HIGH TECH

QNX/UNIX

Анатомия параллелизма

Олег Цилюрик, Егор Горошко

Приложение
Организация обмена сообщениями

Владимир Зайцев

СИМВОЛ®

Санкт-Петербург — Москва
2006

Серия «High tech»

Олег Цилюрик, Егор Горошко

QNX/UNIX: анатомия параллелизма

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*А. Петухов*
Художник	*В. Гренда*
Корректор	*О. Макарова*
Верстка	*О. Макарова*

Цилюрик О., Горошко Е.
QNX/UNIX: анатомия параллелизма. – СПб.: Символ-Плюс, 2006. – 288 с., ил.
ISBN 5-93286-088-X

Книга адресована программистам, работающим в самых разнообразных ОС UNIX. Авторы предлагают шире взглянуть на возможности параллельной организации вычислительного процесса в традиционном программировании. Особый акцент делается на потоках (threads), а именно на тех возможностях и сложностях, которые были привнесены в технику параллельных вычислений этой относительно новой парадигмой программирования. На примерах реальных кодов показываются приемы и преимущества параллельной организации вычислительного процесса. Некоторые из результатов испытаний тестовых примеров будут большим сюрпризом даже для самых бывалых программистов. Тем не менее излагаемые техники вполне доступны и начинающим программистам: для изучения материала требуется базовое знание языка программирования C/C++ и некоторое понимание «устройства» современных многозадачных ОС UNIX.

В качестве «испытательной площадки» для тестовых фрагментов выбрана ОСРВ QNX, что позволило с единой точки зрения взглянуть как на специфические механизмы микроядерной архитектуры QNX, так и на универсальные механизмы POSIX. В этом качестве книга может быть интересна и тем, кто не использует (и не планирует никогда использовать) ОС QNX: программистам в Linux, FreeBSD, NetBSD, Solaris и других традиционных ОС UNIX.

ISBN 5-93286-088-X

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 28.11.2005. Формат 70x100 1/16 . Печать офсетная.
Объем 18 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Зачем написана эта книга и кому она предназначена? Различные аспекты построения программных приложений для операционной системы реального времени QNX, родственные тем, которые мы обсуждаем в данном издании, весьма обстоятельно описаны в литературе. Это и основополагающие труды Э. Дейкстры [10] и других авторов, и общая литература по POSIX (Portable Operating System Interface) и ОС UNIX [2, 3, 5–7]. Другие, сугубо специфические аспекты для ОС QNX, такие как обмен сообщениями микроядра, построение менеджеров ресурсов, пулы потоков и еще ряд других приятных вещей, прекрасно описаны в книге Р. Кертена [1].

Однако все эти источники имеют ряд недостатков:

- Техническая документация по QNX, тщательно описывающая API и детали реализации, оставляет в стороне (возможно, как относительно известные) общие вопросы построения параллельных приложений и их взаимодействия.
- Несмотря на наличие множества кратких примеров кода по использованию *отдельных* вызовов API, в технической документации явно недостаточно примеров их применения в логической последовательности и организации их совместного взаимодействия.
- Издания по UNIX, посвященные общим вопросам, напротив, изобилуют образцами кода, но в силу объективных причин, связанных с длительностью издательского процесса, не отображают новые механизмы, появившиеся в стандартах начиная с конца 90-х годов.
- И наконец, подавляющее большинство переводных книг по программированию для UNIX рассчитано на начальный уровень. Счастливое исключение – книги У. Стивенса, которые мы с удовольствием используем сами и рекомендуем читателям.

В итоге возник замысел написать книгу, в которой будет подведен итог некоторого периода нашего собственного использования ОС QNX и которая будет насыщена примерами в виде законченных проектов или отдельных фрагментов кода. Для удобства читателей `*.tgz`-архив описываемых в книге приложений размещен по адресу *http://www.symbol.ru/library/qnx-unix/pthread.tgz*, что позволит полноценно работать с текстом.

По большей части проекты трансформировались из реальных задач, из которых исключалась вся специфика конкретного заказа, только затуманивающая смысл примера. Многие примеры программного ко-

да и приложения подготовлены так, что несут двойную нагрузку: они построены как тесты тех или иных механизмов ОС и, иллюстрируя рассматриваемые аспекты, одновременно позволяют получить численные характеристики тестируемых параметров. Этим книга существенно отличается от технической документации и описательных статей по QNX, которые не слишком богаты численными показателями, подтверждающими то, о чем в них говорится.

Первоначально предполагалось создать достаточно компактный текст, систематизирующий механизмы, итак хорошо известные и понятные... Но по ходу работы объем материала, который необходимо было хотя бы затронуть, начал разрастаться как снежный ком: уточнение незначительного вопроса порождало два новых и часто гораздо более существенных, чем первоначальный; начали «вылезать» несоответствия документации и результатов тестирования; слабо связанные, на первый взгляд, механизмы (например, примитивы синхронизации POSIX и сигналы UNIX) при перекрестном взаимодействии порождают такие эффекты, которые просто нельзя оставить без внимания.

Для того чтобы хоть как-то бороться с лавинным нарастанием объема, было принято решение выделить те механизмы, программные техники и элементы API, которые наиболее слабо затронуты в литературных источниках, и рассматривать их с максимально возможной обстоятельностью (это относится, например, к базовой форме вызова `spawn()`). Напротив, те элементы, которые достаточно детально описаны и обсуждены или интуитивно понятны (например, все семейство вызовов [1], производных от `spawn()`), лишь поверхностно перечисляются (даже если позже в примерах кода мы и используем именно эти формы). В конце концов, мы не собирались пересказывать техническую документацию QNX, а хотели детально рассмотреть тонкие механизмы и их несоответствия (между собой или с изложением в документации) на работающих образцах кода.

При написании этого материала нам не были доступны никакие внутренние или специфические материалы разработчиков, кроме официальной технической документации ОС QNX, общедоступной литературы, информации, циркулирующей в Интернете в обсуждениях QNX-сообществ (в первую очередь *http://qnx.org.ru* и *http://qnxclub.net*), и контактов с коллегами-разработчиками. Многие тонкие детали приходилось восстанавливать путем тестирования и последующего толкования наблюдаемых результатов.

Как следствие, совершенно не исключено, что некоторые полученные нами результаты были неверно интерпретированы, а описания содержат определенные неточности – текущее состояние текста отображает наше *сегодняшнее* понимание наблюдаемых процессов и механизмов в системе, оно может измениться уже завтра. Мы были бы в высшей степени признательны за всякое указание на сомнительные в этом смысле места, что поможет сделать этот текст максимально адекватным и пригодным для использования.

Более того, не все наши вопросы к системе на сегодняшний день находят удовлетворительное решение – в силу ли ошибочной интерпретации процессов, наблюдаемых нами в системе, или в силу наличия в системе естественных «узлов и закрутов» (А. Ремизов), которые мы пока не в состоянии раскрутить. На таких вопросах мы сознательно акцентируем внимание, даже если пока и не можем предложить на них удовлетворительных ответов.

Но в этой технологии формирования материала по принципу «черного ящика»[1] есть и своя положительная оборотная сторона: описанные и тестированные фрагменты кода отображают состояние и функционирование механизмов ОС «как они есть», а не «как они должны быть». Это вдвойне актуально, учитывая, что разработчик QNX, фирма QSSL, не раз в своей технической документации описывала компоненты как реально существующие, в то время как их только предполагалось реализовать в последующих версиях (маршрутизация QNET «над» IP, механизмы shared memory и др.), или описывала механизмы в таком туманном изложении (дисциплины балансировки нагрузки QoS, HAM), что трудно понять, в какой степени они уже пригодны к практическому применению, а в какой – являются лишь экспериментальными наработками на будущее.

Такая неполнота информации недопустима при разработке целевых систем повышенной надежности для критических областей применения. Хочется надеяться, что наше изложение органично дополнит техническую документацию QNX, что позволит приблизить QNX к практическому использованию. Также надеемся, что эта книга окажется полезной для программистов, работающих в сфере реальных разработок.

Примечание

Предполагается, что читатель уже достаточно обстоятельно знаком с общей техникой программирования в C/C++[2] и имеет некоторый опыт в описываемой области по другим ОС (Linux, MS-DOS, MS Windows и др.). На основах программной техники мы останавливаться не будем, многие образцы кода, использующие общеизвестные и часто употребляемые приемы, будут даваться без комментариев. Более того, текст на протяжении книги очень «неровный» по глубине изложения. Например, текст приложения, предложенный В. Зайцевым, требует существенных предварительных знаний, а еще лучше – опыта разработки в QNX. Однако при первом чтении

[1] По тому же принципу писались книги, ставшие самыми информативными источниками в мировой практике, например описания Даниэля Нортона по MS-DOS или Джеффри Рихтера по Win32.

[2] Все технические описания QNX API сформированы в ориентации на классический C. Напротив, все используемые в тексте примеры кода излагаются в синтаксисе C++, а прилагаемые к тексту приложения транслированы в C++. Это обусловлено рядом аргументов, которые обсуждать не будем, но отметим, что такое различие подходов в любом случае расширяет информационную базу относительно использования QNX API.

такие «усложненные» фрагменты могут быть опущены без ущерба для понимания основного материала.

И наконец, последнее (по порядку, но не по значимости): большая часть излагаемого материала и образцов программного кода базируется на POSIX-стандартизованных механизмах (там, где обратное не оговорено особо) и поэтому может быть отнесена не только к ОС QNX, но и расширена на другие UNIX-подобные системы. Мы же в подавляющем большинстве примеров кода для программной проверки своих положений в качестве эталонной платформы используем ОС QNX как одну из доступных систем UNIX.

На основе ревизии последних лет можно сказать, что в QNX механизмы POSIX реализованы наиболее полно и последовательно относительно других систем, однако и в среде других ОС проводится активная работа по приведению их API в точное соответствие с POSIX. Один из примеров подобной эволюции в ОС Linux мы рассматриваем в тексте; в среде ОС NetBSD заявлено о приведении к соответствию с расширениями POSIX реального времени механизмов потоков и синхронизации. Большая часть кода, приводимого в тексте, может быть перенесена (иногда с незначительными изменениями) в другие системы: Linux, FreeBSD, NetBSD и проч. Отличия же в деталях функционировании рассматриваемых механизмов в разных системах, наоборот, только помогают прояснить детальную картину происходящего. Именно поэтому эта двойственность и была вынесена в заголовок книги: «QNX/UNIX».

Теперь несколько слов о многочисленных примерах программного кода в тексте. Так уж получилось, что по ходу работы над текстом мы постепенно стали нацеливать программные примеры под тестирование возможностей и эффективности иллюстрируемых программных механизмов. Однако такая ориентация, попутно предоставляющая разработчику количественные ориентиры по ОС, не должна затуманивать главное предназначение примеров кода: в них мы стараемся наиболее широко манипулировать разнообразными средствами API, с тем чтобы фрагменты этого кода могли быть непосредственно заимствованы читателями для своих будущих проектов и далее развивались там. Тем не менее в некоторых случаях мы показываем в коде, «как это можно сделать» (когда нужно иллюстрировать специфический механизм), но это вовсе не значит, что «так нужно делать» из соображений производительности, переносимости и т. д. Программный код всех примеров и все необходимое для их сборки, исполнения и проверки находятся в составе файлов архива, доступного по адресу *http://www.symbol.ru/library/qnx-unix/pthread.tgz*.

Чего нет в этой книге...

В этой книге нет множества приятных и полезных вещей: кулинарных рецептов, эротических сцен, кроссвордов... Но здесь мы хотим перечис-

лить те тематические разделы, которые имеют прямое отношение к вопросам параллельного программирования и которые должны были бы войти в книгу, но в силу определенных обстоятельств в нее не вошли:

- Общие UNIX-механизмы IPC (Inter Process Communication). Из всех механизмов, традиционно относимых к IPC, мы детально затрагиваем только один – сигналы. Другие, крайне интересные в применениях и полноценно представленные в API QNX, такие как неименованные (pipe) и именованные (FIFO) каналы, очереди сообщений POSIX (`mq_*`), блокирование записей и файлов (`fcntl()`) и ряд других механизмов, полностью и сознательно обойдены вниманием. Это связано с тем, что: а) в программирование этих механизмов мало что привносит именно «потоковая» (thread) ориентация, положенная во главу угла нашего рассмотрения; б) эти механизмы настолько исчерпывающе описаны У. Стивенсом [2], что добавить что-либо трудно; в) нам крайне не хотелось раздувать объем текста сверх некоторой разумной меры без крайней на то необходимости.
- Совместно используемая (разделяемая) память (shared memory). Это также один из механизмов, традиционно относимый к подмножеству IPC, но мы его выделяем особо. Это именно тот механизм, который **должен** быть описан применительно к QNX самым тщательным образом, но… Самые поверхностные эксперименты наводят на мысль, что именно в QNX реализации механизмов разделяемой памяти выполнены достаточно «рудиментарно»: ряд возможностей, рассматриваемых POSIX как стандартные, не реализован или реализован в ограниченной мере. Поэтому механизмы разделяемой памяти в QNX требуют отдельного пристального изучения и тестирования. Возможно, это должно быть сделано в отдельной публикации, что мы и планируем восполнить в ближайшем будущем.
- Таймеры в системе. Таймерные механизмы в QNX развиты в полной мере, что неудивительно для ОС реального времени, ориентированной во многом на «встраиваемые» (embedded) применения. Однако таймеры а) имеют все же косвенное отношение к вопросам параллелизма и синхронизации и б) блестяще и полно описаны Р. Кертеном [1].

Вообще, при отборе материала для книги мы старались максимально придерживаться следующего алгоритма: чем шире некоторый предмет освещен в литературе (объект или механизм ОС, приемы его использования и тому подобное), по крайней мере, в известной нам литературе, тем меньше внимания мы уделяли ему в своем тексте.

Благодарности

Предварительный вариант книги был вынесен на обсуждение широкой QNX-общественности (да и UNIX/Linux) на форуме *http://qnxclub.net*. Было высказано столько замечаний, пожеланий и рекомендаций, что окончательный текст, в котором все они были учтены, стал радикально

отличаться от исходной редакции. Невозможно перечислить всех членов интернет-сообщества (в первую очередь, конечно, *http://qnx.org.ru* и *http://qnxclub.net*), кто внес свой вклад в это издание – мы благодарны всем без исключения. Но особую признательность мы выражаем:

- Владимиру Зайцеву из г. Харькова, который не только предоставил свой авторский материал, составивший отдельное, самоценное приложение, дополнившее книгу, но и обстоятельно вычитал весь остальной текст, а также внес ряд весьма ценных уточнений.
- Евгению Видревичу из г. Монреаля, который вычитал весь текст с позиции профессионального разработчика и указал на ряд сомнительных мест, а вместо некоторых наших путаных и не совсем внятных формулировок предложил свои – ясные и прозрачные.
- Евгению Тарнавскому из г. Харькова, который высказал ряд очень точных рекомендаций и замечаний, нашедших свое отражение в окончательном варианте текста.

Типографские соглашения

В тексте содержится множество ссылок на программные конструкции: фрагменты кода, имена функций API, символические константы и многое другое, которые при их использовании должны в неизменном виде (именно в таком написании) «перекочевывать» в программный код. Такие фрагменты, конструкции и лексемы выделены моноширинным шрифтом, например `pthread_create()`.

Фрагменты текста, цитируемые из указанных источников, выделены *курсивом*. Таких мест очень немного: прямое цитирование допускалось нами только в отношении крайне принципиальных утверждений.

В отличие от коротких (в две-три строки) фрагментов кода, листинги программ, приводимых и обсуждаемых в тексте, предваряются отчетливо выделенным заголовком. Это указывает на то, что данную программу как законченную программную единицу можно найти в архиве по адресу *http://www.symbol.ru/library/qnx-unix/pthread.tgz*. Помимо крупных законченных проектов там же можно найти и отдельные фрагменты кода, обсуждаемые в тексте. Для удобства поиска названия программных файлов, содержащихся в архиве, приводятся в тексте книги перед соответствующим кодом в скобках, например (*файл s2.cc*).[1]

---

[1] В книге в примерах кода мы часто используем русскоязычные символьные константы для вывода сообщений, например `"Получен сигнал SIGINT"`, что способствует большей доходчивости обсуждаемого кода. Однако в файлах работающих приложений, представленных в архиве, вы увидите только англоязычные эквиваленты выводимых сообщений, поскольку работающие примеры кода являются консольными приложениями, а текстовая консоль QNX не русифицируема в принципе и графические псевдотерминалы (pterm, xterm) имеют определенные сложности.

1

Введение

Параллелизм

Феномен параллелизма при выполнении принципиально последовательного по своей природе компьютерного кода возникает даже раньше, чем он начинает отчетливо требоваться для многозадачных и многопользовательских операционных систем:

- Код обработчиков аппаратных прерываний, являющихся принципиально асинхронными, в самых последовательных ОС выполняется параллельно прерываемому ими коду.
- Для работы многих системных служб необходимо, чтобы они выполнялись параллельно с выполнением пользовательской задачи.

Примечание

Например, в принципиально однозадачной операционной системе MS-DOS исторически первой службой, требующей параллельного выполнения, была подсистема спулинга печати. Но добавлять ее в систему пришлось «по живому», поскольку основная структура системы уже сложилась и стабилизировалась (к версии 2.x), а механизмы параллелизма в этой структуре были изначально отвергнуты на корню. И с этого времени начинается затянувшаяся на многие годы история развития уродливой надстройки над MS-DOS – технологии создания TSR-приложений (terminate and stay resident), программного мультиплексора INT 2F и других.

Новое «пришествие» механизмов параллельного выполнения (собственно, уже хорошо проработанных к этому времени в отрасли мэйнфреймов) начинается с появлением многозадачных ОС, разделяющих во времени выполнение нескольких задач. Для формализации (и стандартизации поведения) развивающихся параллельно программных ветвей создаются абстракции процессов, а позже и потоков. Простейший случай параллелизма – когда N (N>1) задач разделяют между собой ресурсы: время единого процессора, общий объем физической оперативной памяти…

Но многозадачное разделение времени – не единственный случай практической реализации параллельных вычислений. В общем случае программа может выполняться в аппаратной архитектуре, содержащей более одного (M) процессора (SMP-системы). При этом возможны принципиально отличающиеся по поведению ситуации:

- Количество параллельных ветвей (процессов, потоков) N больше числа процессоров M, при этом некоторые вычислительные ветви находятся в блокированных состояниях, конкурируя с выполняющимися ветвями за процессорное время. (Частный случай – наиболее часто имеющее место выполнение N ветвей на одном процессоре.)
- Количество параллельных ветвей (процессов, потоков) N меньше числа процессоров M, при этом все ветви вычисления могут развиваться действительно параллельно, а блокированные состояния возникают только при необходимости синхронизации и обмена данными между параллельными ветвями.

Все механизмы параллелизма проектируются (и это находит прямое отражение в POSIX-стандартах, а еще более в текстах комментариев к стандартам) и должны использоваться так, чтобы неявно не допускались какие-либо предположения об относительных скоростях параллельных ветвей и моментах достижения ими (относительно друг друга) конкретных точек выполнения.[1] Так, в программном фрагменте:

```
void* threadfunc ( void* data ) {
  // оператор 1:
};
...
pthread_create( NULL, NULL, threadfunc, NULL );
// оператор 2:
...
```

нельзя допускать никаких априорных предположений о том, как пойдет дальнейшее выполнение после точки ветвления (точки вызова `pthread_create()`): а) будет выполняться «оператор 2» в родительском потоке; б) будет выполняться «оператор 1» в порожденном потоке; в) на различных процессорах будут действительно одновременно выполняться «оператор 1» и «оператор 2»... Программный код должен быть организован так, чтобы в любых аппаратных конфигурациях (количество процессоров, их скорости, особенности кэширования па-

[1] Это положение напрямую диктуется определением «слабосвязанных процессов», впервые сформулированным Э. Дейкстрой [10]. Заметим, что фундаментальная и стройная «картина мира», выстроенная Э. Дейкстрой и считающаяся классикой, исчерпывающе («необходимо и достаточно») описывает систему процессов равного приоритета. Расширение реальных систем атрибутом приоритета затуманивает прозрачность этой модели и делает все гораздо сложнее...

мяти процессорами и другие характеристики) результаты выполнения были полностью эквивалентны.

Благодаря наличию в составе ОС QNX сетевой подсистемы QNET, органично обеспечивающей «прозрачную» интеграцию сетевых узлов в единую многомашинную систему, возникает дополнительный источник параллелизма (а вместе с тем и дополнительных хлопот), еще более усложняющий общую картину: запросы по QNET к сервисам, работающим на одном сетевом узле, со стороны клиентских приложений, работающих на других. Например, ежедневно выполняя простейшую команду:

```
# cp /net/host/dev/ser1 ./file
```

часто ли мы задумываемся над тем, кого и в каком порядке будет вытеснять код, выполняющий копирование файлов.

Для текущей выполняющейся задачи такой удаленный запрос из сети QNET является скрытым источником параллелизма, а благодаря наследованию приоритетов даже удаленный запрос по сети может привести к немедленному вытеснению локальной задачи, выполняющейся до получения запроса.

Приведенная выше аргументация – это далеко не полный перечень причин, по которым стоит еще пристальнее и с большей заинтересованностью взглянуть на техники параллельной организации вычислительного процесса. В литературе неоднократно отмечалось (например, [11]), что даже в тех случаях, когда приложение заведомо никогда и нигде не будет использоваться на многопроцессорной платформе, более того, когда логика приложения не предполагает естественного параллелизма как «одновременности выполнения», – даже тогда расщепление крупного приложения на логические фрагменты, которые построены как параллельные участки кода, взаимодействующие в ограниченном числе точек контакта, – это путь построения «прозрачного» для написания и понятного для сопровождения программного кода. И как следствие, этот путь (иногда на первый взгляд кажущийся несколько искусственным и привнесенным) – путь построения приложений высокой надежности, свободных от ошибок, характерных для громоздких монолитных приложений, и простых в своем последующем развитии и сопровождении.

Как уже неоднократно отмечалось, параллельная техника выражения в программном коде, пусть даже принципиально последовательных процессов, сопряжена с определенными трудностями: необходимость отличного, «параллельного», взгляда на описываемые процессы и отсутствие привычки применять специфические разделы API, редко используемые в классическом «последовательном» программировании. Единожды освоив эту технику, применять ее в дальнейшем становится легко и просто. Возможно и большее число рутинных приемов ис-

пользования параллельной техники – в своей книге мы постарались «рассыпать» по тексту множество программных иллюстраций.

Наконец, есть еще одна, последняя особенность предлагаемого вашему вниманию материала: значительная часть приводимых здесь примеров и описаний относится ко всему многообразию ОС, поддерживающих POSIX-стандарт, однако акцент делается на не совсем очевидные особенности построения так называемых «приложений реального времени» [4]. В первую очередь это касается принципов синхронизации задач, совместно использующих общий ресурс. К сожалению, приемы программирования, широко распространенные при параллельном выполнении задач общего назначения, могут привести к не совсем предсказуемым результатам (по времени реакции) при построении систем реального времени. Особенности построения параллельно исполняемых систем в сферах реального времени и стали тем ключевым моментом, ориентируясь на который мы строили этот текст.

Семейства API

Общее множество вызовов API (Application Program Interface – интегральное наименование всего множества вызовов из программной среды к услугам операционной системы), реализуемое операционной системой (ОС) реального времени QNX, естественным образом разделяется на три независимых подгруппы:

- Native QNX API – это самодостаточный набор вызовов, развиваемый со времен ранних версий QNX (когда вопрос о совместимости с POSIX еще не стоял); является естественным базисом этой системы, отображающим «микроядерность» ее архитектуры, но по соображениям возможной совместимости и переносимости он является также и исключительной принадлежностью этой ОС.
- POSIX (BSD) API – это уровень API, регламентируемый постоянно расширяющейся системой стандартов группы POSIX, которым должны следовать все ОС, претендующие на принадлежность к семейству UNIX.
- System V API (POSIX) – это та часть API, которая заимствует модели, принятые в UNIX-ах, относящихся к ветви развития System V, а не к ветви BSD.

Native QNX API

Именно этот слой является базовым слоем, реализующим функциональность самой системы QNX. Два последующих слоя в значительной мере являются лишь «обертками», которые ретранслируются в вызовы native QNX API после выполнения реструктуризации или перегруппировки аргументов вызова в соответствии с синтаксисом, требуемым этим вызовом.

Совершенно естественно, что прикладное программное приложение может быть полностью прописано в этом API (как, впрочем, и в каждом другом из описываемых ниже), но это не лучший выбор (на этом акцентирует внимание и техническая документация QNX) по двум причинам: во-первых, из соображений переносимости, а во-вторых, этот слой является самым «мобильным» – разработчики QSSL могут изменить его отдельные вызовы при последующем развитии системы. Примером вызова этого слоя является, в частности, `ThreadCreate()`, применяемый для создания нового потока.

Тем не менее нужно сразу отметить, что многие возможности и модели (например, реакция на сигналы в потоках, тонкое управление поведением мьютексов и другие моменты) не могут быть реализованы в рамках POSIX-модели и выражаются только в native API QNX.

POSIX (BSD) API

Эта часть API наиболее полно соответствует API ОС UNIX, относящихся к ветви BSD (BSD, FreeBSD, NetBSD и другие).[1] Ее наименование можно было бы сузить до «BSD API», так как описанный далее набор API System V также регламентируется POSIX, но мы будем использовать именно термин «POSIX API», следуя терминологии фундаментальной книги У. Стивенса [2]. Эквивалентом названного выше для native API `ThreadCreate()` здесь будет выступать `pthread_create()`.

Именно на API этого слоя и будет строиться последующее изложение и приводимые примеры кода (параллельно с вызовами этого API мы будем для справки кое-где указывать имена комплиментарных им вызовов native API), за исключением случаев использования тех возможностей QNX, которые не имеют эквивалентов в POSIX API. Как раз все, что будет выражено в этом API далее по тексту, может быть перенесено на все UNIX-подобные операционные системы, о чем мы и говорили выше.

Примечание

Самый ранний стандарт POSIX известен как IEEE 1003.1-1988 и, как следует из его названия, относится к 1988 году (если точнее, то ему предшествовал рабочий вариант под названием IEEEIX 1986 года, когда термин POSIX еще не был «придуман»). Более поздняя редакция его развития, IEEE 1003.1-1996, наиболее широко известна как «стандарт POSIX», иногда называемый POSIX.1. Набор стандартов POSIX

[1] На сегодняшний день практически ни одна из ОС UNIX уже не может быть отнесена чисто к System V или BSD, во многом исходя именно из требования совместимости с POSIX, который требует одновременного наличия и того и другого API (хотя в каждом случае комплиментарный набор API реализуется как «обертка» к базовому). Одними из первых (к 1997–1998 гг.) ОС, поддерживающих оба набора API, стали Sun Solaris 2.6 и Digital Unix 4.0B [3].

находится в постоянном развитии и расширении и к настоящему времени включает в себя набор более чем из 30 автономных стандартов.

Для целей операционных систем реального времени возникла потребность определить отдельные механизмы особыми стандартами, на семь из которых ссылаются наиболее часто: 1003.1a, 1003.1b, 1003.1c, 1003.1d, 1003.1j, 1003.21, 1003.2h. Например:

1003.1a (OS Definition) – определяет базовые интерфейсы ОС;

1003.1b (Realtime Extensions) – описывает расширения реального времени, такие как модель сигналов реального времени, диспетчеризация по приоритетам, таймеры, синхронный и асинхронный ввод-вывод, IPC-механизмы (семафоры, разделяемая память, сообщения);

1003.1c (Threads) – определяет функции поддержки потоков, такие как управление потоками, атрибуты потоков, примитивы синхронизации (мьютексы, условные переменные, барьеры и др., но не семафоры), диспетчеризация.

System V API

Этот набор API является базовым для второй ветви[1] UNIX – System V (AT&T Unix System V). Как и оба предыдущих, этот набор API самодостаточен для реализации практически всех возможностей ОС, но использует для этого совершенно другие модели, например сетевую абстракцию TLI вместо сокетов BSD. Для области рассматриваемых нами механизмов – потоков, процессов, синхронизирующих примитивов и др. – в POSIX API и System V API почти всегда существуют функциональные аналоги, отличающиеся при этом как синтаксически, так и семантически. Например, в POSIX API семафор представлен типом `sem_t` и основными операциями с ним `sem_wait()` и `sem_post()`, а в System V API семафор описывается структурой ядра `sem`, а операции (и `wait`, и `post`) осуществляются вызовом `semop()`. Кроме того, операции производятся не над единичными семафорами, а над наборами (массивами) семафоров (в наборе может быть и один семафор). Как отсюда видно, логика использования принципиально единообразных примитивов существенно отличается.

[1] При общей истории UNIX, начинающейся с 1971 г. [7], две ветви API – BSD и System V – в их современном виде сформировались достаточно поздно: BSD к 1983 г., а System V к 1987 г. [3, 7]. Но многие IPC-механизмы System V (например, семафоры) сформировались по времени заметно раньше своих аналогов из BSD. Как отмечается в [3]: «*Информация об истории разработки и развитии функций System V IPC не слишком легко доступна <...> очереди сообщений, семафоры и разделяемая память этого типа были разработаны в конце 70-х в одном из филиалов Bell Laboratories в городе Колумбус... Эта версия называлась Columbus Unix, или CB Unix*».

Примечание

В технической документации присутствие System V API в QNX не упоминается ни одним словом, но он, как того и требует POSIX, действительно предоставляется и в виде библиотек, и в виде необходимых файлов определений (заголовочных файлов). Просто его заголовочные файлы, определяющие структуры данных и синтаксис вызовов, находятся в других относительно POSIX-интерфейсов местах. Так, например, описание семафоров POSIX API (тип `sem_t`) расположено в файле `<semaphore.h>`, а описание семафоров System V API – в файле `<sys/sem.h>` (аналогично относительно всех конструкций, моделируемых этим API).

С позиции программиста System V API присутствует в QNX главным образом для переносимости программных проектов, ранее созданных с использованием этого API, например первоначально созданных для других ОС UNIX (Sun Solaris, HP UNIX и др.). В данной книге это семейство API рассматриваться не будет.

2

Процессы и потоки

При внимательном чтении технической документации [8] и литературы по ОС QNX [1] отчетливо бросается в глаза, что тонкие детали создания и функционирования процессов и потоков описаны крайне поверхностно и на весьма некачественном уровне. Возможно, это связано с тем, что общие POSIX-механизмы уже изучены и многократно описаны на образцах кода в общей литературе по UNIX. Однако большинство литературных источников написано в «допотоковую» эпоху, когда основной исполняемой единицей в системе являлся процесс.

Детальное рассмотрение особенностей именно QNX[1] (версии 6.X после приведения ее в соответствие с POSIX, в отличие от предыдущей 4.25) лишний раз подчеркивает, что:

- Процесс является только «мертвой» статической оболочкой, хранящей учетные данные и обеспечивающей окружение динамического исполнения… Чего? Конечно же, потока, даже если это единственный (главный) исполняемый поток приложения (процесса), как это принято в терминологии, не имеющий отношения к потоковым понятиям.
- Любые взаимодействия, синхронизация, диспетчеризация и другие механизмы имеют смысл только применительно к потокам, даже если это потоки, локализованные в рамках различных процессов. Вот здесь и возникает термин, ставший уже стереотипным: «IPC – средства взаимодействия **процессов**». Для однопотоковых приложений этот терминологический нюанс не вносит ровно никакого различия, но при переходе к многопотоковым приложениям мы должны рассуждать в терминах именно взаимодействующих потоков, локализованных внутри процессов (одного или различных).
- В системах с аппаратной трансляцией адресов памяти (MMU – Memory Menegment Unit) процесс создает для своих потоков дополнительные «границы существования» – защищенное адресное про-

[1] [4]: глава Д. Алексеева «Получение системной информации».

странство. Большинство сложностей, описываемых в литературе в связи с использованием IPC, обусловлено необходимостью взаимодействующих потоков преодолевать адресные барьеры, устанавливаемые процессами для каждого из них. (Что касается MMU, то в данной книге предполагается исключительно x86-архитектура, хотя количество аппаратных платформ, на которых работает ОС QNX, на сегодняшний день уже перевалило за десяток.)

Примечание

Модель потоков QNX в значительной степени напоминает то, что происходит с процессами в MS-DOS или с задачами (task) в существенно более поздней ОС реального времени VxWorks: исполнимые единицы разделяют единое адресное пространство без каких-либо ограничений на использование всего адресного пространства. В рамках подобной модели в QNX можно реализовать и сколь угодно сложный комплекс, трансформировав в потоки отдельные процессы, составляющие этот комплекс, с тем только различием, что в QNX все элементы собственно операционной системы продолжают работать в изолированном адресном пространстве и не могут быть никоим образом включены (и тем самым повреждены) в пространство приложения.

И в технической документации QNX, и в книге Р. Кертена [1] много страниц уделено описанию логики процессов, потоков, синхронизации и многим другим вещам в терминах аллегорических аналогий: коллективное пользование ванной комнатой, кухней... Если согласиться, что такие аллегории более доходчивы для качественного описания картины происходящего (что, похоже, так и есть), то для иерархии «операционная система – процесс – поток» можно найти существенно более близкую аллегорию: «аквариумное хозяйство». Действительно:

- В некотором общем помещении, где имеются все средства жизнеобеспечения – освещение, аэрация, терморегуляция, кормление (операционная система), – размещаются аквариумы (процессы), внутри которых (в одних больше, в других совсем немного) живут активные сущности (растения, рыбы, улитки). Помимо всех прочих «удобств» в помещении время от времени появляется еще одна сущность – «хозяин». Он является внешней по отношению к системе силой, которая асинхронно предпринимает некоторые действия (кормление, пересадка животных), нарушающие естественное «синхронное» течение событий (это служба системного времени операционной системы, которая извне навязывает потокам диспетчеризацию).
- Аквариумы (процессы) являются не только контейнерами, заключающими в себе активные сущности (потоки). Они также ограничивают ареал существования (защищенное адресное пространство) для их обитателей: любое нарушение границ обитания в силу каких-либо форс-мажорных обстоятельств, безусловно, означает гибель нарушителя (ошибка нарушения защиты памяти в потоке).

- Обитатели аквариумов (потоки) легко и непринужденно взаимодействуют между собой (сталкиваются при движении или, напротив, уступают друг другу место) в пределах контейнера (процесса). Однако при этом они не могут взаимодействовать с обитателями других контейнеров (процессов); более того, они даже ничего не знают об их существовании. Если обитатель требует вмешательства, например перемещения его в другой контейнер, то он может лишь способствовать этому, взывая своим поведением (при помощи особых знаков) к инстанции более высокого уровня иерархии, в отличие от контейнера некоторой «общесистемной субстанции», взывающего к хозяину (операционной системе) о вмешательстве (диспетчеризации).
- Все жизненно необходимые ресурсы (кислород, корм, свет) поступают непосредственно к контейнеру как единице распределения (операционная система выделяет ресурсы процессу в целом). Обитатели контейнера (потоки) конкурируют за распределение общих ресурсов контейнера на основании своих характеристик (приоритетов) и некоторой логики (дисциплины) распределения относительно «личностных» характеристик: размера животного, быстроты реакции и движения и т. д.

Такая ассоциативная аналогия, возможно, позволит отчетливее ощутить, что процесс и поток относятся к различным уровням иерархии понятий ОС. Это различие смазывается тем обстоятельством, что в любой ОС (с поддержкой модели потоков или без нее) всякий процесс всегда наблюдается в неразделимом единстве хотя бы с одним (главным) потоком и нет возможности наблюдать и анализировать поведение «процесса без потока».

Отсюда и происходят попытки объединения механизмов создания и манипулирования процессами и потоками «под одной крышей» (единым механизмом). Например, в ОС Linux создание и процесса (`fork()`), и потока (`pthrad_create()`) свели к единому системному вызову `_clone()`, что явилось причиной некоторой иллюзорной эйфории, связанной с непонятной, мифической «дополнительной гибкостью».

