случае успеха, –1 в случае ошибки

```
int     fsync(int filedes);
                <unistd.h>                                             стр. 117
```
Возвращает 0 в случае успеха, –1 в случае ошибки

```
long    ftell(FILE *fp);
                <stdio.h>                                              стр. 198
```
Возвращает значение текущей позиции файла в случае успеха, –1L в случае ошибки

```
off_t   ftello(FILE *fp);
                <stdio.h>                                              стр. 199
```
Возвращает значение текущей позиции файла в случае успеха, (off_t)-1 в случае ошибки

```
key_t   ftok(const char *path, int id);
                <sys/ipc.h>                                            стр. 611
```
Возвращает значение ключа в случае успеха, (key_t)-1 в случае ошибки

```
int     ftruncate(int filedes, off_t length);
                <unistd.h>                                             стр. 150
```
Возвращает 0 в случае успеха, –1 в случае ошибки

```
int     ftrylockfile(FILE *fp);
                <stdio.h>                                              стр. 482
```
Возвращает 0 в случае успеха, ненулевое значение – если блокировка не может быть установлена

```
void    funlockfile(FILE *fp);
                    <stdio.h>                                          стр. 482

int     fwide(FILE *fp, int mode);
                    <stdio.h>                                          стр. 182
                    <wchar.h>
                    Возвращает положительное значение, если поток
                    ориентирован на работу с многобайтными (wide)
                    символами, отрицательное – с однобайтными,
                    0 – если поток не имеет ориентации

size_t  fwrite(const void *restrict ptr, size_t size, size_t nobj,
              FILE *restrict fp);
                    <stdio.h>                                          стр. 196
                    Возвращает количество записанных блоков

const
char    *gai_strerror(int error);
                    <netdb.h>                                          стр. 655
                    Возвращает указатель на строку с описанием ошибки

int     getaddrinfo(const char *restrict host, const char *restrict service,
                    const struct addrinfo *restrict hint,
                    struct addrinfo **restrict res);
                    <sys/socket.h>                                     стр. 654
                    <netdb.h>
                    Возвращает 0 в случае успеха, ненулевой код ошибки
                    в случае неудачи

int     getc(FILE *fp);
                    <stdio.h>                                          стр. 190
                    Возвращает следующий символ в случае успеха,
                    EOF по достижении конца файла или в случае ошибки

int     getchar(void);
                    <stdio.h>                                          стр. 190
                    Возвращает следующий символ в случае успеха,
                    EOF по достижении конца файла или в случае ошибки

int     getchar_unlocked(void);
                    <stdio.h>                                          стр. 483
                    Возвращает следующий символ в случае успеха,
                    EOF по достижении конца файла или в случае ошибки

int     getc_unlocked(FILE *fp);
                    <stdio.h>                                          стр. 483
                    Возвращает следующий символ в случае успеха,
                    EOF по достижении конца файла или в случае ошибки

char    *getcwd(char *buf, size_t size);
                    <unistd.h>                                         стр. 174
                    Возвращает указатель на buf в случае успеха,
                    NULL в случае ошибки
```

```
gid_t   getegid(void);
```
<unistd.h> стр. 267

Возвращает эффективный идентификатор группы вызывающего процесса

```
char    *getenv(const char *name);
```
<stdlib.h> стр. 248

Возвращает указатель на значение переменной окружения с именем *name*, NULL – если переменная не найдена

```
uid_t   geteuid(void);
```
<unistd.h> стр. 267

Возвращает эффективный идентификатор пользователя вызывающего процесса

```
gid_t   getgid(void);
```
<unistd.h> стр. 267

Возвращает реальный идентификатор группы вызывающего процесса

```
struct
group   *getgrent(void);
```
<grp.h> стр. 220

Возвращает указатель в случае успеха, NULL по достижении конца файла или в случае ошибки

```
struct
group   *getgrgid(gid_t gid);
```
<grp.h> стр. 220

Возвращает указатель в случае успеха, NULL в случае ошибки

```
struct
group   *getgrnam(const char *name);
```
<grp.h> стр. 220

Возвращает указатель в случае успеха, NULL в случае ошибки

```
int     getgroups(int gidsetsize, gid_t grouplist[]);
```
<unistd.h> стр. 221

Возвращает количество идентификаторов дополнительных групп в случае успеха, –1 в случае ошибки

```
struct
hostent *gethostent(void);
```
<netdb.h> стр. 651

Возвращает указатель в случае успеха, NULL в случае ошибки

```
int     gethostname(char *name, int namelen);
```
<unistd.h> стр. 226

Возвращает 0 в случае успеха, –1 в случае ошибки

```
char    *getlogin(void);
```
<unistd.h> стр. 320

Возвращает указатель на строку с именем пользователя в случае успеха, NULL в случае ошибки

```
int     getmsg(int filedes, struct strbuf *restrict ctlptr,
               struct strbuf *restrict dataptr, int *restrict flagptr);
                     <stropts.h>                                    стр. 555
                     *flagptr: 0, RS_HIPRI
                     Возвращает неотрицательное значение в случае успеха,
                     −1 в случае ошибки
                     Платформы: Linux 2.4.22, Solaris 9

int     getnameinfo(const struct sockaddr *restrict addr, socklen_t alen,
                    char *restrict host, socklen_t hostlen,
                    char *restrict service, socklen_t servlen,
                    unsigned int flags);
                     <sys/socket.h>                                 стр. 655
                     <netdb.h>
                     Возвращает 0 в случае успеха, ненулевое значение
                     в случае ошибки

struct
netent  *getnetbyaddr(uint32_t net, int type);
                     <netdb.h>                                      стр. 652
                     Возвращает указатель в случае успеха, NULL в случае ошибки

struct
netent  *getnetbyname(const char *name);
                     <netdb.h>                                      стр. 652
                     Возвращает указатель в случае успеха, NULL в случае ошибки

struct
netent  *getnetent(void);
                     <netdb.h>                                      стр. 652
                     Возвращает указатель в случае успеха, NULL в случае ошибки

int     getopt(int argc, const * const argv[], const char *options);
                     <fcntl.h>                                      стр. 888
                     extern int optind, opterr, optopt;
                     extern char *optarg;
                     Возвращает символ следующей опции или −1,
                     если все опции были обработаны

int     getpeername(int sockfd, struct sockaddr *restrict addr,
                    socklen_t *restrict alenp);
                     <sys/socket.h>                                 стр. 660
                        Возвращает 0 в случае успеха, −1 в случае ошибки
pid_t   getpgid(pid_t pid);
                     <unistd.h>                                     стр. 334
                     Возвращает идентификатор группы процессов
                     в случае успеха, −1 в случае ошибки

pid_t   getpgrp(void);
                     <unistd.h>                                     стр. 334
                     Возвращает идентификатор группы процессов
                     вызывающего процесса
```

```
pid_t   getpid(void);
                    <unistd.h>                                          стр. 267
                    Возвращает идентификатор процесса вызывающего процесса

int     getpmsg(int filedes, struct strbuf *restrict ctlptr,
                struct strbuf *restrict dataptr, int *restrict bandptr,
                int *restrict flagptr);
                    <stropts.h>                                         стр. 555
                    *flagptr: 0, MSG_HIPRI, MSG_BAND, MSG_ANY
                    Возвращает неотрицательное значение в случае успеха,
                    −1 в случае ошибки
                    Платформы: Linux 2.4.22, Solaris 9.

pid_t   getppid(void);
                    <unistd.h>                                          стр. 267
                    Возвращает идентификатор родительского процесса

struct
protoent *getprotobyname(const char *name);
                    <netdb.h>                                           стр. 653
                    Возвращает указатель в случае успеха, NULL в случае ошибки

struct
protoent *getprotobynumber(int proto);
                    <netdb.h>                                           стр. 653
                    Возвращает указатель в случае успеха, NULL в случае ошибки

struct
protoent *getprotoent(void);
                    <netdb.h>                                           стр. 653
                    Возвращает указатель в случае успеха, NULL в случае ошибки

struct
passwd  *getpwent(void);
                    <pwd.h>                                             стр. 216
                    Возвращает указатель в случае успеха, NULL в случае ошибки
                    или по достижении конца файла

struct
passwd  *getpwnam(const char *name);
                    <pwd.h>                                             стр. 216
                    Возвращает указатель в случае успеха, NULL в случае ошибки

struct
passwd  *getpwuid(uid_t uid);
                    <pwd.h>                                             стр. 216
                    Возвращает указатель в случае успеха, NULL в случае ошибки

int     getrlimit(int resource, struct rlimit *rlptr);
                    <sys/resource.h>                                    стр. 259
                    Возвращает 0 в случае успеха, ненулевое значение
                    в случае ошибки
```

```
char    *gets(char *buf);
                    <stdio.h>                                    стр. 192
```
Возвращает указатель на *buf* в случае успеха, NULL по достижении конца файла или в случае ошибки

```
struct
servent *getservbyname(const char *name, const char *proto);
                    <netdb.h>                                    стр. 653
```
Возвращает указатель в случае успеха, NULL в случае ошибки

```
struct
servent *getservbyport(int port, const char *proto);
                    <netdb.h>                                    стр. 653
```
Возвращает указатель в случае успеха, NULL в случае ошибки

```
struct
servent *getservent(void);
                    <netdb.h>                                    стр. 653
```
Возвращает указатель в случае успеха, NULL в случае ошибки

```
pid_t   getsid(pid_t pid);
                    <unistd.h>                                   стр. 337
```
Возвращает идентификатор группы процессов лидера сессии в случае успеха, −1 в случае ошибки

```
int     getsockname(int sockfd, struct sockaddr *restrict addr,
                    socklen_t *restrict alenp);
                    <sys/socket.h>                               стр. 660
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     getsockopt(int sockfd, int level, int option, void *restrict val,
                   socklen_t *restrict lenp);
                    <sys/socket.h>                               стр. 681
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
struct
spwd    *getspent(void);
                    <shadow.h>                                   стр. 219
```
Возвращает указатель в случае успеха, NULL в случае ошибки
Платформы: Linux 2.4.22, Solaris 9

```
struct
spwd    *getspnam(const char *name);
                    <shadow.h>                                   стр. 219
```
Возвращает указатель в случае успеха, NULL в случае ошибки
Платформы: Linux 2.4.22, Solaris 9

```
int     gettimeofday(struct timeval *restrict tp, void *restrict tzp);
                    <sys/time.h>                                 стр. 227
```
Всегда возвращает 0

```
uid_t   getuid(void);
                    <unistd.h>                                   стр. 267
```
Возвращает реальный идентификатор пользователя вызывающего процесса

```
struct
tm      *gmtime(const time_t *calptr);
                    <time.h>
```
стр. 229

Возвращает указатель на структуру с временем, разложенным на составляющие

```
int     grantpt(int filedes);
                    <stdlib.h>
```
стр. 788

Возвращает 0 в случае успеха, −1 в случае ошибки.
Платформы: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

```
uint32_t htonl(uint32_t hostint32);
                    <arpa/inet.h>
```
стр. 648

Возвращает 32-битное целое с сетевым порядком байтов

```
uint16_t htons(uint16_t hostint16);
                    <arpa/inet.h>
```
стр. 648

Возвращает 16-битное целое с сетевым порядком байтов

```
const
char    *inet_ntop(int domain, const void *restrict addr, char *restrict str,
                   socklen_t size);
                    <arpa/inet.h>
```
стр. 650

Возвращает указатель на строку с адресом в случае успеха, NULL в случае ошибки

```
int     inet_pton(int domain, const char *restrict str, void *restrict addr);
                    <arpa/inet.h>
```
стр. 650

Возвращает 1 в случае успеха, 0 в случае неверного формата, −1 в случае ошибки

```
int     initgroups(const char *username, gid_t basegid);
                    <grp.h>       /* Linux и Solaris */
                    <unistd.h>    /* FreeBSD и Mac OS X */
```
стр. 221

Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     ioctl(int filedes, int request, ...);
                    <unistd.h>    /* System V */
                    <sys/ioctl.h> /* BSD и Linux */
                    <stropts.h>   /* XSI STREAMS */
```
стр. 125

Возвращает −1 в случае ошибки, любое другое значение в случае успеха

```
int     isastream(int filedes);
                    <stropts.h>
```
стр. 550

Возвращает 1 (истина), если это устройство STREAMS, 0 (ложь) в противном случае
Платформы: Linux 2.4.22, Solaris 9

```
int     isatty(int filedes);
                    <unistd.h>
```
стр. 761

Возвращает 1 (истина), если это терминальное устройство, 0 (ложь) в противном случае

```
int     kill(pid_t pid, int signo);
                    <signal.h>                                          стр. 383
                    Возвращает 0 в случае успеха, −1 в случае ошибки

int     lchown(const char *pathname, uid_t owner, gid_t group);
                    <unistd.h>                                          стр. 147
                    Возвращает 0 в случае успеха, −1 в случае ошибки

int     link(const char *existingpath, const char *newpath);
                    <unistd.h>                                          стр. 154
                    Возвращает 0 в случае успеха, −1 в случае ошибки

int     listen(int sockfd, int backlog);
                    <sys/socket.h>                                      стр. 662
                    Возвращает 0 в случае успеха, −1 в случае ошибки

struct
tm      *localtime(const time_t *calptr);
                    <time.h>                                            стр. 229
                    Возвращает указатель на структуру с временем,
                    разложенным на составляющие

void    longjmp(jmp_buf env, int val);
                    <setjmp.h>                                          стр. 254
                    Эта функция никогда не возвращает управление

off_t   lseek(int filedes, off_t offset, int whence);
                    <unistd.h>                                          стр. 101
                    whence: SEEK_SET, SEEK_CUR, SEEK_END
                    Возвращает новую текущую позицию файла в случае
                    успеха, −1 в случае ошибки

int     lstat(const char *restrict pathname, struct stat *restrict buf);
                    <sys/stat.h>                                        стр. 129
                    Возвращает 0 в случае успеха, −1 в случае ошибки

void    *malloc(size_t size);
                    <stdlib.h>                                          стр. 245
                    Возвращает непустой указатель в случае успеха,
                    NULL в случае ошибки

int     mkdir(const char *pathname, mode_t mode);
                    <sys/stat.h>                                        стр. 165
                    mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                    Возвращает 0 в случае успеха, −1 в случае ошибки

int     mkfifo(const char *pathname, mode_t mode);
                    <sys/stat.h>                                        стр. 606
                    mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                    Возвращает 0 в случае успеха, −1 в случае ошибки

int     mkstemp(char *template);
                    <stdlib.h>                                          стр. 210
                    Возвращает дескриптор файла в случае успеха,
                    −1 в случае ошибки
```

```
time_t  mktime(struct tm *tmptr);
                    <time.h>                                          стр. 229
                    Возвращает календарное время в случае успеха,
                    −1 в случае ошибки

caddr_t mmap(void *addr, size_t len, int prot, int flag, int filedes,
            off_t off);
                    <sys/mman.h>                                      стр. 576
                    prot: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
                    flag: MAP_FIXED, MAP_SHARED, MAP_PRIVATE
                    Возвращает начальный адрес отображенной области
                    в случае успеха, MAP_FAILED в случае ошибки

int     mprotect(void *addr, size_t len, int prot);
                    <sys/mman.h>                                      стр. 579
                    Возвращает 0 в случае успеха, −1 в случае ошибки

int     msgctl(int msqid, int cmd, struct msqid_ds *buf);
                    <sys/msg.h>                                       стр. 617
                    cmd: IPC_STAT, IPC_SET, IPC_RMID
                    Возвращает 0 в случае успеха, −1 в случае ошибки
                    Платформы: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

int     msgget(key_t key, int flag);
                    <sys/msg.h>                                       стр. 616
                    flag: 0, IPC_CREAT, IPC_EXCL
                    Возвращает идентификатор очереди сообщений в случае
                    успеха, −1 в случае ошибки
                    Платформы: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
                    <sys/msg.h>                                       стр. 619
                    flag: 0, IPC_NOWAIT, MSG_NOERROR
                    Возвращает размер блока данных сообщения
                    в случае успеха, −1 в случае ошибки
                    Платформы: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

int     msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
                    <sys/msg.h>                                       стр. 618
                    flag: 0, IPC_NOWAIT
                    Возвращает 0 в случае успеха, −1 в случае ошибки
                    Платформы: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

int     msync(void *addr, size_t len, int flags);
                    <sys/mman.h>                                      стр. 579
                    Возвращает 0 в случае успеха, −1 в случае ошибки

int     munmap(caddr_t addr, size_t len);
                    <sys/mman.h>                                      стр. 580
                    Возвращает 0 в случае успеха, −1 в случае ошибки

uint32_t ntohl(uint32_t netint32);
                    <arpa/inet.h>                                     стр. 648
                    Возвращает 32-битное целое с аппаратным порядком байтов
```

```
uint16_t ntohs(uint16_t netint16);
                    <arpa/inet.h>                                    стр. 648
                    Возвращает 16-битное целое с аппаратным порядком байтов

int     open(const char *pathname, int oflag, ... /* mode_t mode */);
                    <fcntl.h>                                        стр. 97
                    oflag: O_RDONLY, O_WRONLY, O_RDWR;
                           O_APPEND, O_CREAT, O_DSYNC, O_EXCL, O_NOCTTY,
                           O_NONBLOCK, O_RSYNC, O_SYNC, O_TRUNC
                    mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                    Возвращает дескриптор файла в случае успеха,
                    −1 в случае ошибки
                    Платформы: флаг O_FSYNC в FreeBSD 5.2.1 и Mac OS X 10.3

DIR     *opendir(const char *pathname);
                    <direct.h>                                       стр. 167
                    Возвращает указатель на структуру DIR в случае
                    успеха, NULL в случае ошибки

void    openlog(char *ident, int option, int facility);
                    <syslog.h>                                       стр. 511
                    option: LOG_CONS, LOG_NDELAY, LOG_NOWAIT, LOG_ODELAY,
                            LOG_PERROR, LOG_PID
                    facility: LOG_AUTH, LOG_AUTHPRIV, LOG_CRON, LOG_DAEMON,
                              LOG_FTP, LOG_KERN, LOG_LOCAL[0-7], LOG_LPR,
                              LOG_MAIL, LOG_NEWS, LOG_SYSLOG, LOG_USER,
                              LOG_UUCP

long    pathconf(const char *pathname, int name);
                    <unistd.h>                                       стр. 76
                    name: _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED,
                          _PC_FILESIZEBITS, _PC_LINK_MAX, _PC_MAX_CANON,
                          _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC,
                          _PC_PATH_MAX, _PC_PIPE_BUF, _PC_PRIO_IO,
                          _PC_SYMLINK_MAX, _PC_SYNC_IO, _PC_VDISABLE
                    Возвращает соответствующее значение в случае успеха,
                    −1 в случае ошибки

int     pause(void);
                    <unistd.h>                                       стр. 385
                    В случае ошибки возвращает −1 и код ошибки EINTR
                    в переменной errno

int     pclose(FILE *fp);
                    <stdio.h>                                        стр. 594
                    Возвращает код завершения команды cmdstring
                    функции popen, −1 в случае ошибки

void    perror(const char *msg);
                    <stdio.h>                                        стр. 43

int     pipe(int filedes[2]);
                    <unistd.h>                                       стр. 587
                    Возвращает 0 в случае успеха, −1 в случае ошибки
```

```
int     poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
                <poll.h>                                              стр. 566
```
Возвращает количество дескрипторов, готовых к выполнению операции, 0 в случае истечения времени тайм-аута, −1 в случае ошибки
Платформы: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

```
FILE    *popen(const char *cmdstring, const char *type);
                <stdio.h>                                             стр. 594
                type: "r", "w"
```
Возвращает указатель на структуру FILE в случае успеха, NULL в случае ошибки

```
int     posix_openpt(int oflag);
                <stdlib.h>                                            стр. 788
                <fcntl.h>
                oflag: O_RWDR, O_NOCTTY
```
Возвращает дескриптор следующего доступного ведущего PTY в случае успеха, −1 в случае ошибки
Платформы: FreeBSD 5.2.1

```
ssize_t pread(int filedes, void *buf, size_t nbytes, off_t offset);
                <unistd.h>                                            стр. 114
```
Возвращает количество прочитанных байт, 0 по достижении конца файла, −1 в случае ошибки

```
int     printf(const char *restrict format, ...);
                <stdio.h>                                             стр. 200
```
Возвращает количество выведенных символов в случае успеха, отрицательное значение в случае ошибки

```
int     pselect(int maxfdp1, fd_set *restrict readfds,
                fd_set *restrict writefds, fd_set *restrict exceptfds,
                const struct timespec *restrict tsptr,
                const sigset_t *restrict sigmask);
                <sys/select.h>                                        стр. 566
```
Возвращает количество дескрипторов, готовых к выполнению операции, 0 в случае истечения времени тайм-аута, −1 в случае ошибки
Платформы: FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3

```
void    psignal(int signo, const char *msg);
                <signal.h>                                            стр. 427
                <siginfo.h> /* в Solaris */
```

```
int     pthread_atfork(void (*prepare)(void), void (*parent)(void),
                  void (*child)(void));
                <pthread.h>                                           стр. 499
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_attr_destroy(pthread_attr_t *attr);
                <pthread.h>                                           стр. 467
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                    int *detachstate);
                <pthread.h>                                          стр. 468
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                                  size_t *restrict guardsize);
                <pthread.h>                                          стр. 471
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_attr_getstack(const pthread_attr_t *restrict attr, void
                              **restrict stackaddr,
                              size_t *restrict stacksize);
                <pthread.h>                                          стр. 469
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                                  size_t *restrict stacksize);
                <pthread.h>                                          стр. 470
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_attr_init(pthread_attr_t *attr);
                <pthread.h>                                          стр. 467
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
                <pthread.h>                                          стр. 468
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
                <pthread.h>                                          стр. 471
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_attr_setstack(const pthread_attr_t *attr, void *stackaddr,
                              size_t *stacksize);
                <pthread.h>                                          стр. 469
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
                <pthread.h>                                          стр. 470
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_cancel(pthread_t tid);
                <pthread.h>                                          стр. 442
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

void    pthread_cleanup_pop(int execute);
                <pthread.h>                                          стр. 442

void    pthread_cleanup_push(void (*rtn)(void *), void *arg);
                <pthread.h>                                          стр. 442

int     pthread_condattr_destroy(pthread_condattr_t *attr);
                <pthread.h>                                          стр. 480
                Возвращает 0 в случае успеха, код ошибки в случае неудачи
```

```
int     pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
                                    int *restrict pshared);
                <pthread.h>                                              стр. 480
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_condattr_init(pthread_condattr_t *attr);
                <pthread.h>                                              стр. 480
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);
                <pthread.h>                                              стр. 480
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_cond_broadcast(pthread_cond_t *cond);
                <pthread.h>                                              стр. 462
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_cond_destroy(pthread_cond_t *cond);
                <pthread.h>                                              стр. 461
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_cond_init(pthread_cond_t *restrict cond,
                     pthread_condattr_t *restrict attr);
                <pthread.h>                                              стр. 461
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_cond_signal(pthread_cond_t *cond);
                <pthread.h>                                              стр. 462
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_cond_timedwait(pthread_cond_t *restrict cond,
                          pthread_mutex_t *restrict mutex,
                          const struct timespec *restrict timeout);
                <pthread.h>                                              стр. 461
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_cond_wait(pthread_cond_t *restrict cond,
                     pthread_mutex_t *restrict mutex);
                <pthread.h>                                              стр. 461
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_create(pthread_t *restrict tidp,
                  const pthread_attr_t *restrict attr,
                  void *(*start_rtn)(void), void *restrict arg);
                <pthread.h>                                              стр. 434
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_detach(pthread_t tid);
                <pthread.h>                                              стр. 445
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_equal(pthread_t tid1, pthread_t tid2);
                <pthread.h>                                              стр. 433
```
Возвращает ненулевое значение, если потоки эквивалентны, 0 в противном случае

```
void    pthread_exit(void *rval_ptr);
                <pthread.h>                                          стр. 437

int     pthread_getconcurrency(void);
                <pthread.h>                                          стр. 471
                Возвращает текущее значение степени совмещения

void    *pthread_getspecific(pthread_key_t key);
                <pthread.h>                                          стр. 489
                Возвращает адрес области памяти с локальными
                данными потока или NULL, если ключ не ассоциирован
                с локальными данными

int     pthread_join(pthread_t thread, void **rval_ptr);
                <pthread.h>                                          стр. 438
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *);
                <pthread.h>                                          стр. 486
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_key_delete(pthread_key_t *key);
                <pthread.h>                                          стр. 487
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_kill(pthread_t thread, int signo);
                <signal.h>                                           стр. 495
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
                <pthread.h>                                          стр. 472
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr,
                                     int *restrict pshared);
                <pthread.h>                                          стр. 473
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                                  int *restrict type);
                <pthread.h>                                          стр. 474
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_mutexattr_init(pthread_mutexattr_t *attr);
                <pthread.h>                                          стр. 472
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
                <pthread.h>                                          стр. 473
                Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
                <pthread.h>                                          стр. 474
                Возвращает 0 в случае успеха, код ошибки в случае неудачи
```

```
int     pthread_mutex_destroy(pthread_mutex_t *mutex);
                <pthread.h>                                          стр. 448
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr);
                <pthread.h>                                          стр. 448
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_mutex_lock(pthread_mutex_t *mutex);
                <pthread.h>                                          стр. 449
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_mutex_trylock(pthread_mutex_t *mutex);
                <pthread.h>                                          стр. 449
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_mutex_unlock(pthread_mutex_t *mutex);
                <pthread.h>                                          стр. 449
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_once(pthread_once_t *initflag, void (*initfn)(void);
                <pthread.h>                                          стр. 488
                pthread_once_t initflag = PTHREAD_ONCE_INIT;
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
                <pthread.h>                                          стр. 479
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int   pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
                              int *restrict pshared);
                <pthread.h>                                          стр. 479
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
                <pthread.h>                                          стр. 479
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                int pshared);
                <pthread.h>                                          стр. 479
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
                <pthread.h>                                          стр. 457
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                       const pthread_rwlockattr_t *restrict attr);
                <pthread.h>                                          стр. 457
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
                <pthread.h>                                          стр. 457
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
                    <pthread.h>                                    стр. 458
                    Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
                    <pthread.h>                                    стр. 458
                    Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
                    <pthread.h>                                    стр. 457
                    Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
                    <pthread.h>                                    стр. 457
                    Возвращает 0 в случае успеха, код ошибки в случае неудачи

pthread_t pthread_self(void);
                    <pthread.h>                                    стр. 433
                    Возвращает идентификатор вызывающего потока

int     pthread_setcancelstate(int state, int *oldstate);
                    <pthread.h>                                    стр. 491
                    Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_setcanceltype(int type, int *oldtype);
                    <pthread.h>                                    стр. 493
                    Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_setconcurrency(int level);
                    <pthread.h>                                    стр. 471
                    Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_setspecific(pthread_key_t key, const void *value);
                    <pthread.h>                                    стр. 489
                    Возвращает 0 в случае успеха, код ошибки в случае неудачи

int     pthread_sigmask(int how, const sigset_t *restrict set,
                        sigset_t *restrict oset);
                    <signal.h>                                     стр. 494
                    Возвращает 0 в случае успеха, код ошибки в случае неудачи

void    pthread_testcancel(void);
                    <pthread.h>                                    стр. 493

char    *ptsname(int filedes);
                    <stdlib.h>                                     стр. 789
                    Возвращает указатель на имя подчиненного PTY
                    в случае успеха, NULL в случае ошибки
                    Платформы: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

int     putc(int c, FILE *fp);
                    <stdio.h>                                      стр. 192
                    Возвращает символ c в случае успеха, EOF в случае ошибки

int     putchar(int c);
                    <stdio.h>                                      стр. 192
                    Возвращает символ c в случае успеха, EOF в случае ошибки
```

```
int     putchar_unlocked(int c);
                    <stdio.h>                                          стр. 483
```
Возвращает символ *c* в случае успеха, EOF в случае ошибки

```
int     putc_unlocked(int c, FILE *fp);
                    <stdio.h>                                          стр. 483
```
Возвращает символ *c* в случае успеха, EOF в случае ошибки

```
int     putenv(char *str);
                    <stdlib.h>                                         стр. 251
```
Возвращает 0 в случае успеха, ненулевое значение в случае ошибки

```
int     putmsg(int filedes, const struct strbuf *ctlptr,
               const struct strbuf *dataptr, int flag);
                    <stropts.h>                                        стр. 548
                    flag: 0, RS_HIPRI
```
Возвращает 0 в случае успеха, −1 в случае ошибки
Платформы: Linux 2.4.22, Solaris 9

```
int     putpmsg(int filedes, const struct strbuf *ctlptr,
                const struct strbuf *dataptr, int band, int flag);
                    <stropts.h>                                        стр. 548
                    flag: 0, MSG_HIPRI, MSG_BAND
```
Возвращает 0 в случае успеха, −1 в случае ошибки
Платформы: Linux 2.4.22, Solaris 9

```
int     puts(const char *str);
                    <stdio.h>                                          стр. 193
```
Возвращает неотрицательное значение в случае успеха, EOF в случае ошибки

```
ssize_t pwrite(int filedes, const void *buf, size_t nbytes, off_t offset);
                    <unistd.h>                                         стр. 114
```
Возвращает количество записанных байт в случае успеха, −1 в случае ошибки

```
int     raise(int signo);
                    <signal.h>                                         стр. 383
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
ssize_t read(int filedes, void *buf, size_t nbytes);
                    <unistd.h>                                         стр. 105
```
Возвращает количество прочитанных байт в случае успеха, 0 по достижении конца файла, −1 в случае ошибки

```
struct
dirent  *readdir(DIR *dp);
                    <dirent.h>                                         стр. 167
```
Возвращает указатель в случае успеха, NULL по достижении конца каталога, −1 в случае ошибки

```
int     readlink(const char *restrict pathname, char *restrict buf,
                 size_t bufsize);
                    <unistd.h>                                         стр. 160
```
Возвращает количество прочитанных байт в случае успеха, −1 в случае ошибки

```
ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);
                    <sys/uio.h>                                          стр. 571
                    Возвращает количество прочитанных байт
                    в случае успеха, –1 в случае ошибки

void    *realloc(void *ptr, size_t newsize);
                    <stdlib.h>                                           стр. 245
                    Возвращает непустой указатель в случае успеха,
                    NULL в случае ошибки

ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
                    <sys/socket.h>                                       стр. 666
                    flags: 0, MSG_PEEK, MSG_OOB, MSG_WAITALL
                    Возвращает длину сообщения в байтах, 0 – если нет
                    доступных сообщений и на другом конце соединения
                    была запрещена операция записи, –1 в случае ошибки
                    Платформы: флаг MSG_TRUNC в Linux 2.4.22

ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                struct sockaddr *restrict addr, socklen_t *restrict addrlen);
                    <sys/socket.h>                                       стр. 667
                    flags: 0, MSG_PEEK, MSG_OOB, MSG_WAITALL
                    Возвращает длину сообщения в байтах, 0 – если нет
                    доступных сообщений и на другом конце соединения
                    была запрещена операция записи, –1 в случае ошибки
                    Платформы: флаг MSG_TRUNC в Linux 2.4.22

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
                    <sys/socket.h>                                       стр. 668
                    flags: 0, MSG_PEEK, MSG_OOB, MSG_WAITALL
                    Возвращает длину сообщения в байтах, 0 – если нет
                    доступных сообщений и на другом конце соединения
                    была запрещена операция записи, –1 в случае ошибки
                    Платформы: флаг MSG_TRUNC в Linux 2.4.22

int     remove(const char *pathname);
                    <stdio.h>                                            стр. 156
                    Возвращает 0 в случае успеха, –1 в случае ошибки

int     rename(const char *oldname, const char *newname);
                    <stdio.h>                                            стр. 156
                    Возвращает 0 в случае успеха, –1 в случае ошибки

void    rewind(FILE *fp);
                    <stdio.h>                                            стр. 198

void    rewinddir(DIR *dp);
                    <dirent.h>                                           стр. 167

int     rmdir(const char *pathname);
                    <unistd.h>                                           стр. 166
                    Возвращает 0 в случае успеха, –1 в случае ошибки
```

```
int     scanf(const char *restrict format, ...);
                    <stdio.h>                                          стр. 203
```
Возвращает количество введенных элементов, EOF – по достижении конца файла или в случае ошибки перед выполнением преобразования

```
void    seekdir(DIR *dp, long loc);
                    <dirent.h>                                         стр. 167
```

```
int     select(int maxfdp1, fd_set *restrict readfds, fd_set *restrict writefds,
               fd_set *restrict exceptfds, struct timeval *restrict tvptr);
                    <sys/select.h>                                     стр. 561
```
Возвращает количество дескрипторов, готовых к выполнению операции, 0 – по истечении тайм-аута, –1 в случае ошибки

```
int     semctl(int semid, int semnum, int cmd, ... /* union semun arg */);
                    <sys/sem.h>                                        стр. 624
                    cmd: IPC_STAT, IPC_SET, IPC_RMID, GETPID, GETNCNT,
                         GETZCNT, GETVAL, SETVAL, GETALL, SETALL
```
Возвращаемое значение зависит от команды

```
int     semget(key_t key, int nsems, int flag);
                    <sys/sem.h>                                        стр. 623
                    flag: 0, IPC_CREAT, IPC_EXCL
```
Возвращает идентификатор семафора в случае успеха, –1 в случае ошибки

```
int     semop(int semid, struct sembuf semoparray[], size_t nops);
                    <sys/sem.h>                                        стр. 625
```
Возвращает 0 в случае успеха, –1 в случае ошибки

```
ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
                    <sys/socket.h>                                     стр. 664
                    flags: 0, MSG_DONTROUTE, MSG_EOR, MSG_OOB
```
Возвращает количество переданных байт в случае успеха, –1 в случае ошибки
Платформы: флаг MSG_DONTWAIT в FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3, флаг MSG_EOR отсутствует в Solaris 9

```
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
                    <sys/socket.h>                                     стр. 666
                    flags: 0, MSG_DONTROUTE, MSG_EOR, MSG_OOB
```
Возвращает количество переданных байт в случае успеха, -1 в случае ошибки.
Платформы: флаг MSG_DONTWAIT в FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3, флаг MSG_EOR отсутствует в Solaris 9

```
ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags,
               const struct sockaddr *destaddr, socklen_t destlen);
                    <sys/socket.h>                                     стр. 665
                    flags: 0, MSG_DONTROUTE, MSG_EOR, MSG_OOB
```
Возвращает количество переданных байт в случае успеха, –1 в случае ошибки
Платформы: флаг MSG_DONTWAIT в FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3, флаг MSG_EOR отсутствует в Solaris 9

```
void    setbuf(FILE *restrict fp, char *restrict buf);
                    <stdio.h>                                   стр. 185

int     setegid(gid_t gid);
                    <unistd.h>                                  стр. 303
                    Возвращает 0 в случае успеха, –1 в случае ошибки

int     setenv(const char *name, const char *value, int rewrite);
                    <stdlib.h>                                  стр. 251
                    Возвращает 0 в случае успеха, ненулевое значение
                    в случае ошибки

int     seteuid(uid_t uid);
                    <unistd.h>                                  стр. 303
                    Возвращает 0 в случае успеха, –1 в случае ошибки

int     setgid(gid_t gid);
                    <unistd.h>                                  стр. 298
                    Возвращает 0 в случае успеха, –1 в случае ошибки

void    setgrent(void);
                    <grp.h>                                     стр. 220

int     setgroups(int ngroups, const gid_t grouplist[]);
                    <grp.h>    /* в Linux */                    стр. 221
                    <unistd.h> /* в FreeBSD, Mac OS X и Solaris */
                    Возвращает 0 в случае успеха, –1 в случае ошибки

void    sethostent(int stayopen);
                    <netdb.h>                                   стр. 651

int     setjmp(jmp_buf env);
                    <setjmp.h>                                  стр. 254
                    Возвращает 0, если вызывается непосредственно, ненулевое
                    значение, если возврат произошел вследствие вызова longjmp

int     setlogmask(int maskpri);
                    <syslog.h>                                  стр. 511
                    Возвращает предыдущее значение маски приоритета
                    журналируемых сообщений

void    setnetent(int stayopen);
                    <netdb.h>                                   стр. 652

int     setpgid(pid_t pid, pid_t pgid);
                    <unistd.h>                                  стр. 335
                    Возвращает 0 в случае успеха, –1 в случае ошибки

void    setprotoent(int stayopen);
                    <netdb.h>                                   стр. 653

void    setpwent(void);
                    <pwd.h>                                     стр. 216

int     setregid(gid_t rgid, gid_t egid);
                    <unistd.h>                                  стр. 302
                    Возвращает 0 в случае успеха, –1 в случае ошибки
```

```
int     setreuid(uid_t ruid, uid_t euid);
                <unistd.h>                                              стр. 302
                Возвращает 0 в случае успеха, –1 в случае ошибки

int     setrlimit(int resource, const struct rlimit *rlptr);
                <sys/resource.h>                                        стр. 259
                Возвращает 0 в случае успеха, ненулевое значение
                в случае ошибки

void    setservent(int stayopen);
                <netdb.h>                                               стр. 653

pid_t   setsid(void);
                <unistd.h>                                              стр. 336
                Возвращает идентификатор группы процессов
                в случае успеха, –1 в случае ошибки

int     setsockopt(int sockfd, int level, int option, const void *val, socklen_t len);
                <sys/socket.h>                                          стр. 679
                Возвращает 0 в случае успеха, –1 в случае ошибки

void    setspent(void);
                <shadow.h>                                              стр. 219
                Платформы: Linux 2.4.22, Solaris 9

int     setuid(uid_t uid);
                <unistd.h>                                              стр. 298
                Возвращает 0 в случае успеха, –1 в случае ошибки

int     setvbuf(FILE *restrict fp, char *restrict buf, int mode, size_t size);
                <stdio.h>                                               стр. 185
                mode: _IOFBF, _IOLBF, _IONBF
                Возвращает 0 в случае успеха, ненулевое значение
                в случае ошибки

void    *shmat(int shmid, const void *addr, int flag);
                <sys/shm.h>                                             стр. 631
                flag: 0, SHM_RND, SHM_RDONLY
                Возвращает указатель на сегмент разделяемой
                памяти в случае успеха, –1 в случае ошибки

int     shmctl(int shmid, int cmd, struct shmid_ds *buf);
                <sys/shm.h>                                             стр. 630
                cmd: IPC_STAT, IPC_SET, IPC_RMID,
                     SHM_LOCK, SHM_UNLOCK
                Возвращает 0 в случае успеха, –1 в случае ошибки

int     shmdt(void *addr);
                <sys/shm.h>                                             стр. 632
                Возвращает 0 в случае успеха, –1 в случае ошибки

int     shmget(key_t key, int size, int flag);
                <sys/shm.h>                                             стр. 629
                flag: 0, IPC_CREAT, IPC_EXCL
                Возвращает идентификатор сегмента разделяемой
                памяти в случае успеха, –1 в случае ошибки
```

```
int     shutdown(int sockfd, int how);
                 <sys/socket.h>                                          стр. 646
                 how: SHUT_RD, SHUT_WR, SHUT_RDWR
                 Возвращает 0 в случае успеха, –1 в случае ошибки

int     sig2str(int signo, char *str);
                 <signal.h>                                              стр. 428
                 Возвращает 0 в случае успеха, –1 в случае ошибки
                 Платформы: Solaris 9

int     sigaction(int signo, const struct sigaction *restrict act,
                  struct sigaction *restrict oact);
                 <signal.h>                                              стр. 397
                 Возвращает 0 в случае успеха, –1 в случае ошибки

int     sigaddset(sigset_t *set, int signo);
                 <signal.h>                                              стр. 391
                 Возвращает 0 в случае успеха, –1 в случае ошибки

int     sigdelset(sigset_t *set, int signo);
                 <signal.h>                                              стр. 391
                 Возвращает 0 в случае успеха, –1 в случае ошибки

int     sigemptyset(sigset_t *set);
                 <signal.h>                                              стр. 391
                 Возвращает 0 в случае успеха, –1 в случае ошибки

int     sigfillset(sigset_t *set);
                 <signal.h>                                              стр. 391
                 Возвращает 0 в случае успеха, –1 в случае ошибки

int     sigismember(const sigset_t *set, int signo);
                 <signal.h>                                              стр. 391
                 Возвращает 1, если утверждение истинно, 0 – если ложно

void    siglongjmp(sigjmp_buf env, int val);
                 <setjmp.h>                                              стр. 404
                 Эта функция никогда не возвращает управление

void    (*signal(int signo, void (*func)(int)))(int);
                 <signal.h>                                              стр. 367
                 Возвращает предыдущую диспозицию сигнала
                 в случае успеха, SIG_ERR в случае ошибки

int     sigpending(sigset_t *set);
                 <signal.h>                                              стр. 395
                 Возвращает 0 в случае успеха, –1 в случае ошибки

int     sigprocmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
                 <signal.h>                                              стр. 393
                 how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
                 Возвращает 0 в случае успеха, –1 в случае ошибки
```

```
int     sigsetjmp(sigjmp_buf env, int savemask);
                <setjmp.h>                                          стр. 404
```
Возвращает 0, если вызывается непосредственно, ненулевое значение, если возврат произошел вследствие вызова `siglongjmp`

```
int     sigsuspend(const sigset_t *sigmask);
                <signal.h>                                          стр. 408
```
Возвращает −1 с кодом ошибки `EINTR` в переменной `errno`

```
int     sigwait(const sigset_t *restrict set, int *restrict signop);
                <signal.h>                                          стр. 494
```
Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
unsigned
int     sleep(unsigned int seconds);
                <unistd.h>                                          стр. 422
```
Возвращает 0 или количество секунд, оставшихся до окончания приостановки

```
int     snprintf(char *restrict buf, size_t n,
                 const char *restrict format, ...);
                <stdio.h>                                           стр. 200
```
Возвращает количество символов, сохраненных в массиве, в случае успеха, отрицательное значение в случае ошибки преобразования

```
int     sockatmark(int sockfd);
                <sys/socket.h>                                      стр. 683
```
Возвращает 1, если достигнут маркер, 0 – если нет, −1 в случае ошибки

```
int     socket(int domain, int type, int protocol);
                <sys/socket.h>                                      стр. 643
                type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET,
```
Возвращает дескриптор файла (сокета) в случае успеха, −1 в случае ошибки

```
int     socketpair(int domain, int type, int protocol, int sockfd[2]);
                <sys/socket.h>                                      стр. 695
                type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET,
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     sprintf(char *restrict buf, const char *restrict format, ...);
                <stdio.h>                                           стр. 200
```
Возвращает количество символов, сохраненных в массиве, в случае успеха, отрицательное значение в случае ошибки преобразования

```
int     sscanf(const char *restrict buf, const char *restrict format, ...);
                <stdio.h>                                           стр. 203
```
Возвращает количество введенных элементов, `EOF` – по достижении конца файла или в случае ошибки перед выполнением преобразования

```
int     stat(const char *restrict pathname, struct stat *restrict buf);
                    <sys/stat.h>                                        стр. 129
                    Возвращает 0 в случае успеха, –1 в случае ошибки

int     str2sig(const char *str, int *signop);
                    <signal.h>                                          стр. 428
                    Возвращает 0 в случае успеха, –1 в случае ошибки
                    Платформы: Solaris 9

char    *strerror(int errnum);
                    <string.h>                                          стр. 42
                    Возвращает указатель на строку сообщения

size_t  strftime(char *restrict buf, size_t maxsize,
                 const char *restrict format,
                 const struct tm *restrict tmptr);
                    <time.h>                                            стр. 230
                    Возвращает количество символов, сохраненных
                    в массиве, если достаточно места, 0 – в противном случае

char    *strsignal(int signo);
                    <string.h>                                          стр. 428
                    Возвращает указатель на строку с описанием сигнала

int     symlink(const char *actualpath, const char *sympath);
                    <unistd.h>                                          стр. 160
                    Возвращает 0 в случае успеха, –1 в случае ошибки

void    sync(void);
                    <unistd.h>                                          стр. 117

long    sysconf(int name);
                    <unistd.h>                                          стр. 76
                    name: _SC_ARG_MAX, _SC_ATEXIT_MAX, _SC_CHILD_MAX,
                          _SC_CLK_TCK, _SC_COLL_WEIGHTS_MAX,
                          _SC_HOST_NAME_MAX, _SC_IOV_MAX, _SC_JOB_CONTROL,
                          _SC_LINE_MAX, _SC_LOGIN_NAME_MAX, _SC_NGROUPS_MAX,
                          _SC_OPEN_MAX, _SC_PAGESIZE, _SC_PAGE_SIZE,
                          _SC_READER_WRITER_LOCKS, _SC_RE_DUP_MAX,
                          _SC_SAVED_IDS, _SC_SHELL, _SC_STREAM_MAX,
                          _SC_SYMLOOP_MAX, _SC_TTY_NAME_MAX, _SC_TZNAME_MAX,
                          _SC_VERSION, _SC_XOPEN_CRYPT, _SC_XOPEN_LEGACY,
                          _SC_XOPEN_REALTIME, _SC_XOPEN_REALTIME_THREADS,
                          _SC_XOPEN_VERSION
                    Возвращает соответствующее значение в случае успеха,
                    –1 в случае ошибки

void    syslog(int priority, char *format, ...);
                    <syslog.h>                                          стр. 511

int     system(const char *cmdstring);
                    <stdlib.h>                                          стр. 309
                    Возвращает код завершения командной оболочки
```

```
int     tcdrain(int filedes);
                <termios.h>                                          стр. 759
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     tcflow(int filedes, int action);
                <termios.h>                                          стр. 759
                action: TCOOFF, TCOON, TCIOFF, TCION
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     tcflush(int filedes, int queue);
                <termios.h>                                          стр. 759
                queue: TCIFLUSH, TCOFLUSH, TCIOFLUSH
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     tcgetattr(int filedes, struct termios *termptr);
                <termios.h>                                          стр. 748
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
pid_t   tcgetpgrp(int filedes);
                <unistd.h>                                           стр. 339
```
Возвращает идентификатор группы процессов переднего плана в случае успеха, −1 в случае ошибки

```
pid_t   tcgetsid(int filedes);
                <termios.h>                                          стр. 340
```
Возвращает идентификатор группы процессов лидера сессии в случае успеха, −1 в случае ошибки

```
int     tcsendbreak(int filedes, int duration);
                <termios.h>                                          стр. 759
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     tcsetattr(int filedes, int opt, const struct termios *termptr);
                <termios.h>                                          стр. 748
                opt: TCSANOW, TCSADRAIN, TCSAFLUSH
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     tcsetpgrp(int filedes, pid_t pgrpid);
                <unistd.h>                                           стр. 339
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
long    telldir(DIR *dp);
                <dirent.h>                                           стр. 167
```
Возвращает значение текущей позиции в каталоге, ассоциированном с *dp*

```
char    *tempnam(const char *directory, const char *prefix);
                <stdio.h>                                            стр. 209
```
Возвращает указатель на строку с уникальным именем файла

```
time_t  time(time_t *calptr);
                <time.h>                                             стр. 227
```
Возвращает значение текущего времени в случае успеха, −1 в случае ошибки

```
clock_t times(struct tms *buf);
                    <sys/times.h>                                    стр. 321
```
Возвращает значение общего времени выполнения процесса в тактах в случае успеха, −1 в случае ошибки

```
FILE    *tmpfile(void);
                    <stdio.h>                                        стр. 207
```
Возвращает указатель на структуру FILE в случае успеха, NULL в случае ошибки

```
char    *tmpnam(char *ptr);
                    <stdio.h>                                        стр. 207
```
Возвращает указатель на строку с уникальным именем файла

```
int     truncate(const char *pathname, off_t length);
                    <unistd.h>                                       стр. 150
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
char    *ttyname(int filedes);
                    <unistd.h>                                       стр. 761
```
Возвращает указатель на строку с именем специального файла устройства терминала, NULL в случае ошибки

```
mode_t  umask(mode_t cmask);
                    <sys/stat.h>                                     стр. 141
```
Возвращает предыдущее значение маски режима создания файлов

```
int     uname(struct utsname *name);
                    <sys/utsname.h>                                  стр. 225
```
Возвращает неотрицательное значение в случае успеха, −1 в случае ошибки

```
int     ungetc(int c, FILE *fp);
                    <stdio.h>                                        стр. 191
```
Возвращает символ c в случае успеха, EOF в случае ошибки

```
int     unlink(const char *pathname);
                    <unistd.h>                                       стр. 154
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     unlockpt(int filedes);
                    <stdlib.h>                                       стр. 788
```
Возвращает 0 в случае успеха, −1 в случае ошибки
Платформы: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

```
void    unsetenv(const char *name);
                    <stdlib.h>                                       стр. 251

int     utime(const char *pathname, const struct utimbuf *times);
                    <utime.h>                                        стр. 162
```
Возвращает 0 в случае успеха, −1 в случае ошибки

```
int     vfprintf(FILE *restrict fp, const char *restrict format, va_list arg);
                <stdarg.h>                                                    стр. 202
                <stdio.h>
```
Возвращает количество выведенных символов в случае успеха, отрицательное значение в случае ошибки

```
int     vfscanf(FILE *restrict fp, const char *restrict format, va_list arg);
                <stdarg.h>                                                    стр. 203
                <stdio.h>
```
Возвращает количество введенных элементов, EOF – в случае ошибки ввода или по достижении конца файла перед выполнением преобразования

```
int     vprintf(const char *restrict format, va_list arg);
                <stdarg.h>                                                    стр. 202
                <stdio.h>
```
Возвращает количество выведенных символов в случае успеха, отрицательное значение в случае ошибки

```
int     vscanf(const char *restrict format, va_list arg);
                <stdarg.h>                                                    стр. 204
                <stdio.h>
```
Возвращает количество введенных элементов, EOF – в случае ошибки ввода или по достижении конца файла перед выполнением преобразования

```
int     vsnprintf(char *restrict buf, size_t n, const char *restrict format,
                  va_list arg);
                <stdarg.h>                                                    стр. 202
                <stdio.h>
```
Возвращает количество символов, сохраненных в массиве, в случае успеха, отрицательное значение в случае ошибки преобразования

```
int     vsprintf(char *restrict buf, const char *restrict format,
                 va_list arg);
                <stdarg.h>                                                    стр. 202
                <stdio.h>
```
Возвращает количество символов, сохраненных в массиве, в случае успеха, отрицательное значение в случае ошибки преобразования

```
int     vsscanf(const char *restrict buf, const char *restrict format,
                va_list arg);
                <stdarg.h>                                                    стр. 204
                <stdio.h>
```
Возвращает количество введенных элементов, EOF – в случае ошибки ввода или по достижении конца файла перед выполнением преобразования

```
void    vsyslog(int priority, const char *format, va_list arg);
                <syslog.h>                                                    стр. 514
                <stdarg.h>
```

```
pid_t   wait(int *statloc);
                    <sys/wait.h>                                        стр. 279
                    Возвращает идентификатор процесса в случае успеха,
                    −1 в случае ошибки

int     waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                    <sys/wait.h>                                        стр. 285
                    idtype: P_PID, P_PGID, P_ALL
                    options: WCONTINUED, WEXITED, WNOHANG, WNOWAIT, WSTOPPED
                    Возвращает 0 в случае успеха, −1 в случае ошибки
                    Платформы: Solaris 9

pid_t   waitpid(pid_t pid, int *statloc, int options);
                    <sys/wait.h>                                        стр. 279
                    options: 0, WCONTINUED, WNOHANG, WUNTRACED
                    Возвращает идентификатор процесса в случае успеха,
                    −1 в случае ошибки

pid_t   wait3(int *statloc, int options, struct rusage *rusage);
                    <sys/types.h>                                       стр. 287
                    <sys/wait.h>
                    <sys/time.h>
                    <sys/resource.h>
                    options: 0, WNOHANG, WUNTRACED
                    Возвращает идентификатор процесса в случае успеха,
                    0 или −1 в случае ошибки

pid_t   wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
                    <sys/types.h>                                       стр. 287
                    <sys/wait.h>
                    <sys/time.h>
                    <sys/resource.h>
                    options: 0, WNOHANG, WUNTRACED
                    Возвращает идентификатор процесса в случае успеха,
                    0 или −1 в случае ошибки

ssize_t write(int filedes, const void *buf, size_t nbytes);
                    <unistd.h>                                          стр. 106
                    Возвращает количество записанных байт в случае
                    успеха, −1 в случае ошибки

ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
                    <sys/uio.h>                                         стр. 571
                    Возвращает количество записанных байт
                    в случае успеха, −1 в случае ошибки
```

B

Различные исходные тексты

B.1. Наш заголовочный файл

Большинство программ в книге подключают заголовочный файл apue.h, содержимое которого приводится в листинге B.1. Он определяет значения констант (таких как MAXLINE) и прототипы наших собственных функций.

Как правило, программы должны подключать следующие заголовочные файлы: <stdio.h>, <stdlib.h> (где определен прототип функции exit) и <unistd.h> (который содержит прототипы всех стандартных функций UNIX). Поэтому наш заголовочный файл автоматически подключает эти системные заголовочные файлы вместе с файлом <string.h>. Это позволило также сократить размер листингов в книге.

Листинг B.1. Наш заголовочный файл apue.h

```
/*
 * Наш собственный заголовочный файл, который подключается перед любыми
 * стандартными системными заголовочными файлами
 */
#ifndef _APUE_H
#define _APUE_H

#define _XOPEN_SOURCE    600 /* Single UNIX Specification, Version 3 */
#include <sys/types.h>       /* некоторые системы требуют этот заголовок */
#include <sys/stat.h>
#include <sys/termios.h>     /* структура winsize */
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif
#include <stdio.h>           /* для удобства */
#include <stdlib.h>          /* для удобства */
#include <stddef.h>          /* макрос offsetof */
#include <string.h>          /* для удобства */
#include <unistd.h>          /* для удобства */
#include <signal.h>          /* константа SIG_ERR */
```

```
#define MAXLINE 4096        /* максимальная длина строки */

/*
 * Права доступа по умолчанию к создаваемым файлам.
 */
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Права доступа по умолчанию к создаваемым каталогам.
 */
#define DIR_MODE (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void Sigfunc(int);          /* обработчики сигналов */

#if defined(SIG_IGN) && !defined(SIG_ERR)
#define SIG_ERR ((Sigfunc *)-1)
#endif

#define min(a,b) ((a) < (b) ? (a) : (b))
#define max(a,b) ((a) > (b) ? (a) : (b))

/*
 * Прототипы наших собственных функций.
 */
char *path_alloc(int *);              /* листинг 2.3 */
long open_max(void);                  /* листинг 2.4 */
void clr_fl(int, int);                /* листинг 3.5 */
void set_fl(int, int);                /* листинг 3.5 */
void pr_exit(int);                    /* листинг 8.3 */
void pr_mask(const char *);           /* листинг 10.10 */
Sigfunc *signal_intr(int, Sigfunc *); /* листинг 10.12 */

int tty_cbreak(int);                  /* листинг 18.10 */
int tty_raw(int);                     /* листинг 18.10 */
int tty_reset(int);                   /* листинг 18.10 */
void tty_atexit(void);                /* листинг 18.10 */
#ifdef ECHO                 /* только если подключен файл <termios.h> */
struct termios *tty_termios(void);    /* листинг 18.10 */
#endif

void sleep_us(unsigned int);              /* упражнение 14.6 */
ssize_t readn(int, void *, size_t);       /* листинг 14.11 */
ssize_t writen(int, const void *, size_t); /* листинг 14.11 */
void daemonize(const char *);             /* листинг 13.1 */

int s_pipe(int *);                        /* листинги 17.2 и 17.6 */
int recv_fd(int, ssize_t (*func)(int,
           const void *, size_t));        /* листинги 17.13 и 17.15 */
int send_fd(int, int);                    /* листинги 17.12 и 17.14 */
int send_err(int, int,
             const char *);               /* листинг 17.11 */
int serv_listen(const char *);            /* листинги 17.3 и 17.8 */
int serv_accept(int, uid_t *);            /* листинги 17.4 и 17.9 */
int cli_conn(const char *);               /* листинги 17.5 и 17.10 */
int buf_args(char *, int (*func)(int,
             char **));                   /* листинг 17.24 */
```

```
int ptym_open(char *, int);                /* листинги 19.1, 19.2 и 19.3 */
int ptys_open(char *);                     /* листинги 19.1, 19.2 и 19.3 */
#ifdef TIOCGWINSZ
pid_t pty_fork(int *, char *, int, const struct termios *,
               const struct winsize *);    /* листинг 19.4 */
#endif

int lock_reg(int, int, int, off_t, int, off_t);   /* листинг 14.2 */
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

pid_t lock_test(int, int, off_t, int, off_t); /* листинг 14.3 */

#define is_read_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)

void err_dump(const char *, ...);     /* приложение B */
void err_msg(const char *, ...);
void err_quit(const char *, ...);
void err_exit(int, const char *, ...);
void err_ret(const char *, ...);
void err_sys(const char *, ...);

void log_msg(const char *, ...);      /* приложение B */
void log_open(const char *, int, int);
void log_quit(const char *, ...);
void log_ret(const char *, ...);
void log_sys(const char *, ...);

void TELL_WAIT(void);                 /* предок/потомок из раздела 8.9 */
void TELL_PARENT(pid_t);
void TELL_CHILD(pid_t);
void WAIT_PARENT(void);
void WAIT_CHILD(void);

#endif /* _APUE_H */
```

Наш заголовочный файл подключается первым, перед всеми обычными системными заголовочными файлами, по той причине, что это позволяет нам дать определения, которые могут потребоваться другим заголовочным файлам, установить порядок подключения заголовочных файлов, а также переопределить некоторые значения, чтобы сгладить и скрыть различия между системами.

B.2. Стадартные процедуры обработки ошибок

В большинстве наших примеров используются два набора функций обработки ошибочных ситуаций. Один набор включает в себя функции, имена которых начинаются с префикса `err_`, они выводят сообщения об ошибках на стандартное устройство вывода сообщений об ошибках. Другой набор включает в себя функции, имена которых начинаются с префикса `log_`, они предназначены для использования в процессах-демонах (глава 13), которые, как правило, не имеют управляющего терминала.

Эти наборы функций позволяют обрабатывать ошибочные ситуации всего одной строчкой в программе, например

```
if (error condition)
    err_dump(формат в стиле printf с любым количеством аргументов);
```

вместо

```
if (error condition) {
    char buf[200];
    sprintf(buf, формат в стиле printf с любым количеством аргументов);
    perror(buf);
    abort();
}
```

Наши функции обработки ошибок используют возможность передачи списка аргументов переменной длины, которая определяется стандартом ISO C. Дополнительные сведения вы найдете в разделе 7.3 [Kernighan and Ritchie 1988]. Важно понимать, что функциональная возможность передачи списка аргументов переменной длины из стандарта ISO C отличается от функциональности `varargs`, которая предоставлялась ранними версиями системы (такими как SVR3 и 4.3BSD). Имена макроопределений остались теми же, но аргументы некоторых из них изменились.

В табл. B.1 показаны различия между разными функциями обработки ошибок.

Таблица B.1. Наши стандартные функции обработки ошибок

Функция	Добавляет строку от strerror?	Аргументы для strerror	Завершает процесс?
`err_dump`	Да	errno	`abort();`
`err_exit`	Да	Явный параметр	`exit(1);`
`err_msg`	Нет		`return;`
`err_quit`	Нет		`exit(1);`
`err_ret`	Да	errno	`return;`
`err_sys`	Да	errno	`exit(1);`
`log_msg`	Нет		`return;`
`log_quit`	Нет		`exit(2);`

Функция	Добавляет строку от strerror?	Аргументы для strerror	Завершает процесс?
log_ret	Да	errno	return;
log_sys	Да	errno	exit(2);

В листинге B.2 приводятся исходные тексты функций обработки ошибок, которые выводят сообщения на стандартное устройство вывода сообщений об ошибках.