Усилия последующих лет были направлены как раз на разделение этих механизмов, ликвидацию этой «гибкости» и восстановление POSIX-модели. Отсюда же вытекают и разработки последних лет в области новых «экзотических» ОС, направленные на сближение модели процесса и потока, и попытки создания некой «гибридной» субстанции, объединяющей атрибуты процесса и потока, если того захочет программист (на момент создания). По нашему мнению, идея «гибридизации» достаточно сомнительна и согласно нашей аналогии направлена на создание чего-то, в головной своей части напоминающего аквариум, а в задней – рыбу. Получается даже страшнее, чем русалка...

Отмеченный выше дуализм абстракций процессов и потоков (а в некоторых ОС и их полная тождественность) приводит к тому, что крайне

сложно описывать одно из этих понятий, не прибегая к упоминанию атрибутов другого. В итоге, с какой бы из двух абстракций ни начать рассмотрение, нам придется, забегая вперед, ссылаться на атрибутику другой, дуальной ей. В описании процессов нам не обойтись без понятия приоритета (являющегося атрибутикой потока), а в описании потоков мы не сможем не упомянуть глобальные (относительно потока) объекты, являющиеся принадлежностью процесса, например файловые дескрипторы, сокеты и многое другое.

По этой причине наше последующее изложение при любом порядке его «развертывания» обречено на некоторую «рекурсивность». Итак, следуя сложившейся традиции, начнем с рассмотрения процессов.

Процессы

Создание параллельных процессов настолько полно описано в литературе по UNIX, что здесь мы приведем лишь минимально необходимый беглый обзор, останавливаясь только на отличительных особенностях ОС QNX.

Всякое рассмотрение предполагает наличие системы понятий. Интуитивно ясное понятие процесса не так просто поддается формальному определению. Процитируем (во многом качественное) определение, которое дает Робачевский [3]:

> *Обычно программой называют совокупность файлов, будь то набор исходных текстов[1], объектных файлов или собственно выполняемый файл. Для того чтобы программа могла быть запущена на выполнение, операционная система сначала должна создать окружение или среду выполнения задачи, куда относятся ресурсы памяти, возможность доступа к устройствам ввода/вывода и различным системным ресурсам, включая услуги ядра.*

Процесс всегда содержит хотя бы один поток, поскольку мы говорим об исполняемом, развивающемся во времени коде. Для процессов, исходный код которых подготовлен на языке C/C++, **главным потоком** процесса является поток, в котором исполняется функция, текстуально описанная под именем `main()`. Код и данные процесса размещаются в оперативной памяти в **адресном пространстве** процесса. Если операционная система и реализующая платформа (наше рассмотрение ограничено только реализацией x86) поддерживают MMU и виртуализацию адресного пространства на физическую память, то каждый процесс

[1] Здесь Робачевский мимоходом расширяет понятие процесса и на программу, представленную, например, текстом для интерпретатора shell, или языков Perl, Tcl/Tk, или других интерпретаторов. В контексте нашего обсуждения в случаях выполнения таких «программ» «процессом» будет процесс, интерпретирующий текст скрипта, и именно к нему в полной мере относятся все детали нашего рассмотрения относительно процессов.

имеет собственное изолированное и уникальное адресное пространство и у него нет возможности непосредственно обратиться в адресное пространство другого процесса.

Любой процесс может содержать произвольное количество потоков, но не менее одного и не более 32 767 (для QNX версии 6.2). Совокупность данных, необходимых для выполнения любого из потоков процесса, а также контекст текущего выполняемого потока называются **контекстом процесса**.

Согласно ранним «каноническим» спецификациям UNIX [3] ОС должна поддерживать не менее 4095 отдельных процессов (точнее 4096, из которых 0-вой представляет собой процесс, загружающий ОС и, возможно, реализующий в дальнейшем функции ядра). Во всей документации ОС QNX нам не удалось найти предельное значение этого параметра. Но если из этого делается «тайна мадридского двора», то наша задача – найти это значение:

```
int main( int argc, char* argv[] ) {
   unsigned long n = 1;
   pid_t pid;
   while( ( pid = fork() ) >= 0 ) {
      n++;
      if( pid > 0 ) {
         waitpid( pid, NULL, WEXITED );
         exit( EXIT_SUCCESS );
      };
   };
   if( pid == -1 ) {
      cout << "exit with process number: "
           << n << " - " << flush;
      perror( NULL );
   };
};
```

Этот достаточно непривычный по внешнему виду код дает нам следующий результат:

```
# pn
exit with process number: 1743 - Not enough memory
```

Системному сообщению о недостатке памяти достаточно трудно верить: чуть меньше 4 Кбайт программного кода в своих 1743 «реинкарнациях» требуют не более 6,6 Мбайт для своего размещения при свободных более 230 Мбайт в системе, в которой мы испытывали это приложение. Оставим это на совести создателей ОС QNX.

В продолжение нашей основной темы любопытно рассмотреть результаты вывода команды `pidin`, а именно последнюю ее строку с информацией о последнем запущенном в системе процессе:

- до запуска обсуждаемого приложения:

```
47366186    1 /photon/bin/phcalc  10r REPLY       241691
```

- и после его завершения:

```
54652947    1 bin/pidin           10r REPLY       1
```

Легко видеть, что разница PID, равная 54652947 – 47366186 = 7286761, никак не является числом активированных на этом временном промежутке процессов, которое равно 1743. Поэтому к численным значениям PID нужно относиться с заметной осторожностью: это не просто инкрементированное значение числа запущенных процессов, схема формирования PID заметно сложнее.

В любом случае мы можем принять, что в ОС QNX Neutrino 6.2.1, как и в других «канонических» UNIX, количество процессов (если, конечно, эта ОС не дает нам более вразумительных оценок) ограничено цифрой 4095. Видно, что общее количество независимых потоков исполнения в системе может достигать совершенно ошеломляющей цифры. Но как бы много потоков мы ни создавали, им все равно придется конкурировать за доступ к самому главному ресурсу – процессору. В настоящее время реализованные в QNX дисциплины диспетчеризации работают над суммарным полем всех потоков в системе (рис. 2.1): если в системе выполняется N процессов и i-й процесс реализует M_i потоков, то в очередях диспетчеризации одновременно задействовано $\sum_{i=1}^{N} M_i$ управляемых объектов (потоков).

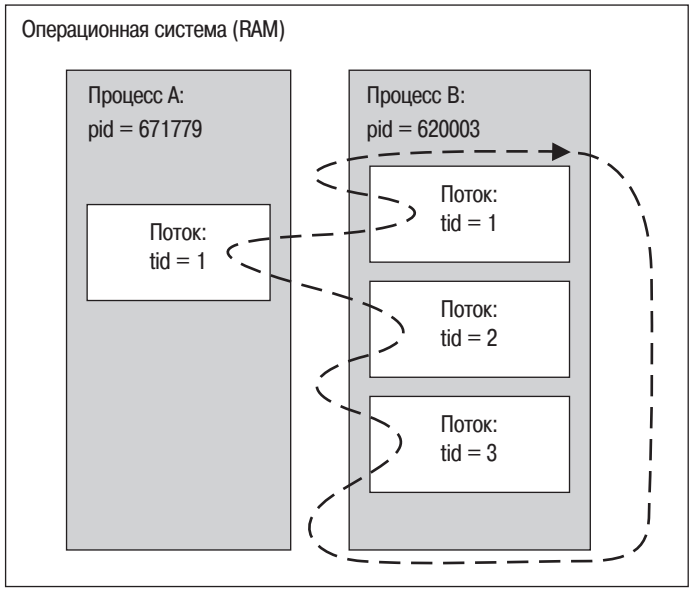

Рис. 2.1. *Диспетчеризация процессов*

На рис. 2.1 изображены два процесса, выполняющиеся под управлением системы. Каждый процесс создал внутри себя различное количество потоков равного приоритета. Обратите внимание, что фактическая диспетчеризация производится не между процессами, а между потоками процессов, даже если иногда для простоты говорят «диспетчеризация процессов». Потоки объединены в циклическую очередь диспетчеризации, и пунктирная линия показывает порядок, в котором (в направлении стрелки) они будут поочередно получать квант времени.

Если ни один из потоков не будет выполнять блокирующих операций (`read()`, `delay()`, `accept()`, `MsgSend()` и множество других), что реально встречается крайне редко, то показанный порядок «следования» потоков при диспетчеризации будет сохраняться неограниченно долго. Как только поток выполнит блокирующий вызов, он будет удален из очереди готовых к выполнению потоков, а после завершения вызова возвращен в очередь, причем (что характерно!) в голову очереди. После этого топология «петли» (порядок чередования), показанной на рисунке пунктиром, может произвольным образом измениться.

Из рисунка хорошо видно, что при диспетчеризации «в рамках системы» (об этом мы будем говорить позже) два запущенных процесса будут выполняться в неравных условиях: на каждый полный цикл диспетчеризации программный код, выполняющийся в рамках процесса A, будет получать 1 квант времени, а код в процессе B – 3 кванта.

Примечание

Стандарт POSIX, определяя названную стратегию диспетчеризации константой `PTHREAD_SCOPE_SYSTEM`, предусматривает и другую стратегию, обозначаемую константой `PTHREAD_SCOPE_PROCESS`, когда потоки конкурируют за процессорный ресурс в пределах процесса, к которому они принадлежат (в Sun Solaris первой стратегии соответствуют «bound thread», а второй – «unbound thread»). Реализация стратегии `PTHREAD_SCOPE_PROCESS` связана с серьезными трудностями. Насколько нам известно, в настоящее время из числа широко распространенных ОС она реализована только в Sun Solaris. В QNX для совместимости с POSIX даже присутствуют системные вызовы относительно стратегии диспетчеризации:

```
int pthread_attr_setscope( pthread_attr_t* attr, int scope );
int pthread_attr_getscope( const pthread_attr_t* attr, int* scope );
```

но в качестве параметра `scope` они допускают... только значение `PTHREAD_SCOPE_SYSTEM` и на поведение потоков никакого влияния не оказывают.

PID (Process ID) – идентификатор процесса, присваиваемый процессу при его создании, например вызовом `fork()`. PID позволяет системе однозначно идентифицировать каждый процесс. При создании нового процесса ему присваивается первый свободный (то есть не ассоциированный ни с каким процессом) идентификатор. Присвоение происходит по возрастающей: идентификатор нового процесса больше иденти-

фикатора процесса, созданного перед ним. Когда последовательность идентификаторов достигает максимального значения (4095), следующий процесс получает минимальный свободный (за счет завершившихся процессов) PID, и весь цикл повторяется снова. Значения PID нумеруются, начиная с 0. Процесс, загружавший ОС, является родительским для всех процессов в системе? и его PID = 0.

Из других важных атрибутов процесса отметим[1]:

- PPID (Parent Process ID) – PID процесса, породившего данный процесс. Таким образом, все процессы в системе включены в единую древовидную иерархию.
- TTY – терминальная линия: терминал или псевдотерминал, ассоциированный с процессом. Если процесс становится процессом-демоном, то он отсоединяется от своей терминальной линии и не имеет ассоциированной терминальной линии. (Запуск процесса как фонового – знак «&» в конце командной строки – не является достаточным основанием для отсоединения процесса от терминальной линии.)
- RID и EUID – реальный и эффективный идентификаторы пользователя. Эффективный идентификатор служит для определения прав доступа процесса к системным ресурсам (в первую очередь к файловым системам). Обычно RID и EUID совпадают, но установка флага SUID для исполняемого файла процесса позволяет расширить полномочия процесса.
- RGID и EGID – реальный и эффективный идентификаторы группы пользователей. Как и в случае идентификаторов пользователя, EGID не совпадает с RGID, если установлен флаг SGID для исполняемого файла процесса.

Часто в качестве атрибутов процесса называют и приоритет выполнения. Однако приоритет является атрибутом не процесса (процесс – это статическая субстанция, контейнер), а потока, но если поток единственный (главный, порожденный функцией `main()`), его приоритет и есть то, что понимается под «приоритетом процесса».

Создание нового процесса

Созданию процессов (имеется в виду создание процесса из программного кода) посвящено столько описаний [1–9], что детальное рассмотрение этого вопроса было бы лишь пересказом. Поэтому мы ограничимся только беглым перечислением этих возможностей, тем более что в ходе обсуждения нас главным образом интересуют не сами процессы, а потоки, заключенные в адресных пространствах процессов.

---

[1] Здесь используется терминология [7]; терминология и аббревиатуры для различных клонов UNIX несколько различаются между собой в описывающих их литературных источниках.

Использование командного интерпретатора

Самый простой способ – запустить из программного кода дочернюю копию командного интерпретатора, которому затем передать команду запуска процесса. Для этого используется вызов:

```
int system( const char * command );
```

где `command` – текстовая строка, содержащая команду, которую предполагается выполнить ровно в том виде, в котором мы вводим ее командному интерпретатору с консоли.

Примечание

Функция имеет еще одну специфическую форму вызова, когда в качестве `command` задается `NULL`. По коду возврата это позволяет выяснить, присутствует ли (и доступен ли) командный интерпретатор в системе (возвращается 0, если интерпретатор доступен).

На время выполнения вызова `system()` вызывающий процесс приостанавливается. После завершения порожденного процесса функция возвращает код завершения вновь созданной копии интерпретатора (или –1, если сам интерпретатор не может быть выполнен), то есть младшие 8 бит возвращаемого значения содержат код завершения выполняемого процесса. Возврат вызова `system()` может анализироваться макросом `WEXITSTATUS()`, определенным в файле `<sys/wait.h>`. Например:

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

int main( void ) {
   int rc = system( "ls" );
   if( rc == -1 ) cout << "shell could not be run" << endl;
   else
      cout << "result of running command is "
           << WEXITSTATUS( rc ) << endl;
    return EXIT_SUCCESS;
};
```

Примечание

Эта функция использует вызов `spawnlp()` для загрузки новой копии командного интерпретатора, то есть «внутреннее устройство» должно быть в общем виде вам понятно. Особенностью QNX-реализации является то, что `spawnlp()` всегда использует вызов `/bin/sh`, независимо от конкретного вида интерпретатора, устанавливаемого переменной окружения SHELL (ksh, bash…). Это обеспечивает независимость поведения родительского приложения от конкретных установок системы, в которой это приложение выполняется.

Вызов `system()` является не только простым, но и очень наглядным, делающим код легко читаемым. Программисты часто относятся к нему

с пренебрежением[1], отмечая множество его недостатков. Однако в относительно простых случаях это может быть оптимальным решением, а недостатки не так и существенны:

- Используя копию командного интерпретатора, вызов `system()` может инициировать процесс, исполняющий и бинарную программу, и скрипт на языке самого командного интерпретатора (shell), а также программный код на интерпретирующих языках, таких как Perl, Tcl/Tk и др. Многие из рассматриваемых ниже «чисто программных» способов могут загружать и исполнять только бинарный исполняемый код (по крайней мере, без использования ими весьма громоздких искусственных и альтернативных возможностей).
- Остановка родительского процесса в ожидании завершения порожденного также легко разрешается: просто запускайте дочерний процесс из отдельного потока[2]:

```
#include <pthread.h>

void* process( void* command ) {
   system( (char*)command );
   delete command;
   return NULL;
}

int main( int argc, char *argv[] ) {
   ...
   char* comstr = "ls -l";
   pthread_create( NULL, NULL, strdup( comstr ), &process );
   ...
};
```

- Часто в качестве недостатка этого способа отмечают «автономность» и невозможность взаимодействия родительского и порожденного процессов.

Но для расширения возможностей взаимосвязи процессов можно прежде всего воспользоваться вызовом `popen()` (POSIX 1003.1a), являющимся в некотором роде эквивалентом, расширяющим возможности `system()`. Возможности `popen()` часто упускаются из виду, так как в описаниях этот вызов относится не к области создания процессов, а к области программных каналов (pipe). Синтаксис этого вызова таков:

```
FILE* popen( const char* command, const char* mode );
```

---

[1] Здесь многое зависит от расстановки приоритетов. Если вы хотите, чтобы всякий, читающий ваш код, тут же воскликнул: «Ну и крутой же парень написал такое!», заведомо используйте `spawn()`. При желании сделать код максимально элегантным используйте `fork()`, а если ставится задача хорошей читаемости и ясности кода, то очень часто достаточно и `system()`.

[2] Детали создания потока и в частности передачи ему параметра обстоятельно рассматриваются далее.

где command – командная строка, как и у system(); mode – режим создаваемого программного канала со стороны порождающего процесса: ввод (mode = «r») или вывод (mode = «w»). Любые другие значения, указанные для mode, дают непредсказуемый результат.

В результате выполнения этой функции создается открытый файловый дескриптор канала (pipe), из которого породивший процесс может (mode = «r») читать (стандартный поток вывода дочернего процесса STDOUT_FILENO) или в который может (mode = «w») писать (стандартный поток ввода дочернего процесса STDIN_FILENO) стандартным образом, как это делается для типа FILE (в частности, с отработкой ситуации EOF).

Рассмотрим следующий пример. Конечно, посимвольный ввод/вывод – это не лучшее решение, и здесь мы используем его только для простоты:

```
int main( int argc, char** argv ) {
   FILE* f = popen( "ls -l", "r" );
   if( f == NULL ) perror( "popen" ), exit( EXIT_FAILURE );
   char c;
   while( ( c = fgetc( f ) ) != EOF )
      cout << ( islower( c ) ? toupper( c ) : c );
   pclose( f );
   return EXIT_SUCCESS;
};
```

Примечание

Новый процесс выполняется с тем же окружением, что и родительский. Процесс, указанный в команде, запускается примерно следующим эквивалентом:

```
spawnlp( P_NOWAIT, shell_command, shell_command,
         "-c", command, (char* )NULL );
```

где shell_command – командный интерпретатор, специфицированный переменной окружения SHELL, или утилита /bin/sh. В этом кроется причина возможного различия в выполнении вызовов system() и popen().

Если popen() возвращает не NULL, то выполнение прошло успешно. В противном случае устанавливается errno: EINVAL – недопустимый аргумент mode, ENOSYS – в системе не выполняется программа менеджера каналов. После завершения работы с каналом, созданным popen(), он должен быть закрыт парной операцией pclose().

При использовании system() в более сложных случаях, например при запуске в качестве дочернего собственного процесса, являющегося составной частью комплекса (до сих пор мы рассматривали в качестве дочерних только стандартные программы UNIX), причем запуск производится из отдельного потока (то есть без ожидания завершения, как предлагалось выше), мы можем реализовать сколь угодно изощренные способы взаимодействия с помощью механизмов IPC, например, открывая в дочернем процессе двунаправленные каналы к родителю.

Клонирование процесса

Вызов `fork()` создает клон (полную копию) вызывающего процесса в точке вызова. Вызов `fork()` является одной из самых базовых конструкций всего UNIX-программирования. Его толкованию посвящено столько страниц в литературе, сколько не уделено никакому другому элементу API. Синтаксис этого вызова (проще по синтаксису не бывает, сложнее по семантике – тоже):

```
#include <process.h>
pid_t fork( void );
```

Действие вызова `fork()` следующее:

- Порождается дочерний процесс, которому системой присваивается новое уникальное значение PID.
- Дочерний процесс получает собственные копии файловых дескрипторов, открытых в родительском процессе в точке выполнения `fork()`. Каждый дескриптор ссылается на тот же файл, который соответствует аналогичному дескриптору родителя. Блокировки файлов (locks), установленные в родительском процессе, наследуются дочерним процессом.
- Для дочернего процесса его значения `tms_utime`, `tms_stime`, `tms_cutime` и `tms_cstime` устанавливаются в значение ноль. Выдержки (alarms) для этих таймеров, установленные к этому времени в родительском процессе, в дочернем процессе очищаются.
- Сигнальные маски (подробнее об этом будет рассказано ниже) для дочернего процесса инициализируются пустыми сигнальными наборами (независимо от сигнальных масок, установленных родительским процессом).

Если вызов функции завершился неудачно, функция возвращает –1 и устанавливает `errno`: `EAGAIN` – недостаточно системных ресурсов; `ENOMEM` – процессы требуют большее количество памяти, чем доступно в системе; `ENOSYS` – функция `fork()` не реализуется в этой модели памяти, например в физической модели адресации памяти (напомним, что QNX – многоплатформенная ОС и число поддерживаемых ею платформ все возрастает).

А вот с кодом возврата этой функции в случае удачи сложнее и гораздо интереснее. Дело в том, что для одного вызова `fork()` одновременно имеют место два возврата в двух различных копиях (но в текстуально едином коде!): в копии кода, соответствующей дочернему процессу, возвращается 0, а в копии кода родителя – PID успешно порожденного дочернего процесса. Это и позволяет разграничить в едином программном коде фрагменты, которые после точки ветвления надлежит выполнять в родительском процессе и которые относятся к дочернему процессу. Типичный шаблон кода, использующего `fork()`, выглядит примерно так:

```
pid_t pid = fork();
if( pid == -1 ) perror( "fork" ), exit( EXIT_FAILURE );
if( pid == 0 ) {
   // ... этот код выполняется в дочернем процессе ...
   exit( EXIT_SUCCES );
};
if( pid > 0 ) {
   // ... этот код выполняется в родительском процессе ...
   do {  // ожидание завершения порожденного процесса:
      wpid = waitpid( pid, &status, 0 );
   } while( WIFEXITED( status ) == 0 );
   exit( WEXITSTATUS( status ) );
};
```

Эта схема порождения процесса, его клонирование, настолько широко употребляется, особенно при построении самых разнообразных серверов, что для нее была создана специальная техника, построенная на вызове `fork()`. Заметьте, что во всех многозадачных ОС обязательно присутствует та или иная техника программного создания нового процесса, однако не во всех существует техника именно клонирования, то есть создания полного дубликата порождающего процесса.

Вот как выглядит простейший ретранслирующий TCP/IP-сервер, заимствованный из нашей более ранней публикации [4] (обработка ошибок полностью исключена, чтобы упростить пример):

Ретранслирующий TCP/IP-сервер[1]

```
int main( int argc, char* argv[] ) {
   // создание и подготовка прослушивающего сокета:
   int rc, ls = socket( AF_INET, SOCK_STREAM, 0 );
   setsockopt( ls, SOL_SOCKET, SO_REUSEADDR, &rc, sizeof( rc ) )
   struct sockaddr_in addr;
   memset( &addr, 0, sizeof( addr ) );
   addr.sin_len = sizeof( addr );     // специфика QNX
   addr.sin_family = AF_INET;
   addr.sin_port = htons( PORT );     // PORT - константа
   addr.sin_addr.s_addr = htonl( INADDR_ANY );
   bind( ls, (struct sockaddr*)&addr, sizeof( sockaddr ) );
   listen( ls, 25 );
   while( true ) {
      rc = accept( ls, NULL, NULL );
      pid_t pid = fork();
      if( pid < 0 ) ... ;             // что-то произошло!
      if( pid == 0 ) {
```

[1] Напоминаем, что листинги, названия которых выделены подобным образом (на сером фоне), представляют собой законченные приложения. Соответствующие файлы можно найти в архивах; они могут быть воспроизведены или модифицированы для тонкого анализа результатов.

```
            close( ls );
            char data[ MAXLINE ];
            int nd = read( rc, &data, MAXLINE );
            if( nd > 0 ) write( rc, &data, nd );
            close( rs );
            exit( EXIT_SUCCESS );
          }
          else close( rs );  // единственное действие родителя
        };
        exit( EXIT_SUCCESS );
    };
```

Приведенный фрагмент может в процессе своей работы породить достаточно много идентичных процессов (один родительский, пассивно прослушивающий канал; остальные – порожденные, активно взаимодействующие с клиентами, по одному на каждого клиента). Все порождаемые процессы наследуют весь набор дескрипторов (в данном случае сокетов), доступных родительскому процессу. Лучшее, что могут сделать процессы (как родительский, так и дочерний), – немедленно после вызова `fork()` (и это хорошая практика в общем случае) тщательно закрыть все унаследованные дескрипторы, не имеющие отношения к их работе.

Примечание

Операция `fork()` должна создать не только структуру адресного пространства нового процесса, но и побайтную копию этой области. В операционных системах общего назначения (Win32, Linux, FreeBSD) для облегчения этого трудоемкого процесса используется виртуализация страниц по технологии COW (copy on write), детально описанная, например, применительно к Win32, Джеффри Рихтером. Накладные расходы процесса копирования здесь демпфированы тем, что копирование каждой физической страницы памяти фактически производится только при записи в эту страницу, то есть затраты на копирование «размазываются» достаточно случайным образом по ходу последующего выполнения дочернего процесса (здесь нет практически никакого итогового выигрыша в производительности, есть только сокрытие от пользователя одноразового размера требуемых затрат).

Системы реального времени не имеют права на такую роскошь: непредсказуемое рассредоточение копирующих операций по всему последующему выполнению, а поэтому и использование в них COW вообще, выглядит весьма сомнительно. В [4] мы описывали эксперименты в QNX, когда в код сервера, построенного на `fork()`, была внесена «пассивная» строка, никак не используемая в программе, но определяющая весьма протяженную инициализированную область данных:

```
static long MEM[ 2500000 ];
```

При этом время реакции (ответа) сервера (затраты времени на выполнение `fork()`) возросло в 50 раз и составило 0,12 сек на процессоре 400 Мгц. Еще раз, но в другом контексте эта особенность будет обсуждена ниже при сравнении затрат производительности на создание процессов и потоков.

Дополнительным вызовом этого класса (для полноты обзора) является использование функции:

```
pid_t vfork( void );
```

В отличие от `fork()`, этот вызов, впервые введенный в BSD UNIX, делает разделяемым для дочернего процесса адресное пространство родителя. Родительский процесс приостанавливается до тех пор, пока порожденный процесс не выполнит `exec()` (загружая новый программный код дочернего процесса) или не завершится с помощью `exit()` или аналогичных средств.

Функция `vfork()` может быть реализована на аппаратных платформах с физической моделью памяти (без виртуализации памяти), а `fork()` – не может (или реализуется с большими накладными расходами), так как требует создания абсолютной копии области адресного пространства, что в физической модели повлечет сложную динамическую модификацию адресных полей кода. Тем не менее (при некоторых кажущихся достоинствах) в BSD также подчеркивалось, что `vfork()` таит в себе серьезную потенциальную опасность, поскольку позволяет одному процессу использовать или даже модифицировать адресное пространство другого, то есть фактически нарушает парадигму защищенных адресных пространств.

Запуск нового программного кода

Наконец, рассмотрим запуск на выполнение нового, отличного от родительского процесса программного кода, образ которого содержится в отдельном исполняемом файле в качестве дочернего процесса. Для этой цели в QNX существуют две группы функций: `exec()` (их всего 8: `execl()`, `execle()`, `execlp()`, `execlpe()`, `execv()`, `execve()`, `execvp()`, `execvpe()`) и `spawn()` (их 10: `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnv()`, `spawnve()`, `spawnvp()`, `spawnvpe()`).

Это множество форм записи отличается синтаксисом, который определяет формат списка аргументов командной строки, полученного нами в качестве параметров функции `main()`, передаваемых программе, а также некоторыми другими дополнительными деталями. Суффиксы в именах функций обозначают следующее:

- `l` – список аргументов определяется через список параметров, заданных непосредственно в самом вызове. Этот список завершается нулевым аргументом `NULL`;
- `e` – окружение для процесса указывается посредством определения массива переменных окружения;
- `p` – относительный путь поиска: если не указан полный путь к файлу программы (то есть имя файла не содержит разделителей «/»), для его поиска используется переменная окружения `PATH`;
- `v` – список аргументов определяется через указатель на массив аргументов.

В нашу задачу не входит описание всех возможностей вызовов, тем более что они обстоятельно описаны в [1, 2, 5, 6], и мы будем использовать по тексту любую, более удобную для нас форму без дополнительных объяснений.

Большинство форм функции exec() являются POSIX-совместимыми, а большая часть форм функции spawn() представляет собой специфическое расширение QNX. Более того, даже для тех функций группы spawn(), которые часто называют POSIX-совместимыми [1], техническая документация QNX определяет степень совместимости примерно в таких терминах: «*...функция spawn( ) является функцией QNX Neutrino (основанной на POSIX 1003.1d черновом стандарте).*»

Функции семейства exec(), напротив, подменяют исполняемый код текущего процесса (не изменяя его идентификатор PID, права доступа, внешние ресурсы процесса, а также находящийся в том же адресном пространстве) исполняемым кодом из другого файла. Поэтому используются эти вызовы непосредственно после fork() для замены копии вызывающего процесса новым (это классическая UNIX-технология использования).

Функции семейства spawn(), напротив, порождают новый процесс (с новым идентификатором PID и в новом адресном пространстве). Все формы вызовов spawn() после подготовительной работы (иногда очень значительной) в конечном итоге ретранслируются в вызов базовой формы spawn()[1], который последним действием своего выполнения и посылает сообщение procnto (менеджер процессов QNX, «территориально» объединенный с микроядром системы в одном файле).

Базовый вызов spawn() определяется следующим образом:

```
#include <spawn.h>
pid_t spawn( const char* path,
             int fd_count,
             const int fd_map[ ],
             const struct inheritance* inherit,
             char* const argv[ ],
             char* const envp[ ] );
```

где path – полное имя исполняемого бинарного файла;

fd_count – размерность следующего за ним массива fd_map;

fd_map – массив файловых дескрипторов, которые вы хотели бы унаследовать в дочернем процессе от родительского. Если fd_count не равен 0 (то есть может иметь значения вплоть до константы OPEN_MAX), то fd_map должен содержать список из fd_count файловых дескрипторов. Если же fd_count равен 0, то дочерний процесс наследует все родитель-

[1] Тем не менее это вовсе не означает, что следует непосредственно использовать вызов spawn(), ведь он самый трудоемкий и чреват ошибками.

ские дескрипторы, исключая те, которые созданы с флагом FD_CLOEXEC функции fcntl();

inherit – системная структура (см. системные определения) типа struct inheritance, содержащая как минимум:

unsigned long flags – один или более установленных бит:

SPAWN_CHECK_SCRIPT – позволить spawn() запускать требуемый командный интерпретатор, интерпретируя path как скрипт (интерпретатор указан в первой строке скрипта path);

SPAWN_SEARCH_PATH – использовать переменную окружения PATH для поиска выполняемого файла path;

SPAWN_SETGROUP – установить для дочернего процесса значение группы, специфицируемое членом (структуры) pgroup. Если этот флаг не установлен, дочерний процесс будет частью текущей группы родительского процесса;

SPAWN_SETND – запустить дочерний процесс на удаленном сетевом узле QNET, сам же удаленный узел специфицируется членом (структуры) nd (см. команду удаленного запуска on);

SPAWN_SETSIGDEF – использовать структуру sigdefault для определения процесса множества (набора) сигналов, для которых будет установлена реакция по умолчанию. Если этот флаг не установлен, дочерний процесс наследует все сигнальные реакции родителя;

SPAWN_SETSIGMASK – использовать sigmask в качестве сигнальной маски дочернего процесса.

pid_t pgroup – группа дочернего процесса; имеет смысл, только если установлен флаг SPAWN_SETGROUP. Если флаг SPAWN_SETGROUP установлен и inherit.pgroup установлен как SPAWN_NEWPGROUP, то дочерний процесс открывает новую группу процессов с идентификатором группы (GID), равным PID этого нового процесса.

sigset_t sigmask – сигнальная маска дочернего процесса, если установлен флаг SPAWN_SETSIGMASK.

sigset_t sigdefault – набор сигналов дочернего процесса, для которых определяется реакция по умолчанию, если установлен флаг SPAWN_SETSIGDEF.

uint32_t nd – это совершенно уникальный (относительно других ОС, а значит, и всего POSIX) параметр QNX – дескриптор узла сети QNET, на котором должен быть запущен новый процесс. Это поле используется, только если установлен флаг SPAWN_SETND.

argv – указатель массива аргументов. Значение argv[0] должно быть строкой (char*), содержащей имя файла, загружаемого как процесс (но может быть NULL, если аргументы не передаются). Последний элемент массива argv обязан быть NULL. Само значение argv никогда не может быть NULL.

envp – указатель массива символьных строк переменных системного окружения (environment). Последний элемент массива envp обязан быть NULL. Каждый элемент массива является строкой (char*) вида: variable = value. Если само значение указателя envp равно NULL, то дочерний процесс полностью наследует копию окружения родителя. (Окружение процесса – всегда «копия», поэтому любые изменения, внесенные в окружение дочерним процессом, никак не отражаются на окружении его родителя.)

Примечание

Если дочерний процесс является скриптом интерпретатора (флаг SPAWN_CHECK_SCRIPT), то первая строка текста скрипта должна начинаться с #!, за которыми должны следовать путь и аргументы того интерпретатора, который будет использоваться для интерпретации этого скрипта. К скрипту не применяется установленный в системе интерпретатор по умолчанию (как это происходит при вызове его по имени из командной строки).

Правила наследования (и ненаследования) параметров дочернего процесса от родителя (RID, RGID и других атрибутов) жестко регламентированы, достаточно сложны (в зависимости от флагов) и могут быть уточнены в технической документации QNX. Отметим, что безусловно наследуются такие параметры, как: а) приоритет и дисциплина диспетчеризации; б) рабочий и корневой каталоги файловой системы. Не наследуются: установки таймеров процесса tms_utime, tms_stime, tms_cutime и tms_cstime, значение взведенного сигнала SIGALRM (это значение сбрасывается в ноль), файловые блокировки, блокировки и отображения памяти (shared memory), установленные родителем.

При успешном завершении вызов функции возвращает PID порожденного процесса. При неудаче возвращается –1 и errno устанавливается:

- E2BIG – количество байт, заданное в списке аргументов или переменных окружения и превышающее ARG_MAX;
- EACCESS – нет права поиска в каталогах префикса имени файла, или для файла не установлены права на выполнение, или файловая система по указанному пути была смонтирована с флагом ST_NOEXEC;
- EAGAIN – недостаточно системных ресурсов для порождения процесса;
- EBADF – недопустим хотя бы один из файловых дескрипторов в массиве fd_map;
- EFAULT – недопустима одна из буферных областей, указанных в вызове;
- ELOOP – слишком глубокий уровень символических ссылок к файлу или глубина префиксов (каталогов) в полном пути к файлу;
- EMFILE – недостаточно ресурсов для отображения файловых дескрипторов в дочерний процесс;
- ENAMETOOLONG – длина полного пути превышает PATH_MAX или длина компонента имени файла в пути превышает NAME_MAX;

- ENOENT – файл нулевой длины или несуществующий префиксный компонент в полном пути;
- ENOEXEC – файл, указанный как программа, имеет ошибочный для исполняемого файла формат;
- ENOMEM – в системе недостаточно свободной памяти для порождения процесса;
- ENOSYS – файловая система, специфицированная полным путевым именем файла, не предназначена для выполнения spawn();
- ENOTDIR – префиксные компоненты пути исполняемого файла не являются каталогами;

Даже из этого очень краткого обзора вызова spawn() становятся очевидными некоторые вещи:

- Эта форма универсальна (самодостаточна), она позволяет обеспечить весь спектр разнообразных форм порождения нового процесса.
- Она же и самая громоздкая форма, тяжеловесная для практического кодирования, поэтому в реальных текстах в большинстве случаев вы вместо нее встретите ее конкретизации: spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(). Все эти формы достаточно полно описаны в [1]. Функционально они эквивалентны spawn(), поэтому мы не станем на них детально останавливаться.
- Хотя вызов spawn() и упоминается в описаниях как POSIX-совместимый, в QNX он существенно расширен и модифицирован и поэтому в лучшем случае может квалифицироваться как «выполненный по мотивам» POSIX.

В качестве примера приведем использованную в [4] (глава Д. Алексеева «Утилита on») форму вызова для запуска программы (с именем, заданным в строке command) на удаленном узле node (например, /net/xxx) сети QNET (как вы понимаете, это совершенно уникальная возможность QNX, говорить о которой в рамках POSIX-совместимости просто бессмысленно):

```
int main ( ) {
   char* command = "...", *node = "...";
   // параметры запуска не используются
   char* const argv[] = { NULL };
   struct inheritance inh;
   inh.flags = 0;
   // флаг удаленного запуска
   inh.flags |= SPAWN_SETND;
   // дескриптор хоста
   inh.nd = netmgr_strtond( node, NULL );
   pid_t pid = spawnp( command, 0, NULL, &inh, argv, NULL );
   ...
};
```

Использованная здесь форма spawnp() наиболее близка к базовой spawn() и отличается лишь тем, что для поиска файла программы используется переменная системного окружения PATH.

Приведем характерный пример вызова группы exec*():

```
int execl( const char* path,
           const char* arg0,
           const char* arg1,
           ...
           const char* argn,
           NULL );
```

где path – путевое имя исполняемого файла; arg0, ..., argn – символьные строки, доступные процессу как список аргументов. Список аргументов должен завершаться значением NULL. Аргумент arg0 должен быть именем файла, ассоциированного с запускаемым процессом.

Примечание

Устоявшаяся терминология «запускаемый процесс» относительно exec*() явно неудачна и лишь вводит в заблуждение. Здесь гораздо уместнее говорить о замещении выполнявшегося до этой точки кода новым, выполнение которого начинается с точки входа главного потока замещающего процесса.

Примечание

Если вызов exec*() выполняется из многопоточного родительского процесса, то все выполняющиеся потоки этого процесса предварительно завершаются. Никакие функции деструкторов для них не выполняются.

Если вызов exec*() успешен, управление никогда уже не возвращается в точку вызова. В случае неудачи возвращается –1 и errno устанавливается так же, как описано выше для spawn().

В качестве примера работы вызова spawn*() (использование exec() аналогично) рассмотрим приложение (*файлы p1.cc, p1ch.cc*), в котором:

- Родительский процесс (*p1*) порождает дочерний (*p1ch*) и ожидает от него поступления сигнала SIGUSR1 (сигналы детально обсуждаются позже, но здесь попутно «вскроем» одну из их особенностей).
- Дочерний процесс периодически посылает родителю сигнал SIGUSR1.
- Родительский процесс может переустановить (с помощью параметров командной строки запуска) для дочернего: период посылки сигнала (1-й параметр задан в нашем приложении константой) и приоритет, с которым будет выполняться дочерний процесс (2-й параметр, в качестве которого ретранслируется единственный параметр команды запуска родителя).

Примечание

В данный момент нас интересует только то приложение, в котором дочерний процесс порождается вызовом `spawnl()`. Используемые приложением механизмы и понятия – сигналы UNIX, приоритеты, наследование и инверсия приоритетов – будут рассмотрены позже, поэтому при первом чтении их можно опустить. Нам не хотелось перегружать текст дополнительными «пустыми» примерами, лишь иллюстрирующими применение одной функции. Это приложение, созданное «на будущее», позволит нам отследить крайне актуальный для систем реального времени вопрос о наличии (или отсутствии) наследования приоритетов при посылке сигналов (допустимо как одно, так и другое решение, но оно должно быть однозначно единственным для ОС).

Итак, родительское приложение (*файл p1.cc*):

Сигналы и наследование приоритетов

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <signal.h>
#include <unistd.h>
#include <sched.h>

// обработчик сигнала:
static void handler( int signo, siginfo_t* info, void* context ) {
   int oldprio = getprio( 0 );
   setprio( 0, info->si_value.sival_int );
   cout << "SIG = " << signo << ": old priority = "
        << oldprio << ", new priority = " << getprio( 0 )
        << endl;
   setprio( 0, oldprio );
};

int main( int argc, char* argv[] ) {
   // установить обработчик сигнала:
   sigset_t sig;
   sigemptyset( &sig );
   //определение: #define SIGUSR1 16
   sigaddset( &sig, SIGUSR1 );
   sigprocmask( SIG_BLOCK, &sig, NULL );
   struct sigaction act;
   act.sa_mask = sig;
   act.sa_sigaction = handler;
   act.sa_flags = SA_SIGINFO;
   if( sigaction( SIGUSR1, &act, NULL ) < 0 )
      perror( "set signal handler" ), exit( EXIT_FAILURE );
   // создать новый (дочерний) процесс:
   const char* prg = "./p1ch", *sdelay = "3";
   pid_t pid =
```

```
      ( ( argc > 1 ) &&
        ( atoi( argv[ 1 ] ) >=
             sched_get_priority_min( SCHED_RR ) ) &&
        ( atoi( argv[ 1 ] ) <=
             sched_get_priority_max( SCHED_RR ) ) ) ?
      spawnl( P_NOWAIT, prg, prg, sdelay, argv[ 1 ], NULL )
      :
      spawnl( P_NOWAIT, prg, prg, sdelay, NULL );
   if( pid == -1 )
      perror( "spawn child process" ), exit( EXIT_FAILURE );
   // размаскировать и ожидать сигнала:
   sigprocmask( SIG_UNBLOCK, &sig, NULL );
   while( true ) {
      if( sleep( 3 ) != 0 ) continue;
      cout << "parent main loop: priority = "
           << getprio( 0 ) << endl;
   };
};
```

Дочернее приложение (*файл p1ch.cc*), которое и будет запускать показанный выше родительский процесс:

```
#include <stdio.h>
#include <iostream.h>
#include <sched.h>
#include <unistd.h>
#include <signal.h>

int main ( int argc, char *argv[] ) {
   int val, del = 5;
   if( ( argc > 1 ) &&
       ( sscanf( argv[ 1 ], "%i", &val ) == 1 ) &&
       ( val > 0 ) ) del = val;
   if( ( argc > 2 ) &&
       ( sscanf( argv[ 2 ], "%i", &val ) == 1 ) &&
       ( val > 0 ) &&
       ( val <= sched_get_priority_max( SCHED_RR ) ) )
      if( setprio( 0, val ) == -1 )
         perror( "set priority" );
   // переодически уведомлять родителя SIGUSR1, используя
   // его как сигнал реального времени (с очередью):
   while( true ) {
      sleep( del );
      union sigval val;
      val.sival_int = getprio( 0 );
      // #define SIGUSR1     16
      sigqueue( getppid(), SIGUSR1, val );
   };
};
```

Примечание

Для многих сигналов действием на их получение, предопределенным по умолчанию, является завершение процесса. (Реже встречается действие по умолчанию – игнорировать полученный сигнал при отсутствии явно установленной для него функции обработчика.) Достаточно странно, что завершение процесса предусмотрено как реакция по умолчанию на получение «пользовательских» сигналов SIGUSR1 и SIGUSR2. Если показанное выше приложение в процессе отладки запустить вызовом из командной строки (из командного интерпретатора или, например, файлового менеджера mqc), то результатом (на первый взгляд не столь ожидаемым) станет завершение интерпретатора командной строки (родительского процесса) и, как следствие, самого тестируемого приложения.

Вот как выглядит начальный участок совместной работы двух процессов:

```
# p1 15
parent main loop: priority = 10
SIG = 16: old priority = 10, new priority = 15
SIG = 16: old priority = 10, new priority = 15
parent main loop: priority = 10
SIG = 16: old priority = 10, new priority = 15
parent main loop: priority = 10
SIG = 16: old priority = 10, new priority = 15
parent main loop: priority = 10
SIG = 16: old priority = 10, new priority = 15
parent main loop: priority = 10
SIG = 16: old priority = 10, new priority = 15
parent main loop: priority = 10
```

Отчетливо видно, что при посылке сигналов реального времени наследование приоритета посылающего процесса не происходит (дочернее приложение, посылающее сигнал, выполняется с приоритетом 15, а обработчик полученного сигнала в родительском процессе выполняется с приоритетом по умолчанию, равным 10).

Забегая вперед, сообщим, что в приведенном коде приложения сделано жалкое подобие имитации наследования приоритета: в качестве ассоциированного с сигналом реального времени значения передается значение приоритета отправителя, которое тут же устанавливается как приоритет для выполнения кода обработчика. Однако слабость в отношении истинного наследования состоит здесь в том, что два первых оператора (сохранение и установка приоритета) выполняются под приоритетом родителя, и в это время обработчик может быть вытеснен диспетчером системы.

Завершение процесса

С завершением процесса дело обстоит достаточно просто, по крайней мере, в сравнении с тем, что происходит при завершении потока, как

это и будет показано очень скоро. Процесс завершается, если программа выполняет вызов `exit()` или выполнение просто доходит до точки завершения функции `main()`, будь то с явным указанием оператора `return` или без оного. Это естественный, внутренний (из программного кода самого процесса) путь завершения.

Другой путь – посылка процессу извне (из другого процесса) сигнала, реакцией на который (предопределенной или установленной) является завершение процесса (подробнее о сигналах и реакциях см. ниже). В противовес естественному завершению такое принудительное завершение извне в [12] (по крайней мере, в отношении потоков) названо **отменой**, и именно этим термином мы будем пользоваться далее, чтобы отчетливо отмечать, о каком варианте завершения идет речь. (Такая же терминология будет использоваться нами и относительно завершения потока.)

Здесь уместно сделать краткое отступление относительно «живучести», как это названо у У. Стивенса [2], или времени жизни объектов IPC, что в равной мере может быть отнесено не только к объектам IPC, но и ко всем прочим объектам операционной системы. У. Стивенс делит все объекты по времени жизни на:

- Объекты, время жизни которых определяется процессом (process-persistent). Такой объект существует до тех пор, пока не будет закрыт последним процессом, который его использует. Примерами такого объекта являются неименованные и именованные программные каналы (pipes, FIFO).
- Объекты, время жизни которых определяется ядром системы (kernel-persistent). Такой объект существует до перезагрузки ядра или явного удаления объекта. Примерами этого класса объектов являются семафоры (именованные) и разделяемая память.
- Объекты, время жизни которых определяется файловой системой (filesystem-persistent). Такой объект отображается на файловую систему и существует до тех пор, пока не будет явно удален. Примерами этого класса объектов в различных ОС в зависимости от реализации могут быть очереди сообщений POSIX, семафоры и разделяемая память.

Квалификация каждого из объектов по времени жизни отнюдь не тривиальная задача. Объекты, отнесенные к одному классу, мигрируют в другой при переходе от одной ОС к другой в зависимости от деталей их реализации.

Проблемы завершения и особенно отмены процесса могут возникать, если процесс оперирует с объектами, время жизни которых превышает process-persistent. Мы еще много раз коснемся этой проблемы при рассмотрении завершения потоков, так как там она может возникать и в отношении всех process-persistent-объектов, и для ее разрешения

в технике потоков даже предложены специальные технологии, о которых мы детально поговорим далее, при рассмотрении потоков.

Соображения производительности

Интересны не только затраты на порождение нового процесса (мы еще будем к ним неоднократно возвращаться), но и то, насколько «эффективно» сосуществуют параллельные процессы в ОС, насколько быстро происходит переключение контекста с одного процесса на другой. Для самой грубой оценки этих затрат создадим простейшее приложение (*файл p5.cc*):

Затраты на взаимное переключение процессов

```
#include <stdlib.h>
#include <inttypes.h>
#include <iostream.h>
#include <unistd.h>
#include <sched.h>
#include <sys/neutrino.h>

int main( int argc, char* argv[] ) {
   unsigned long N = 1000;
   if( argc > 1 && atoi( argv[ 1 ] ) > 0 )
      N = atoi( argv[ 1 ] );
   pid_t pid = fork();
   if( pid == -1 )
      cout << "fork error" << endl, exit( EXIT_FAILURE );
   uint64_t t = ClockCycles();
   for( unsigned long i = 0; i < N; i++ ) sched_yield();
   t = ClockCycles() - t;
   delay( 200 );
   cout << pid << "\t: cycles - " << t
        << "; on sched - " << ( t / N ) / 2 << endl;
   exit( EXIT_SUCCESS );
};
```

Два одновременно выполняющихся процесса настолько симметричны и идентичны, что они даже не анализируют PID после выполнения `fork()`, они только в максимальном темпе «перепасовывают» друг другу активность, как волейболисты делают это с мячом (рис. 2.2).

Рисунок 2.2 иллюстрирует взаимодействие двух идентичных процессов: вся их «работа» состоит лишь в том, чтобы как можно быстрее передать управление партнеру. Такую схему, когда два и более как можно более идентичных потоков или процессов в максимально высоком темпе (на порядок превосходящем последовательность «естественной» RR-диспетчеризации) обмениваются активностью, мы будем неоднократно использовать в дальнейшем для различных механизмов, называя ее для простоты «симметричной схемой».

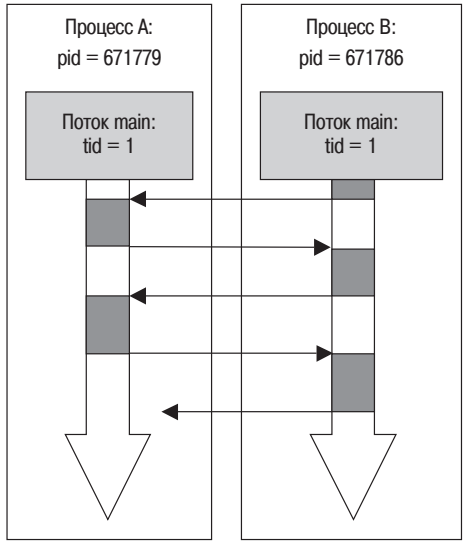

Рис. 2.2. *Симметричное взаимодействие потоков*

Примечание

Чтобы максимально упростить код приложения, при его написании мы не трогали события «естественной» диспетчеризации, имеющие место при RR-диспетчеризации каждые 4 системных тика (по умолчанию это ~4 миллисекунды). Как сейчас покажут результаты, события принудительной диспетчеризации происходят с периодичностью порядка 1 микросекунды, т. е. в 4000 раз чаще, и возмущения, возможно вносимые RR-диспетчеризацией, можно считать не настолько существенными.

Вот результаты выполнения этой программы:

```
# nice -n-19 p5 1000000
1069102 : cycles - 1234175656; on sched - 617
0       : cycles - 1234176052; on sched - 617
# nice -n-19 p5 100000
1003566 : cycles - 123439225; on sched - 617
0       : cycles - 123440347; on sched - 617
# nice -n-19 p5 10000
1019950 : cycles - 12339084; on sched - 616
0       : cycles - 12341520; on sched - 617
# nice -n-19 p5 1000
1036334 : cycles - 1243117; on sched - 621
0       : cycles - 1245123; on sched - 622
# nice -n-19 p5 100
1052718 : cycles - 130740; on sched - 653
0       : cycles - 132615; on sched - 663
```

Видна на удивление устойчивая оценка, практически не зависящая от общего числа актов диспетчеризации, изменяющегося на 4 порядка.

Отбросив мелкие добавки, привносимые инкрементом и проверкой счетчика цикла, можно считать, что передача управления от процесса к процессу требует порядка 600 циклов процессора (это порядка 1,2 микросекунды на компьютере 533 Мгц, на котором выполнялся этот тест).

Потоки

Последующие расширения[1] POSIX специфицируют широкий спектр механизмов «легких процессов» – потоков (группа API `pthread_*()`). Техника потоков вводит новую парадигму программирования вместо уже ставших традиционными UNIX-методов. Это обстоятельство часто недооценивается. Например, использование `pthread_create()` вместо `fork()` может на порядки повысить скорость реакций, особенно в ОС с отсутствием механизмов COW (copy on write) при создании дубликатов физических страниц RAM сегментов данных (таких как QNX, хотя механизмы COW вряд ли вообще применимы в ОС реального времени) [4]. Другой пример: использование множественных потоков вместо ожиданий на множестве дескрипторов в операторе `select()`.

Однако очень часто эти две парадигмы, традиционная и потоковая, не сочетаются в рамках единого кода из-за небезопасности (not thread safe) традиционных механизмов UNIX (`fork()`, `select()` и др.) в многопоточной среде. Тогда приходится использовать либо одну, либо другую парадигму как альтернативы, не смешивая их между собой. Или смешивать, но с большой осторожностью и с хорошим пониманием того, что при этом может произойти в каждом случае.

Поток можно понимать как любой автономный последовательный (линейный) набор команд процессора. Источником этого линейного кода для потока могут служить:

- бинарный исполняемый файл, на основе которого системой или вызовом группы `spawn()` запускается новый процесс и создается его **главный** поток;
- дубликат кода главного потока[2] процесса родителя при клонировании процессов вызовом `fork()` (тоже относительно **главного** потока);
- участок кода, оформленный функцией специального типа (`void* () ( void* )`); это общий случай при создании второго и всех последую-

---

[1] Часто в публикациях ссылаются на расширения реального времени POSIX 1003b (1993). Но POSIX 1003b не описывают группу `pthread_*`, хотя именно в этой редакции определены семафоры `sem_t`. У. Стивенс [2] указывает, что программные потоки POSIX определены в редакции 1003.1 (1995).

[2] Клонирование многопоточных процессов с помощью `fork()` – это отдельная песня. Хотя POSIX и предусматривает реализацию (`pthread_atfork()`) такой возможности, до конца неясно, как это работает. API QNX предоставляет эту возможность, но предупреждает, что пользователь сам отвечает за последствия. Детали механизма `pthread_atfork()` см. в справочном руководстве QNX.

щих потоков процесса (при создании многопоточных процессов) вызовом `pthread_create()`. Такую функцию мы будем называть функцией потока. Это наиболее интересный для нас случай.

В первых двух вариантах мы имеем неявное создание (главного) потока и, как следствие, порождение нового процесса. В последнем случае – явное создание потока, которое в литературе, собственно, и именуется «созданием потока». Хотя сущность происходящего относительно исполняющегося потока во всех случаях все же остается неизменной.

Кроме последовательности команд к потоку нужно отнести и те локальные данные, с которыми работает функция потока, то есть собственный стек потока. Во время приостановки системой выполнения (диспетчеризации) кода текущего потока должна обеспечиваться возможность сохранения текущих значений регистров (включая регистры FPU, сегментные регистры) и, возможно, другой специфической информации. Текущее значение этого набора данных, относящихся к выполнению текущего потока, называется контекстом потока. Контекст потока, кроме того, обеспечивает связь потока с его экземпляром собственных данных, о чем мы детально поговорим чуть позже. Детальная структура и объем данных, составляющих контекст потока, определяются не только самой ОС, но и типом процессорной архитектуры, на которой она выполняется (для многоплатформенных ОС, к которым принадлежит и QNX).

В принципе считается, что время переключения контекстов потоков в пределах одного процесса и время переключения контекстов процессов могут заметно отличаться, особенно для платформ с управлением виртуальной памятью.[1] Однако удобства реализации и стремление к однородности могут перевесить соблазн разработчиков ОС использовать это различие, что мы вскоре и увидим в отношении QNX.

Идентификатором потока (значимым только внутри одного процесса!) является TID (Thread ID), присваиваемый потоку при его создании вызовом `pthread_create()`. TID позволяет процессу (а также системе в рамках процесса) однозначно идентифицировать каждый поток. Нумерация TID в QNX начинается с 1 (это всегда главный поток процесса, порожденный `main()`) и последовательно возрастает по мере создания потоков (до 32767).[2]

---

[1] Собственно с этим и связано употребление применительно к потокам названия «легкие процессы». Впервые этот термин (LWP – lightweight process) ввела в своей технической документации для обозначения понятия, эквивалентного потоку, фирма Sun Microsystems.

[2] Эта схема PID/TID описана в POSIX, но выполняется далеко не во всех UNIX-совместимых ОС. Например, вплоть до самых последних редакций ядра Linux (ситуация стала меняться только сейчас) процессы (`fork()`) и потоки (`pthread_create()`) создавались на базе единого системного вызова (`_clone()`) и TID являлись идентификаторами в едином ряду PID. Это может привести к трудно выявляемым ошибкам при переносе программ между двумя ОС.

Еще одним важнейшим атрибутом потока является приоритет его выполнения. Для каждого из уровней приоритетов, обслуживаемых системой (в QNX 6.2.1 таких уровней 64, в QNX 6.3 – 256), поддерживается **циклическая** очередь потоков, готовых к исполнению (на деле большая часть из таких очередей оказывается пустой). Все политики диспетчеризации работают только с потоками из одной такой очереди: очереди потоков наивысшего из присутствующих в системе приоритетов. Если в системе выполняется поток высокого приоритета, то ни один поток более низкого приоритета не получит управление до тех пор, пока поток высокого приоритета не будет переведен в блокированное состояние в ожидании некоторого события (рис. 2.3).

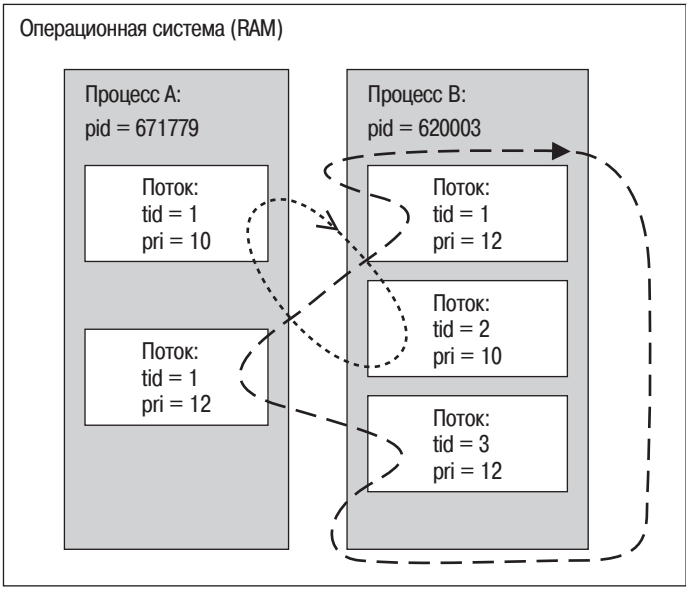

Рис. 2.3. *Диспетчеризация потоков с различными приоритетами*

На рис. 2.3 представлены два процесса, каждый из которых создает внутри себя несколько потоков, но на этот раз различных приоритетов (10 и 12). Жирной пунктирной линией показан порядок, в котором потоки высокого приоритета (12) объединены в циклическую очередь диспетчеризации. Это активная очередь диспетчеризации (наивысшего приоритета). Тонкой линией показан порядок потоков в другой очереди (приоритета 10). До тех пор пока **все** потоки активной очереди не окажутся в силу каких-либо обстоятельств в блокированном состоянии, ни один из потоков очереди приоритета 10 не получит ни единого кванта времени.

Создание нового потока

Создание нового потока в программном коде осуществляет вызов:

```
int pthread_create( pthread_t* thread,
                    const pthread_attr_t* attr,
                    void* (*start_routine)(void* ),
                    void* arg );
```

где thread — NULL или указатель переменной типа pthread_t, значение которой будет загружено идентификатором созданного потока после успешного выполнения функции. Далее это значение (это и есть TID) может использоваться по тексту программы для идентификации созданного потока.

attr — NULL или указатель структуры типа pthread_attr_t. Если это значение NULL, то созданный поток будет иметь набор параметров, устанавливаемых по умолчанию. Если нет, то поток будет создан с параметрами, установленными в структуре attr. Модификация полей attr после создания потока (то есть после вызова функции) не оказывает никакого эффекта на параметры потока, и вообще говоря, структура attr может быть уничтожена сразу же после вызова pthread_create(). Документация предостерегает от прямой манипуляции значениями полей этой структуры, предлагая использовать для этого функции pthread_attr_init() и pthread_attr_set_*().

start_routine — функция типа void* ()(void*), уже упоминавшаяся выше как функция потока; это тот код, который будет фактически выполняться в качестве отдельного потока. Если выполнение этой функции завершается по return, то происходит нормальное завершение потока с вызовом pthread_exit(), использующим значение, возвращаемое start_routine в качестве **статуса завершения**. (Исключением является поток, связанный с main(); он при завершении выполняет вызов exit().)

arg — указатель на блок данных, передаваемых start_routine в качестве входного параметра. Этот параметр подробно рассмотрен далее.

Чаще всего (однако совершенно необязательно) функция потока start_routine представляет собой бесконечный цикл, в котором выполняются некоторые действия с выходом из цикла в том случае, когда нужно завершить выполнение и уничтожить созданный поток. Выглядит это следующим образом:

```
// функция потока:
void* ThreadProc( void* data ) {
   while( true ) {
      // ... выполняется работа ...
      if( ... ) break;
      // после этого поток нам уже не нужен!
   }
   return NULL;
};
```

После успешного создания нового потока он начинает функционировать «параллельно» с породившим его потоком и другими потоками процесса (если быть совсем точными, то со всеми прочими потоками, существующими в системе, так как в QNX существует только одна стратегия диспетчеризации потоков PTHREAD_SCOPE_SYSTEM, и существует она глобально, на уровне всей системы). При этом после точки выполнения pthread_create() невозможно предсказать, какой поток получит управление: породивший, порожденный или вообще произвольный поток из другого процесса. Это важно учитывать при передаче новому потоку данных и других операциях начальной инициализации параметров внутри созданного потока.

В отличие от создаваемых параллельных процессов, рассмотренных ранее, все потоки, создаваемые в рамках одного процесса, разделяют единое адресное пространство процесса, и поэтому все переменные процесса, находящиеся в области видимости любого потока, доступны этому потоку.

Атрибуты потока

В коде реальных приложений очень часто можно видеть простейшую форму вызова, порождающего новый поток, в следующем виде:

```
pthread_create( NULL, NULL, &thread_func, NULL );
```

И для многих целей такого вызова достаточно, так как созданный поток будет обладать свойствами, предусмотренными по умолчанию (преимущественная часть поведенческих характеристик нового потока наследуется от его родителя). Если же нам нужно создать поток с некоторым специфическим поведением, отличающимся от поведения по умолчанию, нам следует обратиться к атрибутной записи создания потока – второму параметру вызова функции создания.

Атрибутная запись потока должна создаваться и обязательно инициализироваться вызовом pthread_attr_init() до точки порождения потока. Любые последующие изменения атрибутной записи создания потока не производят никаких изменений в поведении потока (хотя некоторые из параметров потока, определяемых атрибутной записью при его создании, могут быть изменены позже, уже в ходе выполнения потока, вызовом соответствующих функций). Таким образом, атрибутная запись потока является чисто инициализирующей структурой и может быть даже уничтожена следующим оператором после порождения этого потока.

Эффект повторной инициализации атрибутной записи не определен. Для ее повторного использования (если требуется переопределение значений параметров) должен быть предварительно выполнен вызов pthread_attr_destroy() с последующей повторной инициализацией структуры (он разрушает атрибутную запись, но без освобождения ее памяти):

```
pthread_attr_t* pattr = new pthread_attr_t;
```

```
for( int i = 0; i < N; i++ ) {
   pthread_attr_init( pattr );
   // ... разнообразные настройки для разных потоков ...
   pthread_create( NULL, pattr, &function, NULL );
   pthread_attr_destroy( pattr );
};
delete pattr;
```

Непосредственно манипулировать с полями атрибутной записи, адресуясь к ним по именам полей, крайне опасно. Для этого предусмотрен широкий спектр функций SET/GET:

```
pthread_attr_getdetachstate()
pthread_attr_setdetachstate()
pthread_attr_getguardsize()
pthread_attr_setguardsize()
pthread_attr_getinheritsched()
pthread_attr_setinheritsched()
pthread_attr_getschedparam()
pthread_attr_setschedparam()
pthread_attr_getschedpolicy()
pthread_attr_setschedpolicy()
pthread_attr_getscope()
pthread_attr_setscope()
pthread_attr_getstackaddr()
pthread_attr_setstackaddr()
pthread_attr_getstacklazy()
pthread_attr_setstacklazy()
pthread_attr_getstacksize()
pthread_attr_setstacksize()
```

Мы не станем подробно описывать все параметры потока, которые могут быть переопределены атрибутной записью, ведь для этого есть техническая документация QNX, а рассмотрим только наиболее интересные параметры.

Присоединенность

Это одно из самых интересных свойств потока, но одновременно и одно из самых сложных для понимания, поэтому есть смысл остановиться на нем более подробно. Поток может создаваться как ожидаемый (`PTHREAD_CREATE_JOINABLE`; таковым он и создается по умолчанию; используется также термин «присоединенный») или отсоединенный (`PTHREAD_CREATE_DETACHED`).[1] Например:

```
pthread_attr_t attr;
pthread_attr_init( &attr );
pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_DETACHED );
pthread_create( NULL, &attr, &function, NULL );
```

[1] Русскоязычную терминологию, пусть и не самую благозвучную, мы здесь заимствуем из [12].

Присоединенный поток сохраняет некоторую связь с родителем (мы это рассмотрим, когда речь пойдет о завершения потока), в то время как отсоединенный поток после точки ветвления ведет себя как совершенно автономная сущность: после точки ветвления у родительского потока нет возможности синхронизироваться с его завершением, получить код его завершения или результат выполнения потока.

Можно ожидать завершения присоединенного потока в некотором другом потоке процесса (чаще всего именно в родительском, но это не обязательно) с помощью следующего вызова:

```
int pthread_join( pthread_t thread, void** value_ptr );
```

где `thread` – идентификатор TID ожидаемого потока, который возвращался как первый параметр вызова `pthread_create( pthread_t* thread, ... )` при его создании или был им же получен после своего создания вызовом `pthread_self()`;

`value_ptr` – `NULL` или указатель на область данных (результата выполнения), которую завершающийся поток, возможно, захочет сделать доступной для внешнего мира после своего завершения. Этот указатель функция потока возвращает оператором `return` или вызовом `pthread_exit()`.

Примечание

В API QNX присутствует родственная функция (не POSIX) `pthread_timedjoin()`, отличающаяся тем, что она возвратит ошибку, если синхронизация по завершению не будет достигнута в указанный интервал времени:

```
int pthread_timedjoin( pthread_t thread, void** value_ptr,
                       const struct timespec* abstime );
```

Таким образом, вызов `pthread_join()`: а) блокирует вызывающий поток, б) является средством синхронизации потоков без использования специальных примитивов синхронизации и в) позволяет потоковой функции завершающегося потока возвратить результат своей работы в точку ожидания его завершения.

Примечание

Значение `value_ptr` (если оно не было указано как `NULL`) указывает на возвращенный результат только при нормальном завершении потока. В случае его завершения «извне» (отмены) значение `value_ptr` устанавливается в `PTHREAD_CANCELED` (константа).

Если поток предназначен для выполнения автономной работы, не требует синхронизации и не предполагает возвращать значение, он может создаваться как отсоединенный. Поскольку таких случаев достаточно много, даже большинство (например, все множество параллельных сетевых серверов), то такое поведение потока вполне могло бы быть

умалчиваемым при создании. Причина несколько ограниченного использования отсоединенных потоков относительно тех случаев, когда это может быть оправданным, состоит, скорее всего, в интуитивной боязни программистов «потерять контроль» над параллельно выполняемой ветвью, хотя зачастую этот контроль бывает чисто иллюзорным (принудительное завершение потока мы подробно рассмотрим позже).

По умолчанию потоки создаются именно как присоединенные, и это аргументируется тем обстоятельством, что такой поток всегда может сделать себя (или другой поток) отсоединенным, вызвав из своей функции потока:

```
int pthread_detach( pthread_t thread );
```

Превратить же поток, созданный как отсоединенный, в присоединенный (ожидаемый) нет никакой возможности. Таким образом, это одностороннее преобразование!

Для отсоединенного потока все задействованные им системные ресурсы освобождаются в момент его завершения, а для ожидаемого – в момент выполнения `pthread_join()` для этого потока из какого-либо другого активного потока.

Пример синхронизации порожденных потоков:

```
const int THR_NUM = 5;    // число дочерних потоков
pthread_t thrarrey[ THR_NUM ];
for( int i = 0; i < THR_NUM; i++ )
   pthread_create( &thrarrey[ i ], NULL, &thrfunc, NULL );
...
// синхронизация всех дочерних потоков:
for( int i = 0; i < THR_NUM; i++ )
   pthread_join( &thrarrey[ i ], NULL );
```

Здесь показана стандартная техника использования `pthread_join()`, вызывающая при первом знакомстве вопрос: «А что произойдет, если потоки завершатся в другом порядке, а не в той последовательности, в которой они запускались?» (порядок слежения во 2-м цикле). Но в том-то и состоит приятная особенность этой техники, что ничего не произойдет, – второй цикл является точкой синхронизации **всех** потоков `THR_NUM`, независимо от взаимного порядка их завершения.

Дисциплина диспетчеризации

Для дочернего потока может потребоваться установить иную по отношению к родителю дисциплину (политику) диспетчеризации (`SCHED_FIFO`, `SCHED_RR`, `SCHED_SPORADIC`):

```
pthread_attr_t attr;
pthread_attr_init( &attr );
pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_DETACHED );
pthread_attr_setinheritsched( &attr, PTHREAD_EXPLICIT_SCHED );
pthread_attr_setschedpolicy( &attr, SCHED_RR );
```

Особенностью здесь является то, что после инициализации атрибутной записи значениями по умолчанию в параметре типа наследования атрибутной записи будет стоять PTHREAD_EXPLICIT_SCHED («наследовать от родителя»). Изменение параметров, таких как политика диспетчеризации, приоритет и др., будет иметь силу, только если мы посредством вызова pthread_attr_setinheritsched() принудительно переустановим значение типа наследования в PTHREAD_EXPLICIT_SCHED.

Приоритет

Пожалуй, наиболее часто приходится переопределять именно приоритет, с которым будет выполняться создаваемый поток. При запуске потока с параметрами по умолчанию его приоритет устанавливается равным приоритету порождающего потока.

Примечание

При запуске приложений из командной строки для главного потока приложения (функция main()) значение приоритета устанавливается равным приоритету его родителя, в данном случае командного интерпретатора shell (в какой-то его конкретной реализации: ksh, bash и проч.). Приоритет командного интерпретатора, запускаемого из стартовых скриптов системы, для QNX 6.2.1, например, принимает значение 10, которое и можно квалифицировать как значение «по умолчанию». Важно только отчетливо восстановить «цепочку» возникновения этого «значения по умолчанию» (от стартовой программы, последовательно от одного родительского процесса к дочернему и так далее) и помнить, что она всегда может быть изменена. Таким образом, вся цепочка порождаемых потоков, если они порождаются без вмешательства в атрибутную запись потока, будет иметь тот же приоритет по умолчанию. Как управлять приоритетами создаваемых потоков «персонифицированно», рассказывается в этой главе. Но можно управлять приоритетами всей совокупности потоков приложения (относительно приоритетов всех прочих потоков в системе), изменяя приоритет запуска приложения и используя стандартную UNIX-команду nice. В простейшем виде это выглядит так:

```
# nice -nINC prog …
```

где INC – численное значение инкремента приоритета относительно умалчиваемого, с которым требуется выполнять приложение, причем положительным инкрементам соответствует понижение приоритета, а отрицательным – повышение;

prog – имя приложения со всеми последующими его параметрами. Особенностью реализации команды nice в QNX является то, что она позволяет варьировать приоритет запускаемого приложения только в ограниченных пределах: +9 в сторону уменьшения и –19 в сторону увеличения. Это не позволяет таким простым способом запустить, например, приложение с приоритетом 0 фонового потока procnto (idle-поток) и ограничивает возможность повышения приоритета верхней границей 29 при максимально возможном значении приоритета в системе 63 (все численные значения относятся к редакции QNX 6.2.1; для QNX 6.3 диапазон допустимых значений приоритетов: 0…255). В итоге, чтобы выполнить программу myprog под приоритетом 20, фиксируя при этом время ее выполнения, необходима команда:

```
# nice -n-10 time myprog
```

Значение приоритета создаваемого потока хранится в поле param атрибутной записи (типа sched_param; подробнее эта структура будет рассмотрена при обсуждении диспетчеризации). Для переустановки значений, входящих в структуру sched_param, предоставлена функция:

```
int pthread_attr_setschedparam( pthread_attr_t* attr,
                                const struct sched_param *param );
```

где attr – как и ранее, атрибутная запись потока;

param – указатель структуры sched_param, из которой параметры будут перенесены в атрибутную запись потока.