Листинг B.2. Функции обработки ошибок, которые выводят сообщения на стандртное устройство вывода сообщений об ошибках

```
#include "apue.h"
#include <errno.h>    /* определение переменной errno */
#include <stdarg.h>   /* список аргументов переменной длины ISO C */

static void err_doit(int, int, const char *, va_list);

/*
 * Обработка нефатальных ошибок, связанных с системными вызовами.
 * Выводит сообщение и возвращает управление.
 */
void
err_ret(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
}

/*
 * Обработка фатальных ошибок, связанных с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
err_sys(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Обработка фатальных ошибок, не связанных с системными вызовами.
 * Код ошибки передается в виде аргумента.
 * Выводит сообщение и завершает работу процесса.
 */
```

```
void
err_exit(int error, const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    err_doit(1, error, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Обработка фатальных ошибок, связанных с системными вызовами.
 * Выводит сообщение, создает файл core и завершает работу процесса.
 */
void
err_dump(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    abort();      /* записать дамп памяти в файл и завершить процесс */
    exit(1);      /* этот вызов никогда не должен быть выполнен */
}

/*
 * Обработка нефатальных ошибок, не связанных с системными вызовами.
 * Выводит сообщение и возвращает управление.
 */
void
err_msg(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
}

/*
 * Обработка фатальных ошибок, не связанных с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
err_quit(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
    exit(1);
```

```
}

/*
 * Выводит сообщение и возвращает управление в вызывающую функцию.
 * Вызывающая функция определяет значение флага "errnoflag".
 */
static void
err_doit(int errnoflag, int error, const char *fmt, va_list ap)
{
    char buf[MAXLINE];

    vsnprintf(buf, MAXLINE, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf), ": %s",
                 strerror(error));
    strcat(buf, "\n");
    fflush(stdout); /* в случае, когда stdout и stderr - */
                    /* одно и то же устройство */
    fputs(buf, stderr);
    fflush(NULL);   /* сбрасывает все выходные потоки */
}
```

В листинге B.3 приводятся исходные тексты функций семейства `log_XXX`**. Они требуют, чтобы в вызывающем процессе была определена глобальная переменная** `log_to_stderr`**. Эта переменная должна содержать ненулевое значение, если процесс выполняется не как демон. В этом случае сообщения будут выводиться на стандартное устройство вывода сообщений об ошибках. Если содержит** `log_to_stderr` **0, то для вывода сообщений будет использоваться функция** `syslog` **(раздел 13.4).**

Листинг B.3. Функции обработки ошибок для демонов

```
/*
 * Процедуры обработки ошибок для программ, которые могут работать как демоны.
 */

#include "apue.h"
#include <errno.h>     /* определение переменной errno */
#include <stdarg.h>    /* список аргументов переменной длины ISO C */
#include <syslog.h>

static void log_doit(int, int, const char *, va_list ap);

/*
 * В вызывающем процессе должна быть определена и установлена эта переменная:
 * ненулевое значение - для интерактивных программ, нулевое - для демонов
 */
extern int log_to_stderr;

/*
 * Инициализировать syslog(), если процесс работает режиме демона.
 */
void
log_open(const char *ident, int option, int facility)
```

```
{
    if (log_to_stderr == 0)
        openlog(ident, option, facility);
}

/*
 * Обработка нефатальных ошибок, связанных с системными вызовами. Выводит сообщение,
 * соответствующее содержимому переменной errno, и возвращает управление.
 */
void
log_ret(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Обработка фатальных ошибок, связанных с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
log_sys(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Обработка нефатальных ошибок, не связанных с системными вызовами.
 * Выводит сообщение и возвращает управление.
 */
void
log_msg(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Обработка фатальных ошибок, не связанных с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
log_quit(const char *fmt, ...)
```

```
{
    va_list ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Выводит сообщение и возвращает управление в вызывающую функцию.
 * Вызывающая функция должна определить значения аргументов
 * "errnoflag" и "priority".
 */
static void
log_doit(int errnoflag, int priority, const char *fmt, va_list ap)
{
    int errno_save;
    char buf[MAXLINE];

    errno_save = errno; /* значение,  которое вызывающая функция, возможно, */
                        /* пожелает вывести */
    vsnprintf(buf, MAXLINE, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf), ": %s",
                 strerror(errno_save));
    strcat(buf, "\n");
    if (log_to_stderr) {
        fflush(stdout);
        fputs(buf, stderr);
        fflush(stderr);
    } else {
        syslog(priority, buf);
    }
}
```

C

Варианты решения некоторых упражнений

Глава 1

1.1. Для решения этого упражнения мы будем использовать следующие два аргумента команды `ls`(1): `-i`, который заставляет команду `ls` выводить номера индексных узлов файлов и каталогов (более подробно об индексных узлах рассказывается в разделе 4.14), и `-d`, который заставляет ее выводить информацию только о каталогах.

В результате мы получим следующее:

```
$ ls -ldi /etc/. /etc/..        ключ -i заставляет выводить номера
                                индексных узлов
 162561 drwxr-xr-x  66 root    4096 Feb  5 03:59 /etc/./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /etc/../
$ ls -ldi /. /..                оба каталога . и .. имеют один
                                и тот же номер i-node – 2
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /../
```

1.2. UNIX является многозадачной системой. Следовательно, между запусками нашей программы были запущены какие-то другие процессы.

1.3. Аргумент *ptr* функции `perror` является указателем, поэтому `perror` может изменить содержимое строки, на которую указывает аргумент *ptr*. Однако атрибут `const` говорит о том, что `perror` не изменяет строку, на которую ссылается указатель. С другой стороны, аргумент с кодом ошибки в функции `strerror` является целым числом, а так как он передается по значению, функция `strerror` не сможет изменить его, даже если захочет. (Если вы не совсем понимаете, как передаются и обрабатываются аргументы функций в языке C, обратитесь к разделу 5.2 [Kernighan and Ritchie 1988].)

1.4. Дело в том, что функции `fflush`, `fprintf` и `vprintf` могут изменять содержимое переменной `errno`. Если они изменят это значение, а мы не сохраним его копию, то в результате будет выведено неверное сообщение об ошибке.

1.5. В 2038 году. Проблема может быть решена за счет увеличения размера типа time_t до 64 бит. Если это будет сделано, то для корректной работы всех приложений, использующих 32-битное представление, их необходимо будет пересобрать. Но на самом деле проблема гораздо глубже. Некоторые файловые системы и носители, предназначенные для хранения резервных копий, используют 32-битное представление времени. Они также должны быть обновлены соответствующим образом, но при этом необходимо сохранить совместимость с устаревшим форматом.

1.6. Примерно 248 дней.

Глава 2

2.1. В ОС FreeBSD используется следующий способ. Элементарные типы данных, которые могут быть объявлены в нескольких заголовочных файлах, определяются в файле <machine/_types.h>. Например:

```
#ifndef _MACHINE__TYPES_H_
#define _MACHINE__TYPES_H_

typedef int __int32_t;
typedef unsigned int __uint32_t;
...

typedef __uint32_t __size_t;
...

#endif /* _MACHINE__TYPES_H_ */
```

В каждом из заголовочных файлов, которые могут определять элементарный системный тип данных size_t, можно использовать такую последовательность:

```
#ifndef _SIZE_T_DECLARED
typedef __size_t size_t;
#define _SIZE_T_DECLARED
#endif
```

При таком подходе инструкция typedef для типа size_t будет выполнена всего один раз.

2.3. Если значение константы OPEN_MAX не определено или чрезвычайно велико (то есть равно LONG_MAX), для получения максимально возможного количества открытых файловых дескрипторов для процесса можно использовать функцию getrlimit. Учитывая, что предел для процесса может быть изменен, мы не можем повторно использовать значение, полученное в результате предыдущего вызова (т. к. он мог измениться). Решение приводится в листинге C.1.

Листинг C.1. Альтернативный способ определения максимально возможного количества файловых дескрипторов

```
#include "apue.h"
#include <limits.h>
#include <sys/resource.h>
```

```
#define OPEN_MAX_GUESS 256

long
open_max(void)
{
    long openmax;
    struct rlimit rl;

    if ((openmax = sysconf(_SC_OPEN_MAX)) < 0 ||
      openmax == LONG_MAX) {
        if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
            err_sys("невозможно получить значение предела");
        if (rl.rlim_max == RLIM_INFINITY)
            openmax = OPEN_MAX_GUESS;
        else
            openmax = rl.rlim_max;
    }
    return(openmax);
}
```

Глава 3

3.1. Все дисковые операции ввода-вывода выполняются с использованием буферов блоков, расположенных в пространстве ядра (которые также известны как буферный кэш ядра). Исключением являются операции ввода-вывода с неструктурированными дисковыми устройствами, которые мы не рассматривали. Работа буферного кэша описана в главе 3 [Bach 1986]. Поскольку читаемые или записываемые данные буферизуются ядром, термин *небуферизованный ввод-вывод* скорее означает отсутствие автоматической буферизации в пользовательском процессе при использовании функций `read` и `write`. Каждая из этих функций обращается к единственному системному вызову.

3.3. Каждый вызов функции `open` создает новую запись в таблице файлов. Но поскольку обе операции открывают один и тот же файл, обе записи в таблице файлов будут указывать на одну и ту же запись в таблице виртуальных узлов. Вызов `dup` создает еще одну ссылку на существующую запись в таблице файлов. Диаграмма, соответствующая данной ситуации, показана на рис. С.1. Функция `fcntl` с аргументами `F_SETFD` и `fd1` воздействует только на флаги дескриптора `fd1`. Но с аргументами `F_SETFL` и `fd1` она будет воздействовать на запись в таблице файлов и тем самым на оба дескриптора – `fd1` и `fd2`.

3.4. Если `fd` имеет значение 1, то `dup2(fd, 1)` вернет 1, оставив открытым дескриптор 1. (Вспомните обсуждение из раздела 3.12.) После выполнения трех вызовов `dup2` все три дескриптора будут ссылаться на одну и ту же запись в таблице файлов. Ни один из дескрипторов не будет закрыт.

Однако если `fd` имеет значение 3, после третьего вызова `dup2` на одну и ту же запись в таблице файлов будут ссылаться уже четыре дескриптора. В этом случае нужно закрыть дескриптор с номером 3.

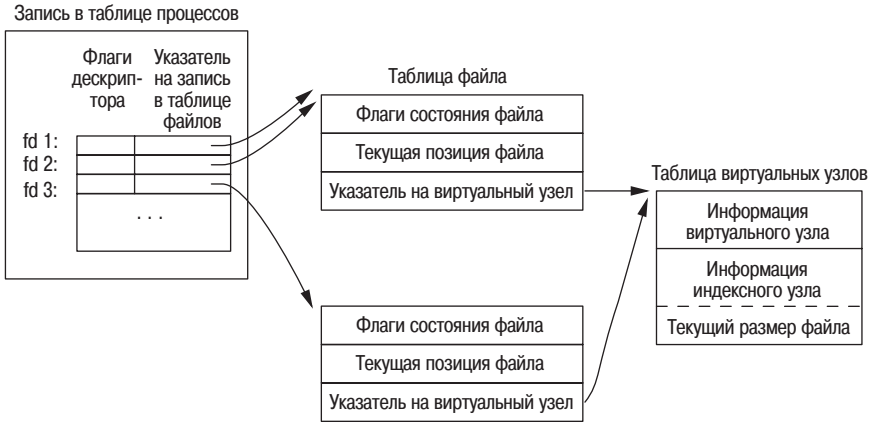


Рис. С.1. *Результат работы функций dup и open*

3.5. Поскольку командные оболочки обрабатывают аргументы командной строки слева направо, команда

```
./a.out > outfile 2>&1
```

сначала перенаправит стандартный вывод в файл `outfile`, а затем продублирует его на дескриптор с номером 2 (стандартный вывод сообщений об ошибках). В результате все, что будет выводиться на стандартный вывод и стандартный вывод сообщений об ошибках, попадет в один и тот же файл. Дескрипторы 1 и 2 будут ссылаться на одну и ту же запись в таблице файлов. Однако команда

```
./a.out 2>&1 > outfile
```

сначала вызовет функцию `dup`, и в результате дескриптор с номером 2 будет ссылаться на терминал (предполагается, что команда была запущена в интерактивном режиме). А затем стандартный вывод будет перенаправлен в файл `outfile`. В результате дескриптор с номером 1 будет ссылаться на запись в таблице файлов, которая соответствует файлу `outfile`, а дескриптор с номером 2 – на запись, которая соответствует терминалу.

3.6. Вы по-прежнему сможете использовать функцию `lseek` и выполнять чтение данных из произвольного места в файле, но вызов функции `write` будет автоматически производить переход в конец файла перед записью данных. Таким образом, в этом случае вы не сможете записать данные в произвольное место в файле.

Глава 4

4.1. Функция `stat` всегда пытается следовать по символическим ссылкам (табл. 4.9), поэтому программа никогда не выведет строку «символическая ссылка». Для приведенного примера, где файл `/dev/cdrom` является символической ссылкой на файл `cdroms/cdrom0` (который в свою очередь

является символической ссылкой на `../scsi/host0/bus0/target0/lun0/cd`), функция `stat` укажет, что файл `/dev/cdrom` является специальным файлом блочного устройства, а не символической ссылкой. Если символическая ссылка ссылается на несуществующий файл, функция `stat` вернет признак ошибки.

4.2. Все биты прав доступа окажутся сброшены:

```
$ umask 777
$ date > temp.foo
$ ls -l temp.foo
---------- 1 sar          0 Feb 5 14:06 temp.foo
```

4.3. Следующий пример показывает, что произойдет, если бит user-read будет сброшен:

```
$ date > foo
$ chmod u-r foo       сбросить бит user-read
$ ls -l foo           проверить права доступа к файлу
--w-r--r--  1 sar         29 Feb 5 14:21 foo
$ cat foo             и попытаться прочитать его
cat: foo: Permission denied
```

4.4. Если попытаться с помощью функции `open` или `creat` создать файл, который уже существует, права доступа к файлу не изменятся. Мы можем убедиться в этом, запустив программу из листинга 4.3:

```
$ rm foo bar          удалить файлы, если они существуют
$ date > foo          создать их и наполнить какими-либо данными
$ date > bar
$ chmod a-r foo bar   сбросить биты права на чтение для всех
$ ls -l foo bar       проверить права доступа
--w-------  1 sar     29 Feb 5 14:25 bar
--w-------  1 sar     29 Feb 5 14:25 foo
$ ./a.out             запустить программу из листинга 4.3
$ ls -l foo bar       проверить права доступа и размеры файлов
--w-------  1 sar     0 Feb 5 14:26 bar
--w-------  1 sar     0 Feb 5 14:26 foo
```

4.5. Размер каталога никогда не может быть равен 0, поскольку файлы каталогов содержат по крайней мере две записи – ссылки на каталоги `.` и `..`. Размер файла символической ссылки определяется количеством символов в имени файла и пути к нему, а имя файла всегда содержит хотя бы один символ.

4.7. При создании файла `core` ядро по умолчанию использует определенные значения битов прав доступа. В данном примере это `rw-r--r--`. Это значение может модифицироваться, а может не модифицироваться значением `umask`. Командная оболочка также определяет значения битов прав доступа по умолчанию, которые устанавливаются для файлов, созданных в результате перенаправления. В данном примере это `rw-rw-rw-`, а это значение всегда модифицируется текущим значением `umask`. В данном примере значением `umask` было число 02.

4.8. Мы не можем воспользоваться командой `du`, так как она требует указания либо имени файла, например

```
du tempfile
```

либо имени каталога:

```
du .
```

Но после возврата из функции `unlink` запись в каталоге для файла `tempfile` исчезает. Команда `du .` не смогла бы показать, что содержимое файла `tempfile` по-прежнему продолжает занимать дисковое пространство. В этом примере мы должны использовать команду `df`, чтобы увидеть фактический объем свободного дискового пространства.

4.9. При удалении ссылки, которая не является последней, сам файл не удаляется. В этом случае обновляется время последнего изменения файла. Но если удаляется последняя ссылка на файл, обновление времени последнего изменения теряет всякий смысл, поскольку вся инфомация о файле (индексный узел) удаляется вместе с файлом.

4.10. Мы рекурсивно вызываем функцию `dopath` после открытия каталога функцией `opendir`. Предположим, что `opendir` использует единственный дескриптор – в этом случае каждый раз, спускаясь на один уровень вглубь иерархии дерева каталогов, мы используем другой дескриптор. (Если исходить из предположения, что дескрипторы не закрываются до тех пор, пока не будет закончен обзор дерева каталогов и не будет вызвана функция `closedir`.) Это ограничивает глубину дерева каталогов, на которую мы можем погрузиться, максимальным количеством одновременно открытых дескрипторов. Обратите внимание: в расширениях XSI стандарта Single UNIX Specification определено, что функция `ftw` позволяет вызывающему процессу задать максимальное количество используемых дескрипторов, допуская закрытие и повторное использование дескрипторов.

4.11. Функция `chroot` используется в Интернете на серверах FTP для повышения безопасности. Пользователи, не имеющие учетных записей в системе (так называемые *анономные пользователи* FTP), попадают в отдельный каталог, и этот каталог делается корневым с помощью функции `chroot`. Это предотвращает возможность доступа к файлам, расположенным за пределами этого нового корневого каталога.

Кроме того, функция `chroot` может использоваться для создания копии дерева каталогов на новом месте, чтобы затем изменять эту новую копию, не опасаясь внести изменения в оригинальную файловую систему. Это полезно, например, для тестирования результатов установки новых программных пакетов.

Только суперпользователь может вызвать функцию `chroot`, и после изменения корневого каталога процесс и все его потомки никогда не смогут вернуться к первоначальному корню файловой системы.

4.13. Прежде всего необходимо вызвать функцию `stat`, чтобы получить три значения времени для файла, затем вызвать `utime`, чтобы изменить

требуемое значение. Значение, которое не должно изменяться в результате вызова utime, должно соответствовать значению, полученному от функции stat.

4.14. Команда finger(1) использует функцию stat для определения атрибутов времени почтового ящика. Время последнего изменения соответствует времени прибытия последнего электронного письма, а время последнего обращения – времени, когда в последний раз была прочитана почта.

4.15. Обе утилиты, cpio и tar, сохраняют в архиве только время последнего изменения (st_mtime). Время последнего обращения не сохраняется, поскольку его значение соответствует времени создания архива, так как при архивировании содержимое файла читается архиватором. Ключ -a команды cpio позволяет переустановить время последнего обращения для каждого файла, который был прочитан. Таким образом, создание архива не влечет за собой изменения времени последнего обращения. (Однако восстановление времени последнего обращения к файлу приводит к изменению времени последнего изменения статуса.) Время последнего изменения статуса не сохраняется в архиве, так как при извлечении файла из архива нет возможности восстановить его, даже если бы оно было сохранено в архиве. (Функция utime может изменять только время последнего изменения файла и время последнего обращения к файлу.)

Когда архиватор tar извлекает файлы из архива, он по умолчанию восстанавливает время последнего изменения извлекаемых файлов. С помощью ключа m можно указать утилите tar, что она не должна восстанавливать время последнего изменения файла, тогда в качестве времени последнего изменения будет использоваться время извлечения из архива. При использовании архиватора tar время последнего обращения к файлу после его извлечения из архива в любом случае будет установлено равным времени извлечения.

С другой стороны, архиватор cpio в качестве времени последнего изменения и времени последнего обращения устанавливает время извлечения из архива. По умолчанию он не пытается восстановить прежнее время последнего изменения файла, сохраненное в архиве. При использовании архиватора cpio для восстановления значений времени последнего обращения и времени последнего изменения, сохраненных в архиве, следует использовать ключ -m.

4.16. Ядро изначально не имеет ограничений на глубину вложенности каталогов. Но большинство команд завершаются ошибкой, если полные имена файлов или каталогов превышают длину PATH_MAX. Программа, показанная в листинге C.2, создает дерево каталогов, состоящее из 100 уровней вложенности, на каждом уровне каталог имеет имя длиной 45 символов. Можно создать эту структуру на любой платформе, однако ни на одной из платформ мы не сможем получить абсолютное полное имя каталога на сотом уровне с помощью функции getcwd. В Linux 2.4.22 и Solaris 9 мы никогда не сможем получить полное имя са-

мого последнего каталога в таком длинном пути. В FreeBSD 5.2.1 и Mac OS X 10.3 программа в состоянии получить полное имя последнего каталога, но нам придется много раз вызвать функцию realloc, чтобы разместить буфер достаточно большого размера. Запуск этой программы в FreeBSD 5.2.1 дал следующие результаты:

```
$ ./a.out
ошибка вызова функции getcwd, размер = 1025: Result too large
ошибка вызова функции getcwd, размер = 1125: Result too large
...                              еще 33 строки
ошибка вызова функции getcwd, размер = 4525: Result too large
длина = 4610
                                 здесь было выведено имя длиной 4610-байт
```

Однако мы не сможем заархивировать это дерево каталогов ни с помощью cpio, ни с помощью tar. Оба архиватора выведут сообщение о слишком длинном имени файла.

Листинг C.2. Создание дерева каталогов с глубокой вложенностью

```
#include "apue.h"
#include <fcntl.h>

#define DEPTH 100 /* глубина вложенности */
#define MYHOME "/home/sar"
#define NAME "alonglonglonglonglonglonglonglonglongname"
#define MAXSZ 8192

int
main(void)
{
    int  i, size;
    char *path;

    if (chdir(MYHOME) < 0)
        err_sys("ошибка вызова функции chdir");

    for (i = 0; i < DEPTH; i++) {
        if (mkdir(NAME, DIR_MODE) < 0)
            err_sys("ошибка вызова функции mkdir, i = %d", i);
        if (chdir(NAME) < 0)
            err_sys("ошибка вызова функции chdir, i = %d", i);
    }
    if (creat("afile", FILE_MODE) < 0)
        err_sys("ошибка вызова функции creat");

    /*
     * Дерево каталогов с большой глубиной вложенности создано,
     * в каталоге создан файл. Теперь попробуем получить его полное имя.
     */
    path = path_alloc(&size);
    for ( ; ; ) {
        if (getcwd(path, size) != NULL) {
            break;
```

```
            } else {
                err_ret("ошибка вызова функции getcwd, размер = %d", size);
                size += 100;
                if (size > MAXSZ)
                    err_quit("превышено наше ограничение");
                if ((path = realloc(path, size)) == NULL)
                    err_sys("ошибка вызова функции realloc");
            }
        }
        printf("длина = %d\n%s\n", strlen(path), path);
        exit(0);
    }
```

4.17. Для каталога `/dev` все биты права на запись сброшены, что не позволяет обычному пользователю удалять файлы из каталога. Это означает, что вызов функции `unlink` будет завершаться неудачей.

Глава 5

5.2. Функция `fgets` будет читать символы до тех пор, пока не встретится символ перевода строки или пока буфер не будет заполнен (с учетом места, которое необходимо оставить для завершающего нулевого символа). Функция `fputs` будет выводить данные из буфера, пока не встретит завершающий нулевой символ – она не обращает внимания на символы перевода строки, которые могут находиться в буфере. Таким образом, если значение `MAXLINE` будет слишком маленьким, обе функции по-прежнему будут работать, просто они будут вызываться намного чаще, чем при использовании буфера большого размера.

Если бы любая из этих функций удаляла или добавляла символ перевода строки (как это делают функции `gets` и `puts`), нам пришлось бы предусматривать размещение буферов достаточно большого объема, чтобы вместить самую длинную строку.

5.3. Вызов

```
printf("");
```

вернет значение 0, поскольку он не выводит ни одного символа.

5.4. Это достаточно распространенная ошибка. Возвращаемое значение функций `getc` и `getchar` имеет тип `int`, а не `char`. Зачастую константа `EOF` определена как –1, и таким образом, если в системе тип `char` имеет знак, этот код будет работать нормально. Но если в системе тип `char` не имеет знака, возвращаемое значение `EOF`, полученное от `getchar`, будет сохранено в переменной с беззнаковым типом `char` и перестанет быть равным –1, вследствие чего цикл никогда не закончится. На всех четырех платформах, описываемых в данной книге, тип `char` имеет знак, поэтому данный пример будет корректно работать на всех этих платформах.

5.5. Пять символов префикса, 4 символа для обеспечения уникальности в пределах процесса и 5 символов для обеспечения уникальности в преде-

лах системы (идентификатор процесса) в сумме дают 14 символов – изначальное ограничение UNIX на длину имени файла.

5.6. Вызывать функцию fsync после каждого вызова fflush. Аргумент функции fsync может быть получен с помощью функции fileno. Вызов fsync без обращения к fflush может не дать ожидаемого результата, если данные все еще находятся во внутренних буферах приложения.

5.7. Когда программа работает в интерактивном режиме, стандартные потоки ввода и вывода буферизуются построчно. Когда вызывается функция fgets, содержимое потока стандартного вывода сбрасывается автоматически.

Глава 6

6.1 Функции доступа к теневому файлу паролей в Linux и Solaris обсуждались в разделе 6.3. Мы не можем для сравнения с зашифрованным паролем использовать значение, возвращаемое в поле pw_passwd функциями, описанными в разделе 6.2, поскольку это поле не содержит зашифрованный пароль. Чтобы получить пароль пользователя в зашифрованном виде, нужно отыскать требуемую учетную запись в теневом файле паролей и извлечь из нее зашифрованный пароль.

В ОС FreeBSD и Mac OS X автоматически используется теневой файл паролей. В структуре passwd, возвращаемой функциями getpwnam и getpwuid, поле pw_passwd содержит зашифрованный пароль (в FreeBSD, однако, только при условии, что вызывающий процесс имеет эффективный идентификатор пользователя 0).

6.2 Программа из листинга C.3 выводит зашифрованный пароль в ОС Linux и Solaris. Если эта программа будет запущена обычным прользователем, вызов функции getspnam завершится неудачей с кодом ошибки EACCES.

Листинг C.3. Вывод зашифрованного пароля в ОС Linux и Solaris

```
#include "apue.h"
#include <shadow.h>

int
main(void) /* версия для Linux/Solaris */
{
    struct spwd *ptr;

    if ((ptr = getspnam("sar")) == NULL)
        err_sys("ошибка вызова функции getspnam");
    printf("sp_pwdp = %s\n", ptr->sp_pwdp == NULL ||
           ptr->sp_pwdp[0] == 0 ? "(null)" : ptr->sp_pwdp);
    exit(0);
}
```

В листинге C.4 приводится исходный текст программы, которая выводит зашифрованный пароль в FreeBSD, если она запущена с привилегиями суперпользователя. В противном случае в поле pw_passwd возвра-

щается символ звездочки. В Mac OS X зашифрованный пароль будет выведен в любом случае, независимо от привилегий, с которыми была запущена программа.

Листинг С.4. Вывод зашифрованного пароля в ОС FreeBSD и Mac OS X

```
#include "apue.h"
#include <pwd.h>

int
main(void) /* FreeBSD/Mac OS X version */
{
    struct passwd *ptr;

    if ((ptr = getpwnam("sar")) == NULL)
        err_sys("ошибка вызова функции getpwnam");
    printf("pw_passwd = %s\n", ptr->pw_passwd == NULL ||
           ptr->pw_passwd[0] == 0 ? "(null)" : ptr->pw_passwd);
    exit(0);
}
```

6.5. Программа из листинга С.5 выводит текущее время и дату в формате утилиты `date`.

Листинг С.5. Вывод текущего времени и даты в формате утилиты date

```
#include "apue.h"
#include <time.h>

int
main(void)
{
    time_t caltime;
    struct tm *tm;
    char   line[MAXLINE];

    if ((caltime = time(NULL)) == -1)
        err_sys("ошибка вызова функции time");
    if ((tm = localtime(&caltime)) == NULL)
        err_sys("ошибка вызова функции localtime");
    if (strftime(line, MAXLINE, "%a %b %d %X %Z %Y\n", tm) == 0)
        err_sys("ошибка вызова функции strftime");
    fputs(line, stdout);
    exit(0);
}
```

Запустив эту программу, мы получили следующее:

```
$ ./a.out                          часовой пояс автора
                                   по умолчанию US/Eastern
Sun Feb 06 16:53:57 EST 2005
$ TZ=US/Mountain ./a.out U.S.      часовой пояс штата Монтана
Sun Feb 06 14:53:57 MST 2005
$ TZ=Japan ./a.out                 Япония
Mon Feb 07 06:53:57 JST 2005
```

Глава 7

7.1. Похоже на то, что возвращаемое значение функции `printf` (количество выведенных символов) стало возвращаемым значением функции `main`. Такое поведение наблюдается не во всех системах.

7.2. Когда программа работает в интерактивном режиме, стандартный вывод обычно буферизуется построчно, таким образом, фактический вывод происходит только при выводе символа перевода строки. Однако если стандартный поток вывода перенаправлен в файл, ему, скорее всего, будет назначен режим полной буферизации, и фактический вывод не будет производиться до тех пор, пока не будет выполнено освобождение ресурсов стандартной библиотеки ввода-вывода.

7.3. В большинстве версий UNIX это невозможно. Копии `argc` и `argv` не сохраняются в глобальных переменных, как, например, `environ`.

7.4. Это дает возможность аварийно завершать процесс при попытке обратиться к памяти по пустому указателю, что является достаточно распространенной ошибкой при программировании на языке C.

7.5. Вот эти определения:

```
typedef void Exitfunc(void);
int atexit(Exitfunc *func);
```

7.6. Функция `calloc` инициализирует выделяемую память, обнуляя все биты. Стандарт ISO C не гарантирует, что в результате это даст числа с плавающей точкой, равные 0, или пустые указатели.

7.7. Куча и стек не размещаются в памяти до тех пор, пока программа не будет запущена одной из функций семейства `exec` (описывается в разделе 8.10).

7.8. Исполняемый файл (`a.out`) содержит отладочную информацию, которая может оказаться полезной при анализе файла `core`. Чтобы удалить эту информацию, можно использовать команду `strip`(1). Удаление отладочной информации из двух файлов `a.out` помогло уменьшить их размеры до 381976 и 2912 байт.

7.9. Когда не используются разделяемые библиотеки, большую часть исполняемого файла занимает стандартная библиотека ввода-вывода.

7.10. Этот код содержит ошибку, поскольку он пытается вернуть ссылку на переменную `val` с автоматическим классом размещения уже после того, как переменная перестала существовать. Автоматические переменные, объявленные после левой, открывающей скобки, с которой начинается составной оператор, не видны за правой, закрывающей скобкой.