Теперь посмотрим, как запустить на выполнение поток с приоритетом на 2 единицы ниже, чем у его родителя:

```
int polisy;
struct sched_param param;
pthread_getschedparam( pthread_self(), &polisy, &param );
param.sched_priority -= 2;
pthread_attr_t attr;
pthread_attr_init( &attr );
pthread_attr_setschedparam( &attr, &param );
pthread_attr_setinheritsched( &attr, PTHREAD_EXPLICIT_SCHED );
pthread_create( NULL, &attr, &func, NULL );
```

Или даже так (хотя это немного грубее):

```
int polisy;
struct sched_param param;
pthread_getschedparam( pthread_self(), &polisy, &param );
pthread_attr_t attr;
pthread_attr_init( &attr );
attr.param.sched_priority = param.sched_priority - 2;
pthread_attr_setinheritsched( &attr, PTHREAD_EXPLICIT_SCHED );
pthread_create( NULL, &attr, &func, NULL );
```

Примечание

Как и при установке политики диспетчеризации, параметры диспетчеризации потока (и приоритет в их составе) будут установлены, только если параметр типа наследования от родителя установлен в PTHREAD_EXPLICIT_SCHED посредством вызова pthread_attr_setinheritsched().

Заметим здесь вскользь (в дальнейшем нам представится возможность использовать эти знания), что помимо «продуктивных» потоков (компонент системы и пользовательских приложений) в системе всегда существует один «паразитный» поток, запущенный с приоритетом 0 (idle-поток). Он «выбирает» весь остаток процессорного времени в те периоды, когда все имеющиеся в системе продуктивные потоки перейдут в блокированные состояния (ожидания). Подобная практика хорошо известна и реализуется также в большинстве других операционных систем.

Отличия от POSIX

Если следовать POSIX-стандарту, то некоторые из атрибутов невозможно переопределить до фактического создания этого стандарта (их можно изменить позже в самом коде потока, но иногда это не совсем правильное решение). Все эти возможности относятся к асинхронному завершению потока; детали функционирования этого механизма рассматриваются позже. К подобного рода атрибутам относятся:

- запретить асинхронное завершение (отмену) потока;
- установить тип завершаемости потока;
- определить, что должно происходить при доставке потоку сигналов.

QNX расширяет возможности POSIX, позволяя по условию `OR` установить соответствующие биты-флаги в поле `flags` атрибутной записи, прежде чем будет произведен вызов, создающий поток. Не существует функций вида `pthread_attr_set_*()`, эквивалентных этим установкам. К этим флагам относятся:

- `PTHREAD_CANCEL_ENABLE` – запрос на завершение будет обрабатываться в соответствии с типом завершаемости, установленным для потока (значение по умолчанию);
- `PTHREAD_CANCEL_DISABLE` – запросы на завершение будут отложены;
- `PTHREAD_CANCEL_ASYNCHRONOUS` – если завершение разрешено, отложенные или текущие запросы будут выполнены немедленно;
- `PTHREAD_CANCEL_DEFERRED` – если завершение разрешено, запросы на завершение будут отложены до достижения точки завершаемости (значение по умолчанию);
- `PTHREAD_MULTISIG_ALLOW` – завершать по сигналу все потоки в процессе (POSIX-умолчание);
- `PTHREAD_MULTISIG_DISALLOW` – завершать по сигналу только тот поток, который принял сигнал.

После запуска потока все атрибуты, связанные с завершаемостью потока, могут быть изменены вызовами `pthread_setcancelstate()` и `pthread_setcanceltype()`.

Передача параметров потоку

Зачастую каждый поток из группы последовательно создаваемых потоков, выполняющих одну и ту же функцию, нужно запускать со своим индивидуальным блоком данных (параметром потока). Для этого предназначен 4-й параметр вызова `pthread_create()` – указатель на блок данных типа `void*`. Характерно, что это может быть произвольная структура данных сколь угодно сложного типа, структуризацию которой вызывающий `pthread_create()` код и функция потока должны понимать единообразно; никакого контроля соответствия типов на этапе вызова не производится.

Достаточно часто встречающийся на практике образец многопоточного кода – это циклическая процедура ожидания наступления некоторого условия (события), после которого порождается новый экземпляр потока, призванный обслужить наступившее событие (типичная схема всего разнообразия многопоточных сетевых серверов). В таких случаях код, порождающий потоки, выглядит подобно следующему фрагменту:

```
// функция потока:
void* ThreadProc( void* data ) {
   // ... выполняется обработка, используя структуру *(DataParam*)data
   return NULL;
};

// порождающий потоки код:
while( true ) {
   // инициализация области параметров
   struct DataParam data( ... );
   if( /* ожидаем нечто */ )
      pthread_create( NULL, &attr, &ThreadProc, &data );
};
```

Этот простейший код крайне опасен: при быстрых циклах и, что намного важнее, непредсказуемых моментах повторных созданий экземпляров потоков из вызывающего цикла необходимо обеспечить, чтобы используемое в функции потока `ThreadProc()` значение данных было адекватным. Оно может быть изменено в вызывающем коде или даже, более того, просто разрушено при выходе из локальной области видимости, как в следующем коде:

```
// порождающий потоки код:
while( true ) {
   if( /* ожидаем нечто */ ) {
      struct DataParam data( ... );
      pthread_create( NULL, &attr, &ThreadProc, &data );
   };
   // ... здесь может идти достаточно длительная обработка
};
```

Здесь блок данных, выделяемый в стеке порождающего потока, к началу его использования в дочернем потоке может быть просто уничтожен.

Единственно надежный способ обеспечить требование актуальности передаваемых данных – это создание копии блока параметров непосредственно при входе в функцию потока, например так (если определена операция копирования):

```
// функция потока:
void* ThreadProc( void* data ) {
   DataParam copy = *(DataParam*)data;
   // ... выполняется обработка, используя структуру copy
   return NULL;
};
```

или так (если определен инициализирующий конструктор структуры данных):

```
// функция потока:
void* ThreadProc( void* data ) {
   DataParam copy( *(DataParam*)data );
   // ... выполняется обработка, используя структуру copy
   return NULL;
};
```

Но и этот код оказывается некорректен. При порождении потока нам нужно, чтобы инициализация копии переданных данных в теле функции потока произошла **до того**, как на очередном цикле оригинал этих данных будет разрушен или изменен. Но дисциплины диспетчеризации равнозначных потоков (в данном случае родительского и порожденного) в операционной системе никак не регламентируют (и не имеют права этого делать!) порядок их выполнения после точки ветвления – `pthread_create()`.

Обеспечить актуальность копии переданных данных можно несколькими искусственными способами:

1. Передачей в качестве аргумента `pthread_create()` специально сделанной ранее временной копии экземпляра данных, например:

```
if( /* нечто */ ) {
   // static обеспечивает неразрушаемость
   static struct DataParam copy;
   copy = data;
   pthread_create( NULL, &attr, &ThreadProc, &copy );
};
```

Этот способ иногда хорошо «срабатывает» для данных типа символьных строк, представленных в стандарте языка C (однако используется он не часто):

```
void* ThreadProc( void *data ) {
   ...
   // можно даже не делать копию – это уже копия:
   printf( "%s", (char*)data );
};
...
while( true ) {
   char *data = ... /* инициализация данных */;
   if( /* нечто */ )
      pthread_create( NULL, &attr,
                      &ThreadProc, strdup( data ) );
};
```

2. Для передачи параметра скалярного типа (`char`, `short`, `int`), не превышающего размер указателя, очень часто в самых разнообразных источниках [1, 3] можно увидеть такой трюк, когда указателю присваивается непосредственное значение скалярной величины:

```
// функция потока:
void* ThreadProc( void* data ) {
   // ... выполняется обработка, используя значение параметра (char)data
   return NULL;
};

// порождающий потоки код:
while( true ) {
   char data = ... /* инициализация параметра */;
   if( /* ожидаем нечто */ )
      pthread_create( NULL, &attr, &ThreadProc, (void*)data );
};
```

Или даже так:

```
pthread_create( NULL, &attr, &ThreadProc, (void*)5 );
pthread_create( NULL, &attr, &ThreadProc, (void*) ( x + y ) );
```

Положительной стороной этого решения (которое тем не менее остается трюкачеством) является то, что параметр в `ThreadProc()` передается по значению, то есть неявным копированием, и любые последующие манипуляции с ним не приведут к порче переданного значения. Таким образом, в `ThreadProc()` нет необходимости создавать локальную копию полученного параметра.

3. **Создание экземпляра данных в родительском потоке для каждого нового экземпляра создаваемого потока с гарантированным уничтожением экземпляра данных при завершении порожденного потока:**

```
void* ThreadProc( void *data ) {
   // используем экземпляр data без копирования...
   ...
   delete data;
   return NULL;
};
...
if( /* нечто */ ) {
   // создание экземпляра вместе с инициализацией
   // (предполагаем, что для DataParam ранее определен
   // копирующий конструктор):
   pthread_create( NULL, &attr, &ThreadProc, new DataParam( data ) );
};
```

Это один из самых безошибочно срабатывающих способов, и единственным его недостатком является то, что объекты создаются в одной структурной единице (родителе), а уничтожаться должны в другой (потомке), которые иногда даже размещаются в различных файлах программного кода, а ошибки с парностью операций над динамической памятью обходятся очень дорого.

4. «Ручной» вызов диспетчеризации в порождающем потоке, по крайней мере при дисциплине по умолчанию для QNX – round-robin:

```
if( /* нечто */ ) {
   pthread_create( NULL, &attr, &ThreadProc, &data );
   sched_yield();
};
```

Мы не можем произвольно изменять последовательность выполнения потоков (чем нарушили бы принципы диспетчеризации) и не можем утверждать, что при наличии многих потоков именно только что порожденный поток получит управление. Но после выполнения `sched_yield()` мы можем гарантировать, что родительский поток будет помещен именно **в хвост** очереди потоков равных приоритетов, готовых к исполнению, и его активизация произойдет позже всех наличных в системе потоков, в том числе и последнего порожденного.

Примечание

В этом месте внимательный читатель вправе оживиться: «Обманывают, обвешивают...». Да, описываемое здесь экзотическое решение не совсем корректно с позиции уже упоминавшегося определения Э. Дейкстры «слабосвязанных процессов» и независимости результата от относительных скоростей: в SMP-системе при количестве процессоров, большем, чем количество параллельных потоков, это решение не будет работать так, как мы ему предписываем. Но к настоящему времени такое «стечение обстоятельств» может быть либо чисто теоретически умозрительным, либо возникать на экспериментальных единичных образцах SMP, содержащих десятки и сотни процессоров..., но где QNX, насколько нам известно, не используется.

В этом варианте и в порожденном потоке можно симметрично сделать так:

```
void* ThreadProc( void *data ) {
   struct DataParam copy( *data );
   sched_yield();
   ...
};
```

Примечание

Иногда для выражения этой техники используется и такая, в общем несколько небрежная, форма записи:

```
pthread_create( NULL, &attr, &ThreadProc, &data );
delay( 1 );    // вместо sched_yield()
```

Фокус здесь состоит не в том, что 1 миллисекунда – это время, заведомо достаточное для копирования экземпляра данных, а в том, что POSIX определяет, что операция `delay()` (а также все родственные ей функции: `sleep()`, `nanosleep()` и другие функции пассивной задержки) является операцией **пассивного** ожидания и должна сопровождаться принудительной диспетчеризацией.

5. Создание потока с приоритетом выше, чем родительский, с последующим возвратом его приоритета на прежний уровень после выполнения требуемой инициализации копии:

```
void* ThreadProc( void* data ) {
   struct sched_param param;
   int polisy;
   pthread_getschedparam( pthread_self(), &polisy, &param );
   param.sched_priority -= 2;
   // инициализация копии блока данных
   ...
   pthread_setschedparam( pthread_self(), polisy, &param );
   ...
   return NULL;
};
...
if( /* нечто */ ) {
   pthread_attr_t attr;
   pthread_attr_init( &attr );
   pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_DETACHED );
   pthread_attr_setinheritsched( &attr, PTHREAD_EXPLICIT_SCHED );
   pthread_attr_setschedpolicy( &attr, SCHED_RR );
   int polisy;
   struct sched_param param;
   pthread_getschedparam( pthread_self(), &polisy, &param );
   attr.param.sched_priority = param.sched_priority + 2;
   pthread_create( NULL, &attr, &ThreadProc, &data );
};
```

Здесь в точке создания порожденный поток сразу же вытесняет своего родителя и выполняет инициализацию копии области параметров, после чего возвращается к нормальной (с равными приоритетами) диспетчеризации. Этот вариант может показаться искусственно усложненным, но отлично вписывается своими побочными достоинствами в создание многопоточных GUI-приложений для графической подсистемы Photon.

Данные потока

В реальном коде часто возникает ситуация, когда одновременно исполняются несколько экземпляров потоков, использующих один и тот же код (при создании потоков указывается одна и та же функция потока). При этом некоторые данные (например, статические объекты, глобальные объекты программного файла, объявленные вне функции потока) будут представлены для различных экземпляров потока в виде единого экземпляра данных, а другие (блок параметров функции потока, локальные данные функции потока) будут представлять собой индивидуальные экземпляры для каждого потока:

```
class DataBlock {
    DataBlock( void );
```

```
    DataBlock( DataBlock& );
};

DataBlock A;

void* ThreadProc( void *data ) {
   static DataBlock B;
   DataBlock C, D( *(DataBlock*)data );
   ...
   delete data;
   return NULL;
};
...
for( int i = 0; i < N; i++ ) {
   DataBlock E;
   // ... обработка и заполнение E ...
   pthread_create( NULL, NULL, &ThreadProc, new DataBlock( E ) );
};
```

В этом простейшем фрагменте кода N потоков разделяют единые экземпляры данных A и B: любые изменения, сделанные в данных потоком i, будут видимы потоку j, если, конечно, корректно выполнена синхронизация доступа к данным и потоки «совместными усилиями» не разрушат целостность блока данных. Другие блоки данных, C и D, представлены одним изолированным экземпляром на каждый поток, и никакие изменения, производимые потоком в своем экземпляре данных, не будут видны другим потокам.

Подобные эффекты не возникают в однопотоковых программах, а если они не учитываются и возникают спонтанно, то порождают крайне трудно выявляемые ошибки.[1] Очень часто такие ошибки возникают после преобразования корректных последовательных программ в потоковые. Рассмотрим простейший фрагмент кода:

```
int M = 0;

void Func_2( void ) {
   static int C = 0;
   M += 2, C++, M -= 2;
};

void Func_1( void ) { Func_2(); };

void* ThreadProc( void *data ) {
   Func_1();
   return NULL;
};
```

[1] В литературе неоднократно отмечалось, что ошибки многопоточных программ часто недетерминированы (могут возникать или нет в идентичных условиях исполнения), трудно воспроизводимы и могут быть крайне трудны для локализации.

```
...
for( int i = 0; i < N; i++ )
   pthread_create( NULL, NULL, &ThreadProc, NULL );
```

Можно ли здесь утверждать, что переменная M сохранит нулевое значение, а переменная C действительно является счетчиком вызовов и ее результирующее значение станет N? Ни в коей мере: после выполнения такого фрагмента в переменных может быть все что угодно. Но цепочка вызовов Func_1() -> Func_2() может быть сколь угодно длинной, описание Func_2() может находиться совершенно в другом файле кода (вместе с объявлением переменной M!) и, наконец, Func_2() в нашей транскрипции может быть любой функцией из библиотек C/C++, писавшейся лет 15 назад и содержащей в своем теле статические переменные!

POSIX.1 требует, чтобы определенные в нем функции были максимально безопасными в многопоточной среде. Но переработка всех библиотек – трудоемкий и длительный процесс. API QNX (и так поступили производители многих POSIX-совместимых ОС) для потенциально небезопасных в многопоточной среде функций ввели их эквиваленты, отличающиеся суффиксом «_r», например: localtime() – localtime_r(), rand() – rand_r() и т. д. Принципиально небезопасна в многопоточной среде одна из самых «любимых» в UNIX функция – select().

Собственные данные потока

Описанной выше схеме общих данных приложения и локальных данных потока, достаточных для большинства «ординарных» приложений, все-таки определенно не хватает гибкости, покрывающей все потребности. Поэтому в расширениях POSIX реального времени вводится третий специфичный механизм создания и манипулирования с данными в потоке – собственные данные потока (thread-specific data). Использование собственных данных потока – самый простой и эффективный способ манипулирования данными, представленными индивидуальными экземплярами данных для каждого потока.

Согласно POSIX операционная система должна поддерживать ограниченное количество объектов собственных данных (POSIX.1 требует, чтобы этот предел **не превышал** 128 объектов на каждый процесс). Ядром системы поддерживается массив из этого количества ключей (тип pthread_key_t; это абстрактный тип, и стандарт предписывает не ассоциировать его с некоторым значением, но реально это небольшие целые значения, и в таком виде вся схема гораздо проще для понимания). Каждому ключу сопоставлен флаг, отмечающий, занят этот ключ или свободен, но это внутренние детали реализации, не доступные программисту. Кроме того, при создании ключа с ним может быть связан адрес функции деструктора, которая будет вызываться при завершении потока и уничтожении его экземпляра данных (рис. 2.4).

Когда поток вызывает pthread_key_create() для создания нового **типа** собственных данных, система разыскивает первое незанятое значение

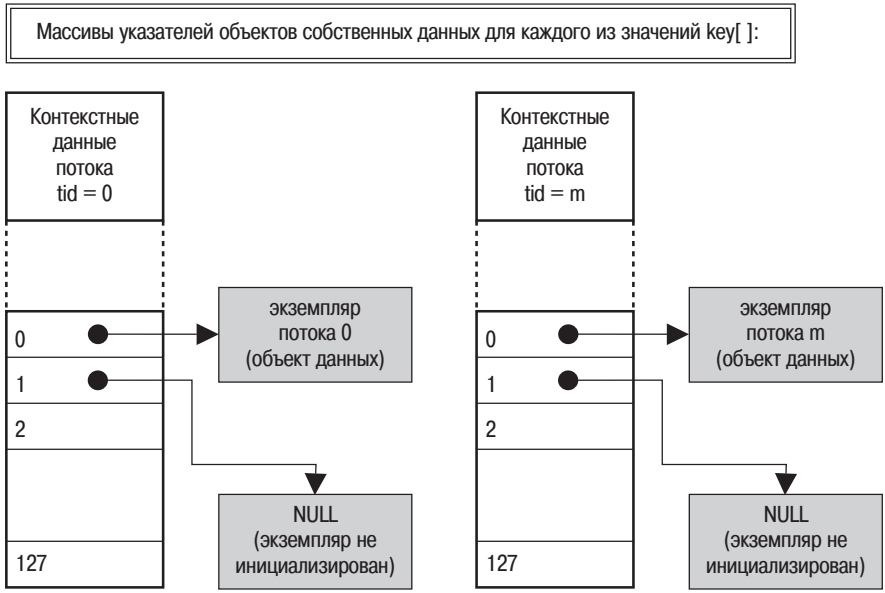

***Рис. 2.4.** Ключи экземпляров данных*

ключа и возвращает его значение (0...127). Для каждого потока процесса (в составе описателя потока) хранится массив из 128 указателей (`void*`) блоков собственных данных, и по полученному ключу поток, индексируя этот массив, получает доступ к своему экземпляру данных, ассоциированных со значением ключа. Начальные значения всех указателей блоков данных – `NULL`, а фактическое размещение и освобождение блоков данных выполняет пользовательская программа (рис. 2.5).

На рис. 2.5 представлен массив структур, создаваемый в единичном экземпляре для каждого **процесса** библиотекой **потоков**. Каждый элемент ключа должен быть предварительно инициализирован вызовом `pthread_key_create()` (однократно для всего процесса). Каждый инициализированный элемент массива определяет объекты единого класса во всех использующих их потоках, поэтому для них здесь же определяется деструктор (это в терминологии языка C!). Деструктор – единый для экземпляров данных в каждом потоке. Даже для инициализированного и используемого ключа в качестве деструктора может быть указан `NULL`, при этом никакие деструктивные действия при завершении потока не выполняются.

После размещения блока программа использует вызов `pthread_setspecific()` для связывания адреса своего экземпляра данных с элементом массива указателей, индексируемого ключом. В дальнейшем каждый поток использует `pthread_getspecific()` для доступа именно к своему экземпляру данных. Это схема, а теперь посмотрим, как она работает.

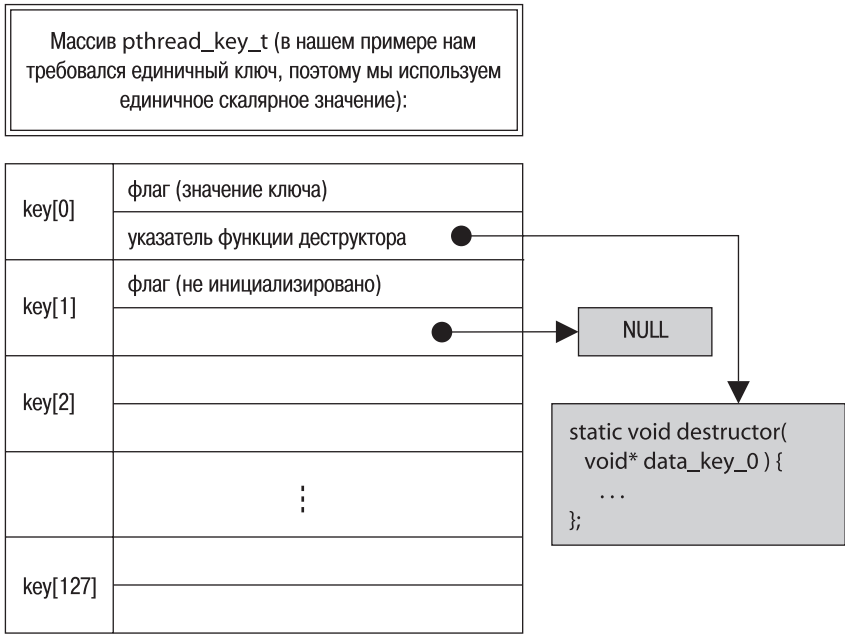

Рис. 2.5. *Поток и его собственные данные*

Положим, что нам требуется создать `N` параллельно исполняющихся идентичных потоков (использующих единую функцию потока), каждый из которых предполагает работать со своей копией экземпляра данных типа `DataBlock`:

```
class DataBlock {
   ~DataBlock() { ... };
   ...
};

void* ThreadProc( void *data ) {
   // ... здесь будет код, который мы рассмотрим
   return NULL;
};
...
for( int i = 0; i < N; i++ )
   pthread_create( NULL, NULL, &ThreadProc, NULL );
```

Последовательность действий потока выглядит следующим образом:

1. Поток запрашивает `pthread_key_create()` – создание ключа для доступа к блоку данных `DataBlock`. Если потоку необходимо иметь несколько (`m`) блоков собственных данных различной типизации (и различного функционального назначения): `DataBlock_1`, `DataBlock_2`, ... `DataBlock_m`, то он запрашивает значения ключей соответствующее число раз для каждого типа (`m`).

2. Неприятность здесь состоит в том, что запросить значение ключа для DataBlock должен только первый пришедший к этому месту поток (когда ключ еще не распределен). Последующие потоки, достигшие этого места, должны только воспользоваться ранее распределенным значением ключа для типа DataBlock. Для разрешения этой сложности в систему функций собственных данных введена функция pthread_once().
3. После этого каждый поток (как создавший ключ, так и использующий его) должен запросить по pthread_getspecific() адрес блока данных и, убедившись, что это NULL, динамически распределить область памяти для своего экземпляра данных, а также зафиксировать по pthread_setspecific() этот адрес в массиве экземпляров для дальнейшего использования.
4. Дальше поток может работать с собственным экземпляром данных (отдельный экземпляр на каждый поток), используя для доступа к нему pthread_getspecific().
5. При завершении любого потока система уничтожит и его экземпляр данных, вызвав для него деструктор, который был установлен вызовом pthread_key_create(), единым для всех экземпляров данных, ассоциированных с этим значением ключа.

Теперь запишем это в коде, заодно трансформировав в новую функцию ThreadProc() код ранее созданной версии этой же функции SingleProc() для исполнения в одном потоке, не являющийся реентерабельным и безопасным в многопоточной среде. (О вопросах реентерабельности мы обязательно поговорим позже.)

```
void* SingleProc( void *data ) {
   static DataBlock db( ... );
   // ... операции с полями DataBlock
   return NULL;
};
```

Примечание

То, что типы параметров и возвращаемое значение SingleProc() «подогнаны» под синтаксис ее более позднего эквивалента ThreadProc(), не является принципиальным ограничением – входную и выходную трансформации форматов данных реально осуществляют именно в многопоточном эквиваленте. Нам здесь важно принципиально рассмотреть общую формальную технику трансформации нереентерабельного кода в реентерабельный.

Далее следует код SingleProc(), преобразованный в многопоточный вид:

```
static pthread_key_t key;
static pthread_once_t once = PTHREAD_ONCE_INIT;

static void destructor( void* db ) {
   delete (DataBlock*)db;
};
```

```
static void once_creator( void ) {
   // создается единый на процесс ключ для данных DataBlock:
   pthread_key_create( &key, destructor );
};

void* ThreadProc( void *data ) {
   // гарантия того, что ключ инициализируется только 1 раз на процесс!
   pthread_once( &once, once_creator );
   if( pthread_getspecific( key ) == NULL )
      pthread_setspecific( key, new DataBlock( ... ) );
   // Теперь каждый раз в теле этой функции или функций, вызываемых
   // из нее, мы всегда можем получить доступ к экземпляру данных:
   DataBlock* pdb = pthread_getspecific( key );
   // ... все те же операции с полями pdb->(DataBlock)
   return NULL;
};
```

Примечание

Обратите внимание, что вся описанная техника преобразования потоковых функций в реентерабельные (как и все программные интерфейсы POSIX) отчетливо ориентирована на семантику классического C, в то время как все свое изложение мы ориентируем и иллюстрируем на C++. При создании экземпляра собственных данных полностью разрушается контроль типизации: разные экземпляры потоков вполне могли бы присвоить своим указателям данные (типа `void*`), ассоциированные с одним значением `key`. Это совершенно различные типы данных, скажем `DataBlock_1*` и `DataBlock_2*`. Но проявилось бы это несоответствие только при завершении функции потока и уничтожении экземпляров данных, когда к объектам совершенно разного типа был бы применен один деструктор, определенный при выделении ключа. Ошибки такого рода крайне сложны в локализации.

Особая область, в которой собственные данные потока могут найти применение и где локальные (стековые) переменные потока не могут быть использованы, – это асинхронное выполнение фрагмента кода в контексте потока, например при получении потоком сигнала.

Еще одно совсем не очевидное применение собственных данных потока (мы не встречали в литературе упоминаний о нем), которое особо органично вписывается в использование именно C++, – это еще один способ возврата в родительский поток результатов работы дочерних. При этом неважно, как были определены дочерние потоки – как присоединенные или как отсоединенные (мы обсуждали это ранее); такое использование в заметной мере нивелирует их разницу. Эта техника состоит в том, что:

- Если при создании ключа не определять деструктор экземпляра данных потока `pthread_key_create( . . ., NULL )`, то при завершении потока над экземпляром его данных не будут выполняться никакие деструктивные действия и созданные потоками экземпляры данных будут существовать и после завершения потоков.

- **Если к этим экземплярам данных созданы альтернативные пути доступа (а они должны быть в любом случае созданы, так как области этих данных в конечном итоге нужно освободить), то благодаря этому доступу порождающий потоки код может использовать данные, «оставшиеся» как результат выполнения потоков.**

В коде (что гораздо нагляднее) это может выглядеть так (код с заметными упрощениями взят из реального завершенного проекта):

```
// описание экземпляра данных потока
struct threwndata {
   . . .
};

static pthread_once_t once = PTHREAD_ONCE_INIT;
static pthread_key_t key;
void createkey( void ) { pthread_key_create( &key, NULL ); };
// STL-очередь, например указателей на экземпляры данных
queue<threwndata*> result;

// функция потока
void* GetBlock( void* ) {
   pthread_once( &once, createkey );
   threwndata *td;
   if( ( td = (threwndata*)pthread_getspecific( key ) )
       == NULL ) {
      td = new threwndata();
      pthread_setspecific( key, (void*)td );
      // вот он - альтернативный путь доступа:
      result.push( td );
   };
   // далее идет плодотворная работа над блоком данных *td
   // . . . . . .
};

int main( int argc, char **argv ) {
   // . . . . . .
   for( int i = 0; i < N; i++ )
      pthread_create( NULL, NULL, GetBlock, NULL );
   // . . . . . к этому времени потоки завершились;
   // ни в коем случае нельзя помещать result.size()
   // непосредственно в параметр цикла!
   int n = result.size();
   for( int i = 0; i < n; i++ ) {
      threwndata *d = result.front();
      // обработка очередного блока *d . . .
      result.pop();
      delete d;
   };
   return EXIT_SUCCESS;
};
```

Примечание

В предыдущих примерах кода мы указывали третий параметр `pthread_create()` в виде `&GetBlock` (адреса функции потока), но в текущем примере мы сознательно записали `GetBlock`. И то и другое верно, ибо компилятор достаточно умен, чтобы при указании имени функции взять ее адрес.

Собственные данные потоков – это настолько гибкий механизм, что он может таить в себе и другие, еще не используемые техники применения.

Безопасность вызовов в потоковой среде

Рассмотрев «в первом приближении» технику собственных данных потоков, мы теперь готовы ответить на вопрос: «В чем же главное предназначение такой в общем-то достаточно громоздкой техники? И зачем для ее введения потребовалось специально расширять стандарты POSIX?» Самое прямое ее предназначение, помимо других «попутных» применений, которые были обсуждены ранее, – это общий механизм превращения существующей функции для однопотокового исполнения в функцию, безопасную (thread safe) в многопоточном окружении. Этот механизм предлагает единую (в смысле «единообразную», а не «единственно возможную») технологию для разработчиков библиотечных модулей.

Примечание

ОС QNX, заимствующая инструментарий GNU-технологии (gcc, make, ...), предусматривает возможность построения как статически связываемых библиотек (имена файлов вида `xxx.a`), так и разделяемых или динамически связываемых (имена файлов вида `xxx.so`). Целесообразность последних при построении автономных и встраиваемых систем (на что главным образом и нацелена ОС QNX) достаточно сомнительна. Однако высказанное выше положение о построении реентерабельных программных единиц относится не только к библиотечным модулям (как статическим, так и динамическим) в традиционном понимании термина «библиотека», но и охватывает куда более широкий спектр возможных объектов и в той же мере относится и просто к любым наборам утилитных объектных модулей (вида `xxx.o`), разрабатываемых в ходе реализации под целевой программный проект.

Если мы обратимся к технической документации API QNX (аналогичная картина будет и в API любого UNIX), то заметим, что только небольшая часть функций отмечена как thread safe. К «небезопасным» отнесены такие общеизвестные вызовы, как `select()`,`rand()` и `readln()`, а многим «небезопасным» в потоковой среде вызовам сопутствуют их безопасные дубликаты с суффиксом `*_r` в написании имени функции, например `MsgSend()` – `MsgSend_r()`.

В чем же состоит небезопасность в потоковой среде? В нереентерабельности функций, подготовленных для выполнения в однопоточной среде, в первую очередь связанной с потребностью в статических данных,

хранящих значение от одного вызова к другому. Рассмотрим классическую функцию rand(), традиционно реализуемую в самых разнообразных ОС примерно так (при «удачном» выборе констант A, B, C):

```
int rand( void ) {
   static int x = rand_init();
   return x = ( A * x + B ) % C;
};
```

Такая реализация, совершенно корректная в последовательной (однопотоковой) модели, становится небезопасной в многопоточной: а) вычисление x может быть прервано событием диспетчеризации, и не исключено, что вновь получивший управление поток в свою очередь обратится к rand() и исказит ход текущего вычисления; б) каждый поток «хотел бы» иметь свою автономную последовательность вычислений x, не зависящую от поведения параллельных потоков. Желаемый результат будет достигнут, если каждый поток будет иметь свой автономный экземпляр переменной x, что может быть получено двумя путями:

1. Изменить прототип объявления функции:

   ```
   int rand_r( int *x ) {
      return x = ( A * ( *x ) + B ) % C;
   };
   ```

   При этом проблема «клонирования» переменной x в каждом из потоков (да и начальной ее инициализации) не снимается, она только переносится на плечи пользователя, что, однако, достаточно просто решается при создании потоковой функции за счет ее стека локальных переменных:

   ```
   void* thrfunc( void* ) {
      int x = rand_init();
      . . . = rand_r( &x );
   };
   ```

   Именно такова форма и многопоточного эквивалента в API QNX – rand_r().

2. В этом варианте мы сохраняем прототип описания функции без изменений за счет использования различных экземпляров собственных данных потока. (Весь приведенный ниже код размещен в отдельной единице компиляции; все имена, за исключением rand(), невидимы и недоступны из точки вызова, что подчеркнуто явным использованием квалификатора static.)

   ```
   static pthread_key_t key;
   static pthread_once_t once = PTHREAD_ONCE_INIT;
   static void destr( void* db ) { delete x; };
   static void once_creator( void ) { pthread_key_create( &key,  destr ); };

   int rand( void ) {
      pthread_once( &once, once_creator );
   ```

```
    int *x = pthread_getspecific( key );
    if( x == NULL ) {
       pthread_setspecific( key, x = new int );
       *x = rand_init();
    };
    return x = ( A * ( *x ) + B ) % C;
};
```

В этом варианте, в отличие от предыдущего, весь код вызывающего фрагмента при переходе к многопоточной реализации остается текстуально неизменным:

```
void* thrfunc( void* ) {
   // . . .
   while( true ) {
      . . . = rand( x );
   };
};
```

Перевод всего программного проекта на использование потоковой среды состоит в замене объектной единицы (объектного файла, библиотеки), содержащей реализацию `rand()`, и новой сборке приложения с этой объектной единицей.

При таком способе изменяются под потоковую безопасность и стандартные общеизвестные библиотечные функции API, написанные в своем первозданном виде 25 лет назад... (по крайней мере, так предлагает это делать стандарт POSIX, вводящий в обиход собственные данные потоков).

Диспетчеризация потоков

Каждому потоку, участвующему в процессе диспетчеризации, соответствует экземпляр структуры, определенной в файле `<sys/target_nto.h>`, в котором находятся все фундаментальные для ОС QNX определения:

```
struct sched_param {
   _INT32 sched_priority;
   _INT32 sched_curpriority;
   union {
      _INT32 reserved[ 8 ];
      struct {
         _INT32          __ss_low_priority;
         _INT32          __ss_max_repl;
         struct timespec __ss_repl_period;
         struct timespec __ss_init_budget;
      } __ss;
   } __ss_un;
};
#define sched_ss_low_priority __ss_un. __ss. __ss_low_priority
#define sched_ss_max_repl     __ss_un. __ss. __ss_max_repl
```

```
#define sched_ss_repl_period  __ss_un. __ss. __ss_repl_period
#define sched_ss_init_budget  __ss_un. __ss. __ss_init_budget
```

Все, что определяется внутри `union __ss_un`, имеет отношение только к спорадической диспетчеризации (спорадическая диспетчеризация была введена значительно позже других, и ей будет уделено достаточно много внимания). Для всех остальных типов диспетчеризации потока это поле заполняется фиктивным полем `reserved`, и именно так (в укороченном виде) определялась структура диспетчеризации в версии QNX 6.1.

Сейчас нас интересуют начальные поля этой структуры, не зависящие от типа диспетчеризации потока:

`sched_priority` — статический приоритет, который присваивается потоку при его создании и который может быть программно изменен по ходу выполнения потока;

`sched_curpriority` — текущий приоритет, с которым выполняется (и согласно которому диспетчеризируется) данный поток в текущий момент времени. Это значение приоритета образуется системой на основе заданного статического приоритета, но оно может динамически изменяться системой, например при отработке дисциплин наследования приоритетов или граничных приоритетов для потока. Программа не имеет средств воздействия на это значение[1], но может его считывать.

Еще раз подчеркнем достаточно очевидную вещь: дисциплина диспетчеризации определяется относительно потока и на уровне потока (но не процесса). Проследить за дисциплиной диспетчеризации (и убедиться в справедливости утверждения предыдущей фразы) можно командой `pidin`. Вот несколько строк ее вывода, относящиеся к составным частям самой системы:

```
  pid tid name               prio STATE       Blocked
  1   1   6/boot/sys/procnto   0f READY
  1   2   6/boot/sys/procnto  10r RUNNING
...
  1   5   6/boot/sys/procnto  63r RECEIVE     1
...
  1   9   6/boot/sys/procnto   6r NANOSLEEP
...
  6   1   roc/boot/devb-eide  10o SIGWAITINFO
```

[1] Пользователь может изменять это поле, однако это лишено смысла и не влечет за собой никаких последствий, ведь значением текущего приоритета «управляет» ОС, например для осуществления наследования приоритетов. С другой стороны, иногда целесообразно считать значение именно этого поля, чтобы определить значение динамического приоритета потока, установленного, например, в результате того же наследования.

В поле prio указывается приоритет (текущий; возможно, последнее из унаследованных значений!) каждого потока с установленной для него дисциплиной диспетчеризации: f – FIFO, r – RR, o – OTHER, s – SPORADIC.

В системе на сегодняшний день реализованы три[1] дисциплины диспетчеризации: очередь потоков равных приоритетов (FIFO – first in first out; еще в ходу термин «невытесняющая»), карусельная (RR – round-robin) и спорадическая. Рассмотрим фрагмент их определения в файле <sched.h>:

```
#if defined(_ _EXT_QNX)
#define SCHED_NOCHANGE  0
#endif
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_OTHER     3
#if defined(_ _EXT_QNX)
#define SCHED_SPORADIC  4  /* Approved 1003.1d D14 */
#define SCHED_ADJTOHEAD 5  /* Move to head of ready queue */
#define SCHED_ADJTOTAIL 6  /* Move to tail of ready queue */
#define SCHED_MAXPOLICY 6  /* Maximum valid policy entry */
#endif
```

Все дисциплины диспетчеризации, кроме спорадической, достаточно полно описаны в литературе [1], поэтому мы лишь перечислим их отличительные особенности:

1. FIFO – это та дисциплина диспетчеризации, которая в литературе по Windows 3.1/3.11 называлась «невытесняющей многозадачностью» (или «кооперативной»). Здесь выполнение потока не прерывается потоками равного приоритета до тех пор, пока сам поток «добровольно» не передаст управление, например вызовом sched_yield() (часто для этой цели используется косвенный эффект вызовов delay(), sleep() и им подобных). В других источниках такой способ диспетчеризации называют очередями потоков равных приоритетов.
2. RR – это та дисциплина диспетчеризации, которая в Windows 98/NT/XP именуется «вытесняющей многозадачностью»; еще в литературе для нее используется термин «режим квантования времени».

[1] В документации неоднократно упоминается еще одна дисциплина – «адаптивная» (SCHED_ADAPTIVE), и даже детально описывается, «как она работает». Видимо, это можно отнести только к тому, что корректировка обширной документации отстает от развития самой системы. На конференции «QNX-Россия 2003» на вопрос по поводу ADAPTIVE-диспетчеризации представители QSSL отвечали так: «Этот вид диспетчеризации был в QNX 4.xx, а в QNX 6.x вместо него введена более продвинутая техника SPORADIC-диспетчеризации». Тем не менее более продвинутая спорадическая диспетчеризация не позволяет абсолютно точно выразить логику адаптивной.

Поток работает непрерывно только в течение предопределенного кванта времени. (В нескольких местах документации утверждается, что значение этого кванта времени составляет 4 системных тика (time-slice), что в QNX 6.2.1 по умолчанию составляет 4 миллисекунды, и только в одном месте документации говорится, что квант диспетчеризации составляет 50 миллисекунд; это определенное разночтение. Справедливым является именно первое утверждение.)

После истечения отведенного ему кванта времени поток вытесняется потоком равного приоритета (при отсутствии других потоков этим новым потоком может быть и только что вытесненный, то есть его выполнение будет продолжено, но передиспетчеризация тем не менее происходит). Установленный квант времени диспетчеризации может быть получен вызовом (стандарт POSIX 1003.1):

```
#include <sched.h>
int sched_rr_get_interval( pid_t pid, struct timespec* interval );
```

где `pid` – это PID процесса, для которого определяется квант времени, как и для многих других подобных функций. Если PID = 0, вызов относится к текущему процессу;

`interval` – указатель на структуру `timespec` (стандарт POSIX 1003.1):

```
#include <time.h>
struct timespec {
   time_t   tv_sec;  // значение секунд
   long     tv_nsec; // значение наносекунд
}
```

При успешном выполнении функция `sched_rr_get_interval()` возвращает 0, в противном случае –1.

Примечание

Две другие функции, часто удобные для работы со структурой `timespec`:

```
#include <time.h>
void nsec2timespec( struct timespec *timespec_p, _uint64 nsec );
```

– это преобразование интервала, выраженного в наносекундах (nsec), в структуру `timespec` («выходной» параметр вызова `timespec_p`);

```
#include <time.h>
_uint64 timespec2nsec( const struct timespec* ts );
```

– это преобразование структуры `timespec` в значение, выраженное в наносекундах (это функция из native API QNX).

3. Спорадическая диспетчеризация – это гораздо более развитая форма «вытесняющей многозадачности», численные характеристики которой (время кванта, численные значения приоритетов и др.) могут детально параметризироваться и даже динамически изменять-

ся по ходу выполнения. Подробнее спорадическая диспетчеризация рассмотрена далее.

Часто задают вопрос: «А как много потоков целесообразно делать? Насколько снижается эффективность многопоточной программы за счет диспетчеризации потоков?» С другой стороны, в литературе часто встречаются (достаточно голословные, на качественном уровне) утверждения, что многопоточная программа будет заметно уступать в эффективности своему последовательному (в одном потоке) эквиваленту. Проверим это на реальной задаче:

Множественные потоки в едином приложении

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <unistd.h>
#include <limits.h>
#include <pthread.h>
#include <inttypes.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>
#include <errno.h>
#include <math.h>

// преобразование процессорных циклов в миллисекунды:
static double cycle2milisec ( uint64_t ccl ) {
   const static double s2m = 1.E+3;
   // это скорость процессора:
   const static uint64_t
      cps = SYSPAGE_ENTRY( qtime )->cycles_per_sec;
   return (double)ccl * s2m / (double)cps;
};

static int nsingl = 1;

// рабочая функция, которая имитирует вычисления:
void workproc( int how ) {
   const int msingl = 30000;
   for( int j = 0; j < how; j++ )
      for( uint64_t i = 0; i < msingl * nsingl; i++ )
         i = ( i + 1 ) - 1;
};

static pthread_barrier_t bstart, bfinish;
struct interv { uint64_t s, f; };
interv *trtime;

void* threadfunc ( void* data ) {
   // все потоки после создания должны "застрять" на входном
   // барьере, чтобы потом одновременно "сорваться" в исполнение...
```

```
   pthread_barrier_wait( &bstart );
   int id = pthread_self() - 2;
   trtime[ id ].s = ClockCycles();
   workproc( (int)data );
   trtime[ id ].f = ClockCycles();
   pthread_barrier_wait( &bfinish );
   return NULL;
};

int main( int argc, char *argv[] ) {
   // здесь только обработка многочисленных ключей...
   int opt, val, nthr = 1, nall = SHRT_MAX;
   while ( ( opt = getopt( argc, argv, "t:n:p:a:" ) )
           != -1 ) {
      switch( opt ) {
         case 't' :
            if( sscanf( optarg, "%i", &val ) != 1 )
               perror( "parse command line failed" ),
               exit( EXIT_FAILURE );
            if( val > 0 && val <= SHRT_MAX ) nthr = val;
            break;
         case 'p' :
            if( sscanf( optarg, "%i", &val ) != 1 )
               perror( "parse command line failed" ),
               exit( EXIT_FAILURE );
            if( val != getprio( 0 ) )
                  if( setprio( 0, val ) == -1 )
                 perror( "priority isn't a valid" ),
                 exit( EXIT_FAILURE );
            break;
         case 'n' :
            if( sscanf( optarg, "%i", &val ) != 1 )
               perror( "parse command line failed" ),
               exit( EXIT_FAILURE );
            if( val > 0 ) nsingl *= val;
            break;
         case 'a' :
            if( sscanf( optarg, "%i", &val ) != 1 )
               perror( "parse command line failed" ),
               exit( EXIT_FAILURE );
            if( val > 0 ) nall = val;
            break;
         default :
            exit( EXIT_FAILURE );
       }
   };
   // ... вот здесь начинается собственно сама программа:
   if( nthr > 1 )
      cout << "Multi-thread evaluation, thread number = "
           << nthr;
   else cout << "Single-thread evaluation";
```

```
    cout << " , priority level: " << getprio( 0 ) << endl;
    _clockperiod clcout;
    ClockPeriod( CLOCK_REALTIME, NULL, &clcout, 0 );
    // интервал диспетчеризации - 4 периода tickslice
    // (системного тика):
    cout << "rescheduling = \t"
         << clcout.nsec * 4. / 1000000. << endl;
    // калибровка времени выполнения в одном потоке
    const int NCALIBR = 512;
    uint64_t tmin = 0, tmax = 0;
    tmin = ClockCycles();
    workproc( NCALIBR );
    tmax = ClockCycles();
    cout << "calculating = \t"
         << cycle2milisec ( tmax - tmin ) / NCALIBR << endl;
    // а теперь контроль времени многих потоков
    if( pthread_barrier_init( &bstart, NULL, nthr )  != EOK )
       perror( "barrier init" ), exit( EXIT_FAILURE );
    if( pthread_barrier_init( &bfinish, NULL, nthr + 1 )
        != EOK )
       perror( "barrier init" ), exit( EXIT_FAILURE );
    trtime = new interv [ nthr ];
    int cur = 0, prev = 0;
    for( int i = 0; i < nthr; i++ ) {
       // границы участков работы для каждого потока:
      cur = (int)floor( (double)nall / (double)nthr * ( i + 1 ) + .5 );
       prev = (int)floor( (double)nall / (double)nthr * i + .5 );
       if( pthread_create( NULL, NULL, threadfunc,
                           (void*)( cur - prev ) ) != EOK )
          perror( "thread create" ), exit( EXIT_FAILURE );
    };
    pthread_barrier_wait( &bfinish );
    for( int i = 0; i < nthr; i++ ) {
       tmin = ( i == 0 ) ? trtime[ 0 ].s : __min( tmin, trtime[ i ].s );
       tmax = ( i == 0 ) ? trtime[ 0 ].f : __max( tmax, trtime[ i ].f );
    };
    cout << "evaluation = \t"
         << cycle2milisec ( tmax - tmin ) / nall << endl;
    pthread_barrier_destroy( &bstart );
    pthread_barrier_destroy( &bfinish );
    delete trtime;
    exit( EXIT_SUCCESS );
};
```

Логика этого приложения крайне проста:

- Есть некоторая продолжительная по времени рабочая функция (`workproc`), выполняющая массированные вычисления.
- Многократно (это число определяется ключом запуска `a`) выполняется рабочая функция. Хорошо (то есть корректнее), если время ее единичного выполнения, которое задается ключом `n`, больше интер-

вала диспетчеризации системы (в системе установлена диспетчеризация по умолчанию – круговая, или карусельная).

- Весь объем этой работы делится поровну (или почти поровну) между несколькими (ключ `t`) потоками.
- Сравниваем усредненное время единичного выполнения рабочей функции для разного числа выполняющих потоков (в выводе "`calculating`" – это время эталонного вычисления в одном главном потоке, а "`evaluation`" – время того же вычисления, но во многих потоках).
- Для того чтобы иметь еще большую гибкость, предоставляется возможность переопределять приоритет, под которым в системе все это происходит (ключ `p`).

Вот самая краткая сводка результатов (1-я строка вывода переносится для удобства чтения):

```
# t1 -n1 -t1000 -a2000
Multi-thread evaluation, thread number = 1000 ,
priority level: 10
rescheduling =  3.99939
calculating =   1.04144
evaluation =    1.08001

# t1 -n1 -t10000 -a20000
Multi-thread evaluation, thread number = 10000 ,
priority level: 10
rescheduling =  3.99939
calculating =   1.04378
evaluation =    1.61946

# t1 -n5 -a2000 -t1
Single-thread evaluation ,
priority level: 10
rescheduling =  3.99939
calculating =   5.07326
evaluation =    5.04726

# t1 -n5 -a2000 -t2
Multi-thread evaluation, thread number = 2 ,
priority level: 10
rescheduling =  3.99939
calculating =   5.06309
evaluation =    5.04649

# t1 -n5 -a2000 -t20
Multi-thread evaluation, thread number = 20 ,
priority level: 10
rescheduling =  3.99939
calculating =   5.06343
evaluation =    4.96956

# t1 -n5 -p51 -a512 -t1
```

```
Single-thread evaluation , priority level: 51
rescheduling =  3.99939
calculating =   4.94502
evaluation =    4.94511

# t1 -n5 -p51 -a512 -t11
Multi-thread evaluation,
thread number = 11 , priority level: 51
rescheduling =  3.99939
calculating =   4.94554
evaluation =    4.94549

# t1 -n5 -p51 -a512 -t111
Multi-thread evaluation,
thread number = 111 , priority level: 51
rescheduling =  3.99939
calculating =   5.02755
evaluation =    4.94487

# t1 -n5 -p51 -a30000 -t10000
Multi-thread evaluation,
thread number = 10000 , priority level: 51
rescheduling =  3.99939
calculating =   4.94575
evaluation =    5.31224
```

Краткий и, возможно, несколько парадоксальный итог этого теста может звучать так: при достаточно высоком уровне приоритета (выше 12–13, когда на его выполнение не влияют процессы обслуживания клавиатуры, мыши и др.) время выполнения в «классическом» последовательном коде и в многопоточном коде (где несколько тысяч потоков!) **практически не различаются**. Различия не более 8%, причем в обе стороны, что мы склонны считать «статистикой эксперимента». К обсуждению этого якобы противоречащего здравому смыслу феномена мы еще вернемся.

А пока посмотрим на текст примера, что и является нашей главной задачей. Обсуждаемое приложение вполне работоспособно в QNX и с большой вероятностью в большинстве других UNIX-систем, но вот в Linux оно завершится аварийно. Причина этого кроется в операторах:

```
int id = pthread_self() - 2;
trtime[ id ].s = ...
```

Это дает повод лишний раз обратиться к вопросу «POSIX-совместимости». POSIX описывает, что TID потока присваивается: а) в рамках процесса, которому принадлежит поток; б) начиная со значения 1, соответствующего главному потоку приложения. В Linux, выполняющем и `pthread_create()`, и `fork()` через единый системный вызов `_clone()`, сделано небольшое «упрощение», навязанное в том числе и гонкой за повышением производительности: TID присваиваются из единого ря-

да PID. И сразу же «вылезает» несовместимость, ведущая к аварийному завершению показанного выше приложения. В последних редакциях ядра Linux делаются изменения по приведению механизмов параллельности к общей POSIX-модели.

Этот момент сам по себе достаточно интересен, поэтому остановимся на нем подробнее, для чего создадим простейший программный тест [1]:

```
#define TCNT 10

void * test (void *in) {
   printf ("pid %ld, tid %ld\n", getpid(), pthread_self() ) ;
   return NULL;
};

int main( int argc, char **argv, char **envp ) {
   pthread_t tid[TCNT];
   int i, status;
   for( i = 0; i < TCNT; i++ ) {
      status = pthread_create(&tid[i], NULL, test, NULL);
      if( status != 0 )
         err( EXIT_FAILURE, "pthread_create()" );
   };
   return(EXIT_SUCCESS);
};
```

Результаты выполнения этого теста в нескольких POSIX-совместимых ОС различны и весьма красноречивы:

```
$ uname -sr
Linux 2.4.21-0.13mdk
$ ./test_pthread
pid 2008, tid 16386
pid 2009, tid 32771
pid 2010, tid 49156
pid 2011, tid 65541
pid 2012, tid 81926
pid 2013, tid 98311
pid 2014, tid 114696
pid 2015, tid 131081
pid 2016, tid 147466
pid 2017, tid 163851
```

А вот результат эволюции в направлении POSIX при переходе от ядра Linux 2.4.x к 2.6.x (алгоритм формирования TID все еще остается загадочным, но уже выполняются требования POSIX о выделении TID в рамках единого PID):

---

[1] Этот тест и его результаты для Linux подсказаны одним из участников (имя нам неизвестно) обсуждений на *http://qnxclub.net.forum*.

```
$ uname -sr
Linux 2.6.3-7mdk
$ ./test_pthread
pid 13929, tid 1083759536
pid 13929, tid 1092156336
pid 13929, tid 1100549040
pid 13929, tid 1108941744
pid 13929, tid 1117334448
pid 13929, tid 1125727152
pid 13929, tid 1134119856
pid 13929, tid 1142512560
pid 13929, tid 1150905264
pid 13929, tid 1159297968
```

И наконец, тот же тест, выполненный в QNX 6.2.1:

```
# uname -a
QNX home 6.2.1 2003/01/08-14:50:46est x86pc x86
# ptid
pid 671779, tid 2
pid 671779, tid 3
pid 671779, tid 4
pid 671779, tid 5
pid 671779, tid 6
pid 671779, tid 7
pid 671779, tid 8
pid 671779, tid 9
pid 671779, tid 10
pid 671779, tid 11
```

Спорадическая диспетчеризация

Системы реального времени принципиально отличаются от систем общего назначения тем, что для таких систем важна не только корректность выполнения возложенных на них функций, но и время, за которое эти функции реализуются. Можно даже сказать, что для задач реального времени опоздание с выполнением практически эквивалентно невыполнению задачи: требуемая реакция или управляющее воздействие не поступили в срок. Предельный срок, в который задача реального времени должна быть выполнена, называют **критическим сроком обслуживания** (deadline).

Если система реального времени реализуется как многопоточная система (а в настоящее время такой вариант рассматривается фактически как стандартный), то при ее разработке зачастую возникает проблема определения того, действительно ли все задачи реального времени, конкурирующие в системе за вычислительный ресурс, успевают выполниться в их критический срок обслуживания.

Примечание

Здесь мы следуем «классической» модели обсуждения из области систем реального времени, хотя уместнее было бы акцентировать внимание не на абсолютной минимизации времени приложения, а именно на том, что приложение обязано «уложиться» в некоторый критический интервал времени (см. выше). Величина же того, насколько быстро приложение выполнит свои критические функции (если оно укладывается в критический интервал) по принципу «меньше – больше», практически уже не имеет никакого значения. Из этого не совсем четкого толкования сложился общий стереотип, состоящий в том, что системы реального времени (в частности, операционные системы реального времени) принято считать «быстрыми» (в том смысле, что они потенциально могут исполнять аналогичные функции быстрее, чем системы общего назначения). Этот взгляд в корне ошибочен: системы реального времени в общем случае, скорее, будут даже «медленнее», чем системы общего назначения, за счет более тщательной отработки операций, например диспетчеризации и переключений контекстов. Во многих случаях можно ожидать, что при многократном выполнении участка кода средняя величина времени его выполнения в ОС общего назначения будет ниже, но вот дисперсия этой средней величины будет намного ниже в системах реального времени.

На сегодняшний день существует несколько систем математического анализа временных характеристик систем реального времени, призванных помочь разработчику в построении системы, распределении приоритетов между задачами и, в конечном счете, определении **диспетчеризуемости** системы. Систему называют **диспетчеризуемой**, если все ее задачи укладываются в свои сроки критического обслуживания.

Одна из наиболее известных систем математического анализа временных характеристик систем реального времени с периодическим поступлением запросов на выполнение задач называется «Частотно-монотонный анализ» (ЧМА – Rate Monotonic Analizing) [13]. Свое название эта система получила от ее основного принципа: **«Чем короче период поступления (выше частота) задачи, тем выше ее приоритет»**. Как уже говорилось, ЧМА предназначен для анализа систем реального времени, в которых каждая задача реального времени обрабатывается со своим периодом, причем еще одним ограничением ЧМА является условие, что период поступления задачи является также и ее критическим сроком обслуживания. В настоящее время появился ряд новых методов анализа характеристик систем реального времени для случаев критических сроков обслуживания, больших или меньших периода поступления, но здесь мы не будем на них останавливаться.

К сожалению, практически невозможно создать эффективную методику анализа систем с полностью случайными сроками поступления задач реального времени. Однако на практике такие ситуации в чистом виде встречаются не особо часто. В отличие от задач с полностью случайным сроком поступления, в математическом анализе систем реального времени рассматриваются так называемые **спорадические задачи,**

то есть задачи, последующий срок поступления которых может наступить не ранее некоторого времени после их предыдущего поступления.

Планирование обслуживания таких задач можно свести к планированию периодических задач и, таким образом, провести для них анализ диспетчеризуемости. Для этого теория ЧМА предлагает введение дополнительной периодической задачи (называемой **спорадический сервер**), которая проводит обслуживание непериодических (спорадических) задач.

Алгоритм работы такого сервера [13] следующий:

- **Шаг 1**. Если спорадический запрос прибывает и сервер не может его обработать, потому что уже занят или не имеет свободного ресурса вычислений, запрос будет поставлен в очередь обработки.
- **Шаг 2**. Если получен спорадический запрос и сервер может его обработать, он делает следующее:
- **Шаг 2a**. Выполняется до служебного завершения или истощения ресурса вычисления.
- **Шаг 2c**. Уменьшает текущий ресурс вычисления на используемое количество и на столько же увеличивает его ресурс вычисления в точке пополнения.

Для реализации теоретически обобщенной модели спорадического сервера в качестве механизма, реализующего эту модель, в QNX 6.2.1 была введена специализированная дисциплина диспетчеризации – **спорадическая**.

Сутью спорадической диспетчеризации в QNX является установка для соответствующего потока двух значений приоритета: основного (normal) и фонового (foreground). В момент запуска потока, подчиняющегося спорадической диспетчеризации (момент времени 0), поток имеет запас времени (C), называемый **начальным бюджетом** (initial budget) потока, в течение которого поток выполняется со своим основным приоритетом (N). Когда же запас времени исчерпывается, его приоритет понижается до уровня фонового (L). Через некоторый период времени T происходит **пополнение** (replenishment) запаса времени потока до значения начального бюджета, и он снова может выполняться с основным приоритетом.

Рассмотрим порядок выполнения такого потока подробнее. В начальный момент времени после запуска поток имеет приоритет N и время C для выполнения с этим приоритетом. Если поток блокируется на время R, то запас времени все равно расходуется и пополнение этого запаса может произойти только через период T после начала выполнения потока. Если же поток вытесняется более приоритетным, то расход его запаса времени прекращается. Когда управление возвращается к потоку, он вновь начинает тратить оставшееся количество времени на основном приоритете. Однако с момента повторного начала выполнения потока начинается отсчет нового периода до момента пополнения.

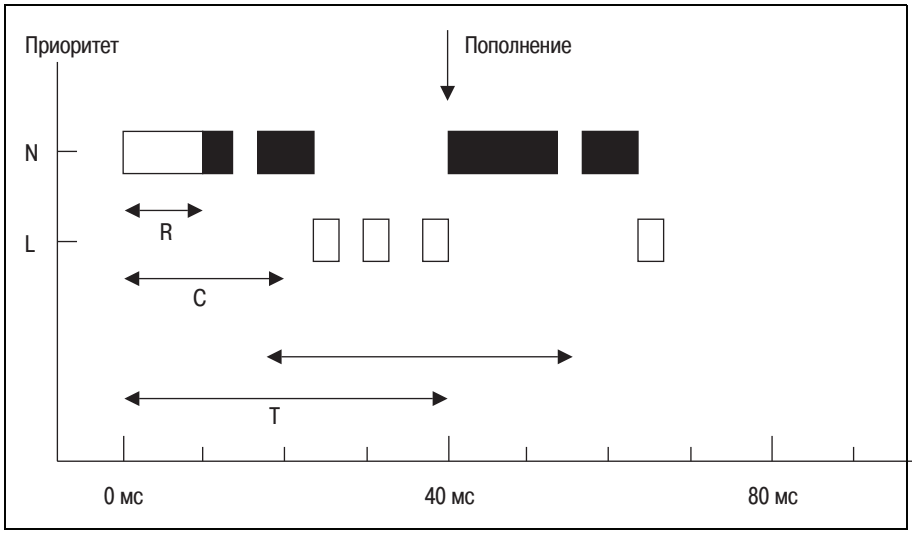

Рис. 2.6. *Периодическое выполнение спорадической задачи*

На рис. 2.6 проиллюстрирована работа спорадического потока. После запуска (момент времени 0) поток переходит в блокированное состояние на время R (10 мс), но его бюджет все равно расходуется. Поток становится активным, но через 3 мс (13 мс от начала выполнения) вытесняется более приоритетным потоком. Факт вытеснения означает, что через период пополнения T (40 мс) бюджет потока будет пополнен на израсходованную величину (13 мс). Еще через 3 мс более приоритетный поток заканчивает свою работу и управление возвращается назад. От начального бюджета потока C (20 мс) осталось еще 7 мс, и поток выполняется это время с основным приоритетом. При этом от повторного начала его выполнения (16 мс) отсчитывается новый период пополнения, то есть через 56 мс бюджет потока будет пополнен на 7 мс. После полного исчерпания бюджета приоритет потока понижается до фонового (L) и поток может вытесняться или нет в зависимости от приоритетов остальных потоков в системе. После наступления очередного времени пополнения бюджет потока восстанавливается на израсходованную в этом периоде величину и т. д.

Если поток много раз вытесняется в период своей работы с основным приоритетом, то его выполнение может превратиться в многократное колебание с высокой частотой между основным и фоновым приоритетами. Поэтому в QNX 6.2.1 в параметрах для спорадической диспетчеризации можно установить (ограничить) максимальное количество пополнений бюджета за период.

Как уже описывалось выше, структура `sched_param` содержит в своем составе, в частности, еще и структуру параметров для спорадической

диспетчеризации (при других типах диспетчеризации эта часть не используется):

```
struct {
   _INT32          __ss_low_priority;
   _INT32          __ss_max_repl;
   struct timespec __ss_repl_period;
   struct timespec __ss_init_budget;
} __ss;
```

где `low_priority` – фоновый приоритет; `max_repl` – максимальное количество пополнений бюджета за период; `repl_period` – период пополнения бюджета и `init_budget` – начальный бюджет.

Соображения производительности

Выполним «симметричный» тест аналогично тому, как это делалось для переключения контекстов процессов (стр. 44), но теперь применительно к потокам (*файл p5t.cc*). При этом мы постараемся максимально сохранить принципы функционирования, имевшие место в приложении «Затраты на взаимное переключение процессов» (*файл p5.cc*) (естественно, из-за принципиального различия механизмов тексты кодов будут существенно отличаться).

Затраты на взаимное переключение потоков

```
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>
#include <iostream.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <sys/neutrino.h>

unsigned long N = 1000;

// потоковая функция:
void* threadfunc ( void* data ) {
   uint64_t t = ClockCycles();
   for( unsigned long i = 0; i < N; i++ ) sched_yield();
   t = ClockCycles() - t;
   // дать спокойно завершиться 2-му потоку до начала вывода
   delay( 100 );
   cout << pthread_self() << "\t: cycles - " << t
        << "; on sched - " << ( t / N ) / 2 << endl;
   return NULL;
};

int main( int argc, char *argv[] ) {
   int opt, val;
   while ( ( opt = getopt( argc, argv, "n:" ) ) != -1 ) {
```

```
      switch( opt ) {
         case 'n' :   // переопределения числа переключений
            if( sscanf( optarg, "%i", &val ) != 1 )
               cout << "parse command line error" << endl,
               exit( EXIT_FAILURE );
            if( val > 0 ) N = val;
            break;
         default :
            exit( EXIT_FAILURE );
       }
   };
   const int T = 2;
   pthread_t tid[ T ];
   // создать взаимодействующие потоки
   for( int i = 0; i < T; i++ )
      if( pthread_create( tid + i, NULL, threadfunc, NULL )
          != EOK )
         cout << "thread create error",
         exit( EXIT_FAILURE );
   // и дожидаться их завершения . . .
   for( int i = 0; i < T; i++ )
      pthread_join( tid[ i ], NULL );
   exit( EXIT_SUCCESS );
};
```

Результаты выполнения программы:

```
# nice -n-19 p5t -n100
2       : cycles - 79490; on sched - 397
3       : cycles - 78350; on sched - 391
# nice -n-19 p5t -n1000
2       : cycles - 753269; on sched - 376
3       : cycles - 752069; on sched - 376
# nice -n-19 p5t -n10000
2       : cycles - 7494255; on sched - 374
3       : cycles - 7493225; on sched - 374
# nice -n-19 p5t -n100000
2       : cycles - 74897795; on sched - 374
3       : cycles - 74895800; on sched - 374
# nice -n-19 p5t -n1000000
2       : cycles - 748850811; on sched - 374
3       : cycles - 748850432; on sched - 374
```

Как и в случае с процессами, результаты отличаются очень высокой устойчивостью при изменении «объема вычислений» на 4 порядка, однако по своим величинам значения для потоков почти в 2 раза меньше, чем для процессов (стр. 45).

Завершение потока

Как и в случае обсуждавшегося ранее завершения процесса, для потоков мы будем отчетливо различать случаи:

- «естественного» завершения выполнения потока из кода самого потока;
- завершения потока извне, из кода другого потока или по сигналу. Для этого действия, в отличие от «естественного» завершения, будем использовать другой термин – отмена.

Завершение потока происходит при достижении функцией потока своего естественного конца и выполнения оператора `return` (явно или неявно) или выполнения потоком вызова:

```
void pthread_exit( void* value_ptr )
```

где `value_ptr` – указатель на результат выполнения потока.

При выполнении `pthread_exit()` поток завершается. Если этот поток принадлежит к категории ожидаемых, он может возвратить результат своей работы другому потоку, ожидающему его завершения на вызове `pthread_join()` (только один поток может получить результат завершения). Если же этот поток отсоединенный, то по его завершении все системные ресурсы, задействованные потоком, освобождаются немедленно.

Перед завершением потока будут выполнены все завершающие процедуры, помещенные в стек завершения, а также деструкторы собственных данных потока, о которых мы говорили ранее. Для последнего потока процесса вызов `pthread_exit()` эквивалентен `exit()`.

Возврат результата потока

Выше отмечено, что вызов `pthread_exit()`, завершающий ожидаемый поток, может передать результат выполнения потока. То же действие может быть выполнено и оператором `return` потоковой функции, которая из прототипа ее определения должна возвращать значение типа `void*`.

В обоих случаях результат может иметь сколь угодно сложный структурированный тип; никакая типизация результата не предусматривается (тип `void*`). Важно, чтобы код, ожидающий результата на вызове `pthread_join()`, понимал его так же, как и функция потока, возвращающая этот результат.

Другим условием является то, что переменная «результат» должна существовать к моменту вызова `pthread_join()`, то есть вполне возможно, что уже далеко после завершения самой функции ожидаемого потока. Этому условию не удовлетворяют, например, любые локальные для функции потока объекты, размещаемые в стеке. Приведем пример часто допускаемой ошибки. Следующая функция потока практически обречена на ошибку защиты памяти:

```
void* threadfunc ( void* data ) {
   int res;          // результат некоторых вычислений
   ...
   res = ...
```

```
   pthread_exit( &res );
};
```

А вот один из многих допустимых вариантов:

```
void* threadfunc ( void* data ) {
   struct data *res = new struct;   // результат некоторых вычислений
   ...
   *res = ...
   pthread_exit( res );
};
...
pthread_t tid;
pthread_create( &tid, NULL, threadfunc, NULL );
struct data *res;
pthread_join( tid, &res );
...
delete res;
```

Недостатком этого варианта является то, что память под блок данных результата выделяется в одной программной единице (в функции потока), а освобождаться должна в другой (в коде, ожидающем результата), при этом сами программные единицы могут размещаться даже в различных файлах исходного кода. (Здесь ситуация зеркально подобна ранее рассмотренному случаю передачи параметров в функцию создаваемого потока.)

Уничтожение (отмена) потока

Корректное завершение выполняющегося потока «извне», из другого потока (то есть асинхронно относительно прерываемого потока), – задача отнюдь не тривиальная; она намного сложнее аналогичной задачи прерывания процесса. Это связано с обсуждавшимся ранее при рассмотрении завершения потоков временем жизни объектов, которые могут быть использованы потоком к моменту его отмены (блоки динамической памяти, файловые дескрипторы, примитивы синхронизации и другие объекты системы).

Если для процесса в перечень «опасных» (с точки зрения завершения) объектов включаются только объекты со временем жизни выше уровня процесса (их число достаточно ограничено), то для потока в число таких объектов включаются уже все объекты со временем жизни процесса (process-persistent). Завершающийся (покидающий процесс) поток обязан оставить все объекты процесса в состоянии, пригодном для их дальнейшего использования другими потоками процесса.

Далее мы подробно рассмотрим то множество предосторожностей, которыми «обложена» отмена потока. Однако именно по причине их «множества» стоит сформулировать краткое правило: не пытайтесь завершать поток извне его функции потока, если для этого нет в высшей степени обоснованной необходимости (а такая необходимость дей-

ствительно бывает, но крайне редко). Даже в крайнем случае следует рассмотреть возможность вместо отмены потока послать ему сигнал (даже не только «сигнал UNIX», а в более широком смысле – «некоторое сообщение»), который, обрабатываясь в контексте потока, после корректных завершающих действий вызовет его завершение. (Как обращаться с сигналами в потоке, будет детально рассмотрено позже.)

Для отмены (принудительного завершения) потока используется вызов:

```
int pthread_cancel( pthread_t thread );
```

где в качестве параметра `thread` указывается TID отменяемого потока. Однако этот вызов не отменяет поток, а только запрашивает завершение потока. В зависимости от статуса отмены, который мы сейчас рассмотрим, поток может перейти (или нет) к действию завершения, которое состоит в том, что:

- выполняются все процедуры завершения, занесенные ранее в стек завершения вызовами `pthread_cleanup_push()`;
- выполняются деструкторы собственных данных потока;
- отменяемый поток завершается;
- процесс отмены – асинхронный с точки зрения вызывающего `pthread_cancel()` кода, поэтому вызывающий отмену поток должен дождаться завершения потока на вызове `pthread_join()`.

Прежде всего, поток может вообще отказаться выполнять любые отмены, вызвав из своей функции потока:

```
int pthread_setcancelstate( int state, int* oldstate );
```

где `state` и `oldstate` – устанавливаемое и установленное ранее (возвращаемое вызовом) состояния отмены потока, которые могут принимать значения `PTHREAD_CANCEL_DISABLE` либо `PTHREAD_CANCEL_ENABLE`. (Естественно, как и во многих функциях с подобным прототипом, значением `oldstate` может быть `NULL`, и тогда нам не нужно возвращать ранее установленное состояние.)

Далее, даже если для потока установлено **состояние** завершаемости (также называемое «состоянием отмены») `PTHREAD_CANCEL_ENABLE` (это значение по умолчанию при создании потока), поток может переопределить еще и **тип** отмены, вызвав:

```
int pthread_setcanceltype( int type, int* oldtype )
```

где `type` и `oldtype` – как и в предыдущем случае, новое и ранее установленное значения типа отмены потока, которые могут принимать значения `PTHREAD_CANCEL_ASYNCHRONOUS` (асинхронный по отмене поток) либо `PTHREAD_CANCEL_DEFERRED` (синхронный по отмене поток). Значением по умолчанию, устанавливаемым при создании потока, является `PTHREAD_CANCEL_DEFERRED`, хотя предписываемым POSIX умолчанием является `PTHREAD_CANCEL_ASYNCHRONOUS`.

Обе рассмотренные функции установок[1] параметров отмены при успешном выполнении возвращают значение EOK.

Итак, действия потока на запрос его завершения будут определяться текущей комбинацией двух установленных для него параметров: состоянием и типом отмены.

Теперь о том, чем же отличается отмена асинхронно и синхронно завершаемых потоков. Поток с асинхронным типом отмены (установленный с PTHREAD_CANCEL_ASYNCHRONOUS) может быть отменен в любой произвольный момент времени, то есть он всегда «свободен» для отмены и отмена производится немедленно. Поток с синхронным типом отмены (установленный с PTHREAD_CANCEL_DEFERRED) может быть остановлен только в тех точках выполнения потока, когда ему «удобно», и соответствующие места в программе называются точками отмены. При поступлении запроса на отмену такого потока (после выполнения извне pthread_cancel()) запрос помещается в очередь, а процесс отмены активизируется только после того, как отменяемый поток в ходе своего выполнения достигнет очередной точки отмены. Как определяются (создаются) точки отмены в коде потока? Для этого служит функция:

```
void pthread_testcancel( void );
```

Каждый вызов pthread_testcancel() тестирует очередь поступивших запросов на отмену на предмет наличия запросов, и если таковой запрос есть, процесс отмены активизируется. Если в коде отсутствуют вызовы pthread_testcancel(), то в нем практически отсутствуют точки отмены и поток становится неотменяемым (подобно установке его состояния отмены в PTHREAD_CANCEL_DISABLE). Поэтому при выполнении длительных вычислений функцию pthread_testcancel() следует периодически вызывать в потоковой функции в тех точках, где потенциальная отмена потока не опасна.

Примечание

(**Очень важно!**) Достаточно много библиотечных функций могут сами устанавливать точки отмены. Более того, такие функции могут косвенно вызываться из других функций в программе и тем самым неявно устанавливать точки отмены. Информацию о таких функциях следует искать в справочной man-странице по функции pthread_testcancel(). В результате этого эффекта можно получить отмену потока не в той точке, которую вы считаете безопасной и которую явно отмечаете вызовом pthread_testcancel(), а ранее этой точки – когда будет вызвана одна из таких функций. А это, очевидно, вовсе не то, на что вы рассчитывали!

[1] Согласно стандарту POSIX установки состояния и типа завершаемости могут быть сделаны только из уже выполняющегося кода потока (при старте потока эти параметры установлены в значения по умолчанию). QNX делает расширение, позволяя установить соответствующие флаги в атрибутной записи еще до создания потока. Подробнее об этом говорилось при обсуждении создания потока.

Если состояние отмены потока, как это описывалось ранее, установлено в PTHREAD_CANCEL_DISABLE, то никакая расстановка точек отмены не имеет эффекта и поток остается неотменяемым.

Покажем, как могут быть использованы все эти предосторожности в коде функции потока, чтобы сделать код безопасным с позиции возможной асинхронной отмены потока извне:

```
void* function( void* data ) {
   int state;
   pthread_setcancelstate( PTHREAD_CANCEL_DISABLE, &state );
   // ... здесь выполняется инициализация ...
   pthread_setcanceltype ( PTHREAD_CANCEL_DEFERRED, NULL );
   pthread_setcancelstate ( &state, NULL );
   while( true ) {
      struct blockdata *blk = new blockdata;
      // ... обработка блока данных blk ...
      delete blk;
      pthread_testcancel ();
   };
};

...
pthread_t tid;
pthread_create ( &tid, NULL, function, NULL );
...
pthread_cancel ( tid );          // отмена потока
void* res;
pthread_join ( tid, &res );      // ожидание отмены
if( res != PTHREAD_CANCELED )
    cout << "Что-то не так!" <<  endl;
```

Наконец, в QNX (но не в POSIX) существует вызов, подобный pthread_cancel(), принудительно отменяющий поток независимо от его установок («желания»):

```
int pthread_abort ( pthread_t thread );
```

В отличие от pthread_cancel(), этот вызов принудительно и немедленно отменяет поток. Кроме того, никакие процедуры завершения и деструкторы собственных данных потока не выполняются. Очевидно, что в результате такого «завершения» состояния объектов процесса будут просто неопределенными, поэтому такой вызов крайне опасен. При таком способе отмены в программный код, ожидающий завершения на pthread_join(), в качестве результата завершения возвращается константа (тип void*) PTHREAD_ABORTED (аналогично возвращается константа PTHREAD_CANCELED при выполнении pthread_cancel()).

Но и этих мер безопасности недостаточно на все случаи жизни, поэтому механизм потоков предусматривает еще один уровень (механизм) страховки.

Стек процедур завершения

Для поддержания корректности состояния объектов процесса каждый поток может помещать (добавлять) в стек процедур завершения (thread's cancellation-cleanup stack) функции, которые при завершении (pthread_exit() или return) или отмене (по pthread_cancel()) выполняются в порядке, обратном помещению. Для манипуляции со стеком процедур завершения предоставляются вызовы (оба вызова реализуются макроопределениями, но это не суть важно[1]):

```
void pthread_cleanup_push( void (routine)(void*), void* arg )
```

где routine – адрес функции завершения, помещаемой в стек; arg – указатель блока данных, который будет передан routine при ее вызове.

Функции завершения (начиная с вершины стека) вызываются со своими блоками данных в случаях, когда:

- поток завершается, выполняя pthread_exit();
- активизируется действие отмены потока, ранее запрошенное по вызову pthread_cancel();
- выполняется второй (комплементарный к pthread_cleanup_push()) вызов с ненулевым значением аргумента:

  ```
  void pthread_cleanup_pop( int execute );
  ```

  Этот вызов выталкивает из стека последнюю помещенную туда pthread_cleanup_push() функцию завершения и, если значение execute ненулевое, выполняет ее.

Вот как может выглядеть в этой технике безопасный (с позиции возможной асинхронной отмены потока) захват мьютекса:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void cleanup( void* arg ) { pthread_mutex_unlock( &mutex );  };

void* thread_function( void* arg ) {
   while( true ) {
      pthread_mutex_lock( &mutex );
      pthread_cleanup_push( &cleanup, NULL );
      {
        // все точки отмены должны быть расставлены в этом блоке!
      };
      pthread_testcancel();
      pthread_cleanup_pop( 1 );
   };
};
```

[1] Разница выражается в том, что макрос pthread_cleanup_push() расширяется в фрагмент кода, открывающийся скобкой «{» без соответствующей скобки «}», аналогично pthread_cleanup_pop() закрывается «}», не имея открывающей скобки. Эти вызовы могут располагаться только парами, в противном случае возникнет лексическая ошибка, обнаруживаемая компилятором.

«Легковесность» потока

Вот теперь, завершив краткий экскурс использования процессов и потоков, можно вернуться к вопросу, который вскользь уже звучал по ходу рассмотрения: почему и в каком смысле потоки часто называют «легкими процессами» (LWP – lightweight process)?

Выполним ряд тестов по сравнительной оценке временных затрат на создание процесса и потока. Начнем с процесса (*файл p2-1.cc*):

Затраты на порождение нового процесса

```
struct mbyte {            // мегабайтный блок данных
#pragma pack( 1 )
   uint8_t data[ 1024 * 1024 ];
#pragma pack( 4 )
};

int main( int argc, char *argv[] ) {
   mbyte *blk = NULL;
   if( argc > 1 && atoi( argv[ 1 ] ) > 0 ) {
      blk = new mbyte[ atoi( argv[ 1 ] ) ];
   };
   uint64_t t = ClockCycles();
   pid_t pid = fork();
   if( pid == -1 ) perror ( "fork" ), exit( EXIT_FAILURE );
   if( pid == 0 ) exit( EXIT_SUCCESS );
   if( pid > 0 ) {
      waitpid( pid, NULL, WEXITED );
      t = ClockCycles() - t;
   };
   if( blk != NULL ) delete blk;
   cout << "Fork time: " << cycle2milisec( t )
        << " msec. [" << t << " cycles]" << endl;
   exit( EXIT_SUCCESS );
};
```

Эта программа сделана так, что может иметь один численный параметр: размер (в мегабайтах) блока условных данных (в нашем случае даже неинициализированных), принадлежащего адресному пространству процесса. (Функцию преобразования процессорных циклов в соответствующий миллисекундный интервал `cycle2milisec()` мы видели раньше, и поэтому в листинг она не включена.)

А теперь оценим временные затраты на создание клона процесса в зависимости от объема программы (мы сознательно использовали клонирование процесса вызовом `fork()`, а не загрузку `spawn*()` или `exec*()`, чтобы исключить из результата время загрузки образа процесса из файла):

```
# p2-1
Fork time: 3.4333 msec. [1835593 cycles]
```

```
# p2-1 1
Fork time: 17.0706 msec. [9126696 cycles]
# p2-1 2
Fork time: 31.5257 msec. [16855024 cycles]
# p2-1 5
Fork time: 70.7234 msec. [37811848 cycles]
# p2-1 20
Fork time: 264.042 msec. [141168680 cycles]
# p2-1 50
Fork time: 661.312 msec. [353566688 cycles]
# p2-1 100
Fork time: 1169.45 msec. [625241336 cycles]
```

Наблюдаются, во-первых, достаточно большие временные затраты на создание процесса (к этому мы еще вернемся), а во-вторых, близкая к линейной зависимость времени создания процесса от размера его образа в памяти и вариации этого времени на несколько порядков. Об этом уже говорилось при рассмотрении функции `fork()`: это следствие необходимости полного копирования образа адресного пространства родительского процесса во вновь создаваемое для дочернего процесса адресное пространство. При этом линейный рост времени копирования от размера образа процесса становится естественным (вот почему для образов таких задач при их построении посредством программы `make` в высшей степени целесообразно выполнить завершающую команду `strip` для уменьшения размера итогового образа задачи). Более того, это «высоко затратная» операция копирования, не в пример привычной функции `memcpy()`. Копирование производится между различными адресными пространствами обращением к средствам системы по принципу: скопировать N байт, начиная с адреса A адресного пространства P, по адресу, начиная с A (тот же адрес!) адресного пространства C. В большинстве других ОС некоторое смягчение вносит использование техники COW (copy on write), но и этот эффект кажущийся (см. выше подробное обсуждение при описании функции `fork()`).

На результаты наших оценок очень существенное влияние оказывают процессы кэширования памяти, что можно легко увидеть, экспериментируя с приложением, но затраты (число процессорных тактов) на выполнение `fork()` будут оценены очень грубо:

```
T = 3000000 + P * 6000
```

где `P` – размер (в килобайтах) файла образа программы, в которой выполняется `fork()`.

Теперь проведем столь же элементарный альтернативный тест (*файл p2-2.cc*) по созданию потока. (В случае потока время гораздо проще измерять и с более высокой точностью, но мы для сравнимости результатов почти текстуально сохраним предыдущий пример с включением в результат операторов завершения дочернего объекта, ожидания результата и т. д.)

Затраты на создание потока

```
void* threadfunc ( void* data ) { pthread_exit( NULL ); };

int main( int argc, char *argv[] ) {
   uint64_t t = ClockCycles();
   pthread_t tid;
   pthread_create( &tid, NULL, threadfunc, NULL );
   pthread_join( tid, NULL );
   t = ClockCycles() - t;
   cout << "Thread time, " << cycle2milisec( t )
        << " msec. [" << t << " cycles]" << endl;
   exit( EXIT_SUCCESS );
};
```

На результаты этого теста (в отличие от предыдущего) уже достаточно существенно влияет приоритет, под которым выполняется задача, поэтому проделаем его с достаточно высоким приоритетом (29):

```
# nice -n-19 p2-2
Thread time: 0.147139 msec. [78667 cycles]
# nice -n-19 p2-1
Fork time: 2.5366 msec. [1356179 cycles]
```

Вот так... время порождения нового «пустого» **процесса**, даже минимального размера (размер исполняемого файла этого процесса чуть больше 4 Кбайт), почти в 20 раз больше затрат на создание **потока**! А для процессов большого объема эта разница может доходить до 3–4 порядков (см. результаты первого теста).

Далее рассмотрим сравнительную эффективность с другой стороны: будет ли диспетчеризация многочисленных потоков, принадлежащих одному процессу, эффективнее диспетчеризации такого же количества отдельных процессов? Для процессов задача текстуально выглядит так (*файл p4-1.cc*):

```
void workproc( int how = 1 ) {
   const int nsingl = 1000, msingl = 30;
   for( int j = 0; j < how; j++ )  // ... имитация вычислений
      for( uint64_t i = 0; i < msingl; i++ )
         for( uint64_t k = 0; k < nsingl; k++ )
            k = ( k + 1 ) - 1;
};

int main( int argc, char *argv[] ) {
   int numpar = 1;
   if( argc > 1 && atoi( argv[ 1 ] ) > 0 )
      numpar = atoi( argv[ 1 ] );
   _clockperiod clcold;
   ClockPeriod ( CLOCK_REALTIME, NULL, &clcold, 0 );
   if( argc > 2 && atoi( argv[ 2 ] ) > 0 ) {
      _clockperiod clcnew = { atoi( argv[ 2 ] ) * 1000, 0 };
```

```
        ClockPeriod( CLOCK_REALTIME, &clcnew, &clcold, 0 );
    };
    timespec interval;
    sched_rr_get_interval( 0, &interval );
    cout << "Rescheduling interval = "
         << (double)interval. tv_nsec / 1000000.
         << " msec." << endl;
    uint64_t t = ClockCycles();
    for( int i = 0; i < numpar; i++ ) {
       pid_t pid = fork();
       if( pid == -1 ) perror ( "fork" ), exit( EXIT_FAILURE );
       if( pid == 0 ) {
          workproc( 1000 );
          exit( EXIT_SUCCESS );
       };
    };
    for( int i = 0; i < numpar; i++ ) wait3( NULL, WEXITED, NULL );
    t = ClockCycles() - t;
    cout << "Forks scheduling time: " << cycle2milisec( t )
         << " msec. [" << t << " cycles]" << endl;
    ClockPeriod( CLOCK_REALTIME, &clcold, NULL, 0 );
    exit( EXIT_SUCCESS );
};
```

Имитатором активной вычислительной нагрузки программы является функция `workproc()`, отличительной особенностью которой является то, что она при активной (хоть и бессмысленной) загрузке процессора не делает на всем интервале своего выполнения никаких системных вызовов, которые могли бы привести к вытеснению выполняющего ее потока.

Первым параметром программы является количество процессов, на которые распределяется общий объем вычислений. Но самое главное: начнем управлять размером периода временного системного тика.

Примечание

По умолчанию системный тик (для QNX 6.2.1) равен 1 мсек., но в принципе его значение можно уменьшать функцией `ClockPeriod()` вплоть до 10 мксек. Кстати, в описании именно этой функции присутствует замечание о том, что «*...период решедулирования равен 4 тикам, и это соотношение в системе нельзя изменить*».

Второй параметр запуска программы (при его наличии) и определяет размер периода системного тика, выраженный в микросекундах. (В конце выполнения задач подобного рода, изменяющих размер системного тика, нужно **обязательно** принять меры к восстановлению его прежнего значения даже в случаях экстремального и аварийного завершения задачи!) Для повышения достоверности тестов величина размера интервала диспетчеризации контролируется независимым образом (вызовом `sched_rr_get_interval()`).

При распараллеливании вычислительного объема между потоками эквивалентный код (*файл p4-2.cc*) будет иметь вид (используется та же функция `workproc()`, которую мы повторно не показываем):

```
void* threadfunc ( void* data ) {
   workproc( 100 );
   pthread_exit( NULL );
};

int main( int argc, char *argv[] ) {
   int numpar = 1;
   if( argc > 1 && atoi( argv[ 1 ] ) > 0 )
      numpar = atoi( argv[ 1 ] );
   pthread_t *tids = new pthread_t [ numpar ];
   _clockperiod clcold;
   ClockPeriod ( CLOCK_REALTIME, NULL, &clcold, 0 );
   if( argc > 2 && atoi( argv[ 2 ] ) > 0 ) {
      _clockperiod clcnew = { atoi( argv[ 2 ] ) * 1000, 0 };
      ClockPeriod( CLOCK_REALTIME, &clcnew, &clcold, 0 );
   };
   timespec interval;
   sched_rr_get_interval( 0, &interval );
   cout << "Rescheduling interval = "
        << (double)interval. tv_nsec / 1000000.
        << " msec." << endl;
   uint64_t t = ClockCycles();
   for( int i = 0; i < numpar; i++ )
      pthread_create( &tids[ i ], NULL, threadfunc, NULL );
   for( int i = 0; i < numpar; i++ )
      pthread_join( tids[ i ], NULL );
   t = ClockCycles() - t;
   cout << "Threads scheduling time: " << cycle2milisec( t )
        << " msec. [" << t << " cycles]" << endl;
   ClockPeriod( CLOCK_REALTIME, &clcold, NULL, 0 );
   exit( EXIT_SUCCESS );
};
```

Наконец, для сравнительного анализа выполним тот же объем вычислительной работы в одиночном потоке, то есть в последовательной «классической» программе (*файл p4-3.cc*):

```
int main( int argc, char *argv[] ) {
   int numpar = 1;
   if( argc > 1 && atoi( argv[ 1 ] ) > 0 )
      numpar = atoi( argv[ 1 ] );
   _clockperiod clcold;
   ClockPeriod( CLOCK_REALTIME, NULL, &clcold, 0 );
   if( argc > 2 && atoi( argv[ 2 ] ) > 0 ) {
      _clockperiod clcnew = { atoi( argv[ 2 ] ) * 1000, 0 };
      ClockPeriod( CLOCK_REALTIME, &clcnew, &clcold, 0 );
   };
   timespec interval;
```

```
    sched_rr_get_interval( 0, &interval );
    cout << "Rescheduling interval = "
         << (double)interval. tv_nsec / 1000000.
         << " msec." << endl;
    uint64_t t = ClockCycles();
    workproc( 1000 * numpar );
    t = ClockCycles() - t;
    cout << "Single scheduling time: " << cycle2milisec( t )
         << " msec. [" << t << " cycles]" << endl;
    ClockPeriod( CLOCK_REALTIME, &clcold, NULL, 0 );
    exit( EXIT_SUCCESS );
};
```

Выполняем 3 полученных теста для различных значений периода системного тика (показано группами по 3 запуска) в таком порядке: одиночный процесс, параллельные потоки, параллельные процессы:

```
# nice -n-19 p4-3 10
Rescheduling interval = 3.99939 msec.
Single scheduling time: 5928.8 msec. [3169850746 cycles]
# nice -n-19 p4-2 10
Rescheduling interval = 3.99939 msec.
Threads scheduling time: 5919.82 msec. [3165049513 cycles]
# nice -n-19 p4-1 10
Rescheduling interval = 3.99939 msec.
Forks scheduling time: 5962.21 msec. [3187713371 cycles]

# nice -n-19 p4-3 10 50
Rescheduling interval = 0.197788 msec.
Single scheduling time: 6427.33 msec. [3436394566 cycles]
# nice -n-19 p4-2 10 50
Rescheduling interval = 0.197788 msec.
Threads scheduling time: 6207.96 msec. [3319104030 cycles]
# nice -n-19 p4-1 10 50
Rescheduling interval = 0.197788 msec.
Forks scheduling time: 6029.23 msec. [3223548073 cycles]

# nice -n-19 p4-3 10 20
Rescheduling interval = 0.077104 msec.
Single scheduling time: 6745.37 msec. [3606433686 cycles]
# nice -n-19 p4-2 10 20
Rescheduling interval = 0.077104 msec.
Threads scheduling time: 6762.69 msec. [3615692975 cycles]
# nice -n-19 p4-1 10 20
Rescheduling interval = 0.077104 msec.
Forks scheduling time: 6647.42 msec. [3554062343 cycles]

# nice -n-19 p4-3 10 11
Rescheduling interval = 0.04358 msec.
Single scheduling time: 7517.74 msec. [4019381476 cycles]
# nice -n-19 p4-2 10 11
Rescheduling interval = 0.04358 msec.
```

```
Threads scheduling time: 7638.89 msec. [4084155676 cycles]
# nice -n-19 p4-1 10 11
Rescheduling interval = 0.04358 msec.
Forks scheduling time: 7679.29 msec. [4105758575 cycles]

# nice -n-19 p4-3 10 10
Rescheduling interval = 0.036876 msec.
Single scheduling time: 7937.35 msec. [4243731124 cycles]
# nice -n-19 p4-2 10 10
Rescheduling interval = 0.036876 msec.
Threads scheduling time: 8136.42 msec. [4350159949 cycles]
# nice -n-19 p4-1 10 10
Rescheduling interval = 0.036876 msec.
Forks scheduling time: 8172.35 msec. [4369372230 cycles]
```

Результаты могут показаться достаточно неожиданными: во всех 3-х вариантах (в группах) это практически одни и те же цифры – различия затрат на выполнение и в едином последовательном потоке, и во многих параллельных процессах (как предельные случаи) не превышают 0,5–2%! Но результат есть результат, и его нужно как-то интерпретировать, ведь, как известно, «из песни слова не выкинешь».

Эти результаты позволяют (пусть грубо и оценочно) «разложить» затраты производительности между обслуживанием системного таймера (службы времени ОС) и диспетчеризацией. Еще раз обратимся к отдельным выборочным результатам:

```
# nice -n-19 p4-3 10
Rescheduling interval = 3.99939 msec.
Single scheduling time: 5928.8 msec. [3169850746 cycles]
```

То есть на протяжении «работы» было 5928,8 / 0,9998475 = 5929 прерываний от службы времени.

```
# nice -n-19 p4-3 10 10
Rescheduling interval = 0.036876 msec.
Single scheduling time: 7937.35 msec. [4243731124 cycles]
```

На этот раз за счет уменьшения периода системного тика на 2 порядка на протяжении «работы» (того же объема полезной работы!) было уже 7937,35 / 0,009219 = 860977 событий диспетчеризации.

Поскольку объем работы программы, выполняемый в этих двух случаях, остается неизменным, то на обслуживание дополнительных 860 977 – 5929 = 855 048 системных тиков (совместно с 855 048 / 4 = 213 762 точками диспетчеризации) и потребовались те 4 243 731 124 – 3 169 850 746 = 1 073 880 378 дополнительных тактов процессора, или около 1256 тактов на один системный тик. Ранее мы уже получали оценки затрат собственно на переключение контекстов между процессами (617) и потоками (374), которые происходят каждый четвертый системный тик, то есть непосредственно переключение контекста «отъедает» в среднем 90–150 (1/4 часть затрат переключения контек-

ста) на каждый системный тик или, другими словами, не более 10% затрат на обслуживание службы системных часов.

Попытаемся осмыслить полученные результаты:

- Время переключения адресных пространств процессов, управляемых MMU аппаратно, в принципе должно быть продолжительнее времени переключения контекстов потоков и тем более восстановления контекста единого последовательного потока, но...
- ...но объем работы по обслуживанию каждого системного тика (прерывания таймера) настолько превышает объем операций переключения контекстов (рис. 2.7), что это практически полностью нивелирует разницу, будь то приложение в виде многих автономных процессов, многопоточное приложение или приложение в виде единого последовательного потока.

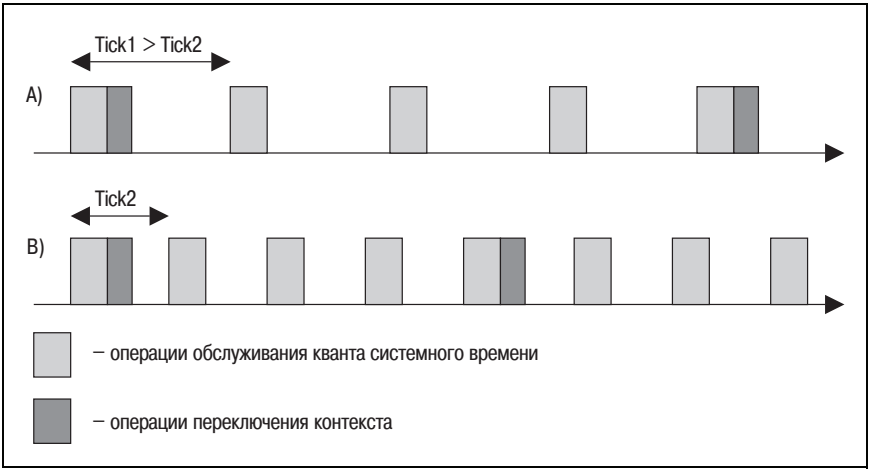

Рис. 2.7. *Эффекты, возникающие при принудительном изменении частоты системных часов*

На рис. 2.7 показана последовательность тиков системных часов и связанная с нею последовательность актов диспетчеризации. При уменьшении периода наступления системных тиков (частоты аппаратных прерываний от системных часов) в силу фиксированных объемов операций, требуемых как для одних, так и для других действий, относительная доля времени, остающаяся для выполнения полезной работы, падает.

- И это будет выполняться не только для потоков, диспетчеризуемых с дисциплиной RR (вытесняемых по истечении бюджета времени выделенного им кванта), но и для потоков с **любой** дисциплиной диспетчеризации, в том числе и FIFO, когда выполняющийся поток (а значит, поток наивысшего приоритета в системе) вообще «не собирается» никому передавать управление.

- Для программиста-разработчика результаты этого теста позволяют сформулировать правило, возможно абсурдное с позиций элементарной (но поверхностной) логики: **Распараллеливание задачи (если это возможно) на N ветвей (будь то использование потоков или процессов) практически не изменяет итоговое время ее выполнения.**

Еще одним побочным результатом рассмотрения можно назвать следующее: эффективность диспетчеризации потоков (сохранения и переключения контекстов), принадлежащих одному процессу, ни в чем не превосходит эффективность диспетчеризации группы потоков, принадлежащих различным процессам. И в этом своем качестве – эффективности периода выполнения – потоки в своей «легковесности» ничем не превосходят автономные параллельные процессы.[1]

В завершение воспользуемся все теми же тестовыми приложениями для ответа на часто задаваемый вопрос: «Насколько эффективно ОС QNX поддерживает приложения, содержащие большое («слишком большое») количество потоков? Посмотрим, как это выглядит. Все выполнения мы делаем при минимально возможном значении системного тика, когда ОС существенно более «озабочена» своими внутренними процессами, нежели процессом вычислений:

```
# nice -n-19 p4-2 2 10
Rescheduling interval = 0.036876 msec.
Threads scheduling time: 1555.43 msec. [831574415 cycles]
# nice -n-19 p4-2 20 10
Rescheduling interval = 0.036876 msec.
Threads scheduling time: 15642 msec. [8362674590 cycles]
# nice -n-19 p4-2 200 10
Rescheduling interval = 0.036876 msec.
Threads scheduling time: 161112 msec. [86134950020 cycles]
```

Наблюдается очень хорошая линейная зависимость итогового времени от числа потоков (от 2 до 200). Таким образом, время выполнения работы в каждом из потоков практически не зависит от общего числа параллельно выполняющихся с ним потоков.

Повторим то же самое, но уже для случая параллельных процессов:

```
# nice -n-19 p4-1 2 10
Rescheduling interval = 0.036876 msec.
Forks scheduling time: 1622.87 msec. [867633362 cycles]
# nice -n-19 p4-1 20 10
Rescheduling interval = 0.036876 msec.
Forks scheduling time: 16682.1 msec. [8918698991 cycles]
```

[1] Мы умышленно делаем акцент на этом месте, так как существует «красивая народная легенда» (и это утверждение встречается порой и в литературе), что потоки на периоде выполнения намного эффективнее (в смысле переключения контекста), чем процессы.

```
# nice -n-19 p4-1 200 10
Rescheduling interval = 0.036876 msec.
Forks scheduling time: 173398 msec. [92703484992 cycles]
```

Здесь наблюдается лишь незначительное увеличение крутизны линейной зависимости, что можно отнести к некоторым накладным расходам на поддержание достаточно большого числа записей о процессах в таблицах менеджера процессов, но величина этого эффекта также весьма малосущественна.

В итоге, в отношении «легковесности» потоков можно сказать следующее:

- При необходимости **динамического** создания параллельных ветвей в ходе выполнения программы (а это достаточно классический случай, например в разнообразных сетевых серверах, создающих ветвь обслуживания для каждого нового клиента) производительность приложения, функционирующего на основе потоков, может быть значительно выше (до нескольких порядков), а время реакции соответственно ниже.
- При **статическом** выполнении (фиксированном количестве параллельных ветвей в приложении) эффективность приложений, построенных на параллельных потоках или параллельных процессах, практически не отличается. Более того, эффективности таких приложений не отличаются и от классической последовательной организации приложения, работающего в одном потоке.
- Существует дополнительный фактор, обеспечивающий «легковесность» потоков в противовес процессам, – это легкость и эффективность их взаимодействия в едином адресном пространстве. В случае процессов для обеспечения таких взаимодействий возникает необходимость привлечения «тяжеловесных» механизмов IPC разнообразной природы (именованные и неименованные каналы, разделяемая память, обмен UNIX-сообщениями и другие). При рассмотрении обмена сообщениями QNX мы еще раз убедимся в том, что обмены и взаимодействия между процессами могут требовать весьма существенных процессорных ресурсов, а при обменах с интенсивным трафиком могут стать доминирующей компонентой, определяющей пределы реальной производительности системы.

Пример: синхронное выполнение кода

Выше приводилось достаточно много подобных примеров, но это были примеры, так сказать, «локальные», фрагментарные, иллюстрирующие использование какой-то одной возможности применительно к потокам. Сейчас мы приведем пример, реализующий часто возникающую на практике возможность. Некоторые программные действия (функции) мы хотели бы запускать периодически с фиксированным временным интервалом T, что весьма напоминает действия в аппарат-

ной реализации, которые должны быть выполнены по каждому импульсу «синхронизирующей последовательности».

Простейшая реализация могла бы выглядеть так:

```
...
while( true ) {
   delay( T );
   func();
};
```

Но это очень «слабое» решение:

- Задержка, обеспечиваемая функцией пассивной задержки `delay()`, согласно требованиям POSIX **не может быть меньше** указанного параметра T, но... может быть **сколь угодно больше!** (В [4] мы писали, что при T = 1 реальная величина задержки будет составлять не 1 мсек., как можно было бы ожидать, а с большой степенью вероятности 3 мсек., и там же мы подробно показывали, как это происходит.)
- Если в системе одновременно с этим приложением работает процесс (поток) более высокого приоритета, то наше приложение может вообще никогда «не проснуться», по крайней мере, пока это не «соизволит» санкционировать параллельное приложение.
- Здесь мы обеспечиваем только одну синхронизированную последовательность вызовов функции `func()`. А если бы нам потребовалось несколько (много) синхросерий, в каждой из которых выполняется своя функция, а периоды серий не кратны друг другу?
- Наконец, время выполнения целевой функции `func()` включается в период одного «кругового пробега» цикла, то есть период T отсчитывается от **конца** предыдущего выполнения функции до **начала** текущего, а это не совсем то, что мы подразумевали при использовании термина «синхронное».
- Более того, если время выполнения функции `func()` достаточно флуктуирует от одного вызова до другого (например, из-за изменений данных, с которыми работает функция), то периоды вызовов начинают «гулять», а дисперсия периода результирующей последовательности вызовов `func()` становится просто непомерно большой.

Ниже показано решение, свободное от многих из этих недостатков (*файл t3.cc*). Приложение представляет собой тестовую программу, осуществляющую 3 цепочки выполнения различных целевых функций (`mon1`, `mon2`, `mon3`) с разными периодами для каждой цепочки (массив `period[]`):

Синхронизация выполнения участка кода

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```
#include <inttypes.h>
#include <errno.h>
#include <iostream.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>
#include <sys/netmgr.h>
#include <pthread.h>
#include <signal.h>
#include <algorithm>

static void out( char s ) {
   int policy;
   sched_param param;
   pthread_getschedparam( pthread_self(), &policy, &param );
   cout << s << param.sched_curpriority << flush;
};

// целевые функции каждой из последовательностей только
// выводят свой символ-идентификатор и следующий за ним
// приоритет, на котором выполняется целевая функция
static void mon1( void ) { out( '.' ); };
static void mon2( void ) { out( '*' ); };
static void mon3( void ) { out( '+' ); };

// это всего лишь перерасчет временных интервалов,
// измеренных в тактах процессора (в наносекундах)
inline uint64_t cycles2nsec( uint64_t c ) {
   const static uint64_t cps =
      // частота процессора
      SYSPAGE_ENTRY( qtime )->cycles_per_sec;
   return ( c * 1000000000 ) / cps;
};

// структура, необходимая только для накопления статистики параметров
// ряда временных отметок: среднего, среднеквадратичного отклонения,
// минимального и максимального значений
struct timestat {
private:
   uint64_t               prev;
public:
   uint64_t               num;
   double                 mean, disp, tmin, tmax;
   timestat( void ) {
      mean = disp = tmin = tmax = 0.0; num = 0;
   };
// новая временная отметка в ряду:
    void operator ++( void ) {
      uint64_t next = ClockCycles(), delta;
      if( num != 0 ) {
         double delta = cycles2nsec( next - prev );
         if( num == 1 ) tmin = tmax = delta;
```

```
            else tmin = min( tmin, delta ),
                 tmax = max( tmax, delta );
            mean += delta;
            disp += delta * delta;
         };
         prev = next;
         num++;
      };
   // подвести итог ряда:
      void operator !( void ) {
         mean /= ( num - 1 );
         disp = sqrt( disp / ( num - 1 ) - mean * mean );
      };
   };

   // предварительное описание функции потока объекта
   void* syncthread( void* );

   class thrblock {
   private:
      static int            code;
      bool                  ok, st;
   public:
      pthread_t             tid;
      struct sigevent event;
      timer_t               timer;
      int                   chid;
      void* ( *func )( void* );
      sched_param param;
      // структура только для статистики:
      timestat              sync;
      // конструктор класса – он не только инициализирует структуру данных
      // создаваемого объекта, но и запускает отдельный поток для его исполнения
      thrblock(
         // параметры конструктора:
         // - целевая функция последовательности
         void ( *dofunc )( void ),
         // - период ее синхронизации
         unsigned long millisec,
         // - приоритет возбуждения синхросерии
         unsigned short priority,
         // - копить ли статистику временных интервалов?
         bool statist = false
      ) {
         // создание канала для получения уведомлений от таймера
         if( !( ok = ( ( chid = ChannelCreate( 0 ) ) >= 0 ) ) ) return;
         // создать соединение по каналу, которое будет использовать таймер
         event.sigev_coid = ConnectAttach( ND_LOCAL_NODE, 0, chid,
                                           NTO_SIDE_CHANNEL, 0 );
         if( !( ok = ( event.sigev_coid >= 0 ) ) ) return;
```

```
        // занести целевую функцию, заодно выполнив
        // трюк преобразования над ее типом
        func = (void* (*)( void* ) ) dofunc;
        int policy;
        // запомнить приоритет вызывающей программы
        // под этим приоритетом и вызывать целевую функцию
        pthread_getschedparam( pthread_self (), &policy, &param );
        st = statist;
        event.sigev_code = code++;
        event.sigev_notify = SIGEV_PULSE;
        // а вот это приоритет, с которым нужно будет пробуждаться от таймера!
        event.sigev_priority = priority;
        // создание таймера
        if( !( ok = ( timer_create( CLOCK_REALTIME,
                    &event, &timer ) = = 0 ) ) ) return;
        // запуск отдельного потока, который по сигналу
        // таймера будет выполнять целевую функцию
        if( !( ok = ( pthread_create( &tid, NULL, &syncthread,
                    (void*)this ) = = EOK ) ) ) return;
        // ... и только после этого можно установить период срабатывания
        // таймера, после чего он фактически и запускается
        struct itimerspec itime;
        nsec2timespec( &itime.it_value, millisec * 1000000ull );
        itime.it_interval =  itime.it_value;
        if( !( ok = ( timer_settime( timer, 0,
                    &itime, NULL ) = = 0 ) ) ) return;
    };
    // признак того, что объект создан успешно и его поток запущен:
    bool OK( void ) { return ok; };
    bool statistic( void ) { return st; };
};

int thrblock::code = _PULSE_CODE_MINAVAIL;

// функция потока объекта
void* syncthread( void *block ) {
    thrblock *p = (thrblock*)block;
    struct _pulse buf;
    pthread_attr_t attr;
    while( true ) {
        // ожидание пульса от периодического таймера объекта
        MsgReceivePulse( p->chid, &buf,
                         sizeof( struct _pulse ), NULL );
        pthread_attr_init( &attr );
        pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_DETACHED );
        // восстановить приоритет целевой функции до уровня того,
        // кто ее устанавливал, вызывая конструктор...
        pthread_attr_setinheritsched( &attr, PTHREAD_EXPLICIT_SCHED );
        pthread_attr_setschedparam( &attr, &p->param );
        // запуск целевой функции в отдельном "отсоединенном" потоке
        pthread_create( NULL, &attr, p->func, NULL );
```

```
      if( p->statistic() ) ++p->sync;
   };
};

// "пустой" обработчик сигнала SIGINT (реакция на ^C)
inline static void empty( int signo ) { };

int main( int argc, char **argv ) {
   // с этой точки стандартная реакция на ^C отменяется...
   signal( SIGINT, empty );
   // массив целевых функций
   void( *funcs[] )( void ) = { &mon1, &mon2, &mon3 };
   // периоды их синхросерий запуска
   int period[] = { 317, 171, 77 };
   // приоритеты, на которых отрабатывается реакция
   // синхросерий на каждый из таймеров синхросерий
   int proirity[] = { 15, 5, 25 };
   int num = sizeof( funcs ) / sizeof( *funcs );
   // запуск 3-х синхронизированных последовательностей
   // выполнения (созданием объектов):
   thrblock** tb = new (thrblock*)[ num ];
   for( int i = 0; i < num; i++ ) {
      tb[ i ] = new thrblock( funcs[ i ], period[ i ],
                              proirity[ i ], true );
      if( !tb[ i ]->OK() )
         perror( "synchro thread create" ),
         exit( EXIT_FAILURE );
   };
   // ... а теперь ожидаем ^C:
   pause();
   // подсчет статистики и завершение программы
   cout << endl << "Monitoring finalisation!" << endl;
   // вывод временных интервалов будем делать в миллисекундах:
   const double n2m = 1000000.;
   for( int i = 0; i < num; i++ ) {
      timestat *p = &tb[ i ]->sync;
      !( *p );  // подсчет статистики по объекту
      cout << i << '\t' << p->num << "\t=> "
           << p->mean / n2m << " [" << p->tmin / n2m
           << "..." << p->tmax / n2m
           << "]\t~" << p->disp / n2m << " ("
           << p->disp / p->mean * 100. << "%)" << endl;
   };
   return EXIT_SUCCESS;
};
```

Вся функциональность программы сосредоточена в одном классе – `thrblock`, который может в неизменном виде использоваться для разных приложений. Необычной особенностью объекта этого класса является то, что он выполнен в технике «активных объектов», навеянной поверхностным знакомством с языками программирования шко-

лы Н. Вирта – ActiveOberon и Zonnon. В ней говорится, что конструктор такого объекта не только создает объект данных, но и запускает (как вариант) отдельный поток выполнения для каждого создаваемого объекта. В нашем случае задача потоковой функции состоит в вызове целевой функции, адрес которой был передан конструктору объекта в качестве одного из параметров.[1]

Ниже представлены отличия нашей реализации от простого цикла с задержкой, обсуждавшейся выше (помимо исправлений очевидных недостатков):

- Для каждого синхронизирующего таймера установлен свой приоритет «пробуждения», и он может быть достаточно высоким, для того чтобы предотвратить вытеснение этого синхронизирующего потока.
- После «пробуждения» по таймеру запускается целевая функция, но выполняется это отдельным потоком, причем потоком «отсоединенным». Другими словами, процесс выполнения целевой функции никак не влияет на общую схему синхронизации.
- Перед запуском целевой функции выполняющему ее потоку восстанавливается приоритет породившего потока (но не потока обслуживания таймера!), ведь нам не нужно, чтобы целевая функция, тем более, возможно и не очень значимая, как в нашем примере, могла влиять вытеснением на процессы синхронизации.

Запустим наше тестовое приложение:

```
# t3
+10+10*10+10+10.10*10+10+10*10+10+10.10*10+10+10+10*10+10.10+10*10+10+10*10+
10.10+10*10+10+10*10+10.10+10+10*10+10+10*10.10+10+10*10+10+10.10*10+10+10+1
0*10+10.10+10*10+10+10*10+10+10.10*10+10+10*10+10+10.10+10*10+10+10*10+10.10
+10*10+10+10*10+10.10+10+10*10+10+10*10^C
Monitoring finalisation!
0  32   => 316.919 [316.867...317.895] ~0.178511 (0.056327%)
1  59   => 170.955 [168.583...173.296] ~0.92472  (0.540914%)
2  132  => 76.9796 [76.942...77.9524]  ~0.085977 (0.111688%)
```

Первое, что мы должны отметить, – это очень приличную точность выдержки периода синхронизации (последняя колонка вывода). Для того чтобы убедиться в том, что целевая функция при этом выполняется под приоритетом породившего ее потока, закомментируем строки, выделенные жирным шрифтом в коде программы:

---

[1] Здесь применена только простейшая форма «активного объекта»; гораздо сложнее, к примеру, определить деструктор такого объекта. Но именно в своем простейшем виде это многообещающая техника активных объектов.