Глава 8

8.1. Чтобы смоделировать ситуацию закрытия стандартного вывода при завершении дочернего процесса, добавьте следующую строку перед вызовом функции `exit` в дочернем процессе:

```
fclose(stdout);
```

Чтобы увидеть, как действует эта строка, замените вызов функции `printf` строками

```
i = printf("pid = %d, glob = %d, var = %d\n",
           getpid(), glob, var);
sprintf(buf, "%d\n", i);
write(STDOUT_FILENO, buf, strlen(buf));
```

Вам также необходимо определить переменные `i` и `buf`.

Здесь предполагается, что стандартный поток `stdout` будет уже закрыт, когда дочерний процесс вызовет функцию `exit`, но дескриптор `STDOUT_FILENO` останется открытым. Некоторые версии стандартной библиотеки ввода-вывода при закрытии стандартного потока вывода закрывают и файловый дескриптор, в результате функция `write` также будет завершаться неудачей. В этом случае с помощью функции `dup` продублируйте стандартный вывод на какой-либо другой дескриптор и используйте его в функции `write`.

8.2. Рассмотрим программу из листинга С.6.

Листинг С.6. Некорректное использование функции vfork

```
#include "apue.h"

static void f1(void), f2(void);

int
main(void)
{
    f1();
    f2();
    _exit(0);
}

static void
f1(void)
{
    pid_t pid;

    if ((pid = vfork()) < 0)
        err_sys("ошибка вызова функции vfork");

    /*
     * Оба процесса, и дочерний и родительский, выполняют возврат
     * в вызывающую функцию.
     */
}
```

```
static void
f2(void)
{
    char buf[1000]; /* переменные с автоматическим классом размещения */
    int  i;

    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}
```

К моменту вызова функции `vfork` указатель стека в родительском процессе будет содержать адрес фрейма стека функции `f1`, которая вызвала `vfork`. Это показано на рис. C.2.

После вызова `vfork` дочерний процесс первым получает управление и выполняет возврат из функции `f1`. После этого потомок вызывает функцию `f2`, и фрейм стека этой функции накладывается на предыдущий фрейм стека функции `f1`. Затем дочерний процесс забивает нулями 1000 байт автоматической переменной `buf`, размещенной на стеке. Затем дочерний процесс выполняет возврат из `f2` и вызывает `_exit`, но содержимое стека ниже фрейма функции `main` уже изменилось. После этого родительский процесс возобновляет работу и производит возврат из функции `f1`. Адрес возврата из функции чаще всего хранится на стеке, но эта информация наверняка уже изменена дочерним процессом. Что может произойти с родительским процессом в данном примере, во многом зависит от различных особенностей реализации конкретной версии UNIX (где в стеке хранится адрес возврата из функции, какая информация на стеке будет уничтожена при изменении содержимого автоматической переменной и тому подобное). Типичный результат – аварийное завершение родительского процесса с созданием файла `core`, но у вас результаты могут быть иными.

8.3. В листинге 8.7 мы заставляли родительский процесс начинать вывод первым. Когда родительский процесс заканчивал вывод, свою строку начинал выводить дочерний процесс, но при этом мы разрешали родительскому процессу завершить работу, не дожидаясь завершения потомка. Что произойдет раньше, завершение работы родительского процесса или завершение вывода дочерним процессом – зависит от реализации алгоритма планирования процессов в ядре (еще одна разновидность гонки за ресурсами). Когда завершается родительский процесс,

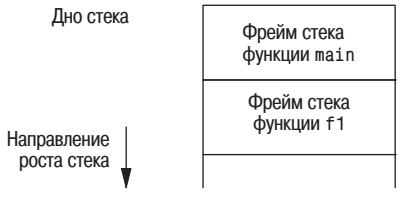


Рис. C.2. *Раскладка фреймов стека при вызове функции vfork*

командная оболочка запускает следующую программу, и вывод этой программы смешивается с выводом дочернего процесса, запущенного предыдущей программой.

Мы можем предотвратить эту ситуацию, запретив родительскому процессу завершать работу раньше, чем дочерний процесс завершит вывод своей строки. Замените код, следующий за вызовом функции `fork`, следующим фрагментом:

```
else if (pid == 0) {
    WAIT_PARENT();          /* родительский процесс стартует первым */
    charatatime("от дочернего процесса\n");
    TELL_PARENT(getppid()); /* сообщить родителю о завершении вывода */
} else {
    charatatime("от родительского процесса\n");
    TELL_CHILD(pid);        /* сообщить потомку о завершении вывода */
    WAIT_CHILD();           /* подождать, пока потомок завершит вывод */
}
```

Мы не сможем наблюдать подобный эффект, если позволим дочернему процессу стартовать первым, поскольку командная оболочка не запустит следующую программу, пока не завершится родительский процесс.

8.4. Аргумент `argv[2]` будет иметь то же самое значение (`/home/sar/bin/testinterp`). Это объясняется тем, что работа функции `execlp` завершается вызовом `execve` с тем же самым значением аргумента *pathname*, что и при непосредственном обращении к функции `execl` (рис. 8.2).

8.5. Не существует каких-либо функций, которые возвращали бы сохраненный set-user-ID. Мы должны сами предусмотреть сохранение идентификатора пользователя при запуске процесса.

8.6. Программа из листинга С.7 создает процесс-зомби.

Листинг С.7. Создает процесс-зомби, состояние которого можно затем проверить с помощью ps

```
#include "apue.h"

#ifdef SOLARIS
#define PSCMD "ps -a -o pid,ppid,s,tty,comm"
#else
#define PSCMD "ps -o pid,ppid,state,tty,command"
#endif

int
main(void)
{

    pid_t pid;

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid == 0) /* потомок */
        exit(0);
```

```
    /* предок */
    sleep(4);
    system(PSCMD);
    exit(0);
}
```

Обычно команда ps обозначает процессы-зомби с помощью символа Z.

```
$ ./a.out
  PID  PPID S TT       COMMAND
 3395  3264 S pts/3    bash
29520  3395 S pts/3    ./a.out
29521 29520 Z pts/3    [a.out] <defunct>
29522 29520 R pts/3    ps -o pid,ppid,state,tty,command
```

Глава 9

9.1. Процесс init знает, когда пользователь производит выход из системы с терминала, потому что init является родительским процессом по отношению к командной оболочке входа и получает сигнал SIGCHLD, когда она завершает работу.

Однако в случае входа в систему через сетевое соединение процесс init никак не задействован. Записи в файлы utmp и wtmp о входе в систему и выходе из системы обычно записываются процессом, который обслуживает вход в систему и определяет момент выхода (в нашем случае – сервер telnetd).

Глава 10

10.1. Программа завершит работу, когда мы пошлем ей первый сигнал. Дело в том, что функция pause возвращает управление сразу же, как только будет перехвачен какой-либо сигнал.

10.3. Схема состояния стека приводится на рис. C.3. Вызов функции longjmp из sig_alrm выполняет переход обратно в функцию main, прерывая работу функции sig_int.

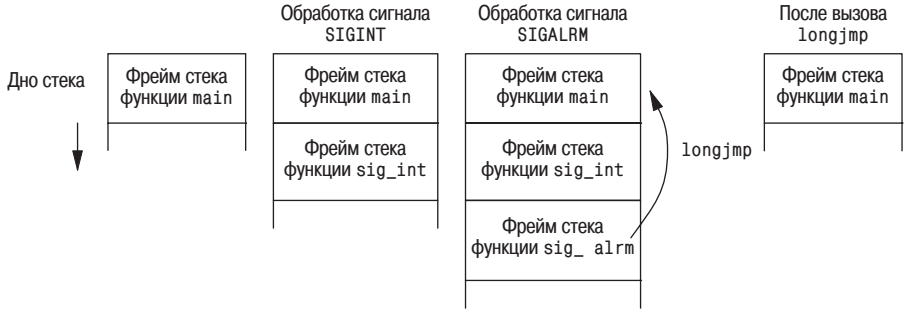


Рис. C.3. *Состояние стека до и после вызова функции longjmp*

10.4. Мы снова столкнулись с состоянием гонки за ресурсами, на этот раз между первым вызовом функции `alarm` и вызовом функции `setjmp`. Если процесс будет заблокирован ядром между этими двумя вызовами и истечет время тайм-аута, процессу будет послан сигнал, для обработки которого будет вызван обработчик сигнала, который в свою очередь вызовет функцию `longjmp`. Но поскольку `setjmp` еще не вызывалась, буфер `env_alrm` не будет заполнен корректными значениями. Поведение функции `longjmp` не определено в случае, когда буфер перехода не инициализирован функцией `setjmp`.

10.5. За примерами обращайтесь к статье Дона Либеса (Don Libes) «Implementing Software Timers» (C Users Journal, vol. 8, no. 11, Nov 1990).

10.7. Если просто вызвать функцию `_exit`, то по коду завершения процесса не будет видно, что он завершился по сигналу `SIGABRT`.

10.8. Если сигнал был послан процессом, который принадлежит некоторому другому пользователю, то этот процесс должен иметь сохраненный set-user-ID, равный либо идентификатору суперпользователя, либо идентификатору пользователя владельца процесса, принимающего сигнал, в противном случае функция `kill` не сможет послать сигнал. Таким образом, реальный идентификатор несет больше информации для процесса, принимающего сигнал.

10.10. В одной из систем, используемых автором, значение количества секунд увеличивалось на 1 каждые 60 – 90 минут. Это отклонение обусловлено тем, что каждый вызов `sleep` планирует событие в будущем, но момент пробуждения процесса не совсем точно соответствует запланированному (из-за нагрузки на центральный процессор). Кроме того, некоторый объем времени требуется для того, чтобы возобновить работу процесса после приостановки и опять вызвать функцию `sleep`.

Такие программы, как `cron`, получают текущее время каждую минуту и в первый раз задают время приостановки таким, чтобы возобновить работу в начале следующей минуты (преобразуя текущее время в локальное и извлекая значение поля `tm_sec`). Каждую минуту они устанавливают величину очередного периода приостановки так, чтобы процесс возобновил работу в начале следующей минуты. Обычно это будут вызовы `sleep(60)` и изредка, для синхронизации с текущим временем, `sleep(59)`. Но иногда, когда выполнение запланированных команд занимает продолжительное время или при высокой нагрузке на систему, может быть выбрано значительно меньшее значение аргумента функции `sleep` .

10.11. В ОС Linux 2.4.22 и Solaris 9 обработчик сигнала `SIGXFSZ` никогда не будет вызван. Но функция `write` вернет число 24, как только размер файла превысит 1024 байта.

Когда размер файла достигнет 1000 байт, в ОС FreeBSD 5.2.1 и Mac OS X 10.3 обработчик сигнала будет вызван при следующей же попытке записать очередные 100 байт, а функция `write` вернет значение –1

с кодом ошибки EFBIG (File too big – файл слишком велик) в переменной errno.

10.12. Результат зависит от реализации стандартной библиотеки ввода-вывода: от того, как функция fwrite обрабатывает прерывание системного вызова write.

Глава 11

11.1. Версия программы, которая выделяет область динамической памяти вместо использования автоматических переменных, приводится в листинге C.8.

Листинг C.8. Корректное использование возвращаемого значения потока

```
#include "apue.h"
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf(s);
    printf("структура по адресу 0x%x\n", (unsigned)fp);
    printf(" foo.a = %d\n", fp->a);
    printf(" foo.b = %d\n", fp->b);
    printf(" foo.c = %d\n", fp->c);
    printf(" foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo *fp;

    if ((fp = malloc (sizeof(struct foo))) == NULL)
        err_sys("невозможно выделить область динамической памяти");
    fp->a = 1;
    fp->b = 2;
    fp->c = 3;
    fp->d = 4;
    printfoo("поток:\n", fp);
    return((void *)fp);
}

int
main(void)
{
    int        err;
    pthread_t  tid1;
    struct foo *fp;
```

```
        err = pthread_create(&tid1, NULL, thr_fn1, NULL);
        if (err != 0)
            err_exit(err, "невозможно создать поток 1");
        err = pthread_join(tid1, (void *)&fp);
        if (err != 0)
            err_exit(err, "невозможно присоединить поток 1");
        printfoo("родительский процесс:\n", fp);
        exit(0);
    }
```

11.2. Чтобы изменить идентификатор потока для задания, ожидающего обработки, необходимо блокировку чтения-записи установить в режиме для записи, чтобы предотвратить возможность поиска по списку, пока не будет произведено изменение идентификатора. Проблема, связанная с текущим определением интерфейсов, заключается в том, что идентификатор задания может быть изменен между моментом, когда задание будет найдено функцией `job_find`, и моментом, кода задание будет исключено из списка функцией `job_remove`. Эта проблема может быть решена за счет добавления счетчика ссылок и мьютекса в структуру `job`, тогда функция `job_find` должна будет увеличивать счетчик ссылок, а код, который производит изменение идентификатора, сможет пропускать те задания в списке, которые имеют ненулевой счетчик ссылок.

11.3. Во-первых, список защищен блокировкой чтения-записи, но переменная состояния должна быть под защитой мьютекса. Во-вторых, каждый поток должен ожидать появления задания для обработки на своей собственной переменной состояния, таким образом, нам придется создать для каждого потока структуру данных, которая представляла бы это состояние. Как вариант, мы могли бы ввести переменную состояния и мьютекс в структуру `queue`, но это означало бы, что все рабочие потоки ожидали бы на одной и той же переменной состояния. При большом количестве рабочих потоков мы могли бы столкнуться с проблемой *гремящего стада* (thundering herd), когда большое количество потоков возобновляют работу, когда фактически для них нет заданий, в результате они впустую расходуют ресурсы процессора, ужесточая борьбу за обладание блокировкой.

11.4. Это зависит от обстоятельств. Вообще оба варианта могут работать вполне корректно, но каждый из них имеет свои недостатки. В первом случае ожидающие потоки будут запланированы на возобновление работы после вызова `pthread_cond_broadcast`. Если программа работает в многопроцессорной среде, некоторые запущенные потоки окажутся сразу же заблокированными, потому что мьютекс все еще заперт (не забывайте, что `pthread_cond_wait` возвращает управление с запертым мьютексом). Во втором случае работающий поток может успеть захватить мьютекс между действиями 3 и 4, среагировать на изменение состояния, сделав его недействительным, и освободить мьютекс. Затем, когда будет вызвана `pthread_cond_broadcast`, состояние больше не будет истинным, и поток отработает понапрасну. По этой причине поток все-

гда должен перепроверить истинность состояния, а не полагаться на то, что оно истинно просто потому, что функция `pthread_cond_wait` вернула управление.

Глава 12

12.1. Эта проблема не связана с многопоточной архитектурой приложения, как может показаться на первый взгляд. Процедуры стандартной библиотеки ввода-вывода в действительности являются безопасными в контексте потоков. Когда мы называем функцию `fork`, каждый процесс получает отдельную копию структур данных стандартной библиотеки ввода-вывода. При запуске программы со стандартным выводом, присоединенным к терминалу, вывод будет буферизоваться построчно, таким образом каждый раз, когда мы выводим строку, стандартная библиотека ввода-вывода будет записывать ее в устройство терминала. Однако если перенаправить стандартный вывод в файл, то библиотека выберет для него режим полной буферизации. Фактическая запись в файл будет произведена только при заполнении буфера или при закрытии потока. В этом примере к моменту вызова функции `fork` буфер уже содержит несколько еще не записанных в файл строк – таким образом, когда родительский и дочерний процессы наконец сбросят свои копии буферов, первоначальное их содержимое будет записано в файл дважды.

12.3. Теоретически, заблокировав доставку всех сигналов при вызове обработчика сигнала, мы могли бы сделать функцию безопасной в контексте обработки асинхронных сигналов. Проблема в том, что мы не знаем, не разблокирует ли какая-либо функция, к которой мы обращаемся, какой-либо из заблокированных сигналов, сделав тем самым возможным повторное вхождение в обработчик другого сигнала.

12.4. В FreeBSD 5.2.1 мы получили непрерывный поток сообщений об ошибках, и через некоторое время программа была аварийно завершена с созданием файла `core`. С помощью отладчика `gdb` удалось определить, что программа застряла в бесконечном цикле инициализации. В процессе инициализации программа вызывала функции инициализации потоков, которые обращаются к функции `malloc`. Функция `malloc` в свою очередь вызывает функцию `getenv`, чтобы получить значение переменной окружения `MALLOC_OPTIONS`. Наша реализация `getenv` вызывает функции библиотеки `pthread`, которые затем пытаются вызвать функции инициализации потоков. В результате этой ошибки программа застревает в своего рода бесконечном цикле, пока не будет вызвана функция `abort`. После создания примерно полумиллиона фреймов стека процесс был завершен аварийно с созданием файла `core`.

12.5. Функция `fork` по-прежнему необходима, если мы пожелаем запустить одну программу из другой (то есть вызывать `fork` перед вызовом `exec`).

12.6. В листинге C.9 приводится безопасная в многопоточной среде реализация функции `sleep`, которая для организации задержки использует функцию `select`. Она безопасна в многопоточной среде потому, что не

использует никаких незащищенных глобальных или статических данных и вызывает только безопасные функции.

12.7. Реализация переменной состояния, скорее всего, использует мьютекс для защиты ее внутренней структуры. Поскольку это уже относится к области реализации конкретных версий UNIX и скрыто от нас, какого-либо переносимого способа захватить или отпустить блокировку в момент ветвления процесса не существует. Поскольку мы не можем определить состояние внутренней блокировки в переменной состояния после вызова функции `fork`, использование переменных состояния в дочернем процессе будет небезопасным.

Листинг С.9. Реализация функции sleep, безопасная в многопоточной среде

```
#include <unistd.h>
#include <time.h>
#include <sys/select.h>

unsigned
sleep(unsigned nsec)
{
    int            n;
    unsigned       slept;
    time_t         start, end;
    struct timeval tv;

    tv.tv_sec = nsec;
    tv.tv_usec = 0;
    time(&start);
    n = select(0, NULL, NULL, NULL, &tv);
    if (n == 0)
        return(0);
    time(&end);
    slept = end - start;
    if (slept >= nsec)
        return(0);
    return(nsec - slept);
}
```

Глава 13

13.1. Если процесс вызовет функцию `chroot`, он не сможет открыть устройство `/dev/log`. Решение заключается в том, чтобы вызвать функцию `openlog` с флагом `LOG_NDELAY` в аргументе *option* перед обращением к функции `chroot`. Таким образом демон откроет специальный файл устройства (сокет дейтаграмм из домена UNIX), что даст ему дескриптор, который останется действительным даже после вызова `chroot`. С подобным алгоритмом можно столкнуться в таких демонах, как `ftpd` (демон службы передачи файлов по протоколу FTP), где функция `chroot` используется из соображений безопасности, но для регистрации ошибок в системном журнале используется `syslog`.

13.3. Решение приводится в листинге C.10. Результат зависит от платформы. Вспомните, что функция daemonize закрывает все дескрипторы файлов и снова открывает первые три на устройстве /dev/null. Это означает, что процесс не имеет управляющего терминала, в результате функция getlogin не сможет отыскать запись о процессе в файле utmp. Таким образом, в ОС Linux 2.4.22 и Solaris 9 мы обнаружим, что демоны не имеют имени пользователя.

Однако в FreeBSD 5.2.1 и Mac OS X 10.3 имя пользователя сохраняется в таблице процессов и копируется в дочерний процесс при вызове функции fork. Это означает, что процесс всегда может узнать имя пользователя, если только он не был запущен одним из процессов, которые не имеют имени пользователя (как, например, процесс init).

Листинг C.10. Вызов функции daemonize и попытка определить имя пользователя

```
#include "apue.h"

int
main(void)
{
    FILE *fp;
    char *p;

    daemonize("getlog");
    p = getlogin();
    fp = fopen("/tmp/getlog.out", "w");
    if (fp != NULL) {
        if (p == NULL)
            fprintf(fp, "процесс не имеет имени пользователя\n");
        else
            fprintf(fp, "имя пользователя: %s\n", p);
    }
    exit(0);
}
```

Глава 14

14.1. Тестовая программа приводится в листинге C.11.

Листинг C.11. Проверка поведения механизма блокировки записей в файле

```
#include "apue.h"
#include <fcntl.h>
#include <errno.h>

void
sigint(int signo)
{
}

int
main(void)
{
```

```
pid_t pid1, pid2, pid3;
int   fd;

setbuf(stdout, NULL);
signal_intr(SIGINT, sigint);
/*
 * Создать файл.
 */
if ((fd = open("lockfile", O_RDWR|O_CREAT, 0666)) < 0)
    err_sys("невозможно открыть/создать файл блокировки");

/*
 * Установить блокировку для чтения.
 */
if ((pid1 = fork()) < 0) {
    err_sys("ошибка вызова функции fork");
} else if (pid1 == 0) { /* потомок */
    if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
        err_sys("потомок 1: невозможно заблокировать
                            файл для чтения");
    printf("потомок 1: установлена блокировка для чтения\n");
    pause();
    printf("потомок 1: выход после паузы\n");
    exit(0);
} else { /* предок */
    sleep(2);
}

/*
 * Родительский процесс продолжается ...
 * снова установить блокировку для чтения.
 */
if ((pid2 = fork()) < 0) {
    err_sys("ошибка вызова функции fork");
} else if (pid2 == 0) { /* потомок */
    if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
        err_sys("потомок 2: невозможно заблокировать
                            файл для чтения");
    printf("потомок 2: установлена блокировка для чтения\n");
    pause();
    printf("потомок 2: выход поле паузы\n");
    exit(0);
} else { /* родительский процесс */
    sleep(2);
}

/*
 * Родительский процесс продолжается ... блокируется
 * при попытке установить блокировку для записи.
 */
if ((pid3 = fork()) < 0) {
    err_sys("ошибка вызова функции fork");
```

```
    } else if (pid3 == 0) { /* потомок */
        if (lock_reg(fd, F_SETLK, F_WRLCK, 0, SEEK_SET, 0) < 0)
            printf("потомок 3: невозможно заблокировать
                                файл для записи:%s\n",
                    strerror(errno));
        printf("потомок 3: останов, пока не получит блокировку...\n");
        if (lock_reg(fd, F_SETLKW, F_WRLCK, 0, SEEK_SET, 0) < 0)
            err_sys("потомок 3: невозможно заблокировать
                                 файл для записи");
        printf("потомок 3 сумел установить блокировку для записи???\n");
        pause();
        printf("потомок 3: выход после паузы\n");
        exit(0);
    } else { /* родительский процесс */
        sleep(2);
    }

    /*
     * Проверить, будет ли заблокирована попытка получить
     * блокировку для записи очередной попыткой установки
     * блокировки для чтения.
     */
    if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
        printf("родитель: невозможно заблокировать файл для чтения: %s\n",
                strerror(errno));
    else
        printf("родитель: установлена дополнительная
                           блокировка для чтения,"
               " запрос на установку блокировки для записи ожидает\n");
    printf("останавливается потомок 1...\n");
    kill(pid1, SIGINT);
    printf("останавливается потомок 2...\n");
    kill(pid2, SIGINT);
    printf("останавливается потомок 3...\n");
    kill(pid3, SIGINT);
    exit(0);
}
```

На всех четырех платформах, рассматриваемых в книге, был получен одинаковый результат: дополнительные читающие процессы могут оставить пишущие процессы ни с чем. Запустив программу, мы получили следующие результаты:

```
потомок 1: установлена блокировка для чтения
потомок 2: установлена блокировка для чтения
потомок 3: невозможно заблокировать файл для записи: Resource temporarily
           unavailable
потомок 3 about to block in write-lock...
родитель: установлена дополнительная блокировка для чтения, запрос
на установку блокировки для записи ожидает
останавливается потомок 1...
потомок 1: выход после паузы
```

```
останавливается потомок 2...
потомок 2: выход после паузы
останавливается потомок 3...
потомок 3: невозможно заблокировать файл для записи: Interrupted system call
```

14.2 В большинстве систем тип данных `fd_set` определен как структура, которая содержит всего одно поле – массив длинных целых чисел. Каждый бит в этом массиве соответствует одному дескриптору. Макросы `FD_` работают с этим массивом длинных целых чисел, включая, выключая и возвращая состояние отдельных битов.

Одна из причин, по которым этот тип данных был объявлен как структура, содержащая массив, а не просто как массив, заключается в том, это дает возможность присваивать значение одной переменной типа `fd_set` другой переменной типа `fd_set` обычным оператором присваивания языка C.

14.3. Большинство систем допускают возможность определения константы `FD_SETSIZE` перед подключением заголовочного файла `<sys/select.h>`. Например, с помощью инструкций

```
#define FD_SETSIZE 2048
#include <sys/select.h>
```

можно определить размер типа `fd_set` таким, чтобы он мог вместить 2048 дескрипторов. Этот прием работает в ОС FreeBSD 5.2.1, Mac OS X 10.3 и Solaris 9. В Linux 2.4.22 все это реализовано несколько иначе.

14.4 В следующей таблице перечислены функции, которые решают сходные задачи.

`FD_ZERO`	`sigemptyset`
`FD_SET`	`sigaddset`
`FD_CLR`	`sigdelset`
`FD_ISSET`	`sigismember`

В семействе `FD_XXX` нет функции, которая соответствовала бы функции `sigfillset`. При работе с сигналами указатель на набор сигналов всегда передается в первом аргументе, а номер сигнала – во втором. При работе с наборами дескрипторов в первом аргументе передается номер дескриптора, а в следующем – указатель на набор дескрипторов.

14.5 Функция `getmsg` может возвращать до пяти типов различной информации: собственно данные, объем данных, управляющую информацию, объем управляющей информации и флаги.

14.6 В листинге C.12 показана реализация с использованием функции `select`.

Листинг C.12. Реализация функции sleep_us на основе функции select

```
#include "apue.h"
#include <sys/select.h>
```

```
void
sleep_us(unsigned int nusecs)
{
    struct timeval tval;

    tval.tv_sec = nusecs / 1000000;
    tval.tv_usec = nusecs % 1000000;
    select(0, NULL, NULL, NULL, &tval);
}
```

В листинге C.13 показана аналогичная реализация с использованием функции `poll`.

Листинг C.13. Реализация функции sleep_us на основе функции poll

```
#include <poll.h>

void
sleep_us(unsigned int nusecs)
{
    struct pollfd dummy;
    int    timeout;

    if ((timeout = nusecs / 1000) <= 0)
        timeout = 1;
    poll(&dummy, 0, timeout);
}
```

Как утверждает страница справочного руководства `usleep`(3) в BSD, функция `usleep` использует в своей работе интервальный таймер `setitimer` и при каждом обращении к ней производит восемь обращений к системным вызовам. Она корректно взаимодействует с другими таймерами, установленными вызывающим процессом, и не прерывается в случае перехвата сигнала.

14.7 Нет. В этом случае `TELL_WAIT` должна была бы создать временный файл длиной в два байта, где один байт отводится для родительского и один байт – дочернего процесса. Функция `WAIT_CHILD` могла бы заставить родительский процесс ожидать снятия блокировки с байта дочернего процесса, а `TELL_PARENT` – снимать блокировку с байта дочернего процесса. Проблема, однако, состоит в том, что функция `fork` снимает все блокировки в дочернем процессе, поэтому дочерний процесс не может быть запущен с какими-либо установленными блокировками.

14.8 Решение приводится в листинге C.14.