```
# t3
+25+25*5+25+25.15*5+25+25*5+25+25.15*5+25+25+25*5+25.15+25*5+25+25*5+25.15+2
5*5+25+25*5*5+25.15+25+25*5+25+25*5.15+25+25*5*5+25+25.15*5+25+25+25*5+25.15+2
5*5+25+25*5+25+25.15*5+25+25*5+25+25^C
Monitoring finalisation!
0  32  => 316.919 [316.797...317.915] ~0.185331 (0.0584792%)
1  60  => 170.955 [168.964...173.925] ~0.47915 (0.280279%)
2  134 => 76.9796 [76.8895...77.9694] ~0.0937379 (0.12177%)
```

В этом варианте (и диагностический вывод это подтверждает) мы искусственно ликвидировали наследование приоритета по цепочке порождения: сработавший таймер – функция потока – целевая функция объекта. Это не совсем соответствует цели, намеченной в начале этого раздела, но все же этот вариант иллюстрирует, что именно наш предыдущий вариант удовлетворял всем поставленным целям.

3

Сигналы

Сигналы инициируются некоторыми событиями в системе и посылаются процессу для его уведомления о том, что произошло нечто неординарное, требующее определенной реакции. Порождающее сигнал событие может быть действием пользователя или может быть вызвано другим процессом или ядром операционной системы. Сигналы являются одним из самых старых и традиционных механизмов UNIX.[1]

Уже из этого краткого описания можно заключить, что:

- действия, вызываемые для обработки сигнала, являются принципиально **асинхронными**;
- сигналы могут быть использованы как простейшее, но мощное средство межпроцессного взаимодействия.

Все сигналы определяются целочисленными константами, но для программиста это в принципе не так важно, поскольку каждому сигналу приписано символическое наименование вида `SIG*`. Все относящиеся к сигналам определения находятся в заголовочном файле `<signal.h>`, который должен быть включен в любой код, работающий с сигналами. В последующих примерах мы не будем показывать эту включающую директиву, чтобы не загромождать текст.

Посланный сигнал обрабатывается получившим его потоком или процессом (как уже обсуждалось ранее, собственно процесс ничего не может обрабатывать: это пассивная субстанция; понятно, что речь в таком контексте идет об обработке сигнала главным потоком процесса). Поток может отреагировать на полученный сигнал следующими способами (иногда действие, установленное для конкретного сигнала, называют **диспозицией** сигнала):

- **Стандартное действие** – выполнение действия, предписанного для обработки этого сигнала по умолчанию. Для многих сигналов действием по умолчанию является завершение, но это необязательно;

---

[1] Именно поэтому, наверное, стандарту POSIX так сложно органично согласовать их с нововведениями, например с парадигмой потоков.

есть сигналы, которые по умолчанию игнорируются. Для большей части сигналов, чьим действием по умолчанию является принудительное завершение процесса, предписано действие создания дампа памяти при завершении (core dump), но для некоторых, например SYSINT (завершение по [Ctrl+C]), определено такой дамп при завершении не создавать.

- **Игнорирование сигнала** – сигнал не оказывает никакого воздействия на ход выполнения потока получателя.
- **Вызов обработчика** – по поступлению сигнала вызывается функция реакции, определенная пользователем. Если для сигнала устанавливается функция-обработчик, то говорят, что сигнал перехватывается (относительно стандартного действия).

Для различных сигналов программа может установить различные механизмы обработки. Более того, в ходе выполнения программа может динамически переопределять реакции, установленные для того или иного сигнала.

Для большинства сигналов их воздействие можно перехватить из программного кода или игнорировать, но не для всех. Например, сигналы SIGKILL и SIGSTOP не могут быть ни перехвачены, ни проигнорированы. Другой пример – описываемые далее специальные сигналы QNX, начиная с SIGSELECT и далее.

В QNX определены 64 сигнала в трех диапазонах:

- 1…40 – 40 POSIX-сигналов общего назначения;
- 41…56 – 16 POSIX-сигналов реального времени, введенных в стандарт позже (от SIGRTMIN до SIGRTMAX);
- 57…64 – 8 сигналов, используемых в QNX Neutrino для специальных целей.

Начнем со специальных сигналов. Эти сигналы не могут быть проигнорированы или перехвачены: попытка вызвать signal() или sigaction() (или вызов ядра SignalAction() native API) завершится для них с ошибкой EINVAL. Более того, эти сигналы всегда блокированы в пользовательском приложении, и для них установлено разрешение очереди обслуживания (очереди сигналов будут подробно рассмотрены далее). Попытка разблокировать эти сигналы, используя sigprocmask() (SignalProcmask() – native API), будет проигнорирована. Эти сигнальные механизмы используются, в частности, графической системой Photon для ожидания событий GUI и функцией select() для ожидания ввода/вывода на множестве дескрипторов (один из фундаментальных механизмов UNIX). Вот определения некоторых из этих сигналов:

```
#define SIGSELECT (SIGRTMAX + 1)
#define SIGPHOTON (SIGRTMAX + 2)
```

В силу своей недоступности эта группа сигналов представляет небольшой интерес для разработчика приложений.

Далее перейдем к POSIX-сигналам общего назначения (1...40), из которых назовем только самые употребляемые[1] (пока мы описываем все сигналы в классической UNIX-нотации, не углубляясь в нюансы, например такие, как особенности реакции на них в многопоточном окружении):

- SIGABRT (+) [6] – аварийное завершение процесса (process abort signal). Посылается процессу при вызове им функции abort(). В результате обработки сигнала SIGABRT произойдет то, что спецификация XSI описывает как **аварийное завершение**.
- SIGALRM [14] – наступление тайм-аута таймера сигналов (real-time alarm clock). Посылается при срабатывании ранее установленного пользовательского таймера (таймер устанавливается заданием первого параметра setitimer(), равным ITIMER_REAL), например при помощи системного вызова alarm().
- SIGBUS (+) [10] – сигнал ошибки на шине (bus error). Этот сигнал посылается при возникновении некоторых аппаратных ошибок (зависит от платформы); обычно он генерируется при попытке обращения к допустимому виртуальному адресу, для которого нет физической страницы.
- SIGCHLD [18] – сигнал завершения или остановка дочернего процесса (child process terminated or stopped). По умолчанию родительский процесс игнорирует этот сигнал, поэтому если нужно получать уведомление о завершении порожденного процесса, этот сигнал нужно перехватывать. Для этого сигнала определен синоним:

```
#define  SIGCLD  SIGCHLD
```

- SIGCONT [25] – продолжение работы остановленного процесса (continue executing if stopped). Это сигнал управления выполнением процесса, который возобновляет его выполнение, если ранее он был остановлен сигналом SIGSTOP; если процесс не остановлен, он игнорирует этот сигнал.
- SIGDEADLK [7] – сигнал, говорящий о возникновении «мертвой» блокировки потока на мьютексе (mutex deadlock occurred). Эта ситуация будет подробно рассмотрена далее, когда речь пойдет о примитивах синхронизации (глава 4).
- SIGEMT [7] – EMT-инструкция (emulator trap). Как видно из сравнения с предыдущим сигналом, у них один код, то есть в QNX SIGDEADLK и SIGEMT – это один сигнал.

---

[1] Наличие круглых скобок после имени сигнала – признак того, что для этих сигналов реакция по умолчанию – принудительное завершение процесса; наличие (+) означает, что для этих сигналов предусмотрено создание дампа завершения, а наличие (–) – для этих сигналов дамп не создается. В квадратных скобках после имени сигнала указано численное значение, соответствующее данному сигналу, как оно определено в QNX.

- SIGFPE (+) [8] – недопустимая арифметическая операция с плавающей точкой (floating point exception).
- SIGHUP [1] – сигнал освобождения линии, разрыв связи с управляющим терминалом (hangup signal). Посылается всем процессам, подключенным к управляющему терминалу при отключении терминала (обычно управляющий терминал группы процесса является терминалом пользователя, но это не всегда так). Этот сигнал также посылается всем членам сеанса, если завершает работу лидер сеанса (обычно процесс командного интерпретатора), связанного с управляющим терминалом (это гарантирует, что, если не были приняты специальные меры, при выходе пользователя из системы будут завершены все запущенные им фоновые процессы).
- SIGILL [4] – попытка выполнения недопустимой инструкции процессора (illegal instruction); обработка сигнала не сбрасывается, когда он перехватывается.
- SIGINT (-) [2] – принудительное прерывание процесса (interrupt); это тот сигнал, который генерируется при нажатии [Ctrl+C]. Это обычный способ завершения выполняющегося процесса.
- SIGKILL (+) [9] – уничтожение процесса (kill); этот сигнал может активизироваться командой `kill -9 <PID>`. Это сигнал безусловного завершения, посылаемый процессу другим процессом или системой (при завершении работы системы). Этот сигнал не может быть проигнорирован или перехвачен.
- SIGPIPE [13] – попытка выполнить недопустимую запись в канал или сокет, для которых принимающий процесс уже завершил работу (write on pipe or socket when recipient is terminated, write on pipe with no reader).
- SIGPOLL [22] – сигнал уведомления об одном из опрашиваемых событий (pollable event). Этот сигнал генерируется системой, когда некоторый открытый дескриптор файла готов для ввода или вывода. Этот сигнал имеет синоним (наименование SIGPOLL более известно из UNIX System V):

```
#define SIGIO       SIGPOLL
```

- SIGPROF [29] – сигнал профилирующего таймера (profiling timer expired). Как и сигнал SIGALRM, этот сигнал возбуждается по истечении времени таймера (но это другой таймер), который используется для измерения времени выполнения процесса в пользовательском и системном режимах (таймер устанавливается заданием первого параметра setitimer(), равным ITIMER_PROF).
- SIGQUIT (-) [3] – выход из процесса (quit).
- SIGSEGV (+) [11] – обращение к некорректному адресу памяти, ошибка защиты памяти, нарушение границ сегмента памяти (invalid memory reference, segmentation violation).

- SIGSTOP [23] – временная остановка процесса (sendable stop signal not from tty). Если пользователь вводит с терминала [Ctrl+Z], то активному процессу посылается этот сигнал и процесс приостанавливается. Позже процесс может быть возобновлен с точки остановки при получении сигнала SIGCONT. Этот сигнал нельзя проигнорировать или перехватить.
- SIGSYS (+) [12] – некорректный системный вызов (invalid system call, bad argument to system call).
- SIGTERM [15] – программный сигнал завершения (software termination signal from kill). Программист может использовать этот сигнал для того, чтобы дать процессу время для «наведения порядка», прежде чем посылать ему сигнал SIGKILL. Именно этот сигнал посылается по умолчанию командой kill без параметра, указывающего сигнал.
- SIGTRAP [5] – сигнал трассировочного прерывания (trace trap). Это особый сигнал, который в сочетании с системным вызовом ptrace() используется отладчиками: sdb, adb, gdb. По умолчанию сигнал приводит к аварийному завершению. Обработка сигнала не сбрасывается, когда он перехватывается.
- SIGTSTP [24] – терминальный сигнал остановки (terminal stop signal). Генерируется при нажатии специальной комбинации остановки [Ctrl+Z]. Аналогичен сигналу SIGSTOP, но, в отличие от последнего, может быть перехвачен или проигнорирован.
- SIGTTIN [26] – остановка фонового процесса, если он пытается прочитать данные со своего управляющего терминала (background process attempting read).
- SIGTTOU [27] – остановка фонового процесса, если он пытается писать данные на свой управляющий терминал (background process attempting write).
- SIGURG [21] – сигнал о поступлении в буфер сокета срочных (приоритетных) данных (higs bandwidth data is available at a socket, urgent condition on I/O channel) уведомляет процесс, что по открытому им сетевому соединению получены внеочередные данные.
- SIGUSR1 [16], SIGUSR2 [17] – зарезервированные сигналы пользователя. Для этих сигналов предопределенной реакцией в QNX является завершение процесса (хотя естественнее ожидать, и так это предлагает POSIX, реакцию «игнорировать сигнал»), и реакцию на них должен определять пользователь. Так же как и сигнал SIGTERM, эти сигналы никогда не посылаются системой.
- SIGVTALRM [28] – сигнал виртуального таймера (virtual timer expired). Подобно SIGPROF и SIGALRM, этот сигнал возбуждается по истечении времени таймера (это третий из доступных таймеров), который измеряет время процессора только в пользовательском режиме

(таймер устанавливается заданием первого параметра `setitimer()`, равным `ITIMER_VIRTUAL`).

- `SIGXCPU [30]` – сигнал о превышении лимита процессорного времени (CPU time limit exceeded). Посылается процессу при исчерпании им ранее установленного лимита процессорного времени. Действие по умолчанию – аварийное завершение.
- `SIGXFSZ [31]` – сигнал о превышении предела, установленного на размер файла (file size limit exceeded). Действие по умолчанию – аварийное завершение.
- `SIGWINCH [20]` – сигнал, который генерируется (в консольном режиме `pterm` и `xterm` эмулируют его вручную при изменении их размеров) при изменении размера окна (window size change) для запущенного в окне приложения (mc, mqc...), чтобы оно перерисовало свой экран вывода.

Примечание

В QNX определено еще два специфических сигнала, которые вряд ли должны представлять для нас интерес:

- `SIGIOT [6]` – IOT-инструкция; никогда не генерируется для платформы x86.
- `SIGPWR [19]` – сигнал power-fail restart, о котором в технической документации QNX ничего не говорится, но в преамбуле, описывающей нововведения версии 6.2.1, сказано: «corrected SIGPWR to SIGTERM», то есть этот сигнал, очевидно, – рудимент прежних версий системы.

Примечание

POSIX допускает, что не все сигналы могут быть реализованы. Более того, допускается ситуация, когда некоторое символическое имя сигнала определено, но сам сигнал отсутствует в системе (изменения такого рода вполне могут наблюдаться при переходе от одной версии QNX к другой). Для диагностики реального наличия сигнала можно воспользоваться рекомендацией, приведенной в информативной части стандарта POSIX 1003.1: наличие поддержки сигнала сообщает вызов функции `sigaction()` с аргументами `act` и `oact`, установленными в `NULL`. Приведем простейший тест (*файл s1.cc*), реализующий рекомендацию POSIX в QNX 6.2.1:

```
#include <stdlib.h>
#include <stream.h>
#include <errno.h>
#include <signal.h>
int main( int argc, char *argv[] ) {
   cout << "SIGNO";
   for( int i = _SIGMIN; i <= _SIGMAX; i++ ) {
      if( i % 8 == 1 ) cout << endl << i << ':';
      int res = sigaction( i, NULL, NULL );
      cout << '\t' << ( ( res != 0 && errno == EINVAL ) ? '-' : '+' );
   };
   cout << endl;
   return EXIT_SUCCESS;
};
```

И результат его выполнения:

```
SIGNO
1:    +    +    +    +    +    +    +    +
9:    +    +    +    +    +    +    +    +
17:   +    +    +    +    +    +    +    +
25:   +    +    +    +    +    +    +    +
33:   +    +    +    +    +    +    +    +
41:   +    +    +    +     +   +    +    +
49:   +    +    +    +    +    +    +    +
57:   -    -    -    -    -    -    -    -
```

Система «считает» все сигналы 1…56 реализуемыми, а последние 8 специфических сигналов QNX, как упоминалось выше, не допускают применения к ним `sigaction()`. Здесь с учетом цитировавшейся выше раскладкой сигналов QNX есть небольшая загадка: максимальным номером POSIX-сигнала, определенного в `<signal.h>`, является 31 (`SIGXFSZ` – 31); там же в комментарии есть фраза: «допустимый диапазон пользовательских сигналов – от 1 до 56, используемых ядром – от 57 до 64». Непонятно, в каком качестве используются сигналы 32 – 40, непосредственно предшествующие сигналам реального времени (41 – 56) и диагностируемые `sigaction()` как действительные (valid)? Позже мы увидим, что они обслуживаются системой наравне с документированными сигналами.

Традиционная обработка сигнала

В этой части изложения мы рассмотрим традиционные модели перехвата сигналов и установки для них собственных обработчиков (в том числе и игнорирование или восстановление стандартной обработки по умолчанию). Термин «традиционный» здесь означает, что мы бегло рассмотрим обработку сигналов применительно к процессам и стандартным сигналам UNIX (не сигналам реального времени), то есть в том изложении, как она традиционно рассматривается в литературе по UNIX (и здесь сигнал воспринимается, конечно же, единственным потоком приложения, а не процессом, но в этом случае различие не принципиально). Позже мы рассмотрим модель обработки сигналов реального времени и расширим ее на многопоточные приложения.

«Старая» модель обработки сигнала

В ранних версиях UNIX была принята единственная модель обработки сигналов, основанная на функции `signal()`, которая подразумевает семантику так называемых «ненадежных сигналов», принятую в этих ОС. Позже эта модель была подвержена радикальной критике, вскрывшей ее «ненадежность». Данная модель сохранена для совместимости с ранее разработанным программным обеспечением. Она обладает существенными недостатками, основными из которых являются:

- процесс не может заблокировать сигнал, то есть отложить получение сигнала на период выполнения критических участков кода;

- каждый раз при получении сигнала его диспозиция устанавливается на действие по умолчанию, и при необходимости продолжить обработку поступающих сигналов требуется повторно восстанавливать требуемый обработчик.

Вот пример (*файл s2.cc*) использования этой модели в коде, который уже стал иллюстративным образцом и кочует из одного источника в другой:

Ненадежная модель реакции на сигнал

```
#include <iostream.h>
#include <signal.h>
#include <unistd.h>

// обработчик сигнала SIGINT:
static void handler(int signo) {
   // восстановить обработчик:
   signal( SIGINT, handler );
   cout << "Получен сигнал SYSINT" << endl;
};

int main() {
   // устанавливаются диспозиции сигналов:
   signal( SIGINT, handler );
   signal( SIGSEGV, SIG_DFL );
   signal( SIGTERM, SIG_IGN );
   while( true ) pause();
};
```

Примечание

Макросы SIG_DFL и SIG_IGN определяются так:

```
#define SIG_ERR          ( ( void(*)( _SIG_ARGS ) ) -1 )
#define SIG_DFL          ( ( void(*)( _SIG_ARGS ) )  0 )
#define SIG_IGN          ( ( void(*)( _SIG_ARGS ) )  1 )
#define SIG_HOLD         ( ( void(*)( _SIG_ARGS ) )  2 )
```

где _SIG_ARGS – это фактически тип int. SIG_DFL и SIG_IGN устанавливают диспозиции сигнала «по умолчанию» и «игнорировать» соответственно, а о SIG_HOLD мы будем отдельно говорить позже.

Выполнение этой программы вам будет не так просто прекратить: на комбинацию завершения [Ctrl+C] она отвечает сообщением о получении сигнала... и все. Воспользуемся для этого посылкой программе опять же сигнала, но из другого процесса (другого экземпляра командного интерпретатора). Смотрим PID запущенного процесса:

```
# pidin
...
2207786   1 ./s2               10r STOPPED
...
```

И посылаем процессу сигнал завершения:

```
# kill -9 2207786 или kill -SIGKILL 2207786
```

Таким же образом, как показано командой `kill`, мы будем посылать сигналы процессам «извне» и в описываемых далее тестах, не останавливаясь подробно, как это происходит, в том числе и для сигналов реального времени (41...56).

Предыдущий пример можно переписать (*файл s4.cc*) для обеспечения часто требуемой на практике защиты от немедленного прерывания выполнения по [Ctrl+C], чтобы дать программе возможность выполнить все требуемые операции по завершению (сбросить буферы данных на диск, закрыть файлы, сокеты и другие используемые объекты):

```
#include <stdlib.h>
#include <iostream.h>
#include <signal.h>
#include <unistd.h>

static void handler( int signo ) {
   cout << "Saving data ... wait.\r" << flush;
   sleep( 2 ); // ... здесь выполняются все завершающие действия!
   cout << "                                  " << flush;
   exit( EXIT_SUCCESS );
};

int main() {
   signal( SIGINT, handler );
   signal( SIGSEGV, SIG_DFL );
   signal( SIGTERM, SIG_IGN );
   while( true ) pause();
};
```

Оператор ожидания `pause()` при поступлении сигналов завершается с возвратом −1, а переменная `errno` устанавливается в `EINTR`. Этот оператор дает нам еще один способ (*файл s3.cc*) неявного (без явной установки обработчиков) использования сигналов:

```
#include <stream.h>
#include <stdlib.h>
#include <unistd.h>

int main( void ) {
    alarm( 5 );
    cout << "Waiting to die in 5 seconds ..." << endl;
    pause();
    return EXIT_SUCCESS;
};
```

Описываемая модель обработки сигналов обладает рядом недостатков, считается устаревшей и, более того, как было показано, не обеспечивает надежную обработку сигналов. Тем не менее эту модель достаточно

широко применяют в простых случаях, например при необходимости установить тайм-аут для некоторой операции. Вот как, к примеру, устанавливается тайм-аут ожидания установления соединения в TCP/IP-клиенте [9]:

```
void alarm_handler( int sig ) { return; };

int main() {
...
   signal( SIGALRM, alarm_handler );
   alarm( 5 );
   int rc = connect( ... );
   alarm( 0 );
   if( rc < 0 && errno == EINTR )
      cout << "Истек тайм-аут" << endl, exit( EXIT_FAILURE );
...
};
```

Здесь уместно напомнить немаловажное обстоятельство, связанное с сигналами, которое обделяется вниманием во многих руководствах по программированию: большинство блокирующих вызовов API (`connect()`, `delay()`, `wait()`, `waitid()` и многие другие) будут разблокированы при получении блокированным потоком **любого** сигнала. Такие вызовы API, как `pause()` и `sigwait()`, вообще предназначены только для выполнения пассивной блокировки до момента поступления сигнала. Многие их них возвращают значение или устанавливают в качестве кода системной ошибки `errno` значение `EINTR`, специально отведенное для отражения такого результата завершения, как прерывание поступившим извне сигналом. Мы неоднократно будем использовать это обстоятельство в тексте примеров программного кода, например:

```
if( delay( 100 ) != 0 ) . . .
```

В данном случае учитываем, что функция `delay()` возвращает нереализованный остаток «заказанного» ей ожидания, который может быть ненулевым только при прерывании этого ожидания сигналом извне (нулевое значение соответствует «естественному» истечению времени задержки).

Модель надежных сигналов

В более поздней («новой») модели обработки сигналов (называемой еще моделью надежных сигналов) используются не единичные сигналы, а наборы сигналов – тип `sigset_t`.

Примечание

POSIX требует, чтобы в реализации тип `sigset_t` определялся таким образом, чтобы он мог «вместить» все определенные в системе сигналы; для QNX это число равно 64. Определение типа `sigset_t` в QNX, как и большинство фундаментальных для системы определений, находится в заголовочном файле `<target_nto.h>`:

```
struct { long bits[ 2 ]; }
```

Понятно, что в этом случае тип sigset_t – это битовая маска, но на практике знание представления этого типа не имеет никакой ценности для программиста, так как все операции над ним выполняются набором специальных операций, так что совершенно обоснованно этот тип можно считать абстрактным.

Для формирования сигнальных наборов определяется набор специальных операций:

- sigemptyset(sigset_t *set) – инициализирует набор set, исключая из него все сигналы;
- sigfillset(sigset_t *set) – инициализирует набор set, включая в него все сигналы;
- sigaddset(sigset_t *set, int signo) – добавляет в инициализированный набор set единичный сигнал signo;
- sigdelset(sigset_t *set, int signo) – удаляет из инициализированного набора set единичный сигнал signo.

В качестве signo в функциях добавления и удаления единичных сигналов используется символическая константа, соответствующая сигналу (такая как SIGINT), либо численное значение сигнала, но в этом случае код становится зависимым от системы. Легко увидеть, что, пользуясь совокупностью этих 4-х операций, можно сформировать любой произвольный набор сигналов. Например:

```
sigset_t sig;
sigemptyset( &sig );
sigaddset( &sig, SIGPOLL );
sigaddset( &sig, SIGALRM );
```

Этот фрагмент кода формирует сигнальный набор, состоящий из двух сигналов: SIGPOLL и SIGALRM.

Диспозиция обработки каждого сигнала в этой модели устанавливается функцией:

```
int sigaction( int signo, const struct sigaction *act, struct sigaction *oact );
```

где signo – номер (имя) сигнала, для которого устанавливается диспозиция;

act – определение нового обработчика сигнала;

oact – структура (если указано не NULL), где будет сохранено описание ранее установленного обработчика (например, для последующего восстановления реакции).

Структура описания обработчика sigaction определена так (мы исключили из определения часть структуры, предназначенную для компилятора Watcom, QNX 4.X):

```
struct sigaction {
#define sa_handler   un._sa_handler
#define sa_sigaction un._sa_sigaction
   union {
      void (*_sa_handler)( _SIG_ARGS );
      void (*_sa_sigaction)( int, siginfo_t*, void* );
   } un;
   int         sa_flags;
   sigset_t    sa_mask;
};
```

Примечание

Это определение по форме, но не по содержанию отличается от описания, показанного в POSIX и используемого во многих традиционных UNIX [5] (обратите внимание на изменение порядка следования полей маски и флагов; это может стать преградой для прямой инициализации структуры в стиле C++ из соображений переносимости):

```
struct sigaction {
   /* указатель на функцию обработчика сигнала  */
   void (*sa_handler) ( int );
   /* сигналы, блокирующиеся во время обработки */
   sigset_t sa_mask;
   /* флаги, влияющие на поведение сигнала      */
   int sa_flags
   /* указатель на функцию обработчика сигнала */
   void (*sa_siaction) ( int, siginfo_t*, void* );
};
```

Определения `#define` в первых строках описания – это обычная в QNX практика переопределения имен для компиляторов, «не понимающих» анонимных (неименованных) объединений (`union`). Легко видеть, что даже размеры структур в этих двух определениях (QNX и POSIX) будут отличаться, что подсказывает необходимость соблюдения здесь особой тщательности при использовании.

Первое поле `sa_handler` определяет обработчик, устанавливаемый для сигнала в традиционной модели. Это может быть:

- `SIG_DFL` – восстановить обработчик сигнала, принятый по умолчанию (определения `SIG_DFL` и `SIG_IGN` см. в предыдущем разделе);
- `SIG_IGN` – игнорировать данный сигнал;
- адрес функции-обработчика, устанавливаемой как реакция на поступление этого сигнала. Эта функция будет выполняться при поступлении сигнала `signo`, и в качестве аргумента вызова она получит значение `signo` (одна функция может выступать как обработчик целой группы сигналов). Управление будет передано этой функции, как только процесс получит сигнал, какой бы участок кода при этом ни выполнялся. После возврата из функции управление будет возвращено в ту точку, в которой выполнение процесса было прервано.

Второе поле sa_mask демонстрирует первое применение набора сигналов: сигналы, установленные в sa_mask, будут блокироваться на время выполнения обработчика sa_handler (при вызове sa_handler и сам сигнал signo будет неявно добавлен в набор sa_mask, поэтому его можно не указывать явно). Это не значит, что поступившие в это время сигналы будут игнорироваться и теряться, просто их обработка будет отложена до завершения работы обработчика sa_handler.[1]

Поле sa_flags может использоваться для изменения характера реакции на сигнал signo. Возможны следующие значения поля флагов:

- SA_RESETHAND – после выполнения функции обработчика будет восстановлен обработчик по умолчанию (SIG_DFL, что соответствует духу модели «ненадежных сигналов» и позволяет воспроизводить ее поведение);
- SA_NOCLDSTOP – используется только для сигнала SIGCHLD; флаг указывает системе не генерировать для родительского процесса SIGCHLD от порожденных процессов, которые завершаются посредством SIGSTOP.
- SA_SIGINFO – при этом будет использована обработка сигналов на базе очереди сигналов (модель сигналов реального времени). По умолчанию используется простая обработка: результат воздействия нескольких сигналов определяется последним поступившим. В случае установки этого флага будет использована расширенная форма обработчика sa_siaction (при этом поле sa_handler не будет использоваться)[2]. Обработчику будет передаваться дополнительная информация о сигнале – структура siginfo_t (его номер, PID пославшего сигнал процесса, действующий идентификатор пользователя этого процесса). Эта весьма объемная структура будет очень кратко рассмотрена ниже.[3] Ее описание вынесено в отдельный заголовочный файл <sys/siginfo.h> и может быть изучено там.

Приведем несколько небольших и самых простых примеров использования модели надежных сигналов.

---

1 Все это и делает механизм обработки более надежным по сравнению с более ранним механизмом, который описывался выше.

2 Спецификация XSI требует, чтобы процесс использовал либо поле sa_handler, либо поле sa_sigaction, но не оба поля одновременно (в случае «классической» структуры sigaction, см. выше). Реализация QNX за счет объединения двух обработчиков под одним union обеспечивает это требование автоматически, хотя определения при этом становятся несколько более громоздкими.

3 Модель очереди сигналов введена главным образом для обеспечения сигналов реального времени и будет рассмотрена ниже.

Модель надежных сигналов

1. Перехватчик сигнала SIGINT (реакция на пользовательский ввод [Ctrl+C])[1] (*файл s8.cc*):

```
void catchint( int signo ) {
   cout << "SIGINT: signo = " << signo << endl;
};

int main() {
   static struct sigaction act = { &catchint, 0, (sigset_t)0 };
   // запрещаем любые сигналы на время обработки SIGINT:
   sigfillset( &(act.sa_mask) );
   // до этого вызова реакцией на Ctrl+C будет завершение задачи:
   sigaction( SIGINT, &act, NULL );
   for( int i = 0; i < 20; i++ )
      sleep( 1 ), cout << "Cycle # " << i << endl;
};
```

   Результатом нормального (без вмешательства оператора) выполнения приложения будет последовательность из 20 циклов секундных ожиданий, но если в процессе этих ожиданий пользователь пытается прервать работу процесса по [Ctrl+C], то он получит вывод, подобный следующему:

```
...
Cycle # 10
... здесь пользователь пытается прервать программу ...
SIGINT: signo = 2
Cycle # 11
...
```

2. Запрет прерывания выполнения программы с терминала. Для этого достаточно заменить строку инициализации структуры sigaction на:

```
static struct sigaction act = { SIG_IGN, 0, (sigset_t)0 };
```

   Можно проигнорировать сразу несколько сигналов (прерывающих выполнение программы с клавиатуры):

```
sigaction( SIGINT, &act, NULL );
sigaction( SIGQUIT, &act, NULL );
```

Далее остановимся еще на одном вызове API-сигналов, который широко используется в этой и последующих моделях обработки (сигналы реального времени, реакция в потоках):