Листинг C.14. Подсчет емкости неименованного канала с помощью неблокирующей операции записи

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
```

```
    int i, n;
    int fd[2];

    if (pipe(fd) < 0)
        err_sys("ошибка вызова функции pipe");
    set_fl(fd[1], O_NONBLOCK);

    /*
     * Записывать по 1 байту, пока канал не заполнится.
     */
    for (n = 0; ; n++) {
        if ((i = write(fd[1], "a", 1)) != 1) {
            printf("функция write вернула число %d, ", i);
            break;
        }
    }
    printf("емкость канала = %d\n", n);
    exit(0);
}
```

В следующей таблице показаны значения, полученные на наших четырех платформах.

Платформа	Емкость канала в байтах
FreeBSD 5.2.1	16 384
Linux 2.4.22	4 096
Mac OS X 10.3	8 192
Solaris 9	9 216

Эти значения могут отличаться от значения константы `PIPE_BUF`, поскольку эта константа определяет максимальный объем данных, которые могут быть записаны в канал *атомарно*. Здесь же мы получили объем данных, которые могут находиться в канале, не принимая во внимание атомарность их записи.

14.10. Изменит ли программа из листинга 14.12 время последнего обращения к исходному файлу, зависит от операционной системы и типа файловой системы, в которой размещается файл.

Глава 15

15.1. Если конец канала, открытый для записи, не будет закрыт, то процесс, читающий данные из канала, никогда не увидит признак конца файла. Таким образом, программа постраничного просмотра окажется «навечно» заблокированной в операции чтения со стандартного ввода.

15.2. Родительский процесс завершится сразу же после записи в канал последней строки. Конец канала, открытый для чтения, автоматически закроется при завершении родительского процесса. Но родительский процесс наверняка опережает потомка на один буфер, поскольку дочер-

ний процесс (программа постраничного просмотра) ожидает, пока пользователь не просмотрит выведенную перед ним страницу. Если запустить программу в командной оболочке, которая работает в диалоговом режиме, такой как Korn shell, то оболочка наверняка изменит режим терминала по завершении работы родительского процесса и выведет свое приглашение. Это несомненно повлияет на программу постраничного просмотра, так как она тоже изменяет режим терминала. (Большинство программ постраничного просмотра в ожидании перехода к следующей странице переводят терминал в неканонический режим.)

15.3. Функция `popen` вернет указатель на структуру `FILE`, потому что она запустит командную оболочку. Но сама командная оболочка не сможет выполнить несуществующую команду и потому выведет строку

```
sh: line 1: ./a.out: No such file or directory
```

на стандартное устройство вывода сообщений об ошибках и завершится с кодом завершения 127. Функция `pclose` вернет код завершения команды, который будет получен от функции `waitpid`.

15.4. После завершения родительского процесса посмотрите код его завершения. В Bourne shell, Bourne-again shell и Korn shell это можно сделать с помощью команды `echo $?`. Она выведет число, равное сумме числа 128 и номера сигнала.

15.5. Прежде всего нужно добавить объявление

```
FILE *fpin, *fpout;
```

Затем с помощью функции `fdopen` связать дескриптор канала с потоком ввода-вывода и назначить ему построчный режим буферизации. Сделать это необходимо перед входом в цикл `while`, где производится чтение со стандартного ввода:

```
if ((fpin = fdopen(fd2[0], "r")) == NULL)
    err_sys("ошибка вызова функции fdopen");
if ((fpout = fdopen(fd1[1], "w")) == NULL)
    err_sys("ошибка вызова функции fdopen");
if (setvbuf(fpin, NULL, _IOLBF, 0) < 0)
    err_sys("ошибка вызова функции setvbuf");
if (setvbuf(fpout, NULL, _IOLBF, 0) < 0)
    err_sys("ошибка вызова функции setvbuf");
```

Обращения к функциям `read` и `write` в цикле заменить строками

```
if (fputs(line, fpout) == EOF)
    err_sys("ошибка вывода в канал");
if (fgets(line, MAXLINE, fpin) == NULL) {
    err_msg("дочерний процесс закрыл канал");
    break;
}
```

15.6. Функция `system` вызовет `wait`, и первым завершится дочерний процесс, запущенный функцией `popen`. Поскольку это не тот потомок, который

был запущен функцией system, она снова вызовет функцию wait и заблокируется до тех пор, пока не завершится работа команды sleep. После этого функция system вернет управление. Когда pclose вызовет wait, она вернет признак ошибки, поскольку все дочерние процессы уже завершили работу. Вслед за ней и сама pclose вернет признак ошибки.

15.7. Функция select пометит дескриптор как доступный для чтения. Когда функция read будет вызвана после считывания всех данных из канала, она вернет значение 0 в качестве признака конца файла. В случае с функцией poll (при условии, что каналы реализованы на базе механизма STREAMS) будет возвращено событие POLLHUP, а оно может быть возвращено, даже если в канале еще имеются данные, доступные для чтения. Однако когда функция read прочитает все данные, она вернет значение 0 как признак конца файла. После прочтения всех данных событие POLLIN возвращено не будет, даже если нам еще только предстоит прочитать признак конца файла (возвращаемое значение 0 функции read).

В случае дескриптора, открытого на запись, когда закрывается дескриптор, открытый на чтение, функция select пометит дескриптор как доступный для записи. Но при вызове функции write процессу будет послан сигнал SIGPIPE. Если мы игнорируем этот сигнал или возвращаем управление из обработчика обычным образом, функция write вернет код ошибки EPIPE. Однако в случае с функцией poll, если каналы реализованы на базе механизма STREAMS, функция poll вернет управление с событием POLLHUP для заданного дескриптора.

15.8. Все, что будет выведено дочерним процессом на стандартный вывод сообщений об ошибках, будет отправлено на стандартный вывод сообщений об ошибках родительского процесса. Чтобы отправить данные со стандартного вывода сообщений об ошибках родительскому процессу, включите в *cmdstring* операцию перенаправления 2>&1.

15.9. Функция popen создает дочерний процесс, а он запускает командный интерпретатор. Командный интерпретатор в свою очередь вызывает fork, и новый дочерний процесс командного интерпретатора запускает командную строку. Родительский командный интерпретатор дожидается, когда *cmdstring* завершится, и также завершает работу, чего в свою очередь ожидает функция waitpid в pclose.

15.10. Хитрость заключается в том, что канал FIFO надо открыть дважды: один раз для чтения и один раз для записи. Мы вообще не используем дескриптор, открытый для записи, но оставляем его открытым для предотвращения генерации признака конца файла, когда количество клиентов уменьшается с 1 до 0. Открытие FIFO в два приема требует некоторых дополнительных действий, так как оно должно производиться в неблокирующем режиме. Сначала мы должны открыть FIFO только для чтения в неблокирующем режиме, а затем вызвать open в блокирующем режиме, чтобы открыть канал только для записи. (Если мы сначала попытаемся открыть FIFO в неблокирующем режиме

только для записи, функция open вернет признак ошибки.) Затем мы должны сбросить флаг неблокирующего режима в дескрипторе, открытом для чтения. В листинге C.15 показано, как это делается.

Листинг C.15. Открытие канала FIFO для чтения и записи без блокировки процесса

```
#include "apue.h"
#include <fcntl.h>

#define FIFO "temp.fifo"

int
main(void)
{
    int fdread, fdwrite;

    unlink(FIFO);
    if (mkfifo(FIFO, FILE_MODE) < 0)
        err_sys("ошибка вызова функции mkfifo");
    if ((fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("ошибка открытия для чтения");
    if ((fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("ошибка открытия для записи");
    clr_fl(fdread, O_NONBLOCK);
    exit(0);
}
```

15.11. Беспорядочное чтение сообщений из активной очереди может повлечь за собой конфликты между сервером и клиентом из-за несоблюдения протокола обмена, так как в этом случае могут быть утеряны либо запросы клиента, либо отклики сервера. Чтобы получить возможность чтения из очереди, процесс должен знать ее идентификатор, а очередь должна иметь установленный бит world-read (доступ на чтение для всех остальных).

15.13. Мы никогда не должны хранить фактические адреса в сегменте разделяемой памяти, поскольку существует вероятность, что сервер и все клиенты подключат этот сегмент к различным адресам. Вместо адресов в связанном списке, который строится в сегменте разделяемой памяти, следует использовать величины смещений объекта от начала сегмента разделяемой памяти. Эти смещения формируются путем вычитания адреса начала сегмента разделяемой памяти из адреса объекта.

15.14. В табл. C.1 приводится схема происходящих событий.

Глава 16

16.1 В листинге C.16 приводится программа, которая определяет порядок байтов для аппаратной архитектуры, на которой она запущена.

Таблица С.1. Чередование периодов работы родительского и дочернего процессов из листинга 15.12

Значение i в родителе	Значение i в потомке	Разделяемое значение	Возвращаемое значение update	Комментарий
		0		Инициализируется функцией mmap
	1			Потомок запускается первым и затем блокируется
0				Запускается родитель
		1		
			0	Затем родитель блокируется
		2		Потомок возобновляет работу
			1	
	3			Затем потомок блокируется
2				Родитель возобновляет работу
		3		
			2	Затем родитель блокируется
		4		
			3	
	5			Затем потомок блокируется
4				Родитель возобновляет работу

Листинг С.16. Определение порядка байтов

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int
main(void)
{
    uint32_t i;
    unsigned char *cp;

    i = 0x04030201;
    cp = (unsigned char *)&i;
    if (*cp == 1)
        printf("прямой (little-endian)\n");
    else if (*cp == 4)
        printf("обратный (big-endian)\n");
    else
        printf("неизвестный?\n");
    exit(0);
}
```

16.3. Каждый из сокетов, который будет принимать запросы на соединение, должен быть привязан к своему адресу, и для каждого дескриптора должна быть создана соответствующая запись в fd_set. Для ожидания прибытия запросов на соединение на несколько адресов мы будем использовать функцию select. В разделе 16.4 мы уже говорили, что по прибытии запроса на соединение дескриптор сокета будет отмечен как доступный для чтения. Прибывающие запросы на соединение мы будем принимать и обслуживать, как и прежде.

16.5. Для этого нужно установить обработчик сигнала SIGCHLD, обратившись к функции signal (листинг 10.12), которая устанавливает обработчик сигнала с помощью функции sigaction, позволяющей определить возможность перезапуска прерванных системных вызовов. Затем следует убрать вызов waitpid из функции serve. После запуска дочернего процесса родитель закрывает новый дескриптор и переходит к ожиданию новых запросов на соединение. И наконец, нам нужен сам обработчик сигнала SIGCHLD:

```
void
sigchld(int signo)
{
    while (waitpid((pid_t)-1, NULL, WNOHANG) > 0)
        ;
}
```

16.6. Чтобы разрешить асинхронный режим работы сокета, необходимо назначить процесс владельцем сокета с помощью команды F_SETOWN функции fcntl и разрешить асинхронную доставку сигнала с помощью команды FIOASYNC функции ioctl. Чтобы запретить асинхронный режим работы сокета, достаточно будет просто запретить асинхронную доставку сигнала. Смешивание вызовов функций fcntl и ioctl необходимо для обеспечения переносимости. Код функций приводится в листинге C.17.

Листинг C.17. Функции разрешения и запрещения асинхронного режима работы сокета

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/ioctl.h>

#if defined(BSD) || defined(MACOS) || defined(SOLARIS)
#include <sys/filio.h>
#endif

int
setasync(int sockfd)
{
    int n;

    if (fcntl(sockfd, F_SETOWN, getpid()) < 0)
```

```
        return(-1);
    n = 1;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}

int
clrasync(int sockfd)
{
    int n;

    n = 0;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}
```

Глава 17

17.3. *Объявление* определяет атрибуты (такие как тип данных) набора идентификаторов. Если объявление предполагает выделение памяти под объявленные объекты, то такое объявление называется *определением*.

В заголовочном файле `opend.h` мы объявляем три глобальные переменные с классом хранения `extern`. Эти объявления не подразумевают выделение памяти для хранения значений переменных. В файле `main.c` мы определяем три глобальные переменные. Иногда определение глобальной переменной может сопровождаться ее инициализацией, но мы, как правило, позволяем языку C инициализировать ее значением по умолчанию.

17.5. Обе функции, `select` и `poll`, возвращают количество дескрипторов, готовых к выполнению операции. Цикл обхода массива `client` может быть завершен раньше, когда число обработанных дескрипторов достигнет значения, полученного от функции `select` или `poll`.

Глава 18

18.1. Обратите внимание: поскольку терминал находится в неканоническом режиме, ввод команды `reset` должен завершаться символом перевода строки, а не символом возврата каретки.

18.2. Она строит таблицу для каждого из 128 символов и затем устанавливает самый старший бит (бит паритета) в соответствии с указаниями пользователя. После этого она использует 8-битный ввод-вывод, самостоятельно обслуживая бит паритета.

18.3. Если вы используете терминал с оконной системой, вам не нужно входить в систему дважды. Вы можете проделать этот эксперимент в двух отдельных окнах. В Solaris запустите команду `stty -a`, перенаправив стандартный ввод окна, в котором запущен редактор `vi`. Это позволит

увидеть, что vi устанавливает параметры MIN и TIME в значение 1. Вызов функции read будет ожидать ввода хотя бы одного символа, но когда символ будет введен, функция read, прежде чем вернуть управление, будет ждать ввода дополнительных символов не дольше одной десятой доли секунды.

Глава 19

19.1. Оба сервера, telnetd и rlogind, работают с привилегиями суперпользователя, поэтому они могут без ограничений пользоваться функциями chown и chmod.

19.3. Запустите pty -n stty -a, чтобы предотвратить инициализацию структур termios и winsize ведомого терминала.

19.5. К сожалению, команда F_SETFL функции fcntl не позволяет изменять состояние режима «для чтения и для записи».

19.6. Здесь присутствуют три группы процессов: (1) командная оболочка входа, (2) дочерний и родительский процессы программы pty, (3) процесс cat. Первые две группы составляют единую сессию, в которой в качестве лидера выступает командная оболочка входа. Вторая сессия содержит только процесс cat. Первая группа процессов (командная оболочка входа) является группой процессов фонового режима, а две другие – группами процессов переднего плана.

19.7. Первым завершится процесс cat, когда получит от своего модуля дисциплины обслуживания терминала признак конца файла. Это приведет к завершению подчиненного PTY, что вызовет завершение ведущего PTY. Это, в свою очередь, приведет к тому, что родительский процесс, который получает ввод от ведущего PTY, получит признак конца файла. Родительский процесс пошлет сигнал SIGTERM дочернему процессу, вследствие чего дочерний процесс прекратит работу. (Дочерний процесс не перехватывает этот сигнал.) И, наконец, родительский процесс вызовет функцию exit(0) в конце функции main.

Ниже приводится вывод программы из листинга 8.16, соответствующий данному случаю.

```
cat    e =    270, chars =    274, stat =     0:
pty    e =    262, chars =     40, stat =    15: F      X
pty    e =    288, chars =    188, stat =     0:
```

19.8. Сделать это можно с помощью команд echo и date(1), запустив их в подоболочке:

```
#!/bin/sh
( echo "Сбор данных запущен " `date`;
  pty "${SHELL:-/bin/sh}";
  echo " Сбор данных завершен " `date` ) | tee typescript
```

19.9. В модуле дисциплины обслуживания терминала, расположенном выше подчиненного PTY, разрешен эхо-вывод, поэтому все, что читает

pty со своего стандартного ввода и записывает в ведущий PTY, по умолчанию выводится в виде эха. Эхо-вывод производится модулем дисциплины обслуживания терминала, расположенным выше подчиненного PTY, даже если программа (ttyname) не читает данные.

Глава 20

20.1. Наш консерватизм в установке блокировки в функции _db_dodelete обусловлен стремлением избежать состояния гонки в функции db_nextrec. Если вызов _db_writedat не будет защищен блокировкой, то может возникнуть ситуация, когда запись с данными будет стерта в то время, когда db_nextrec читает ее: функция db_nextrec может прочитать индексную запись, убедиться в том, что она не пуста, и приступить к чтению записи с данными, которая может быть стерта функцией _db_dodelete между вызовами _db_readidx и _db_readdat в db_nextrec.

20.2. Предположим, что db_nextrec вызывает _db_readidx, которая считывает индекс в буфер процесса. Этот процесс затем приостанавливается ядром, и управление передается другому процессу. Другой процесс вызывает db_delete, и удаляет запись, прочитанную первым процессом. Обе записи – ключ и данные – оказываются затертыми пробелами. Затем управление переходит к первому процессу, который вызывает _db_readdat (из db_nextrec) и считывает запись с данными, затертую пробелами. Блокировка для чтения, устанавливаемая в db_nextrec, позволяет выполнить чтение индексной записи и записи с данными атомарно (относительно других процессов, использующих ту же самую базу данных).

20.3. Использование принудительных блокировок окажет влияние на другие читающие и пишущие процессы. Они будут заблокированы ядром до тех пор, пока не будут сняты блокировки, установленные функциями _db_writeidx и _db_writedat.

20.5. Используя такой порядок записи (сначала данные, потом индекс), мы защищаем файлы базы данных от повреждения в том случае, если процесс будет завершен между двумя операциями записи. Если процесс сначала запишет индексную запись, но перед записью данных будет неожиданно завершен, мы получим корректную индексную запись, которая указывает на некорректные данные.

Глава 21

21.5. Несколько подсказок. Проверять наличие заданий можно в двух местах: в очереди демона печати и во внутренней очереди сетевого принтера. Вы должны не допустить, чтобы один пользователь получил возможность отменить задание печати другого пользователя. Разумеется, суперпользователь должен иметь возможность отменить печать любого задания.

Список литературы

1. Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. 1986. «Mach: A New Kernel Foundation for UNIX Development», *Proceedings of the 1986 Summer USENIX Conference*, pp. 93–113, Atlanta, GA.

   Введение в операционную систему Mach.

2. Adobe Systems Inc. 1999. *PostScript Language Reference Manual, Third Edition*. Addison-Wesley, Reading, MA.

   Справочное руководство по языку PostScript.

3. Aho, A. V., Kernighan, B. W., and Weinberger, P. J. 1988. *The AWK Programming Language*. Addison-Wesley, Reading, MA.

   Замечательная книга по языку программирования awk. Версия awk, описываемая в книге, иногда называется nawk («new awk»).

4. Andrade, J. M., Carges, M. T., and Kovach, K. R. 1989. «Building a Transaction Processing System on UNIX Systems», *Proceedings of the 1989 USENIX Transaction Processing Workshop*, vol. May, pp. 13–22, Pittsburgh, PA.

   Описание системы обработки запросов AT&T Tuxedo.

5. Arnold, J. Q. 1986. «Shared Libraries on UNIX System V», *Proceedings of the 1986 Summer USENIX Conference*, pp. 395–404, Atlanta, GA.

   Описание реализации разделяемых библиотек в SVR3.

6. AT&T. 1989. *System V Interface Definition, Third Edition*. Addison-Wesley, Reading, MA.

   Этот четырехтомник описывает интерфейсы исходного кода System V и ее поведение во время выполнения. Третья редакция соответствует SVR4. Пятый том содержит обновленные версии команд и функций из томов 1–4, был издан в 1991 г. В настоящее время тираж распродан.

7. AT&T. 1990a. *UNIX Research System Programmer's Manual, Tenth Edition, Volume I*. Saunders College Publishing, Fort Worth, TX.

   Версия «Руководства программиста UNIX» для 10-й редакции Research UNIX System (V10). В этой книге содержатся традиционные для UNIX страницы справочного руководства (разделы 1–9).

8. AT&T. 1990b. *UNIX Research System Papers, Tenth Edition, Volume II*. Saunders College Publishing, Fort Worth, TX.

   Том II руководства программиста для UNIX Version 10 (V10) содержит 40 статей, описывающих различные аспекты системы.

9. AT&T. 1990c. *UNIX System V Release 4 BSD/XENIX Compatibility Guide*. Prentice-Hall, Englewood Cliffs, NJ.

Содержит страницы справочного руководства, описывающие библиотеку совместимости.

10. AT&T. 1990d. *UNIX System V Release 4 Programmer's Guide: STREAMS*. Prentice-Hall, Englewood Cliffs, NJ.

Описывает систему STREAMS в SVR4.

11. AT&T. 1990e. *UNIX System V Release 4 Programmer's Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ.

Это справочное руководство программиста к реализации SVR4 для процессора Intel 80386. Содержит разделы 1 (команды), 2 (системные вызовы), 3 (подпрограммы), 4 (форматы файлов) и 5 (различные возможности).

12. AT&T. 1991. *UNIX System V Release 4 System Administrator's Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ.

Справочное руководство системного администратора к реализации SVR4 для процессора Intel 80386. Содержит разделы 1 (команды), 4 (форматы файлов), 5 (различные возможности) и 7 (специальные файлы).

13. Bach, M. J. 1986. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, NJ.

Книга подробно описывает архитектуру и реализацию операционной системы UNIX. Хотя исходный код UNIX и не приводится (поскольку в то время он был собственностью AT&T), все же в книге представлено большое количество алгоритмов и структур данных, используемых ядром UNIX. Эта книга описывает SVR2.

14. Bolsky, M. I., and Korn, D. G. 1995. *The New KornShell Command and Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ.

Книга описывает работу с командной оболочкой Korn shell – как с командным интерпретатором, так и с языком программирования.

15. Chen, D., Barkley, R. E., and Lee, T. P. 1990. «Insuring Improved VM Performance: Some No-Fault Policies», *Proceedings of the 1990 Winter USENIX Conference*, pp. 11–22, Washington, D.C.

Описывает изменения, внесенные в реализацию виртуальной памяти SVR4 для повышения производительности (главным образом функций `fork` и `exec`).

16. Comer, D. E. 1979. «The Ubiquitous B-Tree», *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137 (June).

Хорошая подробная статья о двоичных деревьях.

17. Date, C. J. 2004. *An Introduction to Database Systems, Eighth Edition*. Addison-Wesley, Boston, MA.[1]

Обширный обзор систем управления базами данных.

[1] К. Дж. Дейт «Введение в системы баз данных», 8-е издание, Вильямс, 2006.

18. Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. 1979. «Extendible Hashing – A Fast Access Method for Dynamic Files», *ACM Transactions on Databases*, vol. 4, no. 3, pp. 315–344 (September).

Статья, описывающая методику расширяемого хеширования.

19. Fowler, G. S., Korn, D. G., and Vo, K. P. 1989. «An Efficient File Hierarchy Walker», *Proceeding of the 1989 Summer USENIX Conference*, pp. 173–188, Baltimore, MD.

Описывает альтернативную библиотеку функций, используемых для обхода дерева каталогов файловой системы.

20. Gallmeister, B. O. 1995. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Sebastopol, CA.

Описывает интерфейсы реального времени стандарта POSIX.

21. Garfinkel, S., Spafford, G., and Schwartz, A. 2003. *Practical UNIX & Internet Security, Third Edition*. O'Reilly & Associates, Sebastopol, CA.

Подробная книга о безопасности операционной системы UNIX.

22. Gingell, R. A., Lee, M., Dang, X. T., and Weeks, M. S. 1987. «Shared Libraries in SunOS», *Proceedings of the 1987 Summer USENIX Conference*, pp. 131–145, Phoenix, AZ.

Описывает реализацию разделяемых библиотек в SunOS.

23. Gingell, R. A., Moran, J. P., and Shannon, W. A. 1987. «Virtual Memory Architecture in SunOS», *Proceedings of the 1987 Summer USENIX Conference*, pp. 81–94, Phoenix, AZ.

Описывает первоначальную реализацию функции `mmap` и проблемы, связанные с архитектурой виртуальной памяти.

24. Goodheart, B. 1991. *UNIX Curses Explained*. Prentice-Hall, Englewood Cliffs, NJ.

Полное руководство по terminfo и библиотеке `curses`. В настоящее время тираж распродан.

25. Hume, A. G. 1988. «A Tale of Two Greps», *Software Practice and Experience*, vol. 18, no. 11, pp. 1063–1072.

Интересная статья, в которой обсуждается вопрос повышения производительности утилиты `grep`.

26. IEEE. 1990. *Information Technology – Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]*. IEEE (Dec.).

Это был первый из стандартов POSIX, и он определял стандартные системные интерфейсы языка программирования C на основе ОС UNIX. Нередко он называется POSIX.1. В настоящее время входит в состав стандарта Single UNIX Specification, опубликованного The Open Group [2004].

27. ISO. 1999. *International Standard ISO/IEC 9899—Programming Language C*. ISO/IEC.

Официальный стандарт языка программирования C и его библиотек. Электронную версию стандарта в формате PDF можно получить по адресу *http://www.ansi.org* или *http://www.iso.org*.

28. Kernighan, B. W., and Pike, R. 1984. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, NJ.[1]

    Общее руководство по программированию в UNIX. Книга охватывает множество команд и утилит UNIX, таких как `grep`, `sed`, `awk` и Bourne shell.

29. Kernighan, B. W., and Ritchie, D. M. 1988. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ.[2]

    Книга о версии ANSI языка программирования C. Приложение B содержит описание библиотек, определяемых стандартом ANSI.

30. Kleiman, S. R. 1986. «Vnodes: An Architecture for Multiple File System Types in Sun Unix», *Proceedings of the 1986 Summer USENIX Conference*, pp. 238–247, Atlanta, GA.

    Описание оригинальной реализации концепции виртуальных узлов.

31. Knuth, D. E. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition.* Addison-Wesley, Boston, MA.[3]

    Описывает алгоритмы сортировки и поиска.

32. Korn, D. G., and Vo, K. P. 1991. «SFIO: Safe/Fast String/File IO», *Proceedings of the 1991 Summer USENIX Conference*, pp. 235–255, Nashville, TN.

    Описание альтернативной библиотеки ввода-вывода. Библиотека доступна по адресу *http://www.research.att.com/sw/tools/sfio*.

33. Krieger, O., Stumm, M., and Unrau, R. 1992. «Exploiting the Advantages of Mapped Files for Stream I/O», *Proceedings of the 1992 Winter USENIX Conference*, pp. 27–42, San Francisco, CA.

    Альтернатива стандартной библиотеке ввода-вывода, основанная на отображаемых файлах.

34. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA.

    Книга целиком посвящена операционной системе 4.3BSD. Она описывает версию Tahoe 4.3BSD. В настоящее время тираж распродан.

35. Lennert, D. 1987. «How to Write a UNIX Daemon», *;login:*, vol. 12, no. 4, pp. 17–23 (July/August).

    Рассказывает о написании демонов для UNIX.

---

[1] Б. Керниган, Р. Пайк «UNIX. Программное окружение», Символ-Плюс, 2003.

[2] Б. Керниган, Д. Ритчи «Язык программирования Си», Невский Диалект, 2000.

[3] Дональд Э. Кнут «Искусство программирования. Том 3. Сортировка и поиск», 2-е издание, Вильямс, 2005.

36. Libes, D. 1990. «expect: Curing Those Uncontrollable Fits of Interaction», *Proceedings of the 1990 Summer USENIX Conference*, pp. 183–192, Anaheim, CA.

Описание программы `expect` и ее реализации.

37. Libes, D. 1991. «expect: Scripts for Controlling Interactive Processes», *Computing Systems*, vol. 4, no. 2, pp. 99–125 (Spring).

В статье представлены многочисленные сценарии для программы `expect`.

38. Libes, D. 1994. *Exploring Expect*. O'Reilly & Associates, Sebastopol, CA.

Книга по работе с программой `expect`.

39. Lions, J. 1977. *A Commentary on the UNIX Operating System*. AT&T Bell Laboratories, Murray Hill, NJ.

Описывает исходные тексты 6-й Редакции UNIX (6th Edition UNIX System). Доступна только для специалистов и служащих AT&T, хотя некоторые копии просочились за пределы AT&T.

40. Lions, J. 1996. *Lions' Commentary on UNIX 6th Edition*. Peer-to-Peer Communications, San Jose, CA.

Общедоступная версия классического издания 1977 г. описывает 6-ю Редакцию ОС UNIX.

41. Litwin, W. 1980. «Linear Hashing: A New Tool for File and Table Addressing», *Proceedings of the 6th International Conference on Very Large Databases*, pp. 212–223, Montreal, Canada.

Статья, описывающая метод линейного хеширования.

42. McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, Reading, MA.

Книга целиком посвящена операционной системе 4.4BSD.

43. McKusick, M. K., and Neville-Neil, G. V. 2005. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, Boston, MA.[1]

Книга целиком посвящена операционной системе FreeBSD 5.2.

44. Mauro, J., and McDougall, R. 2001. *Solaris Internals*. Prentice-Hall, Upper Saddle River, NJ.

Книга о внутреннем устройстве операционной системы Solaris. Охватывает версии Solaris 2.5.1, 2.6 и 2.7 (известную также как Solaris 7).

45. Morris, R., and Thompson, K. 1979. «UNIX Password Security», *Communications of the ACM*, vol. 22, no. 11, pp. 594–597 (Nov.).

Описание истории развития схемы паролей, используемой в системах UNIX.

[1] Маршалл К. МакКузик, Джордж В. Невилл-Нил «FreeBSD: архитектура и реализация», КУДИЦ-Образ, 2006.

46. Nemeth, E., Snyder, G., Seebass, S., and Hein, T. R. 2001. *UNIX System Administration Handbook, Third Edition*. Prentice-Hall, Upper Saddle River, NJ.[1]

Книга, в которой подробно рассматривается администрирование UNIX.

47. Olander, D. J., McGrath, G. J., and Israel, R. K. 1986. «A Framework for Networking in System V», *Proceedings of the 1986 Summer USENIX Conference*, pp. 38–45, Atlanta, GA.

Описывает оригинальную реализацию служебных интерфейсов STREAMS и TLI для System V.

48. The Open Group. 2004. *The Single UNIX Specification, Version 3*. The Open Group, Berkshire, UK.

Стандарты POSIX и X/Open, объединенные в один справочник. С версией в формате HTML можно познакомиться по адресу *http://www.opengroup.org*. Там же можно приобрести CD-ROM, содержащий стандарт целиком.

49. Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. 1995. «Plan 9 from Bell Labs», *Plan 9 Programmer's Manual Volume 2*. AT&T, Reading, MA.

Описание операционной системы Plan 9, разработанной в том же подразделении, что и система UNIX.

50. Plauger, P. J. 1992. *The Standard C Library*. Prentice-Hall, Englewood Cliffs, NJ.

Книга о библиотеке ANSI C. Содержит полную реализацию библиотеки языка C.

51. Presotto, D. L., and Ritchie, D. M. 1990. «Interprocess Communication in the Ninth Edition UNIX System», *Software Practice and Experience*, vol. 20, no. S1, pp. S1/3–S1/17 (June).

Описывает возможности IPC, предоставляемые 9-й Редакцией UNIX (Ninth Edition Research UNIX System), разработанной в AT&T Bell Laboratories. Функциональные возможности основаны на потоковой системе ввода-вывода и включают дуплексные каналы, передачу файловых дескрипторов между процессами и создание уникальных соединений между клиентами и серверами. Копия этой статьи имеется также в [8].

52. Rago, S. A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, MA.

Книга описывает программирование в сетевом окружении UNIX System V Release 4, основанное на использовании механизмов STREAMS.

53. Raymond, E. S., ed. 1996. *The New Hacker's Dictionary, Third Edition*. MIT Press, Cambridge, MA.

Определения множества терминов из лексикона хакера.

[1] Э. Немет, Г. Снайдер, С. Сибасс, Т. Р. Хейн «UNIX: руководство системного администратора. Для профессионалов», Питер, 2003.

54. Ritchie, D. M. 1984. «A Stream Input-Output System», *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1897–1910 (Oct.).

Оригинальный документ о Streams.

55. Salus, P. H. 1994. *A Quarter Century of UNIX*. Addison-Wesley, Reading, MA.

История развития UNIX с 1969 по 1994 гг.

56. Seltzer, M., and Olson, M. 1992. «LIBTP: Portable Modular Transactions for UNIX», *Proceedings of the 1992 Winter USENIX Conference*, pp. 9–25, San Francisco, CA.

Модификация библиотеки `db(3)` из 4.4BSD, которая реализует механизм транзакций.

57. Seltzer, M., and Yigit, O. 1991. «A New Hashing Package for UNIX», *Proceedings of the 1991 Winter USENIX Conference*, pp. 173–184, Dallas, TX.

Описание библиотеки `dbm(3)` и ее реализации, а также новейшего пакета хеширования.

58. Stevens, W. R. 1990. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, NJ.[1]

Книга подробно описывает программирование сетевых приложений для UNIX. Первое издание очень сильно отличается по своему содержанию от более поздних изданий.

59. Stevens, W. R., Fenner, B., and Rudoff, A. M. 2004. *UNIX Network Programming, Volume 1, Third Edition*. Addison-Wesley, Boston, MA.[2]

Подробно описывается программирование сетевых приложений для UNIX. Переработана и разбита на два тома во 2-ом издании, дополнена в третьем.

60. Stonebraker, M. R. 1981. «Operating System Support for Database Management», *Communications of the ACM*, vol. 24, no. 7, pp. 412–418 (July).

Описывает службы операционной системы и их влияние на работу базы данных.

61. Strang, J. 1986. *Programming with curses*. O'Reilly & Associates, Sebastopol, CA.

Книга о версии библиотеки `curses` из Беркли.

62. Strang, J., Mui, L., and O'Reilly, T. 1988. *termcap & terminfo, Third Edition*. O'Reilly & Associates, Sebastopol, CA.

Книга посвящена `termcap` и `terminfo`.

63. Sun Microsystems. 2002. *STREAMS Programming Guide*. Sun Microsystems, Santa Clara, CA.

Описывает STREAMS программирование на платформе Solaris.

---

[1] Стивенс У. «UNIX: разработка сетевых приложений», Питер, 2003.

[2] Стивенс У., Феннер Б., Рудофф Э. «UNIX: разработка сетевых приложений», Питер, 2006.

64. Thompson, K. 1978. «UNIX Implementation», *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1931–1946 (July–Aug.).

Описывает некоторые аспекты реализации Version 7.

65. Vo, Kiem-Phong. 1996. «Vmalloc: A General and Efficient Memory Allocator», *Software Practice and Experience*, vol. 26, no. 3, pp. 357–374.

Описывает гибкий менеджер динамической памяти.

66. Weinberger, P. J. 1982. «Making UNIX Operating Systems Safe for Databases», *The Bell System Technical Journal*, vol. 61, no. 9, pp. 2407–2422 (Nov.).

Описывает некоторые проблемы реализации баз данных в ранних версиях UNIX.

67. Weinstock, C. B., and Wulf, W. A. 1988. «Quick Fit: An Efficient Algorithm for Heap Storage Allocation», *SIGPLAN Notices*, vol. 23, no. 10, pp. 141–148.

Описывает алгоритм управления динамической памятью, который подходит для широкого круга приложений.

68. Williams, T. 1989. «Session Management in System V Release 4», *Proceedings of the 1989 Winter USENIX Conference*, pp. 365–375, San Diego, CA.

Описывает архитектуру сессии в SVR4, на которой были основаны интерфейсы POSIX.1. Рассматриваются группы процессов, управление заданиями, управляющие терминалы и вопросы безопасности существующих механизмов.

69. X/Open. 1989. *X/Open Portability Guide*. Prentice-Hall, Englewood Cliffs, NJ.

Издание состоит из семи томов, которые охватывают команды и утилиты (том 1), системные интерфейсы и заголовочные файлы (том 2), дополнительные определения (том 3), языки программирования (том 4), управление данными (том 5), управление окнами (том 6), сетевые службы (том 7). Хотя это издание в настоящее время отсутствует в продаже, его заменяет Single UNIX Specification [Open Group 2004].

Алфавитный указатель

Ссылка на «определение функции» означает, что на указанной странице вы сможете найти прототип функции, ее описание и исходный код. Функции, определяемые в книге для использования в последующих примерах, такие как `set_fl` из листинга 3.5, также включены в предметный указатель. Кроме того, в предметный указатель включены определения внешних функций, которые входят в состав больших примеров (главы 17, 19, 20 и 21), чтобы вам было проще разобраться в этих примерах. Также в предметный указатель включены страницы, где наиболее важные функции, такие как `select` или `poll`, встречаются в каких-либо примерах. В предметный указатель не были включены ссылки на тривиальные функции, такие как `exit`, которые используются практически в каждом примере.

A

abort, функция, 277, 414, 921, 961
  определение, 414
accept, функция, 670, 673, 700, 894, 921
  определение, 662
access, функция, 139, 921
  определение, 139
acct, структура, 314
acct, функция, 313
accton, программа, 314
ACOMPAT, константа, 314
ACORE, константа, 314
add_job, функция, 891, 897, 901, 905
add_option, функция, 908, 912
addrinfo, структура, 654, 877, 878
add_worker, функция, 891, 901, 905
AEXPND, константа, 314
AF_INET, константа, 643, 878
AF_INET6, константа, 643
AF_IPX, константа, 643
AF_LOCAL, константа, 643
AFORK, константа, 314
AF_UNIX, константа, 643
AF_UNSPEC, константа, 643
Aho, Kernighan, and Weinberger, 306
AI_ADDRCONFIG, константа, 655
AI_ALL, константа, 655
AI_CANONNAME, константа, 655, 878
AI_NUMERICHOST, константа, 655
AI_NUMERICSERV, константа, 655
AI_PASSIVE, константа, 655
AI_V4MAPPED, константа, 655
AIX, операционная система, 68
alarm, функция, 378, 385–390, 405, 424, 676, 922, 981
  определение, 385
alloca, функция, 248
ALTWERASE, константа, 739, 750
argc, аргумент функции main, 234
ARG_MAX, константа, 76, 294
argv, аргумент функции main, 234
asctime, определение функции, 229, 922
ASU, константа, 314
atexit, функция, 76, 237, 239, 803, 922, 976
  определение, 237
ATEXIT_MAX, константа, 76
atoi, функция, 49, 723, 916, 917
atol, функция, 842, 843, 844, 895
attributes-charset, атрибут, 868
attributes-natural-language, атрибут, 868, 912
awk, программа, 305
AXSIG, константа, 314

B

B0, константа, 758
B110, константа, 758
B115200, константа, 758
B1200, константа, 758
B134, константа, 758
B150, константа, 758
B1800, константа, 758
B19200, константа, 758
B200, константа, 758
B2400, константа, 758
B300, константа, 758
B38400, константа, 758
B4800, константа, 758
B50, константа, 758
B57600, константа, 758
B600, константа, 758
B75, константа, 758
B9600, константа, 758
bdflush, программа, 506
/bin/false, программа, 215
/bin/true, программа, 215
bind, функция, 663, 681, 697, 699, 702, 922
  определение, 659
Bourne shell, командная оболочка, 29, 992
Bourne-again shell, командная оболочка, 29, 992
BRKINT, константа, 738, 750
BSDLY, константа, 740, 750
buf_args, определение функции, 957
BUFSIZ, константа, 84, 185
build_qonstart, функция, 894, 899
  определение, 891
BUS_ADRALN, константа, 400
BUS_ADRERR, константа, 400
BUS_OBJERR, константа, 400

C

C shell, командная оболочка, 29
caddr_t, тип данных, 93
calloc, функция, 553, 597, 837, 922, 976
  определение, 245
cat, программа, 347, 998
CBAUDEXT, константа, 737, 750
cbreak mode, посимвольный режим ввода, 734
cc, программа, 33
CCAR_OFLOW, константа, 737, 750
CCTS_OFLOW, константа, 737, 750
cd, программа, 49
CDSR_OFLOW, константа, 737, 750
CDTR_IFLOW, константа, 737, 751
cfgetispeed, функция, 741, 922
  определение, 758
cfgetospeed, функция, 741, 922
  определение, 758
cfsetispeed, определение функции, 758, 922
cfsetospeed, определение функции, 758, 922
CHAR_BIT, константа, 70
CHAR_MAX, константа, 70, 71
CHAR_MIN, константа, 70, 71
chdir, функция, 172, 328, 509, 922, 972
  определение, 172
CHILD_MAX, константа, 76
chmod, программа, 136
chmod, функция, 143, 702, 795, 797, 922, 998
  определение, 143
chown, функция, 146, 328, 795, 922, 998
  определение, 147
chroot, функция, 970, 985
CIBAUDEXT, константа, 737, 751
CIGNORE, константа, 737, 751
CLD_CONTINUED, константа, 401
CLD_DUMPED, константа, 401
CLD_EXITED, константа, 401
CLD_KILLED, константа, 401
CLD_STOPPED, константа, 401
CLD_TRAPPED, константа, 401
cleanup_push, определение функции, 442
clearenv, функция, 250
clearerr, функция, 191, 922
  определение, 190
cli_conn, определение функции, 693, 695, 701, 957
client_cleanup, определение функции, 891, 906
client_thread, определение функции, 891, 901
CLOCAL, константа, 737, 751
clock_t, тип данных, 48, 93
clone, функция, 269
close, функция, 35, 87, 101, 516, 590, 591, 597, 603, 634, 645, 671, 673, 682, 688, 693, 695, 699, 700, 702, 718, 719, 721, 728, 730, 731, 791, 792, 796, 798,

801, 813, 838, 903, 904, 907, 911, 914, 923
определение, 101
closedir, функция, 34, 764, 901, 923
определение, 167
closelog, функция, 511, 923
close-on-exec, флаг, 295
clrasync, функция, 997
clr_fl, функция, 525, 957
cmsgcred, структура, 712
CMSG_DATA, функция, 710, 716, 923
определение, 709
CMSG_FIRSTHDR, функция, 709, 923
cmsghdr, структура, 708
CMSG_LEN, функция, 709, 923
CMSG_NXTHDR, функция, 709, 923
CMSPAR, константа, 740, 751
COLL_WEIGHTS_MAX, константа, 76
COLUMNS, переменная окружения, 249
compression, атрибут, 869
comp_t, тип данных, 93
configfile, функция, 879
connect, функция, 661, 702, 923
определение, 660
connect_retry, функция, 669, 884, 911
определение, 661, 877
connld, модуль, 691
Content-Length, атрибут, 870
Content-Type, атрибут, 870
cooked mode, подготовленный режим ввода, 734
core, файл, 969, 976
cpio, программа, 971
CR, служебный символ, 742, 745
CRDLY, константа, 740, 751
CREAD, константа, 737, 751
creat, функция, 100, 534, 693, 902, 903, 923, 969, 972
определение, 100
cron, программа, 506, 981
CRTSCTS, константа, 737, 751
CRTS_IFLOW, константа, 737, 751
CRTSXOFF, константа, 737, 751
crypt, функция, 328
csetispeed, функция, 741
csetospeed, функция, 741
CSIZE, константа, 737, 751
csopen, функция, 718
определение, 719, 724
CSTOPB, константа, 737, 751
ctermid, определение функции, 760, 923
ctime, функция, 228, 923
определение, 229
cupsd, программа, 506, 870
curses, библиотка, 778
cuserid, функция, 320

D

daemonize, функция, 518, 521, 671, 674, 678, 727, 891, 957, 986
определение, 508
date, программа, 40, 229, 975, 998
DATEMSK, переменная окружения, 249
DATLEN_MAX, константа, 831
DATLEN_MIN, константа, 831
db, библиотека, 819
DB, структура, 833
_db_alloc, функция, 834, 837
определение, 834
db_close, функция, 820, 825, 830, 838
определение, 820
db_delete, функция, 831, 845, 859, 999
определение, 821
_db_dodelete, функция, 834, 845, 846, 853, 858, 865, 999
db_fetch, функция, 823, 830, 839, 859
определение, 821
_db_find_and_lock, функция, 834, 839, 845, 851, 840, 864
_db_findfree, функция, 834, 852, 854, 858
_db_free, функция, 834, 835, 838
_db_hash, функция, 834, 840, 841, 865
DB_INSERT, константа, 821, 831
dbm, библиотека, 818
db_nextrec, функция, 826, 831, 856, 858, 865, 999
определение, 821
db_open, функция, 820, 825, 830, 834
определение, 820
_db_readdat, функция, 839, 857, 999
определение, 834
_db_readidx, функция, 840, 842, 854, 857, 999
определение, 834
_db_readptr, функция, 840, 842, 847, 852–854, 865
определение, 834
DB_REPLACE, константа, 821, 831
db_rewind, функция, 831, 837, 856
определение, 821

DB_STORE, константа, 821, 831
db_store, функция, 822, 825, 831, 851, 859, 865
  определение, 820
_db_writedat, функция, 834, 846, 848, 852–854, 858, 865, 999
_db_writeidx, функция, 834, 849, 852, 853, 858, 865, 999
db_writeidx, функция, 847
_db_writeptr, функция, 834, 847, 850, 852, 853, 855
detachstate, атрибут потока, 467
/dev/fd, каталог, 126
/dev/ptmx, устройство клонирования, 790, 797
/dev/tty, специальный файл устройства, 338, 760
dev_t, тип данных, 93
DIR, структура, 34, 168, 899
dirent, структура, 34, 168, 899
DISCARD, служебный символ, 742, 745
document-format, атрибут, 868, 912
document-name, атрибут, 868
document-natural-language, атрибут, 868
do_driver, функция, 803
  определение, 813
DSUSP, служебный символ, 742, 745
du, программа, 970
dup, функция, 87, 115, 509, 645, 924, 967, 968, 977
  определение, 115
dup2, функция, 115, 591, 597, 603, 645, 673, 689, 719, 801, 813, 924, 967
  определение, 115

E

EACCES, константа, 43, 974
EAGAIN, константа, 44, 524, 663, 794
EBADF, константа, 87
EBUSY, константа, 44
ECHO, константа, 739, 751
echo, программа, 992, 998
ECHOCTL, константа, 739, 752
ECHOE, константа, 739, 752
ECHOK, константа, 739, 752
ECHOKE, константа, 739, 752
ECHONL, константа, 739, 752
ECHOPRT, константа, 739, 752
EFBIG, константа, 982
EIDRM, константа, 619
EINTR, константа, 44, 309, 375, 937
EINVAL, константа, 82
EIO, константа, 793, 900, 904
ENAMETOOLONG, константа, 99
endgrent, функция, 220, 924
endhostent, функция, 652, 924
endnetent, функция, 652, 924
endprotoent, функция, 653, 924
endpwent, функция, 216, 217, 924
endservent, функция, 653, 924
endspent, функция, 219, 924
ENFILE, константа, 44
ENOBUFS, константа, 44
ENOENT, константа, 43, 793, 821, 851
ENOLCK, константа, 44
ENOMEM, константа, 44
ENOSPC, константа, 44
ENOSR, константа, 44
environ, глобальная переменная, 241
EOF, служебный символ, 742, 745
EOL, служебный символ, 742, 745
EOL2, служебный символ, 742, 746
EPERM, константа, 299
ERANGE, константа, 85
ERASE, служебный символ, 742, 746
ERASE2, служебный символ, 743, 746
err_doit, определение функции, 960, 962
err_dump, функция, 370, 707, 711, 716, 834–849, 854–857, 959
  определение, 958, 961
err_exit, функция, 959
  определение, 958, 961
err_msg, функция, 688, 959
  определение, 958, 961
errno, глобальная переменная, 41
err_quit, функция, 658, 727, 744, 803, 825, 849, 850, 884, 886, 959
  определение, 958, 961
err_ret, функция, 39, 47, 707, 711, 715, 959
  определение, 958, 960
err_sys, функция, 34, 37, 47, 82, 87, 239, 378, 688, 697, 707, 711, 718–720, 724, 725, 729, 749, 775, 777, 803, 825, 836, 884, 886, 959
  определение, 958, 960
ESPIPE, код ошибки, 102
ESRCH, константа, 384
/etc/group, файл групп, 213, 219

/etc/master.passwd, теневой файл паролей (FreeBSD), 223
/etc/networks, файл с перечнем сетей, 223
/etc/passwd, файл паролей, 213
/etc/protocols, файл с перечнем сетевых протоколов, 223
/etc/pwd.db, хешированный файл паролей, 222
/etc/services, файл с перечнем сетевых служб, 223
/etc/shadow, теневой файл паролей, 218
/etc/spwd.db, хешированный теневой файл паролей, 222
/etc/termcap, файл, 778
ETIME, константа, 881
EWOULDBLOCK, константа, 44, 524, 663
exec, функция, 38, 76, 291, 984
execl, функция, 305, 328, 420, 592, 597, 603, 673, 689, 719, 924, 979
  определение, 292
execle, функция, 296, 924
  определение, 292
execlp, функция, 39, 47, 297, 813, 924, 979
  определение, 292
execv, определение функции, 292, 924
execve, определение функции, 292, 924
execvp, функция, 803, 925
  определение, 292
_Exit, функция, 277, 925
  определение, 235
_exit, функция, 277, 925
  определение, 235
exit, функция, 34, 276, 277, 925
  определение, 235
expect, программа, 785
EXTPROC, константа, 739, 753

F

fattach, функция, 694, 925
  определение, 690
fchdir, функция, 172, 645, 925
  определение, 172
fchmod, функция, 143, 542, 645, 925
  определение, 143
fchown, функция, 146, 645, 925
  определение, 147
fclose, функция, 236, 767, 879, 925, 977
  определение, 189
fcntl, функция, 118, 295, 524, 528, 533, 645, 684, 861, 862, 925, 967, 996
  определение, 118
fdatasync, функция, 117, 645, 925
  определение, 117
FD_CLOEXEC, флаг дескриптора, 119
FD_CLR, функция, 728, 925, 989
  определение, 563
fdetach, определение функции, 691, 926
FD_ISSET, функция, 727, 894, 926, 989
  определение, 563
fdopen, функция, 187, 597, 926, 992
  определение, 187
fd_set, тип данных, 93, 989, 996
FD_SET, функция, 727, 728, 881, 893, 926, 989
  определение, 563
FD_SETSIZE, константа, 564, 989
F_DUPFD, команда функции fcntl, 118, 119
F_DUPFD, константа, 925
FD_ZERO, функция, 881, 893, 926, 989
  определение, 563
feof, функция, 190, 926
  определение, 190
ferror, функция, 190, 317, 591, 596, 603, 688, 926
  определение, 190
FFDLY, константа, 740, 753
fflush, функция, 416, 600, 926, 962, 964, 965, 974
  определение, 186
fg, команда, 342
fgetc, функция, 190, 926
  определение, 190
F_GETFD, команда функции fcntl, 119
F_GETFD, константа, 925
F_GETFL, команда функции fcntl, 119
F_GETFL, константа, 925
F_GETLK, константа, 528, 530, 925
F_GETOWN, команда функции fcntl, 120
F_GETOWN, константа, 925
fgetpos, определение функции, 199, 926
fgets, функция, 37, 39, 47, 591, 596, 600, 603, 678, 688, 718, 879, 926, 973, 974, 992
  определение, 192
FIFO, именованные каналы, 605

FILE, структура, 986, 992
FILE_MODE, константа, 994
fileno, функция, 598, 926, 974
  определение, 205
FILESIZEBITS, константа, 77
finger, программа, 215, 971
FIOASYNC, константа, 684, 996
FIOSETOWN, константа, 684
FIPS, стандарт, 64
flock, структура, 528
flock, функция, 527
flockfile, определение функции, 482, 926
FLUSHO, константа, 739, 753
FNDELAY, константа, 524
F_OK, константа, 139, 921
fopen, функция, 187, 259, 317, 591, 595, 767, 879, 927, 986
  определение, 187
FOPEN_MAX, константа, 72
fork, функция, 38, 39, 47, 268, 269, 284, 296, 381, 419, 508, 542, 590, 597, 634, 673, 688, 719, 805, 813, 927, 979, 984, 986, 987, 990
  определение, 268
fpathconf, функция, 69, 74, 744, 927
  определение, 76
FPE_FLTDIV, константа, 400
FPE_FLTINV, константа, 400
FPE_FLTOVF, константа, 400
FPE_FLTRES, константа, 400
FPE_FLTSUB, константа, 400
FPE_FLTUND, константа, 400
FPE_INTDIV, константа, 400
FPE_INTOVF, константа, 400
fpos_t, тип данных, 93
fprintf, функция, 43, 525, 670, 677, 803, 927, 965, 986
  определение, 200
fputc, определение функции, 192, 927
fputs, функция, 82, 596, 600, 603, 688, 767, 927, 962, 964, 973, 975, 992
  определение, 193
F_RDLCK, константа, 528, 987
fread, функция, 317, 927
  определение, 196
free, функция, 246, 449, 454, 762, 763, 839, 907, 910, 911, 914, 919, 927
  определение, 245
freeaddrinfo, функция, 910, 927
  определение, 654
FreeBSD, операционная система, 67
freopen, функция, 187, 927
  определение, 187
fscanf, определение функции, 203, 928
fseek, определение функции, 198, 928
fseeko, определение функции, 199, 928
F_SETFD, команда функции fcntl, 119
F_SETFD, константа, 925, 967
F_SETFL, команда функции fcntl, 120
F_SETFL, константа, 684, 925, 967, 998
F_SETLK, константа, 528, 530, 925, 987
F_SETLKW, константа, 528, 530, 925
F_SETOWN, команда, 996
F_SETOWN, команда функции fcntl, 120
F_SETOWN, константа, 682, 684, 925
fsetpos, определение функции, 199, 928
fstat, функция, 30, 129, 542, 581, 645, 836, 884, 911, 928
  определение, 129
fsync, функция, 117, 865, 928, 974
  определение, 117
ftell, определение функции, 198, 928
ftello, определение функции, 199, 928
ftok, определение функции, 611, 928
ftpd, программа, 985
ftruncate, функция, 150, 516, 646, 928
  определение, 150
ftrylockfile, определение функции, 482, 928
F_UNLCK, константа, 528
funlockfile, определение функции, 482, 929
fwide, определение функции, 182, 929
fwrite, функция, 929, 982
  определение, 196
F_WRLCK, константа, 528

G

gai_strerror, функция, 674, 679, 884, 929
  определение, 655
Garfinkel, S., 218
gawk, программа, 306
gcc, программа, 33
gdb, программа, 984
gdbm, библиотека, 819
getaddrinfo, функция, 658, 669, 671, 676, 679, 878, 929
  определение, 654

getaddrlist, функция, 880, 884, 893
определение, 877, 878
GETALL, константа, 625, 946
getc, функция, 37, 190, 767, 929, 973
определение, 190
getchar, функция, 190, 600, 929, 973
определение, 190
getchar_unlocked, определение функции, 483, 929
getcwd, функция, 85, 172, 174, 929, 971, 972
определение, 174
getegid, функция, 714, 930
определение, 267
getenv, функция, 250, 592, 930, 984
определение, 248
geteuid, функция, 301, 714, 930
определение, 267
getgid, определение функции, 267, 930
getgrent, определение функции, 220, 930
getgrgid, функция, 224, 930
определение, 220
getgrnam, функция, 224, 930
определение, 220
getgroups, определение функции, 221, 930
gethostbyaddr, функция, 224
gethostent, определение функции, 652, 930
gethostname, функция, 76, 671, 674, 678, 892, 930
определение, 226
getlogin, функция, 320, 930, 986
определение, 320
getmsg, функция, 554, 557, 646, 707, 931, 989
определение, 555
getnameinfo, определение функции, 655, 931
GETNCNT, константа, 625, 946
getnetbyaddr, функция, 224, 931
определение, 652
getnetbyname, функция, 224, 931
определение, 652
getnetent, определение функции, 652, 931
get_newjobno, функция, 891, 89, 902
getopt, функция, 727, 802, 883, 887, 931
определение, 888
getpass, функция, 328, 339, 768
определение, 767
getpeername, определение функции, 660, 931
getpgid, определение функции, 334, 931
getpgrp, определение функции, 334, 931
GETPID, константа, 625, 946
getpid, функция, 435, 714, 932
определение, 267
getpmsg, функция, 554, 646, 932
определение, 555
getppid, функция, 284, 805, 932
определение, 267
get_printaddr, функция, 896
определение, 877, 880
get_printserver, функция, 884
определение, 877, 880
getprotobyname, функция, 224, 932
определение, 653
getprotobynumber, функция, 224, 932
определение, 653
getprotoent, определение функции, 653, 932
getpwent, определение функции, 216, 932
getpwnam, функция, 224, 328, 378, 893, 932, 974, 975
определение, 216, 217
getpwuid, функция, 224, 932, 974
определение, 216
getrlimit, функция, 88, 259, 263, 508, 932, 966, 967
определение, 259
getrusage, функция, 321
gets, функция, 933, 973
определение, 192
getservbyname, функция, 224, 933
определение, 653
getservbyprot, функция, 224, 933
определение, 653
getservent, определение функции, 653, 933
getsid, определение функции, 337, 933
getsockname, определение функции, 660, 933
getsockopt, определение функции, 681, 933
getspent, определение функции, 219, 933
getspnam, функция, 224, 933, 974
определение, 219

gettimeofday, gettimeofday, функция, 477, 933
  определение, 227
getty, программа, 327
gettytab, конфигурационный файл программы getty, 327
getuid, определение функции, 267, 933
GETVAL, константа, 625, 946
GETZCNT, константа, 946
GETZNCNT, константа, 625
gid_t, тип данных, 93
gmtime, функция, 228, 934
  определение, 229
grantpt, функция, 789, 791, 796, 798, 934
  определение, 788, 795, 797
grep, программа, 49, 237
guardsize, атрибут потока, 467

H

HOME, переменная окружения, 249, 328
hostent, структура, 652
HOST_NAME_MAX, константа, 76, 226, 672
HP-UX, операционная система, 68
htonl, функция, 903, 904, 912, 934
  определение, 648
htons, функция, 908, 911, 934
  определение, 648
HTTP, протокол передачи гипертекста, 867, 869
HTTP_INFO, функция, 916
  определение, 889
HTTP_SUCCESS, функция, 916
  определение, 889
HUPCL, константа, 737, 753

I

ICANON, константа, 739, 753
ICRNL, константа, 738, 753
IDXLEN_MAX, константа, 831
IDXLEN_MIN, константа, 831
IEEE P1003.1a, стандарт, 57
IEEE P1003.2b, предварительный стандарт, 57
IEEE POSIX, стандарт, 55
IEEE Standard 1003.1-1990, стандарт, 56
IEEE Standard 1003.1-2001, стандарт, 56
IEEE Standard 1003.1b-1993, стандарт, 56
IEEE Standard 1003.1c-1995, стандарт, 56
IEEE Standard 1003.1d-1999, стандарт, 56, 57
IEEE Standard 1003.1g-2000, стандарт, 57
IEEE Standard 1003.1i-1995, стандарт, 56
IEEE Standard 1003.1j-2000, стандарт, 56, 57
IEEE Standard 1003.1q-2000, стандарт, 56, 57
IEEE Standard 1003.2-1993, стандарт, 57
IEEE Standard 1003.2d-1994, стандарт, 57
IEEE Std 1003.1–1990, стандарт, 56
IEXTEN, константа, 739, 753
I_FIND, константа, 793
IGNBRK, константа, 738, 753
IGNCR, константа, 738, 753
IGNPAR, константа, 738, 753
ILL_BADSTK, константа, 400
ILL_COPROC, константа, 400
ILL_ILLADR, константа, 400
ILL_ILLOPC, константа, 400
ILL_ILLOPN, константа, 400
ILL_ILLTRP, константа, 400
ILL_PRVOPC, константа, 400
ILL_PRVREG, константа, 400
IMAXBEL, константа, 738, 753
in6_addr, структура, 650
in_addr, структура, 649
INADDR_ANY, константа, 660
INET6_ADDRSTRLEN, константа, 651
inet_addr, функция, 650
INET_ADDRSTRLEN, константа, 651
inetd, программа, 331, 506
inet_ntoa, функция, 650
inet_ntop, определение функции, 651, 934
inet_pton, определение функции, 651, 934
init, программа, 267
initgroups, функция, 328, 934
  определение, 221
init_printer, функция, 891, 892, 896, 910
init_request, функция, 891, 892, 895
initserver, функция, 671, 674, 679, 877, 893
  определение, 663, 681

INLCR, константа, 738, 753
i-node, 965
ino_t, тип данных, 94
INPCK, константа, 738, 754
INT_MAX, константа, 70
INT_MIN, константа, 70
INTR, служебный символ, 743, 746
ioctl, функция, 124, 338, 552, 646, 684, 694, 706, 707, 777, 792, 797, 801, 803, 934, 996
  определение, 125

_IOFBF, константа, 186, 948
_IOLBF, константа, 186, 948, 992
_IONBF, константа, 186, 948
iovec, структура, 571
IOV_MAX, константа, 76
IPC_CREAT, константа, 611, 936, 946, 948
IPC_EXCL, константа, 611, 936, 946, 948
IPC_NOWAIT, константа, 618, 936
ipc_perm, структура, 611
IPC_PRIVATE, константа, 610
IPC_RMID, константа, 617, 624, 630, 936, 946, 948
IPC_SET, константа, 617, 624, 630, 936, 946, 948
IPC_STAT, константа, 617, 624, 630, 936, 946, 948
IPP, протокол печати через Интернет, 866, 867
ipp-attribute-fidelity, атрибут, 868
ipp_hdr, структура, 875
I_RECVFD, константа, 705
IRIX, операционная система, 68
isastream, функция, 550, 551, 552, 695, 934
  определение, 550
isatty, функция, 744, 761, 765, 778, 802, 811, 934
  определение, 761
isdigit, функция, 916, 917
I_SENDFD, константа, 705
ISIG, константа, 739, 754
ISO C, стандарт, 53
ISO/IEC 9945-1, стандарт, 56
ISO/IEC 9945-2, стандарт, 57
is_read_lockable, определение функции, 533, 958
ISRIP, константа, 738
isspace, функция, 916, 917
ISTRIP, константа, 754
isupper, функция, 600
is_write_lockable, функция, 533, 958
IUCLC, константа, 738, 754
IXANY, константа, 738, 754
IXOFF, константа, 738, 754
IXON, константа, 738, 754

J

job, структура, 983
job_find, функция, 983
job-impressions, атрибут, 869
job-k-octets, атрибут, 869
job-media-sheets, атрибут, 869
job-name, атрибут, 868
job_remove, функция, 983

K

Kernighan and Ritchie, 246
kill, определение функции, 383, 935
kill, программа, 357, 370
KILL, служебный символ, 743, 746
kill, функция, 46, 357, 383, 416, 427, 805, 981, 988
kill_workers, функция, 891, 906, 907
Korn shell, командная оболочка, 29, 992
kupdated, программа, 506

L

LANG, переменная окружения, 249
LC_ALL, переменная окружения, 249
LC_COLLATE, переменная окружения, 249
LC_CTYPE, переменная окружения, 249
lchown, функция, 146, 935
  определение, 147
LC_MESSAGES, переменная окружения, 249
LC_MONETARY, переменная окружения, 249
LC_NUMERIC, переменная окружения, 249
LC_TIME, переменная окружения, 249
LDAP, облегченный протокол доступа к каталогам, 223
ldterm, модуль STREAMS, 783
libmalloc, библиотека функций распределения памяти, 247

limit, встроенная команда, 261
LINE_MAX, константа, 76
LINES, переменная окружения, 249
link, функция, 153, 935
  определение, 154
LINK_MAX, константа, 77
lint, программа, 237
Linux, операционная система, 67
listen, функция, 663, 682, 699
  определение, 662, 935
LLONG_MAX, константа, 71
LLONG_MIN, константа, 71
LNEXT, служебный символ, 743, 746
localtime, функция, 228, 975
  определение, 229, 935
lockd, программа, 506
lockf, функция, 527
lockfile, функция, 516
lock_reg, функция, 958, 987
lock_test, функция, 958
LOG_ALERT, константа, 513
LOG_AUTH, константа, 513, 937
LOG_AUTHPRIV, константа, 513, 937
LOG_CONS, константа, 512, 937
LOG_CRIT, константа, 513
LOG_CRON, константа, 513, 937
LOG_DAEMON, константа, 513, 937
LOG_DEBUG, константа, 513
log_doit, определение функции, 962, 964
LOG_EMERG, константа, 513
LOG_ERR, константа, 513
LOG_FTP, константа, 513, 937
logger, команда, 514
login, программа, 216, 327
LOG_INFO, константа, 513
LOGIN_NAME_MAX, константа, 76
LOG_KERN, константа, 513, 937
LOG_LOCAL, константа, 937
LOG_LOCAL0, константа, 513
LOG_LOCAL1, константа, 513
LOG_LOCAL2, константа, 513
LOG_LOCAL3, константа, 513
LOG_LOCAL4, константа, 513
LOG_LOCAL5, константа, 513
LOG_LOCAL6, константа, 513
LOG_LOCAL7, константа, 513
LOG_LPR, константа, 513, 937
LOG_MAIL, константа, 513, 937
log_msg, функция, 729, 731, 880, 896, 900–905, 907, 910, 911, 913, 916, 918, 919, 959
  определение, 958
LOGNAME, переменная окружения, 249, 328
LOG_NDELAY, константа, 512, 937, 985
LOG_NEWS, константа, 513, 937
LOG_NOTICE, константа, 513
LOG_NOWAIT, константа, 512, 937
LOG_ODELAY, константа, 512, 937
log_open, функция, 727, 958, 962
LOG_PERROR, константа, 512, 937
LOG_PID, константа, 512, 937
log_quit, функция, 726, 893, 895, 907, 908, 959
  определение, 958, 963
log_ret, функция, 894, 905, 913, 914, 960
  определение, 958, 963
log_sys, функция, 727, 729, 731, 893, 895–897, 915, 960
  определение, 958, 963
LOG_SYSLOG, константа, 513, 937
LOG_USER, константа, 513, 937
LOG_UUCP, константа, 513, 937
LOG_WARNING, константа, 513
longjmp, функция, 255, 257, 387, 390, 980
  определение, 254, 935
longjump, функция, 252
LONG_MAX, константа, 71, 966
LONG_MIN, константа, 71
loop, функция, 803, 805
lp, программа, 870
lpd, программа, 870
lpsched, программа, 870
ls, программа, 41, 965
lseek, функция, 35, 101, 542, 581, 646, 842, 843, 845, 850, 856, 896, 968
  определение, 101, 935
lstat, функция, 129, 935
L_tmpnam, константа, 208

M

Mac OS X, операционная система, 67
main, функция, 234
mallinfo, функция, 247
malloc, функция, 50, 246, 449, 454, 477, 632, 671, 674, 678, 710, 714, 715, 725, 729, 762, 837, 838, 892, 897, 905, 916, 982, 984
  определение, 245, 935

MALLOC_OPTIONS, переменная окружения, 984
mallopt, функция, 247
MAP_ANON, константа, 635
MAP_ANONYMOUS, константа, 636
MAP_FAILED, константа, 936
MAP_FIXED, константа, 578, 936
MAP_PRIVATE, константа, 578, 635, 936
MAP_SHARED, константа, 578, 634, 635, 936
M_ASYNC, константа, 580
MAX_CANON, константа, 77, 735
MAX_INPUT, константа, 77, 734
MB_LEN_MAX, константа, 71
M_DATA, тип сообщения STREAMS, 548
MDMBUF, константа, 737, 754
memcpy, функция, 581, 897
memset, функция, 699, 701
mgetty, программа, 330
mkdir, функция, 165, 972
  определение, 165, 935
mkfifo, функция, 994
  определение, 606, 935
mkstemp, определение функции, 210, 935
mktime, функция, 228
  определение, 229, 936
mlock, функция, 260
mmap, функция, 581, 634, 646
  определение, 576, 936
mode_t, тип данных, 94
mount, функция, 692
M_PCPROTO, тип сообщения STREAMS, 548
mprotect, определение функции, 579, 936
M_PROTO, тип сообщения STREAMS, 548
MSG_ANY, константа, 932
MSG_BAND, константа, 932, 944
msgctl, определение функции, 617, 936
MSG_CTRUNC, константа, 668
MSG_DONTROUTE, константа, 665, 946
MSG_DONTWAIT, константа, 665, 668, 946
MSG_EOR, константа, 665, 668, 946
msgget, определение функции, 616, 936
msghdr, структура, 666, 708
MSG_HIPRI, константа, 932, 944
MSG_NOERROR, константа, 619, 936
MSG_OOB, константа, 665, 667, 668, 682, 945, 946
MSG_PEEK, константа, 667, 945
msgrcv, определение функции, 619, 936
msgsnd, определение функции, 618, 936
MSG_TRUNC, константа, 667, 668, 945
MSGVERB, переменная окружения, 249
MSG_WAITALL, константа, 667, 945
M_SIG, тип сообщения STREAMS, 548
MS_INVALIDATE, константа, 580
msqid_ds, структура, 615
M_SYNC, константа, 580
msync, определение функции, 579, 936
munmap, определение функции, 580, 936

N

NAME_MAX, константа, 77
nanosleep, функция, 477
nawk, программа, 306
NCCS, константа, 736
ncurses, библиотека, 779
ndbm, библиотека, 819, 820
netent, структура, 652
newgrp, программа, 220
nfsd, программа, 506
NGROUPS_MAX, константа, 76, 222
NI_DGRAM, константа, 656
NI_NAMEREQD, константа, 656
NI_NOFQDN, константа, 656
NI_NUMERICHOST, константа, 656
NI_NUMERICSERV, константа, 656
NIS, сетевая информационная служба, 223
NL, служебный символ, 743, 746
NL_ARGMAX, константа, 75
NLDLY, константа, 740, 754
nlink_t, тип данных, 94
NL_LANGMAX, константа, 75
NL_MSGMAX, константа, 75
NL_NMAX, константа, 75
NL_SETMAX, константа, 75
NLSPATH, переменная окружения, 250
NL_TEXTMAX, константа, 75
NOFLSH, константа, 739, 755
NOKERNINFO, константа, 739, 755
nologin, программа, 215
ntohl, функция, 902, 918
  определение, 648, 936

ntohs, функция, 918
определение, 648, 937
NZERO, константа, 75

O

O_ACCMODE, константа, 119
O_APPEND, константа, 98, 937
O_CREAT, константа, 98, 835, 937
OCRNL, константа, 740, 755
od, программа, 104
O_DSYNC, константа, 98, 937
O_EXCL, константа, 98, 937
OFDEL, константа, 740, 755
offsetof, функция, 697, 699, 700, 702, 912
определение, 698
off_t, тип данных, 94
OFILL, константа, 740, 755
OLCUC, константа, 740, 755
O_NDELAY, константа, 68, 98, 524
ONLCR, константа, 740, 755, 811
ONLRET, константа, 740, 755
ONOCR, константа, 740, 755
O_NOCTTY, константа, 98, 338, 937, 938
ONOEOT, константа, 740, 755
O_NONBLOCK, константа, 68, 98, 524, 937, 994
OP_CANCEL_JOB, константа, 874
OP_CREATE_JOB, константа, 874
open, функция, 35, 41, 97, 338, 509, 516, 542, 551, 552, 581, 634, 695, 721, 731, 791, 792, 794, 796, 797, 798, 820, 835, 884, 895, 900, 910, 967, 969, 987, 993
определение, 97, 937
opendir, функция, 34, 295, 763, 900, 970
определение, 167, 937
openlog, функция, 509, 963, 985
определение, 511, 937
OPEN_MAX, константа, 77, 966
open_max, определение функции, 957, 967
openpt, определение функции, 794
OP_GET_JOB_ATTR, константа, 874
OP_GET_JOBS, константа, 874
OP_HOLD_JOB, константа, 874
OPOST, константа, 740, 755
OP_PAUSE_PRINTER, константа, 874
OP_PRINT_JOB, константа, 874
OP_PRINT_URI, константа, 874
OP_PURGE_JOBS, константа, 874
OP_RELEASE_JOB, константа, 874
OP_RESTART_JOB, константа, 874
OP_RESUME_PRINTER, константа, 874
OP_SEND_DOC, константа, 874
OP_SEND_URI, константа, 874
optarg, глобальная переменная, 888
opterr, глобальная переменная, 888
optind, глобальная переменная, 888
optopt, глобальная переменная, 888
OP_VALIDATE_JOB, константа, 874
O_RDONLY, константа, 97, 98, 937, 994
O_RDWR, константа, 98, 937
O_RSYNC, константа, 99, 937
O_RWDR, константа, 938
O_SYNC, константа, 99, 937
O_TRUNC, константа, 98, 825, 937
O_WRONLY, константа, 98, 937, 994
OXTABS, константа, 740, 755

P

PAGE_SIZE, константа, 77
PAGESIZE, константа, 77
P_ALL, константа, 286, 955
PARENB, константа, 738, 756
PAREXT, константа, 738, 756
PARMRK, константа, 738, 756
PARODD, константа, 738, 756
passwd, структура, 213, 975
PATH, переменная окружения, 250, 292, 328
path_alloc, функция, 972
определение, 957
pathconf, функция, 69, 74
определение, 76, 937
PATH_MAX, константа, 77, 971
pause, функция, 369, 372, 381, 385, 387, 405, 980
определение, 385, 937
_PC_ASYNC_IO, константа, 90, 927, 937
_PC_CHOWN_RESTRICTED, константа, 90, 927, 937
_PC_FILESIZEBITS, константа, 77, 927, 937
pckt, модуль STREAMS, 783
_PC_LINK_MAX, константа, 77, 927, 937
pclose, функция, 311, 594, 596, 601, 671, 678, 992, 993
определение, 594, 937

_PC_MAX_CANON, константа, 77, 927, 937
_PC_MAX_INPUT, константа, 77, 927, 937
_PC_NAME_MAX, константа, 77, 927, 937
_PC_NO_TRUNC, константа, 90, 927, 937
_PC_PATH_MAX, константа, 77, 927, 937
_PC_PIPE_BUF, константа, 77, 927, 937
_PC_PRIO_IO, константа, 90, 937
PC_PRIO_IO, константа, 927
_PC_SYMLINK_MAX, константа, 77, 927, 937
_PC_SYNC_IO, константа, 90, 927, 937
_PC_VDISABLE, константа, 90, 927, 937
PENDIN, константа, 739, 756
perror, функция, 43, 381, 965
  определение, 43, 937
pid_t, тип данных, 94
pipe, функция, 590, 591, 597, 602, 693, 991
  определение, 587, 937
PIPE_BUF, константа, 77, 589, 691, 991
poll, функция, 566, 646, 729, 993
  определение, 566, 938
POLL_ERR, константа, 401
POLLERR, константа, 568
pollfd, структура, 567
POLL_HUP, константа, 401
POLLHUP, константа, 568, 993
POLL_IN, константа, 401
POLLIN, константа, 568, 993
POLL_MSG, константа, 401
POLLNVAL, константа, 568
POLL_OUT, константа, 401
POLLOUT, константа, 568
POLL_PRI, константа, 401
POLLPRI, константа, 568
POLLRDBAND, константа, 568
POLLRDNORM, константа, 568
POLLWRBAND, константа, 568
POLLWRNORM, константа, 568
popen, функция, 311, 594, 595, 600, 671, 678, 992
  определение, 594, 938
POSIX.1, стандарт, 56
_POSIX2_LINE_MAX, константа, 75
_POSIX_ADVISORY_INFO, константа, 60
_POSIX_ARG_MAX, константа, 73
_POSIX_ASYNCHRONOUS_IO, константа, 60
_POSIX_ASYNC_IO, константа, 90
_POSIX_BARRIERS, константа, 60
_POSIX_CHILD_MAX, константа, 73
_POSIX_CHOWN_RESTRICTED, константа, 90
_POSIX_CLOCK_SELECTION, константа, 60
_POSIX_CPUTIME, константа, 60
_POSIX_FSYNC, константа, 60
_POSIX_HOST_NAME_MAX, константа, 73
_POSIX_JOB_CONTROL, константа, 89
_POSIX_LINK_MAX, константа, 73
_POSIX_LOGIN_NAME_MAX, константа, 73
_POSIX_MAPPED_FILES, константа, 60
_POSIX_MAX_CANON, константа, 73
_POSIX_MAX_INPUT, константа, 73
_POSIX_MEMLOCK, константа, 60
_POSIX_MEMLOCK_RANGE, константа, 61
_POSIX_MEMORY_PROTECTION, константа, 61
_POSIX_MESSAGE_PASSING, константа, 61
_POSIX_MONOTONIC_CLOCK, константа, 61
_POSIX_NAME_MAX, константа, 73
_POSIX_NGROUPS_MAX, константа, 73
_POSIX_NO_TRUNC, константа, 90
_POSIX_OPEN_MAX, константа, 73
posix_openpt, функция, 789, 798
  определение, 788, 797, 938
_POSIX_PATH_MAX, константа, 73
_POSIX_PIPE_BUF, константа, 73
_POSIX_PRIO_IO, константа, 90
_POSIX_PRIORITIZED_IO, константа, 61
_POSIX_PRIORITIZED_SCHEDULING, константа, 61
_POSIX_RAW_SOCKETS, константа, 61
_POSIX_READER_WRITER_LOCKS, константа, 89
_POSIX_RE_DUP_MAX, константа, 73
_POSIX_SAVED_IDS, константа, 89, 299, 384
_POSIX_SEMAPHORES, константа, 61

_POSIX_SHARED_MEMORY_OBJECTS, константа, 61
_POSIX_SHELL, константа, 89
_POSIX_SOURCE, константа, 49
_POSIX_SPAWN, константа, 61
_POSIX_SPIN_LOCKS, константа, 61
_POSIX_SPORADIC_SERVER, константа, 61
_POSIX_SSIZE_MAX, константа, 73
_POSIX_STREAM_MAX, константа, 73
_POSIX_SYMLINK_MAX, константа, 73
_POSIX_SYMLOOP_MAX, константа, 73
_POSIX_SYNCHRONIZED_IO, константа, 61
_POSIX_SYNC_IO, константа, 90
_POSIX_THREAD_ATTR_STACKADDR, константа, 62, 469
_POSIX_THREAD_ATTR_STACKSIZE, константа, 62, 469
_POSIX_THREAD_CPUTIME, константа, 61
_POSIX_THREAD_PRIO_INHERIT, константа, 61
_POSIX_THREAD_PRIO_PROTECT, константа, 62
_POSIX_THREAD_PRIORITY_SCHEDULING, константа, 62
_POSIX_THREAD_PROCESS_SHARED, константа, 62, 472
_POSIX_THREADS, константа, 61
_POSIX_THREADS, макроопределение, 432
_POSIX_THREAD_SAFE_FUNCTIONS, константа, 62, 480
_POSIX_THREAD_SPORADIC_SERVER, константа, 62
_POSIX_TIMEOUTS, константа, 61
_POSIX_TIMERS, константа, 61
_POSIX_TRACE, константа, 62
_POSIX_TRACE_EVENT_FILTER, константа, 61
_POSIX_TRACE_INHERIT, константа, 62
_POSIX_TRACE_LOG, константа, 62
_POSIX_TTY_NAME_MAX, константа, 73
_POSIX_TYPED_MEMORY_OBJECTS, константа, 62
_POSIX_TZNAME_MAX, константа, 73
_POSIX_V6_ILP32_OFF32, константа, 105
_POSIX_V6_ILP32_OFFBIG, константа, 105
_POSIX_V6_LP64_OFF64, константа, 105
_POSIX_VDISABLE, константа, 90, 743
_POSIX_VERSION, константа, 89
POST, метод, 870
PostScript, формат документов, 871
P_PGID, константа, 286, 955
P_PID, константа, 286, 955
pread, функция, 114
  определение, 114, 938
pr_exit, определение функции, 957
print, программа, 871
printd, программа, 871
printer_status, функция, 914
  определение, 891, 915
printer_thread, определение функции, 891, 909
printer-url, атрибут, 868
printf, функция, 37, 38, 39, 49, 81, 435, 761, 775, 777, 973, 976
  определение, 200, 938
printreq, структура, 877
printresp, структура, 877
pr_mask, определение функции, 957
proc, структура, 354
PROT_EXEC, константа, 577, 936
PROT_NONE, константа, 577, 936
protoent, структура, 653
PROT_READ, константа, 577, 936
PROT_WRITE, константа, 577, 936
ps, программа, 347, 980
pselect, функция, 561
  определение, 566, 938
psignal, определение функции, 427, 938
ptem, модуль STREAMS, 783
pthread_atfork, функция, 501
  определение, 499, 938
pthread_attr, определение функции, 938
pthread_attr_destroy, функция, 468
  определение, 467
pthread_attr_getdetachstate, определение функции, 468, 939
pthread_attr_getguardsize, определение функции, 471, 939
pthread_attr_getstack, определение функции, 469, 939

pthread_attr_getstackaddr, функция, 469
pthread_attr_getstacksize, определение функции, 470, 939
pthread_attr_init, функция, 468
  определение, 467, 939
pthread_attr_setdetachstate, функция, 468
  определение, 468, 939
pthread_attr_setguardsize, определение функции, 471, 939
pthread_attr_setstack, определение функции, 469, 939
pthread_attr_setstackaddr, функция, 469
pthread_attr_setstacksize, определение функции, 470, 939
pthread_cancel, функция, 906
  определение, 442, 939
PTHREAD_CANCEL_ASYNCHRONOUS, константа, 493
PTHREAD_CANCEL_DEFERRED, константа, 493
PTHREAD_CANCEL_DISABLE, константа, 490
PTHREAD_CANCELED, константа, 442
PTHREAD_CANCEL_ENABLE, константа, 490
pthread_cleanup, определение функции, 442, 939
pthread_cleanup_pop, функция, 443, 905
pthread_cleanup_push, функция, 443, 901
  определение, 939
pthread_condattr_destroy, определение функции, 480, 939
pthread_condattr_getpshared, определение функции, 480, 940
pthread_condattr_init, определение функции, 480, 940
pthread_condattr_setpshared, определение функции, 480, 940
pthread_cond_broadcast, функция, 983
  определение, 462, 940
pthread_cond_destroy, определение функции, 461, 940
pthread_cond_init, определение функции, 461, 940
PTHREAD_COND_INITIALIZER, константа, 891
pthread_cond_signal, функция, 463, 496, 898
  определение, 462, 940
pthread_cond_timedwait, определение функции, 461, 940
pthread_cond_wait, функция, 463, 497, 910, 983
  определение, 461, 940
pthread_create, функция, 434, 436, 439, 440, 443, 468, 497, 501, 519, 894, 983
  определение, 940
PTHREAD_CREATE_DETACHED, константа, 468
PTHREAD_CREATE_JOINABLE, константа, 468
PTHREAD_DESTRUCTOR_ITERATIONS, константа, 466, 487
pthread_detach, определение функции, 445, 940
pthread_equal, функция, 460
  определение, 433, 940
pthread_exit, функция, 277, 438, 440, 902–905
  определение, 437, 941
pthread_getconcurrency, определение функции, 471, 941
pthread_getspecific, функция, 489
  определение, 489, 941
pthread_join, функция, 439, 440, 444, 983
  определение, 438, 941
pthread_key_create, функция, 489
  определение, 486, 941
pthread_key_delete, определение функции, 487, 941
PTHREAD_KEYS_MAX, константа, 466, 487
pthread_kill, определение функции, 495, 941
pthread_mutexattr_destroy, функция, 484
  определение, 472, 941
pthread_mutexattr_getpshared, определение функции, 473, 941
pthread_mutexattr_gettype, определение функции, 474, 941
pthread_mutexattr_init, функция, 478, 484
  определение, 472, 941
pthread_mutexattr_setpshared, определение функции, 473, 941

pthread_mutexattr_settype, функция, 478, 484
определение, 474, 941
PTHREAD_MUTEX_DEFAULT, константа, 473
pthread_mutex_destroy, функция, 450, 453, 455
определение, 448, 942
PTHREAD_MUTEX_ERRORCHECK, константа, 473
pthread_mutex_init, функция, 449, 452, 454
определение, 448, 942
PTHREAD_MUTEX_INITIALIZER, константа, 448, 890, 891
pthread_mutex_lock, функция, 450, 452, 454, 463, 478, 484, 489, 496, 497, 501, 897, 898, 905–907, 909, 910
определение, 449, 942
PTHREAD_MUTEX_NORMAL, константа, 473
PTHREAD_MUTEX_RECURSIVE, константа, 473
pthread_mutex_trylock, определение функции, 449, 942
pthread_mutex_unlock, функция, 450, 452, 454, 463, 478, 484, 490, 496, 497, 501, 897, 898, 906, 907, 910
определение, 449, 942
pthread_once, функция, 484, 489
определение, 488, 942
PTHREAD_ONCE_INIT, константа, 488, 942
PTHREAD_PROCESS_PRIVATE, константа, 472
PTHREAD_PROCESS_SHARED, константа, 472
pthread_rwlockattr_destroy, определение функции, 479, 942
pthread_rwlockattr_getpshared, определение функции, 479, 942
pthread_rwlockattr_init, определение функции, 479, 942
pthread_rwlockattr_setpshared, определение функции, 479, 942
pthread_rwlock_destroy, определение функции, 457, 942
pthread_rwlock_init, функция, 459
определение, 457, 942
pthread_rwlock_rdlock, функция, 460
определение, 457, 942
pthread_rwlock_tryrdlock, определение функции, 458, 943
pthread_rwlock_trywrlock, определение функции, 458, 943
pthread_rwlock_unlock, функция, 459, 460
определение, 457, 943
pthread_rwlock_wrlock, функция, 459
определение, 457, 943
pthread_self, функция, 433, 435, 901
определение, 943
pthread_setcancelstate, определение функции, 491, 943
pthread_setcanceltype, определение функции, 493, 943
pthread_setconcurrency, определение функции, 471, 943
pthread_setspecific, функция, 490
определение, 489, 943
pthread_sigmask, функция, 497, 519, 892
определение, 494, 943
PTHREAD_STACK_MAX, константа, 466
PTHREAD_STACK_MIN, константа, 466
pthread_t, тип, 433
pthread_testcancel, определение функции, 493, 943
P_tmpdir, константа, 209
ptrdiff_t, тип данных, 94
ptsname, функция, 789, 791, 796
определение, 789, 794, 797, 943
pty, программа, 781, 802, 808–811, 814, 998
pty_fork, определение функции, 799, 800, 958
pty_fork, функция, 803
ptym_open, функция, 800
определение, 790, 791, 795, 798, 958
ptys_open, функция, 801
определение, 790, 791, 796, 798, 958
putc, функция, 37, 767
определение, 192, 943
putchar, функция, 600, 601
определение, 192, 943
putchar_unlocked, определение функции, 483, 944

putc_unlocked, определение функции, 483, 944
putenv, функция, 250
    определение, 251, 944
putmsg, функция, 548, 646
    определение, 548, 944
putpmsg, функция, 548, 646
    определение, 548, 944
puts, функция, 973
    определение, 193, 944
PWD, переменная окружения, 250
pwd, программа, 41
pwrite, функция, 114
    определение, 114, 944

Q

queue, структура, 983
quick-fit, алгоритм распределения памяти, 247
QUIT, служебный символ, 743, 746

R

raise, функция, 383
    определение, 383, 944
raw mode, прозрачный режим ввода, 734
read, функция, 35, 36, 105, 389, 390, 426, 525, 542, 575, 590, 603, 646, 688, 718, 720, 728, 730, 775, 805, 842, 844, 845, 881, 886, 895, 900, 913, 993
    определение, 105, 944
readdir, функция, 34, 763, 900
    определение, 167, 944
readlink, функция, 160
    определение, 161, 944
read_lock, функция, 542
    определение, 532, 958
readmore, функция, 917, 918
    определение, 891, 915
readn, функция, 574, 811, 887
    определение, 574, 957
readv, функция, 75, 76, 571, 646, 843
    определение, 571, 945
readw_lock, функция, 840, 857
    определение, 532, 958
realloc, функция, 85, 245, 725, 915, 972
    определение, 245, 945
recv, функция, 669
    определение, 666, 945
recv_fd, функция, 719, 724
    определение, 704, 707, 711, 957
recvfrom, функция, 676, 678
    определение, 667, 945
recvmsg, функция, 711, 715
    определение, 668, 945
recv_ufd, определение функции, 715
RE_DUP_MAX, константа, 77
regcomp, функция, 73
regexec, функция, 73
remove, функция, 153
    определение, 156, 945
remove_job, функция, 910
    определение, 891, 899
rename, функция, 153
    определение, 156, 945
replace_job, функция, 914
    определение, 891, 898
REPRINT, служебный символ, 743, 746
request, функция, 728, 730
    определение, 721, 731
requesting-user-name, атрибут, 868
reset, программа, 997
restrict, спецификатор, 54
rewind, определение функции, 198, 945
rewinddir, определение функции, 167, 945
RLIM_INFINITY, константа, 260, 967
rlimit, структура, 260, 967
RLIMIT_AS, константа, 260
RLIMIT_CORE, константа, 260
RLIMIT_CPU, константа, 260
RLIMIT_DATA, константа, 260
RLIMIT_FSIZE, константа, 260
RLIMIT_LOCKS, константа, 260
RLIMIT_MEMLOCK, константа, 260
RLIMIT_NOFILE, константа, 260, 967
RLIMIT_NPROC, константа, 260
RLIMIT_RSS, константа, 261
RLIMIT_SBSIZE, константа, 261
RLIMIT_STACK, константа, 261
RLIMIT_VMEM, константа, 261
rlim_t, тип данных, 94
rlogind, программа, 998
rm, программа, 888
rmdir, функция, 165
    определение, 166, 945
RMSGD, константа, 556
RMSGN, константа, 556
RNORM, константа, 556
R_OK, константа, 139, 921
root, имя суперпользователя, 44
rpciod, программа, 506

RPROTDAT, константа, 557
RPROTDIS, константа, 557
RPROTNORM, константа, 557
RS_HIPRI, константа, 931

S

SA_INTERRUPT, константа, 398
SA_NOCLDSTOP, константа, 398
SA_NOCLDWAIT, константа, 398
SA_NODEFER, константа, 398
SA_ONSTACK, константа, 399
SA_RESETHAND, константа, 399
SA_RESTART, константа, 375, 399
SA_SIGINFO, константа, 399
S_BANDURG, константа, 570
sbrk, системный вызов, 50
sbrk, функция, 246
scan_configfile, функция, 880
scanf, определение функции, 203, 946
_SC_ARG_MAX, константа, 76, 951
_SC_ATEXIT_MAX, константа, 76, 951
_SC_CHILD_MAX, константа, 76, 951
_SC_CLK_TCK, константа, 76, 321, 951
_SC_COLL_WEIGHTS_MAX, константа, 76, 951
SCHAR_MAX, константа, 70, 71
SCHAR_MIN, константа, 70, 71
_SC_HOST_NAME_MAX, константа, 76, 672, 892, 951
_SC_IOV_MAX, константа, 76, 951
_SC_JOB_CONTROL, константа, 89, 951
_SC_LINE_MAX, константа, 76, 951
_SC_LOGIN_NAME_MAX, константа, 76, 951
SCM_RIGHTS, константа, 708
_SC_NGROUPS_MAX, константа, 76, 951
_SC_OPEN_MAX, константа, 77, 951
_SC_PAGE_SIZE, константа, 77, 578, 951
_SC_PAGESIZE, константа, 77, 578, 951
_SC_READER_WRITER_LOCKS, константа, 89, 951
_SC_RE_DUP_MAX, константа, 77, 951
script, программа, 781, 784
_SC_SAVED_IDS, константа, 89, 299, 951
_SC_SHELL, константа, 89, 951
_SC_STREAM_MAX, константа, 77, 951
_SC_SYMLOOP_MAX, константа, 77, 951
_SC_THREAD_DESTRUCTOR_ITERATIONS, константа, 466
_SC_THREAD_KEYS_MAX, константа, 466
_SC_THREAD_PROCESS_SHARED, константа, 472
_SC_THREADS, константа, 432
_SC_THREAD_SAFE_FUNCTIONS, константа, 481
_SC_THREAD_STACK_MAX, константа, 466
_SC_THREAD_STACK_MIN, константа, 466
_SC_TTY_NAME_MAX, константа, 77, 951
_SC_TZNAME_MAX, константа, 77, 951
_SC_V6_ILP32_OFF32, константа, 105
_SC_V6_ILP32_OFFBIG, константа, 105
_SC_V6_LP64_OFF64, константа, 105
_SC_V6_LP64_OFFBIG, константа, 105
_SC_VERSION, константа, 89, 951
_SC_XOPEN_CRYPT, константа, 89, 951
_SC_XOPEN_LEGACY, константа, 89, 951
_SC_XOPEN_REALTIME, константа, 89, 951
_SC_XOPEN_REALTIME_THREADS, константа, 89, 951
_SC_XOPEN_VERSION, константа, 89, 951
SEEK_CUR, константа, 101, 843, 928, 935
seekdir, определение функции, 167, 946
SEEK_END, константа, 101, 848, 858, 928, 935
SEEK_SET, константа, 101, 843, 849, 895, 928, 935, 987
SEGV_ACCERR, константа, 400
SEGV_MAPPER, константа, 400
select, функция, 561, 646, 727, 881, 894, 984, 985, 993, 996
  определение, 561, 946
sembuf, структура, 625
semctl, определение функции, 624, 946
semget, определение функции, 623, 946
semid_ds, структура, 622
semop, определение функции, 625, 946
semun, объединение, 624

SEM_UNDO, константа, 627
send, функция, 671
  определение, 664, 946
send_err, функция, 721, 731
  определение, 704, 705, 957
send_fd, функция, 705, 721, 731
  определение, 704, 706, 709, 713, 957
sendmsg, функция, 710, 714
  определение, 666, 946
sendto, функция, 676, 678
  определение, 665, 946
S_ERROR, константа, 570
serv_accept, функция, 727, 729
  определение, 693, 694, 700, 957
serve, функция, 672, 679
  определение, 673, 678
servent, структура, 654
serv_listen, функция, 727, 729
  определение, 693, 698, 957
session, структура, 352
SETALL, константа, 625, 946
setasync, определение функции, 996
setbuf, функция, 322, 767
  определение, 185, 947
setegid, функция, 303
  определение, 303, 947
setenv, функция, 250
  определение, 251, 947
seteuid, функция, 303
  определение, 303, 947
set_fl, функция, 525, 542, 991
  определение, 957
setgid, функция, 328
  определение, 298, 947
setgrent, определение функции, 220, 947
set-group-ID, флаг, 134, 313
setgroups, определение функции, 221, 947
sethostent, определение функции, 652, 947
setitimer, функция, 990
setjmp, функция, 255, 257, 387, 390, 981
  определение, 254, 947
setjump, функция, 252
setlogmask, определение функции, 511, 947
setnetent, определение функции, 652, 947
set_noecho, функция, 803
  определение, 804
setpgid, определение функции, 335, 947
setprotoent, определение функции, 653, 947
setpwent, функция, 217
  определение, 216, 947
setregid, функция, 302
  определение, 302, 947
setreuid, функция, 302
  определение, 302, 948
setrlimit, функция, 88, 259
  определение, 259, 948
setservent, определение функции, 653, 948
setsid, функция, 508, 800
  определение, 336, 948
setsockopt, функция, 681, 715
  определение, 679, 948
setspent, определение функции, 219, 948
settimeofday, функция, 227
setuid, функция, 328, 893
  определение, 298, 948
set-user-ID, флаг, 134, 312, 313
SETVAL, константа, 625, 946
setvbuf, функция, 259, 992
  определение, 185, 948
S_HANGUP, константа, 570
SHELL, переменная окружения, 250, 328
S_HIPRI, константа, 570
shmat, функция, 632
  определение, 631, 948
shmctl, функция, 632
  определение, 630, 948
shmdt, определение функции, 632, 948
shmget, функция, 632
  определение, 629, 948
shmid_ds, структура, 628
SHMLBA, константа, 631
SHM_LOCK, константа, 630, 948
SHM_RDONLY, константа, 631, 948
SHM_RND, константа, 631, 948
SHM_UNLOCK, константа, 631, 948
SHRT_MAX, константа, 70
SHRT_MIN, константа, 70
shutdown, определение функции, 646, 949
SHUT_RD, константа, 646, 949
SHUT_RDWR, константа, 646, 949
SHUT_WR, константа, 646, 949
SI_ASYNCIO, константа, 401
sig2str, определение функции, 428, 949

SIGABRT, сигнал, 277, 281, 362, 414, 981
sigaction, структура, 397
sigaction, функция, 375, 397, 402, 403, 416, 419, 424, 509, 519, 521, 676, 892, 996
  определение, 397, 949
sigaddset, функция, 395, 408, 410, 419, 424, 427, 497, 521, 767, 892, 989
  определение, 391, 392, 949
SIGALRM, сигнал, 359, 362, 378, 385
sig_atomic_t, тип данных, 94
SIG_BLOCK, константа, 393, 892, 949
SIGBUS, сигнал, 359, 362, 579
SIGCANCEL, сигнал, 359, 362
SIGCHLD, сигнал, 279, 359, 362, 379, 980, 996
SIGCLD, сигнал, 379
SIGCONT, сигнал, 342, 359, 363, 384
sigdelset, функция, 416, 424, 989
  определение, 391, 392, 949
SIG_DFL, константа, 368
sigemptyset, функция, 395, 408, 410, 419, 424, 427, 497, 509, 519, 521, 767, 892, 989
  определение, 391, 949
SIGEMT, сигнал, 359, 363
sigfillset, функция, 416, 519, 989
  определение, 391, 949
SIGFPE, сигнал, 281, 359, 363
SIGFREEZE, сигнал, 359, 363
SIGHUP, сигнал, 350, 359, 363, 892, 907
SIG_IGN, константа, 368
SIGILL, сигнал, 359, 363
SIGINFO, сигнал, 359, 364
siginfo, структура, 400
SIGINT, сигнал, 47, 359, 364
SIGIO, сигнал, 359, 364, 560, 684
SIGIOT, сигнал, 359, 364
sigismember, функция, 394, 395, 989
  определение, 391, 393, 949
SIGKILL, сигнал, 359, 364, 807
siglist[], массив имен сигналов, 427
siglongjmp, функция, 403, 405
  определение, 404, 949
SIGLWP, сигнал, 359, 364
signal, функция, 47, 367, 369, 372, 375, 378, 381, 386, 387, 388, 389, 390, 395, 404, 408, 410, 418, 426, 427, 602, 688, 775, 996
  определение, 367, 402, 949
signal_intr, функция, 805
  определение, 403, 957
signal_thread, определение функции, 891, 907
sigpending, функция, 394, 395
  определение, 395, 949
SIGPIPE, сигнал, 360, 364, 892, 993
SIGPOLL, сигнал, 360, 364, 560
sigprocmask, функция, 393–395, 409, 410, 416, 419, 424, 427, 767
  определение, 393, 949
SIGPROF, сигнал, 360, 365
SIGPWR, сигнал, 360, 365
SIGQUIT, сигнал, 360, 365
SIGSEGV, сигнал, 360, 365
sigset, функция, 375
sigsetjmp, функция, 403, 405
  определение, 404, 950
SIG_SETMASK, константа, 393, 949
sigset_t, тип данных, 94
SIGSTKFLT, сигнал, 360, 365
SIGSTOP, сигнал, 360, 365
sigsuspend, функция, 407, 409, 410, 424
  определение, 408, 950
SIGSYS, сигнал, 360, 365
SIGTERM, сигнал, 360, 365, 805, 892, 907, 998
SIGTHAW, сигнал, 360, 366
SIGTRAP, сигнал, 360, 366
SIGTSTP, сигнал, 350, 360, 366, 426, 807
SIGTTIN, сигнал, 342, 360, 366
SIGTTOU, сигнал, 343, 360, 366
SIG_UNBLOCK, константа, 393, 949
SIGURG, сигнал, 361, 366, 682
SIGUSR1, сигнал, 361, 366
SIGUSR2, сигнал, 361, 366
sigvec, функция, 375
SIGVTALRM, сигнал, 361, 367
sigwait, функция, 496, 518, 907
  определение, 494, 950
SIGWAITING, сигнал, 361, 367
SIGWINCH, сигнал, 361, 367, 777, 815
SIGXCPU, сигнал, 361, 367
SIGXFSZ, сигнал, 361, 367, 981
SIGXRES, сигнал, 361, 367
SI_MESGQ, константа, 401
Single UNIX Specification, стандарт, 63
S_INPUT, константа, 570
SIOCSPGRP, константа, 684
SI_QUEUE, константа, 401
S_IRGRP, константа, 136, 143

S_IROTH, константа, 136, 143
S_IRUSR, константа, 136, 143, 795, 895
S_IRWXG, константа, 143
S_IRWXO, константа, 143
S_IRWXU, константа, 143
S_ISBLK(), макроопределение, 131
S_ISCHR(), макроопределение, 131, 132
S_ISDIR(), макроопределение, 131, 132
S_ISFIFO(), макроопределение, 131
S_ISGID, константа, 143
S_ISLNK(), макроопределение, 131
S_ISREG(), макроопределение, 131, 132
S_ISREG, функция, 884
S_ISSOCK, макроопределение, 131
S_ISSOCK, функция, 700
S_ISUID, константа, 143
S_ISVTX, константа, 143, 923, 925, 935
SI_TIMER, константа, 401
SI_USER, константа, 401
S_IWGRP, константа, 136, 143, 795
S_IWOTH, константа, 136, 143
S_IWUSR, константа, 136, 143, 795, 895
S_IXGRP, константа, 136, 143
S_IXOTH, константа, 136, 143
S_IXUSR, константа, 136, 143
size_t, тип данных, 94
sleep, функция, 269, 284, 381, 395, 422, 914, 980, 981, 984, 987
  определение, 422, 950
sleep_us, определение функции, 957, 990
S_MSG, константа, 570
SNDPIPE, константа, 554
SNDZERO, константа, 554
snprintf, функция, 797, 962, 964
  определение, 200, 950
SO_ACCEPTCONN, константа, 680
SO_BROADCAST, константа, 680
sockaddr, структура (Linux), 649
sockaddr, структура (FreeBSD), 649
sockaddr_in, структура, 649
sockaddr_in, структура (Linux), 650
sockaddr_in6, структура, 650
sockaddr_un, структура, 696, 697
sockatmark, определение функции, 683, 950
SOCK_DGRAM, константа, 644, 950
socket, функция, 663, 676, 681, 697, 699, 701, 884, 911
  определение, 643, 950
socketpair, функция, 696
  определение, 695, 950
SOCK_RAW, константа, 644
SOCK_SEQPACKET, константа, 644, 950
SOCK_STREAM, константа, 644, 878, 950
SO_DEBUG, константа, 680
SO_DONTROUTE, константа, 680
SO_ERROR, константа, 680
SO_KEEPALIVE, константа, 680
Solaris, операционная система, 68
SO_LINGER, константа, 680
SOL_SOCKET, константа, 708
SOMAXCONN, константа, 662
SO_OOBINLINE, константа, 680
SO_RCVBUF, константа, 680
SO_RCVLOWAT, константа, 680
SO_RCVTIMEO, константа, 680
SO_REUSEADDR, константа, 680
SO_SNDBUF, константа, 680
SO_SNDTIMEO, константа, 680
SO_TYPE, константа, 680
S_OUTPUT, константа, 570
s_pipe, функция, 688, 719, 813
  определение, 690, 696, 957
sprintf, функция, 516, 602, 671, 678, 702, 719, 721, 724, 731, 836, 849, 850, 895, 896, 900, 902–904, 910, 912–914
  определение, 200, 950
spwd, структура, 974
S_RDBAND, константа, 570
S_RDNORM, константа, 570
sscanf, функция, 602, 879
  определение, 203, 950
ssize_t, тип данных, 94
stackaddr, атрибут потока, 467
stacksize, атрибут потока, 467
START, служебный символ, 743, 746, 747
stat, структура, 130
stat, функция, 30, 129, 700, 763, 970
  определение, 129, 951
STATCLASS_CLIERR, функция, 873
STATCLASS_INFO, функция, 873
STATCLASS_OK, функция, 873, 919
STATCLASS_REDIR, функция, 873
STATCLASS_SRVERR, функция, 873
STAT_CLI_ACCERR, константа, 873
STAT_CLI_ATTRCON, константа, 873
STAT_CLI_BADFMT, константа, 873
STAT_CLI_BADREQ, константа, 873

STAT_CLI_COMPERR, константа, 873
STAT_CLI_FMTERR, константа, 873
STAT_CLI_FORBID, константа, 873
STAT_CLI_NOAUTH, константа, 873
STAT_CLI_NOCHAR, константа, 873
STAT_CLI_NOCOMP, константа, 873
STAT_CLI_NOPERM, константа, 873
STAT_CLI_NOSCHM, константа, 873
STAT_CLI_NOTFND, константа, 873
STAT_CLI_NOTPOS, константа, 873
STAT_CLI_NOTSUP, константа, 873
STAT_CLI_OBJGONE, константа, 873
STAT_CLI_TIMOUT, константа, 873
STAT_CLI_TOOBIG, константа, 873
STAT_CLI_TOOLNG, константа, 873
st_atime, поле структуры stat, 161
STAT_OK, константа, 873
STAT_OK_ATTRCON, константа, 873
STAT_OK_ATTRIGN, константа, 873
STAT_SRV_BADVER, константа, 874
STAT_SRV_CANCEL, константа, 874
STAT_SRV_DEVERR, константа, 874
STAT_SRV_INTERN, константа, 873
STAT_SRV_NOMULTI, константа, 874
STAT_SRV_NOTSUP, константа, 873
STAT_SRV_REJECT, константа, 874
STAT_SRV_TMPERR, константа, 874
STAT_SRV_TOOBUSY, константа, 874
STAT_SRV_UNAVAIL, константа, 873
STATUS, служебный символ, 743, 747
st_ctime, поле структуры stat, 161
stderr, стандартный поток вывода сообщений об ошибках, 183
STDERR_FILENO, константа, 97, 183
stdin, константа, 37
stdin, стандартный поток ввода, 183
STDIN_FILENO, константа, 36, 97, 183, 813
stdout, константа, 37, 962
stdout, стандартный поток вывода, 183
STDOUT_FILENO, константа, 36, 97, 183, 813, 977
sticky bit, бит закрепления в памяти, 146
st_mtime, поле структуры stat, 161
STOP, служебный символ, 743, 747
str2sig, определение функции, 428, 951
strbuf, структура, 547
strcat, функция, 763, 835, 837, 962, 964
strchr, функция, 844
strcmp, функция, 217, 723, 762, 841, 900
strcpy, функция, 49, 697, 723, 794, 834, 857, 885, 886, 908, 913
STREAM_MAX, константа, 77
STREAMS, 544
strerror, функция, 521, 670, 721, 900, 902–904, 907, 911, 962, 964, 965
  определение, 42, 951
strftime, функция, 228, 975
  определение, 230, 951
strip, программа, 976
strlen, функция, 39, 47, 270, 688, 705, 718, 719, 724, 796, 837, 849, 851, 855, 908, 977
str_list, структура, 552
str_mlist, структура, 552
strncasecmp, функция, 917
strncat, функция, 886
strncmp, функция, 795, 903
strncpy, функция, 763, 794, 795, 798, 800, 902, 903, 904
strrchr, функция, 518, 520, 592
strrecvfd, структура, 706
strsignal, функция, 428, 951
strtok, функция, 722
stty, команда, 757
stty, программа, 342, 998
submit_file, функция, 884
  определение, 883, 885
SunOS, реализация ОС UNIX, 65
SUSP, служебный символ, 743, 747
swapper, системный процесс с идентификатором 0, 267
S_WRBAND, константа, 570
S_WRNORM, константа, 570
symlink, функция, 160
  определение, 160, 951
SYMLINK_MAX, константа, 77
SYMLOOP_MAX, константа, 77
sync, функция, 117
  определение, 117, 951
sysconf, функция, 48, 69, 74, 82, 671, 674, 678, 892, 967
  определение, 76, 951
syslog, функция, 509, 516, 518, 520, 670, 671, 673, 678, 679, 962
  определение, 511, 951
syslogd, программа, 506
system, функция, 308, 417, 418, 980, 993
  определение, 309, 419, 951

T

TABDLY, константа, 740, 756
TAG_BOOLEAN, константа, 875
TAG_CHARSET, константа, 875
TAG_DATETIME, константа, 875
TAG_END_OF_ATTR, константа, 874, 912
TAG_ENUM, константа, 875
TAG_INTEGER, константа, 875
TAG_INTRANGE, константа, 875
TAG_JOB_ATTR, константа, 874
TAG_MIMETYPE, константа, 875, 912
TAG_NAMEWLANG, константа, 875
TAG_NAMEWOLANG, константа, 875
TAG_NATULANG, константа, 875, 912
TAG_NONE, константа, 875
TAG_OCTSTR, константа, 875
TAG_OPERATION_ATTR, константа, 874, 912
TAG_PRINTER_ATTR, константа, 874
TAG_RESOLUTION, константа, 875
TAG_TEXTWLANG, константа, 875
TAG_TEXTWOLANG, константа, 875
TAG_UNKNOWN, константа, 875
TAG_UNSUPP_ATTR, константа, 874
TAG_UNSUPPORTED, константа, 875
TAG_URI, константа, 875, 912
TAG_URISCHEME, константа, 875
tar, программа, 971
tcdrain, функция, 741, 759
  определение, 759, 952
tcflow, функция, 741
  определение, 759, 952
tcflush, функция, 735, 741
  определение, 759, 952
tcgetattr, функция, 741, 744, 749, 761, 767, 771, 803, 804
  определение, 748, 952
tcgetpgrp, функция, 339, 741
  определение, 339, 952
tcgetsid, функция, 339, 741
  определение, 340, 952
TCIFLUSH, константа, 759, 952
TCIOFF, константа, 759, 952
TCIOFLUSH, константа, 759, 952
TCION, константа, 759, 952
TCOFLUSH, константа, 759, 952
TCOOFF, константа, 759, 952
TCOON, константа, 759, 952
TCP/IP, семейство протоколов, 867
TCSADRAIN, константа, 748, 952
TCSAFLUSH, константа, 748, 952
TCSANOW, константа, 748, 952
tcsendbreak, функция, 741
  определение, 759, 952
tcsetattr, функция, 735, 741, 744, 750, 767, 772, 801, 804
  определение, 748, 952
tcsetpgrp, функция, 339, 741
  определение, 339, 952
TELL_CHILD, функция, 534, 542, 634, 979
  определение, 412, 593, 958
telldir, определение функции, 167, 952
TELL_PARENT, функция, 534, 635, 979, 990
  определение, 412, 593, 958
TELL_WAIT, функция, 534, 634, 990
  определение, 412, 593, 958
telnet, программа, 331
telnetd, программа, 980, 998
tempnam, определение функции, 209, 952
TENEX C shell, командная оболочка, 29
TERM, переменная окружения, 250
terminal line discipline, дисциплина обслуживания терминала, 735
terminfo, 779
termios, структура, 735, 799, 957, 998
time, программа, 49
time, функция, 405, 975, 985
  определение, 227, 952
times, функция, 322
  определение, 321, 953
timespec, структура, 462
time_t, тип данных, 94
timeval, структура, 228, 562, 881, 985, 990
TIOCGPTN, константа, 797
TIOCGWINSZ, константа, 956
TIOCPKT, константа, 814
TIOCREMOTE, константа, 814
TIOCSCTTY, константа, 338, 801
TIOCSIG, константа, 815
TIOCSIGNAL, константа, 815
TIOCWINSZ, константа, 815
tm, структура, 228, 975
TMPDIR, переменная окружения, 209, 250
tmpfile, функция, 207
  определение, 207, 953

TMP_MAX, константа, 72
tmpnam, определение функции, 207, 953
tms, структура, 321
tolower, функция, 600
TOSTOP, константа, 739, 756
touch, команда, 164
TRAP_BRKPT, константа, 400
TRAP_TRACE, константа, 401
tread, функция, 882, 902, 915, 916
  определение, 877, 881
treadn, функция, 901
  определение, 877, 882
truncate, функция, 150
  определение, 953
ttcompat, модуль STREAMS, 783
tty, структура, 353
tty_atexit, определение функции, 774, 957
tty_cbreak, функция, 776
  определение, 771, 957
ttymon, программа, 330
ttyname, программа, 999
ttyname, функция, 765
  определение, 761, 764, 953
TTY_NAME_MAX, константа, 77
tty_raw, функция, 775, 803
  определение, 772, 957
tty_reset, функция, 774, 775, 776
  определение, 774, 957
tty_termios, определение функции, 774, 957
TZ, переменная окружения, 250
TZNAME_MAX, константа, 77

U

UCHAR_MAX, константа, 70, 71
ucred, структура, 712
uid_t, тип данных, 94
UINT_MAX, константа, 70
ulimit, встроенная команда, 261
ulimit, программа, 87
ULLONG_MAX, константа, 71
ULONG_MAX, константа, 71
umask, функция, 140, 508
  определение, 141, 953
uname, определение функции, 225, 953
ungetc, функция, 191
  определение, 191, 953
UNIX System Release 3.2, реализация ОС UNIX, 65
UNIX System V Release 4, 65
UnixWare, операционная система, 68
unlink, функция, 153, 693, 699, 700, 702, 900, 903, 904, 914, 970, 973
  определение, 154, 953
un_lock, функция, 837, 839, 845, 847, 848, 854, 855, 857
  определение, 532, 958
unlockpt, функция, 789, 791, 796, 798
  определение, 788, 795, 797, 953
unsetenv, функция, 250
  определение, 251, 953
unslept, функция, 388
update_jobno, функция, 910
  определение, 891, 896
USER, переменная окружения, 328
USHRT_MAX, константа, 71
usleep, функция, 990
utime, функция, 162, 970
  определение, 162, 953
utmp, структура, 224
utmp, файл, 225, 806, 980, 986
utsname, структура, 225

V

va_arg, функция, 835
va_end, функция, 835, 960
varargs, функция, 959
va_start, функция, 835, 960
VDISCARD, константа, 742
VDSUSP, константа, 742
VEOF, константа, 742, 744
VEOL, константа, 742
VEOL2, константа, 742
VERASE, константа, 742
VERASE2, константа, 743
vfork, функция, 274, 275, 977
vfprintf, определение функции, 202, 954
vfscanf, определение функции, 204, 954
vi, программа, 998
VINTR, константа, 743, 744
vipw, функция, 216
VKILL, константа, 743
VLNEXT, константа, 743
vmalloc, библиотека функций распределения памяти, 247
VMIN, константа, 769
v-node, 967
volatile, спецификатор, 256
vprintf, функция, 965

определение, 202, 954
VQUIT, константа, 743
VREPRINT, константа, 743
vscanf, определение функции, 204, 954
vsnprintf, функция, 962, 964
определение, 202, 954
vsprintf, определение функции, 202, 954
vsscanf, определение функции, 204, 954
VSTART, константа, 743
VSTATUS, константа, 743
VSTOP, константа, 743
VSUSP, константа, 743
vsyslog, определение функции, 514, 954
VTDLY, константа, 740, 756
VTIME, константа, 769
VWERASE, константа, 743

W

wait, функция, 277, 279, 286, 992
определение, 279, 955
wait3, функция, 286
определение, 287, 955
wait4, функция, 286
определение, 287, 955
WAIT_CHILD, функция, 534, 635, 979, 990
определение, 413, 593, 958
waitid, функция, 285, 286
определение, 285, 955
WAIT_PARENT, функция, 534, 542, 635, 979
определение, 412, 593, 958
waitpid, функция, 38, 39, 47, 277, 279, 285, 286, 296, 305, 542, 591, 598, 673, 992, 993
определение, 279, 955
wall, программа, 792
wchar_t, тип данных, 94
WCONTINUED, константа, 283, 286, 955
WCOREDUMP, макроопределение, 280, 281
WERASE, служебный символ, 743, 747
WEXITED, константа, 286, 955
WEXITSTATUS, макроопределение, 280, 281
who, программа, 40
WIFCONTINUED, макроопределение, 280
WIFEXITED, макроопределение, 280, 281
WIFSIGNALED, макроопределение, 280, 281
WIFSTOPPED, макроопределение, 281
winsize, структура, 354, 777, 799, 998
WNOHANG, константа, 283, 286, 955
WNOWAIT, константа, 286, 955
W_OK, константа, 139, 921
worker_thread, структура, 889
write, программа, 792
write, функция, 35, 36, 106, 389, 390, 426, 516, 525, 534, 542, 575, 581, 590, 602, 603, 646, 669, 676, 706, 718, 837, 850, 896, 903, 904, 913, 968, 977, 981, 991
определение, 106, 955
write_lock, функция, 542, 895
определение, 532, 958
writen, функция, 574, 705, 805, 811, 886, 902–905
определение, 574, 957
writev, функция, 75, 76, 571, 646, 719, 724, 848, 850, 913
определение, 571, 955
writew, функция, 840
writew_lock, функция, 534, 836, 846, 848, 849, 854, 865
определение, 532, 958
WSTOPPED, константа, 286, 955
WSTOPSIG, макроопределение, 280
WTERMSIG, макроопределение, 280, 281
wtmp, файл, 225, 980
WUNTRACED, константа, 283, 955

X

X/Open System Interface, набор системных интерфейсов, 63
xargs, команда, 294
XCASE, константа, 739, 757
Xenix, реализация ОС UNIX, 65
xinetd, программа, 506
X_OK, константа, 139, 921
_XOPEN_CRYPT, константа, 89
_XOPEN_IOV_MAX, константа, 75
_XOPEN_LEGACY, константа, 63, 89
_XOPEN_NAME_MAX, константа, 75
_XOPEN_PATH_MAX, константа, 75
_XOPEN_REALTIME, константа, 89

_XOPEN_REALTIME_THREADS, константа, 89
_XOPEN_STREAMS, константа, 62
_XOPEN_UNIX, константа, 62
_XOPEN_VERSION, константа, 89
XSI IPC, 609

А

абсолютный путь, 31
адресация, 647
альтернативы стандартной библиотеке ввода-вывода, 211
анонимные области отображаемой памяти, 635
анонимный пользователь FTP, 970
аргументы командной строки, 240
асинхронный ввод-вывод, 569, 683
атомарные операции, 113
атрибуты
  блокировок чтения-записи, 479
  мьютексов, 472
  потока, 466
  синхронизации, 472
  файла, 30

Б

библиотеки функций, 27, 49
биты прав доступа к файлу, 969
блокировка записей, 527
блокировка чтения-записи, 456
блокировки в конце файла, 537

В

ввод-вывод двоичных данных, 196
виртуальный узел, 967
восстановление после ошибок, 43
временные характеристики процесса, 320
время работы процесса, 48
вход с терминала, 325

Г

группы процессов, 333
  переднего плана, 337
  фоновых процессов, 337

Д

двоичные деревья, 819
двоичный семафор, 621
демон, 320, 504
дескрипторы сокетов, 643
дескрипторы файлов, 35, 96
децентрализованный доступ, 828
динамическое хеширование, 819
диспозиция сигнала, 358
дистанционный режим, 814
домашний каталог, 34
домен UNIX, 643
домен Интернета IPv4, 643
домен Интернета IPv6, 643
дополнительные группы, 46, 220
дочерний процесс, 40, 268
дырки в файлах, 148

Ж

жесткие ссылки, 152

З

завершение работы процесса, 235
заголовок HTTP, 870
заголовок IPP, 869
запись в поток, 189
запись со слиянием, 571
зомби, 278

И

идентификатор
  группы, 45
  пользователя, 44, 320
  потока, 41, 432, 433
  процесса, 38, 974
  сессии, 336
имена сигналов, 427
именованные каналы STREAMS, 690
именованные сокеты домена UNIX, 696
имя файла, 31
индексный узел, 965
индексный файл, 823
интерпретация файлов, 304
исходные тексты, 956

К

календарное время, 48, 227
канал FIFO, 993
каналы STREAMS, 686
канонический режим, 766

каталог, 30
код завершения, 236
командная оболочка, 29
командный интерпретатор, 28
концепция сигналов, 356
копирование при записи, 268

Л

лидер сессии, 336
линейное хеширование, 819
локальные данные потоков, 485

М

макроопределения контроля функциональных особенностей, 92
маркер срочности, 683
маска режима создания файла, 140
маска сигналов, 383
медленные системные вызовы, 373
межпроцессное взаимодействие, 585
модель клиент-сервер, 522
мультиплексирование ввода-вывода, 558
мьютекс, 448

Н

наборы дескрипторов, 562
наборы сигналов, 391
надежные сигналы, 356, 382
неблокирующий ввод-вывод, 523
небуферизованный ввод-вывод, 35, 96
неименованные каналы, 586
неканонический режим, 769, 992
ненадежные сигналы, 371
необязательные конфигурационные параметры, 88
неопределенный домен, 643

О

обработка ошибок, 41
обработчик сигнала, 368
обработчики выхода, 237
объявление, 997
одновременный доступ, 828
операционная система, 28
опрос, 288, 526
ориентация потока, 182
осиротевшая группа процессов, 349, 350
отказ в обслуживании, 881
относительный путь, 31
очереди сообщений, 615
очередь печати, 870

П

пакетный режим, 814
передача дескрипторов файлов, 703
переменные окружения, 248
переменные состояния, 460
перехватчик сигнала, 368
позиционирование в потоке, 198
полная буферизация, 183
порядок байтов, 647
посимвольный ввод-вывод, 189
построчная буферизация, 184
построчный ввод-вывод, 189
поток управления, 431
потоки, 41, 181
  и сигналы, 494
права доступа к файлу, 135
правила наследования блокировок, 535
правила программирования демонов, 506
пределы, 69
  времени выполнения, 69
  времени компиляции, 69
  для потоков, 465
представление времени, 48
прерванные системные вызовы, 373
принудительные блокировки, 539
прототипы функций, 921
процесс, 38
процессорное время, 48
прямой ввод-вывод, 189
псевдотерминалы, 331, 781
  в BSD, 793
  в Linux, 797
  на основе STREAMS, 790
пустой сигнал, 357
путь к файлу, 31

Р

рабочий каталог, 34
разделяемая память, 628
разделяемые библиотеки, 243, 976
размер окна терминала, 776
размер файла, 148
распределение памяти, 244
расширенные возможности IPC, 686

расширенные операции ввода-вывода, 523
расширяемое хеширование, 819
реальный идентификатор процесса, 981
реентерабельность, 480
реентерабельные функции, 376
рекомендательные блокировки, 539
решения, 965
родительский процесс, 40

С

сегмент разделяемой памяти, 994
семантика сигнала SIGCLD, 379
семафоры, 621
сервер FTP, 970
сессия, 335
сигналы, 46
сигналы управления заданиями, 424
символические ссылки, 152, 157, 968
синхронизация потоков, 445
системные вызовы, 27, 49
смонтированные потоки, 692
сокеты, 643
  домена UNIX, 695
сообщения STREAMS, 546
сопроцессы, 601
специальные файлы устройств, 175
список переменных окружения, 240
среда окружения процесса, 234
стандартный
  ввод, 35
  вывод, 35
  вывод сообщений об ошибках, 35
суперпользователь, 44
сценарий командной оболочки, 29

Т

теневой файл паролей, 218, 974
теневые пароли, 217
типы файлов, 130

У

указатель на файл, 183
указатель на цепочку, 823
уникальные соединения, 691, 698
управление заданиями, 340
управление потоками, 465
управляющие символы, 41
управляющий процесс, 337
управляющий терминал, 337
усечение файлов, 149
условие гонки за ресурсами, 287
устройство клонирования, 790
учет использования ресурсов, 313

Ф

файл групп, 219
файл паролей, 213
файловые системы, 30, 150
форматированный ввод-вывод, 199, 203
форматы адресов, 649
функции ввода, 190
функции для работы с датой и временем, 227

Х

характеристики демонов, 504

Ц

централизованный доступ, 827

Ч

чтение вразброс, 571
чтение из потока, 189
чтение каталогов, 167

Э

экстренные данные, 682
элементарные системные типы данных, 93
эффективность функций стандартного ввода-вывода, 193

Я

ядро, 27

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru - Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-089-8, название «UNIX. Профессиональное программирование», 2-е издание – покупка в Интернет-магазине «Books.Ru - Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.