```
int sigprocmask( int how, const sigset_t *set, sigset_t *oset );
```

---

[1] Инициализации, используемые в примерах вида `sigaction act = { &catchint, 0, (sigset_t)0 };`, будут зависимыми от системы из-за описанных ранее различий определения `struct sigaction` в разных ОС UNIX.

Этот вызов позволяет прочитать текущее значение (если set установлено в NULL, то параметр how игнорируется) или переустановить сигнальную маску для текущего потока. Параметры вызова:

- set – это то значение, в соответствии с которым корректируется сигнальная маска процесса;
- how – указывает, какое именно действие переустановки сигнальной маски требуется осуществить:
  - SIG_BLOCK – добавить сигналы, указанные в set к маске процесса (заблокировать реакцию на эти сигналы);
  - SIG_UNBLOCK – сбросить указанные set сигналы в сигнальной маске;
  - SIG_SETMASK – переустановить сигнальную маску процесса на значение, указанное в set.
- oset – значение, в котором будет сохранено значение маски, предшествующее вызову (старое значение).

Как и большинство сигнальных функций, данная функция возвращает нулевое значение в результате успешного выполнения и –1 в случае неудачи, при этом код ошибки устанавливается в errno.

Именно эта функция снимает одно из самых существенных ограничений, свойственных модели «ненадежных сигналов», – позволяет заблокировать реакцию на сигналы при выполнении критических участков кода и восстановить ее при завершении выполнения этих участков.

Модель сигналов реального времени

Сигналы реального времени были добавлены в POSIX относительно недавно (1996 г.). Эта новая модель в различных ОС UNIX реализуется с разной степенью полноты и с отклонениями от спецификаций, и QNX не исключение. Модель еще до конца не отработана, поэтому возможны сюрпризы (и сейчас они будут).

Модель сигналов реального времени, которую специфицирует POSIX, устанавливается флагом SA_SIGINFO (который уже упоминался выше) при вызове sigaction(). В нижеследующем перечислении того, что предусматривает эта модель, мы излагаем в первую очередь качественную картину происходящего, предлагаемую POSIX, кое-где уточняя ее конкретными данными реализации QNX (артефакты в поведении QNX будут отдельно отмечены позже):

1. Сигналы, называемые сигналами реального времени, могут принимать значения между SIGRTMIN и SIGRTMAX. Количество таких сигналов определяется в системе константой RTSIG_MAX, которая должна быть не менее 8 (POSIX). В QNX: SIGRTMIN = 41, SIGRTMAX = 56.
2. Обработка сигналов реального времени строится на основе очереди. Если сигнал порожден N раз, то он должен быть и N раз получен ад-

ресатом (в описываемых ранее моделях это не так, в них процесс получает только единичный экземпляр сигнала). Повторные экземпляры одного и того же сигнала в модели реального времени доставляются обработчику в порядке FIFO.

3. Помимо 8-битного кода с сигналом реального времени ассоциируется 32-битное значение (`si_value`, мы им займемся позже), заполняемое отправителем и доставляемое получателю (что позволяет «различать» экземпляры сигналов в очереди, о которой говорилось выше).

4. Для работы с сигналами реального времени добавлено несколько новых функций. В частности, в этой модели для отправки сигнала некоторому процессу используется `sigqueue()` вместо `kill()`.

   Эти два вызова определяются очень близкими формами:

```
int kill( pid_t pid, int signo );
int sigqueue( pid_t pid, int signo, const union sigval value );
```

Примечание

Как мы вскоре увидим, эти две синтаксические формы одного и того же вызова отличаются лишь тем, помещают ли они в сигнал указанное значение или оставляют его нулевым. Если процесс устанавливает обработку сигнала на основании очереди, он будет получать почти одинаковым образом сигналы, посланные обоими вызовами. Разница «почти» состоит в том, что получатель на основании анализа поля `si_code` в `siginfo_t` в состоянии отличить, каким вызовом ему был послан сигнал.

Примечание

При ошибке выполнения `sigqueue()` (код возврата –1) могут устанавливаться (в `errno`) следующие коды ошибок:

- `EAGAIN` – недостаточно ресурсов для помещения сигнала в очередь;
- `EINVAL` – недопустимое значение `signo` или неподдерживаемый сигнал;
- `ENOSYS` – вызов `sigqueue()` не поддерживается реализацией (возможно, версией);
- `EPERM` – у процесса недостаточно привилегий для посылки сигнала принимающему процессу;
- `ESRCH` – несуществующий PID процесса получателя.

Последний случай особо интересен, так как при указании в качестве номера сигнала `signo = 0` реальная посылка сигнала не производится, но устанавливается код ошибки. Это простейший и эффективный способ выяснить, выполняется ли в системе процесс с заданным PID.

5. Когда в очередь помещаются различные не заблокированные процессом (потоком) сигналы в диапазоне `SIGRTMIN...SIGRTMAX`, то сигналы с меньшими номерами доставляются обработчику из FIFO-очереди раньше сигналов с большими номерами (то есть сигналы с меньшими номерами имеют более высокий приоритет).

6. Обработчик для сигналов реального времени устанавливается с флагом SA_SIGINFO, а функция обработчика объявляется теперь с другим прототипом:

```
void func( int signo, siginfo_t* info, void* context );
```

Обработчик имеет больше параметров и получает больше информации. POSIX требует, чтобы тип siginfo_t содержал как минимум:

```
typedef struct {
   int si_signo;
   int si_code;
   union sigval si_value;  /* целое или указатель от отправителя*/
} siginfo_t;
```

В QNX sigval определяется так (подобное определение дают и другие ОС UNIX):

```
union sigval {
    int         sival_int;
    void       *sival_ptr;
};
```

Это 32-битное значение предназначено для посылки совместно с сигналом данных для получателя, которые, как видно из синтаксиса определения sigval, могут быть целочисленным значением или указателем неспецифицированного типа.

7. Поле si_code типа siginfo_t, передаваемое получателю, определяет природу возбуждения сигнала:
   - SI_ASINCIO – сигнал порожден завершением операций асинхронного ввода/вывода, запущенного одной из функций POSIX aio_*();
   - SI_MESGQ – сигнал возбуждается при помещении сообщения в пустую очередь сообщений UNIX;
   - SI_QUEUE – сигнал был отправлен функцией sigqueue() (в этом разделе нас интересуют только такие сигналы);
   - SI_TIMER – сигнал был порожден по истечении установленного времени интервального таймера;
   - SI_USER – сигнал был отправлен функцией kill().
8. Допускается, что при возбуждении сигнала еще каким-либо механизмом (сверх перечисленных, что может определяться специфическими особенностями ОС) значение si_code может отличаться от перечисленных. Однако значение поля si_value считается актуальным только в тех случаях, когда si_code имеет одно из значений: SI_ASINCIO, SI_MESGQ, SI_QUEUE, SI_TIMER.
9. Согласно POSIX сигналы, обработчики для которых также устанавливаются с флагом SA_SIGINFO, но не входящие в диапазон сигналов реального времени, например стандартные сигналы UNIX, могут

обрабатываться как на основе помещения их в очередь, так и без ее использования; выбор оставляется на усмотрение разработчика ОС.

Мы перечислили основные требования POSIX к модели обработки сигналов реального времени. Дополнения, отличия и специфические структуры данных QNX будут рассмотрены немного позже.

Весьма доходчивый пример для проверки и иллюстрации обработки сигналов реального времени приведен У. Стивенсом [2]. Мы же построим приложение, реализующее его основную идею:[1]

Приоритеты сигналов реального времени

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <signal.h>
#include <unistd.h>

static void handler( int signo, siginfo_t* info, void* context ) {
   cout << "received signal " << signo
        << " code = " << info->si_code
        << " val = " << info->si_value.sival_int << endl;
};

int main( int argc, char *argv[] ) {
   cout << "signal SIGRTMIN=" << (int)SIGRTMIN
        << " - signal SIGRTMAX=" << (int)SIGRTMAX << endl;
   int opt, val, beg = SIGRTMAX, num = 3,
       fin = SIGRTMAX - num, seq = 3;
   // обработка параметров запуска:
   while ( ( opt = getopt( argc, argv, "b:e:n:") ) != -1 ) {
      switch( opt ) {
         case 'b' : // начальный сигнал серии
            if( sscanf( optarg, "%i", &val ) != 1 )
               perror( "parse command line failed" ),
               exit( EXIT_FAILURE );
            beg = val;
            break;
         case 'e' :  // конечный сигнал серии
            if( sscanf( optarg, "%i", &val ) != 1 )
               perror( "parse command line failed" ),
               exit( EXIT_FAILURE );
            fin = val;
            break;
```

[1] Повторить приложение У. Стивенса в QNX в чистом виде не удастся – оно аварийно завершится по сигналу. Тонкий анализ этого факта интересен сам по себе, но он выходит за рамки нашего рассмотрения. Мы обращаем внимание на это обстоятельство, чтобы лишний раз сделать акцент на достаточно ощутимых отличиях реализаций QNX от схем POSIX (или того, как эти схемы понимаются в других ОС).

```
      case 'n' : // количество сигналов в группе посылки
         if( sscanf( optarg, "%i", &val ) != 1 )
            perror( "parse command line failed" ),
            exit( EXIT_FAILURE );
         seq = val;
         break;
      default :
         exit( EXIT_FAILURE );
   }
};
num = fin - beg;
fin += num > 0 ? 1 : -1;
sigset_t sigset;
sigemptyset( &sigset );
for( int i = beg; i != fin; i += ( num > 0 ? 1 : -1 ) )
     sigaddset( &sigset, i );
pid_t pid;
// вот здесь ветвление на 2 процесса...
if( pid = fork() == 0 ) {
   // дочерний процесс: здесь будут приниматься посланные сигналы
   sigprocmask( SIG_BLOCK, &sigset, NULL );
   for( int i = beg; i != fin;
        i += ( num > 0 ? 1 : -1 ) ) {
      struct sigaction act, oact;
      sigemptyset( &act.sa_mask );
      act.sa_sigaction = handler;
      // вот оно - реальное время!
      act.sa_flags = SA_SIGINFO;
      if( sigaction( i, &act, NULL ) < 0 )
         perror( "set signal handler: " );
   };
   cout << "CHILD: signal mask set" << endl;
   sleep( 3 );  // пауза для посылки сигналов родителем
   cout << "CHILD: signal unblock" << endl;
   sigprocmask( SIG_UNBLOCK, &sigset, NULL );
   sleep( 3 );  // пауза для приема всех сигналов
   exit( EXIT_SUCCESS );
}
// родительский процесс: отсюда будут посылаться сигналы
sigprocmask( SIG_BLOCK, &sigset, NULL );
// пауза для установки обработчиков дочерним процессом
sleep( 1 );
union sigval value;
for( int i = beg; i != fin;
     i += ( num > 0 ? 1 : -1 ) ) {
   for( int j = 0; j < seq; j++ ) {
      value.sival_int = j;
      sigqueue( pid, i, value );
      cout << "signal sent: " << i
           << " with val = " << j << endl;
   };
```

```
    };
    cout << "PARENT: finished!" << endl;
    exit( EXIT_SUCCESS );
};
```

Идея этого теста крайне проста:

- Создаются два процесса, один из которых (родительский) посылает серию последовательных (по номерам) сигналов, а второй (дочерний) должен их принять и обработать.
- Начальный и конечный номера сигналов в серии могут быть переопределены ключами `-b` и `-e` соответственно.
- Посылается не одиночный сигнал, а их повторяющаяся группа; размер группы повторений может быть переопределен ключом `-n`.
- В качестве значения, передаваемого с каждым сигналом, устанавливается последовательный номер его посылки в группе.

Таким образом, мы можем изменять последовательность сигналов на передаче и наблюдать последовательность их доставки к принимающему процессу. Запустим полученное приложение и сразу же командой `pidin` посмотрим его состояние:

```
1077295   1 ./s5               10r NANOSLEEP
1081392   1 ./s5               10r NANOSLEEP
```

Это то, что мы и предполагали получить. Рассмотрим теперь результат выполнения приложения со значениями сигналов по умолчанию (сигналы 56...54, именно в порядке убывания, в каждой группе посылается 3 сигнала):

```
# ./s5
signal SIGRTMIN=41 - signal SIGRTMAX=56
CHILD: signal mask set
signal sent: 56 with val = 0
signal sent: 56 with val = 1
signal sent: 56 with val = 2
signal sent: 55 with val = 0
signal sent: 55 with val = 1
signal sent: 55 with val = 2
signal sent: 54 with val = 0
signal sent: 54 with val = 1
signal sent: 54 with val = 2
PARENT: finished!
# CHILD: signal unblock
received signal 56 code = -2 val = 0
received signal 56 code = -2 val = 1
received signal 56 code = -2 val = 2
received signal 55 code = -2 val = 0
received signal 55 code = -2 val = 1
received signal 55 code = -2 val = 2
received signal 54 code = -2 val = 0
```

```
received signal 54 code = -2 val = 1
received signal 54 code = -2 val = 2
```

Первый сюрприз, который нас ожидает, – это общее количество сигналов реального времени, выводимое программой в первой строке. Документация (HELP QNX) утверждает:

There are 24 POSIX 1003.1b realtime signals, including:
SIGRTMIN – First realtime signal.
SIGRTMAX – Last realtime signal.

Здесь есть некоторое несоответствие: тест дает значения констант SIGRTMIN = 41 и SIGRTMAX = 56, а общее количество сигналов = 16 (POSIX, как вы помните, требует минимум 8). «Потерявшиеся» сигналы (24 – 16 = 8), очевидно, и есть те сигналы из диапазона 32...40, которые выходят за пределы общих UNIX-сигналов (1...31), но не отнесены к диапазону сигналов реального времени (41...56).

Но гораздо больший сюрприз состоит в порядке доставки сигналов из очереди FIFO принимающему процессу. Документация об этом сообщает (выделено нами):

The POSIX standard includes the concept of queued realtime signals. QNX Neutrino supports optional queuing of any signal, not just realtime signals. The queuing can be specified on a signal-by-signal basis within a process. Each signal can have an associated 8-bit code and a 32-bit value.

This is very similar to message pulses described earlier. The kernel takes advantage of this similarity and uses common code for managing both signals and pulses. The signal number is mapped to a pulse priority using _SIGMAX -- signo. ***As a result, signals are delivered in priority order with lower signal numbers having higher priority.*** *This conforms with the POSIX standard, which states that existing signals have priority over the new realtime signals.*

Изменим временной порядок возбуждения сигналов – от сигналов с меньшими номерами к сигналам с большими номерами:

```
# ./s5 -b54 -e56 -n2
signal SIGRTMIN=41 - signal SIGRTMAX=56
CHILD: signal mask set
signal sent: 54 with val = 0
signal sent: 54 with val = 1
signal sent: 55 with val = 0
signal sent: 55 with val = 1
signal sent: 56 with val = 0
signal sent: 56 with val = 1
PARENT: finished!
# CHILD: signal unblock
received signal 56 code = -2 val = 0
received signal 56 code = -2 val = 1
received signal 55 code = -2 val = 0
```

```
received signal 55 code = -2 val = 1
received signal 54 code = -2 val = 0
received signal 54 code = -2 val = 1
```

Независимо от порядка отправки сигналов порядок доставки их из очереди принимающему процессу сохраняется от старших номеров сигналов, которые являются более приоритетными, к младшим. А это в точности противоположно тому, что обещает документация, и соответствует картине, которую У. Стивенс наблюдал в ОС Sun Solaris 2.6 и которую он же комментирует словами: «Похоже, что в реализации Solaris 2.6 есть ошибка».

Выполним ту же задачу, но теперь не относительно сигналов диапазона реального времени, а относительно стандартных UNIX-сигналов. Выберем для этого произвольный диапазон сигналов (<signal.h>):

```
#define SIGVTALRM   28  /* virtual timer expired */
#define SIGPROF     29  /* profileing timer expired */
#define SIGXCPU     30  /* exceded cpu limit */
#define SIGXFSZ     31  /* exceded file size limit */
```

Посмотрим результат в этом случае:

```
# ./s5 -b28 -e31 -n2
signal SIGRTMIN=41 - signal SIGRTMAX=56
CHILD: signal mask set
signal sent: 28 with val = 0
signal sent: 28 with val = 1
signal sent: 29 with val = 0
signal sent: 29 with val = 1
signal sent: 30 with val = 0
signal sent: 30 with val = 1
signal sent: 31 with val = 0
signal sent: 31 with val = 1
PARENT: finished!
# CHILD: signal unblock
received signal 31 code = -2 val = 0
received signal 31 code = -2 val = 1
received signal 30 code = -2 val = 0
received signal 30 code = -2 val = 1
received signal 29 code = -2 val = 0
received signal 29 code = -2 val = 1
received signal 28 code = -2 val = 0
received signal 28 code = -2 val = 1
```

QNX при установке обработчика с флагом SA_SIGINFO использует модель работы реального времени не только относительно сигналов диапазона SIGRTMIN...SIGRTMAX, но и для всего множества стандартных UNIX-сигналов. Это расширение, впрочем, не противоречит приведенному выше утверждению POSIX, что такое решение факультативно (см. пункт 9 описания модели сигналов реального времени) и оставляется на усмотрение разработчика ОС.

Любопытным может оказаться и рассмотрение реакции на сигналы, которые никак не определены в QNX (в исследуемый диапазон для сравнения включены как неопределенные, так и определенные сигналы системы):

```
# ./s5 -b39 -e41 -n2
signal SIGRTMIN=41 - signal SIGRTMAX=56
CHILD: signal mask set
signal sent: 39 with val = 0
signal sent: 39 with val = 1
signal sent: 40 with val = 0
signal sent: 40 with val = 1
signal sent: 41 with val = 0
signal sent: 41 with val = 1
PARENT: finished!
# CHILD: signal unblock
received signal 41 code = -2 val = 0
received signal 41 code = -2 val = 1
received signal 40 code = -2 val = 0
received signal 40 code = -2 val = 1
received signal 39 code = -2 val = 0
received signal 39 code = -2 val = 1
```

Для них реакция никак не отличается от реакции на другие сигналы, что, впрочем, неудивительно, учитывая замечание в цитировавшемся выше фрагменте документации о том, что возбуждение сигнала и посылка импульса (сообщения) микроядра в QNX – одно и то же, и обрабатываются они единым механизмом.

Посмотрим также реакцию системы на специальные сигналы QNX, номера которых выше SIGRTMAX (в исследуемый диапазон опять для сравнения включим как специальные сигналы (57...64), так и сигналы из диапазона 1... SIGRTMAX):

```
# ./s5 -b56 -e59 -n2
signal SIGRTMIN=41 - signal SIGRTMAX=56
set signal handler: : Invalid argument
set signal handler: : Invalid argument
set signal handler: : Invalid argument
CHILD: signal mask set
signal sent: 56 with val = 0
signal sent: 56 with val = 1
signal sent: 57 with val = 0
signal sent: 57 with val = 1
signal sent: 58 with val = 0
signal sent: 58 with val = 1
signal sent: 59 with val = 0
signal sent: 59 with val = 1
PARENT: finished!
# CHILD: signal unblock
received signal 56 code = -2 val = 0
received signal 56 code = -2 val = 1
```

Из вывода видно, что на сигнал с номером 56 реакция ожидаемая, а на остальные сигналы реакции нет вовсе. Как и следует из предупреждения в документации, заблокировать или изменить реакцию на эти сигналы невозможно, и попытка установки `sigaction()` для них завершается ошибкой.

Таким образом, система фактически никак не выделяет сигналы диапазона реального времени (41…56), но обрабатывает аналогичным образом и стандартные сигналы UNIX (1…31), и специальные сигналы QNX (57…64), и даже сигналы, никак не специфицируемые системой вообще (32…40). Корректнее говорить не об обработке сигналов реального времени и даже не о модели сигналов реального времени, а об еще одном способе работы с любыми сигналами – обработке сигналов на базе очередей сигналов.

Примечание

Для полноты картины приведем конкретную спецификацию типа `siginfo_t` для QNX (выше мы рассматривали минимальную спецификацию этого типа, требуемую POSIX). Спецификация весьма объемна (вся информация до конца раздела может быть безболезненно пропущена теми, кому это неинтересно), но предоставляет программисту исчерпывающую информацию о полученном сигнале:

```
typedef struct {
   int      si_signo;
   int      si_code;          /* if SI_NOINFO, only si_signo is valid */
   int      si_errno;
   union {
      int            __pad[7];
      struct {
         pid_t               __pid;
         union {
            struct {
               uid_t               __uid;
               union sigval        __value;
            }  __kill;          /* si_code <= 0 SI_FROMUSER */
            struct {
               _CSTD clock_t __utime;
               /* CLD_EXITED status, else signo */
               int           _status;
               _CSTD clock_t __stime;
            }__chld;
            /* si_signo=SIGCHLD si_code=CLD_* */
         } __pdata;
      } __proc;
      struct {
         int __fltno;
         void*__fltip;
         void*__addr;
      }__fault;     /* si_signo=SIGSEGV,ILL,FPE,TRAP,BUS */
   } __data;
```

```
} siginfo_t;
#define si_pid     __data.__proc.__pid
#define si_value     __data.__proc.__pdata.__kill.__value
#define si_uid       __data.__proc.__pdata.__kill.__uid
#define si _status   __data.__proc.__pdata.__chld.__status
#define si_utime   __data.__proc.__pdata.__chld.__utime
#define si_stime   __data.__proc.__pdata.__chld.__stime
#define si_fltno   __data.__fault.__fltno
#define si_trapno  si_fltno
#define si_addr    __data.__fault.__addr
#define si_fltip   __data.__fault.__fltip
```

И полный перечень определений символьных констант, используемых для установки значений поля `si_code`:

```
#define SI_USER             0
#define SI_RESERVED1        (-1)
#define SI_QUEUE            (-2)
#define SI_TIMER            (-3)
#define SI_ASYNCIO          (-4)
#define SI_MESGQ            (-5)
#define SI_NOTIFY           (-6)
#define SI_IRQ              (-7)
// QNX managers will never use this range:
#define SI_MINAVAIL         (-128)
#define SI_MAXAVAIL         (-64)
#define SI_NOINFO           127
#define SI_MAXSZ            126
```

Значение `SI_QUEUE`, равное –2, – это и есть то значение, которое мы видели в выводе тестовой задачи выше.

Соображения производительности

Ранее мы производили оценку затрат производительности процессора на переключение между контекстами для процессов и для потоков (тестовые задачи, которые мы по их структуре именовали как «симметричные»). Проделаем теперь то же самое, но синхронизацию процессов выполним посылкой и приемом сигнала (переключение контекстов будет происходить именно на операторе `pause()` – переходе к приему сигнала) (*файл pbs.cc*):

Затраты на переключение процессов посылкой сигналов

```
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>
#include <iostream.h>
#include <unistd.h>
#include <sched.h>
#include <sys/neutrino.h>
```

```
// "пустые" обработчики сигналов
static void nhand( int signo ) { };
static void qhand( int signo, siginfo_t* info, void* context ) { };

int main( int argc, char *argv[] ) {
   unsigned long N = 1000;
   bool que = false;
   int opt, val;
   while ( ( opt = getopt( argc, argv, "n:q" ) ) != -1 ) {
      switch( opt ) {
         case 'n' :
            if( sscanf( optarg, "%i", &val ) != 1 )
               cout << "parse command line error" << endl,
               exit( EXIT_FAILURE );
            if( val > 0 ) N = val;
            break;
         // ключ q определяет схему обработки сигнала
         case 'q' :
            que = true;
            break;
         default :
            exit( EXIT_FAILURE );
       }
   };
   // установка сигнальных обработчиков
   sigset_t sig;
   sigemptyset( &sig );
   sigaddset( &sig, SIGUSR1 );
   sigprocmask( SIG_UNBLOCK, &sig, NULL );
   struct sigaction act;
   act.sa_mask = sig;
   act.sa_sigaction = qhand;
   act.sa_handler = nhand;
   act.sa_flags = que ? SA_SIGINFO : 0;
   if( sigaction( SIGUSR1, &act, NULL ) < 0 )
      cout << "set signal handler" << endl,
      exit( EXIT_FAILURE );
   pid_t pid = fork();
   if( pid == -1 )
      cout << "fork error" << endl, exit( EXIT_FAILURE );
   // кому отправлять сигнал? :
   pid_t did = ( pid == 0 ? getppid() : pid );
   unsigned long i = 0;
   uint64_t t = ClockCycles();
   while( true ) {
      kill( did, SIGUSR1 );
      if( ++i == N ) break;
      pause();
   };
   t = ClockCycles() - t;
   cout << getpid() << " -> " << did << "\t: cycles - "
```

```
        << t << "; on signal - " << ( t / N ) / 2 << endl;
    exit( EXIT_SUCCESS );
};
```

Этим приложением мы можем тестировать и традиционную схему обработки сигналов (модель надежных сигналов), и схему обработки с очередью поступления сигналов (модель сигналов реального времени), когда при старте программы указан ключ `-q`. Посмотрим на результаты тестовых запусков:

```
# nice -n-19 p6s -n1000
2904115 -> 2912308   : cycles - 5792027; on signal - 2896
2912308 -> 2904115   : cycles - 5828952; on signal - 2914
# nice -n-19 p6s -n10000
2920499 -> 2928692   : cycles - 57522753; on signal - 2876
2928692 -> 2920499   : cycles - 57530378; on signal - 2876
# nice -n-19 p6s -n100000
2936883 -> 2945076   : cycles - 573730469; on signal - 2868
2945076 -> 2936883   : cycles - 573738122; on signal - 2868
# nice -n-19 p6s -n1000000
2953267 -> 2961460   : cycles - 5747418203; on signal - 2873
2961460 -> 2953267   : cycles - 5747425310; on signal - 2873
```

Вспомним, что при изучении тестов простого переключения процессов (см. в главе 2) мы получали цифру порядка 600 процессорных циклов на переключение. Сейчас у нас затраты заметно больше: порядка 2850 циклов, из которых «лишние» 2250 – это не что иное, как затраты на посылку и прием сигнала, возбуждение функции обработчика и ее завершение (разделить их по компонентам мы не можем). Это и может служить ориентировочной оценкой трудоемкости обмена сигналами.

Проделаем то же самое, но уже при обработке сигналов в порядке очереди их поступления:

```
# nice -n-19 p6s -n1000 -q
2838579 -> 2846772   : cycles - 5772106; on signal - 2886
2846772 -> 2838579   : cycles - 5782138; on signal - 2891
# nice -n-19 p6s -n10000 -q
2854963 -> 2863156   : cycles - 57194634; on signal - 2859
2863156 -> 2854963   : cycles - 57199831; on signal - 2859
# nice -n-19 p6s -n100000 -q
2871347 -> 2879540   : cycles - 571543013; on signal - 2857
2879540 -> 2871347   : cycles - 571550847; on signal - 2857
# nice -n-19 p6s -n1000000 -q
2887731 -> 2895924   : cycles - 5715903548; on signal - 2857
2895924 -> 2887731   : cycles - 5715908318; on signal - 2857
```

Это практически те же цифры, поэтому мы можем предположить, что, вообще-то говоря, для всех рассмотренных ранее схем обработки реализуется один и тот же внутренний механизм приема сигналов, который только лишь модифицируется в зависимости от используемой схемы.

Сигналы в потоках

Модель реакций на сигналы многопоточных приложений не проработана до конца в рамках POSIX и находится на стадии предварительных предложений. Тем не менее в системах с развитой многопоточностью (а QNX – именно такая система) эта сторона вопроса не может игнорироваться, и не только потому, что потоки в комбинации с сигналами могут создавать мощные конструктивные элементы программ, а еще и потому, что непроизвольные разблокирующие или завершающие операции, инициируемые сигналами, могут породить очень серьезные проблемы в случае многопоточности (мы еще будем возвращаться к этим вопросам по тексту). А раз так, то в этих случаях система должна обязательно предлагать некоторую модель функционирования (удачную или не очень).

Для того чтобы не допускать разночтений в вопросе, обратимся сначала к оригинальному фрагменту документации, описывающему принятую модель:

> *The original POSIX specification defined signal operation on processes only. In a multi-threaded process, the following rules are followed:*
>
> **The signal actions are maintained at the process level. If a thread ignores or catches a signal, it affects all threads within the process.*
>
> **The signal mask is maintained at the thread level. If a thread blocks a signal, it affects only that thread.*
>
> **An un-ignored signal targeted at a thread will be delivered to that thread alone.*
>
> **An un-ignored signal targeted at a process is delivered to the first thread that doesn't have the signal blocked. If all threads have the signal blocked, the signal will be queued on the process until any thread ignores or unblocks the signal. If ignored, the signal on the process will be removed. If unblocked, the signal will be moved from the process to the thread that unblocked it.*

Все достаточно однозначно: обработчики сигналов определяются на уровне процесса, а вот сигнальные маски, определяющие, реагировать ли на данный сигнал, – на уровне каждого из потоков.

Для манипулирования сигнальными масками на уровне потоков нам придется использовать функцию `SignalProcmask()`[1] (естественно, из состава native API, поскольку эта модель не декларирована POSIX):

```
#include <sys/neutrino.h>
int SignalProcmask ( pid_t pid, int tid, int how, const sigset_t* set,
                     sigset_t* oldset );
```

[1] Функция `SignalProcmask()` имеет свой реентерабельный (безопасный в потоках) эквивалент: `SignalProcmask_r()`.

Видна прямая аналогия с рассматривавшейся ранее функцией `sigprocmask()`. Да это и неудивительно, поскольку `sigprocmask()` является POSIX-«оберткой» к `SignalProcmask()`. Только рассматриваемый вызов имеет два «лишних» начальных параметра: PID и TID потока, к маске которого применяется действие. Если `pid` = 0, то предполагается текущий процесс, если `tid` = 0, то `pid` игнорируется и предполагается текущий поток, вызывающий функцию.

Остальные параметры соответствуют параметрам `sigprocmask()` (дополнительно появляется возможное значение `SIG_PENDING` для `how`).

Рассмотрим, как это работает на примере простейшего кода (*файл s6.cc*):

Сигналы, обрабатываемые в потоках

```
#include <stdio.h>
#include <iostream.h>
#include <signal.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <sys/neutrino.h>

static void handler( int signo, siginfo_t* info, void* context ) {
   cout << "SIG = " << signo << "; TID = "
        << pthread_self() << endl;
};

sigset_t sig;

void* threadfunc ( void* data ) {
   SignalProcmask( 0, 0, SIG_UNBLOCK, &sig, NULL );
   while( true ) pause();
};

int main() {
   sigemptyset( &sig );
   sigaddset( &sig, SIGRTMIN );
   sigprocmask( SIG_BLOCK, &sig, NULL );
   cout << "Process " << getpid() << ", waiting fot signal "
        << SIGRTMIN << endl;
   struct sigaction act;
   act.sa_mask = sig;
   act.sa_sigaction = handler;
   act.sa_flags = SA_SIGINFO;
   if( sigaction( SIGRTMIN, &act, NULL ) < 0 )
      perror( "set signal handler: " );
   const int thrnum = 3;
   for( int i = 0; i < thrnum; i++ )
      pthread_create( NULL, NULL, threadfunc, NULL );
   pause();
```

```
    exit( EXIT_SUCCESS );
};
```

Для анализа этого и последующих фрагментов нам будет недостаточно команды `kill`, поэтому сделаем простейший «передатчик» плотной (в смысле минимального интервала следования) последовательности повторяющихся сигналов (*файл k6.cc*). Выполнение этого тестера, например по команде:

```
# k6 -p214005 -s41 -n100
```

направляет процессу с PID = 214005 последовательность из 100 сигналов с кодом 41 (`SIGRTMIN`). Посылая нашему процессу-тестеру последовательность из N сигналов, мы получим N сообщений вида:

```
SIG = 41; TID = 4
```

Примечание

Здесь удобный случай показать разницу между обработкой сигналов на базе очереди и простой обработкой (модель надежных сигналов). Для этого заменим две строки заполнения структуры `sigaction` на:

```
act.sa_handler = handler;
act.sa_flags = 0;
```

а заголовок функции `handler()` перепишем так: `static void handler(int signo)`. Если теперь мы в точности повторим предыдущий тест, то при посылке процессу-тестеру последовательности из N сигналов мы получим всего одно сообщение все того же вида. Это наблюдение интересно еще и тем, что оно показывает, что алгоритм взаимодействия сигнала с потоками не зависит от того, какая обработка установлена для этого сигнала: на основе модели сигналов реального времени или на основе модели надежных сигналов.

Сколько бы раз мы ни повторяли тестирование, идентификатор потока, получающего и обрабатывающего сигнал, всегда будет равен 4. Что же происходит:

- главный поток (TID = 1) создает 3 новых потока (TID = 2, 3, 4);
- главный поток переходит в пассивное ожидание сигналов, но в его маске доставка посылаемого сигнала (41) заблокирована;
- выполнение функции потока начинается с разблокирования ожидаемого сигнала;
- … 3 потока (TID = 2, 3, 4) ожидают поступления сигнала;
- при поступлении серии сигналов вся их очередь доставляется и обрабатывается одним потоком с TID = 4, который тут же в цикле возвращается к ожиданию следующих сигналов.

Таким образом, сигнал доставляется одному и только одному потоку, который не блокирует этот сигнал. Обработчик сигнала вызывается в контексте (стек, области собственных данных) этого потока. После

выполнения обработчика сигнал поглощается. Какому из потоков, находящихся в состоянии блокирования в ожидании сигналов (в масках которых разблокирован данный сигнал), будет доставлен экземпляр сигнала, предсказать невозможно; это так и должно быть исходя из общих принципов диспетчеризации потоков. Но реально этим потоком является поток, **последним** перешедший в состояние ожидания. Для того чтобы убедиться в этом, заменим предпоследнюю строку программы (`pause();`) на:

```
threadfunc( NULL );
```

Теперь у нас 4 равнозначных потока, ожидающих прихода сигнала, переходящих в состояние ожидания в последовательности: TID = 2, 3, 4, 1. Реакция процесса на приход сигнала изменится на:

```
SIG = 41; TID = 1
```

Изменим текст функции потока на (*файл s7.cc*):

```
void* threadfunc ( void* data ) {
   while( true ) {
      SignalProcmask( 0, 0, SIG_UNBLOCK, &sig, NULL );
      delay( 1 );
      SignalProcmask( 0, 0, SIG_BLOCK, &sig, NULL );
      delay( 10 );
   };
};
```

Поведение приложения радикально изменится – происходит смена обрабатывающего потока (чтобы сократить объем вывода, серии посылаемых сигналов состоят из одного сигнала). Следует отметить, что смена обрабатывающего потока происходит между сериями, но ни в коем случае не внутри длинных серий, что можно проследить экспериментально.

```
SIG = 41; TID = 1
SIG = 41; TID = 4
SIG = 41; TID = 4
SIG = 41; TID = 1
SIG = 41; TID = 1
SIG = 41; TID = 4
SIG = 41; TID = 4
SIG = 41; TID = 1
SIG = 41; TID = 2
SIG = 41; TID = 2
SIG = 41; TID = 3
SIG = 41; TID = 4
```

Такая модель вряд ли может быть названа в полной мере «сигналами в потоках», так как сигнал в ней в конечном итоге направляется процессу как контейнеру, содержащему потоки (можно сказать и так: в оболочку адресного пространства процесса). И только после этого

в контексте одного из потоков (и в случае множественных потоков, разблокированных на обработку единого сигнала, невозможно предсказать, в контексте какого из них) выполняется обработчик сигнала. Главный поток процесса (TID = 1) в этой схеме участвует в равнозначном качестве (здесь хорошо видно, что устоявшееся понятие «реакция процесса на сигнал» в строгом смысле некорректно).

Перейдем к более конкретным вопросам: как можно продуктивно использовать эту схему в многопоточных приложениях? Рассмотрим сначала случай, когда каждый из рабочих потоков разблокирован на получение одного, свойственного только ему сигнала (*файл s9.cc*):

Чередование потоковых сигналов

```
#include <stdio.h>
#include <iostream.h>
#include <signal.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <sys/neutrino.h>

static void handler( int signo, siginfo_t* info, void* context ) {
   cout << "SIG = " << signo << "; TID = "
        << pthread_self()  << endl;
};

void* threadfunc ( void* data ) {
   // блокировать реакцию на все сигналы
   sigset_t sig;
   sigfillset( &sig );
   SignalProcmask( 0, 0, SIG_BLOCK, &sig, NULL );
   // разблокировать реакцию на свой сигнал
   sigemptyset( &sig );
   sigaddset( &sig, (int)data );
   SignalProcmask( 0, 0, SIG_UNBLOCK, &sig, NULL );
   // цикл ожидания приходящих сигналов:
   while( true ) pause();
};

int main() {
   // для обработки всей группы сигналов управления потоками используем
   // единую функцию реакции, иначе все было бы гораздо проще.
   struct sigaction act;
   sigemptyset( &act.sa_mask );
   act.sa_sigaction = handler;
   act.sa_flags = SA_SIGINFO;
   // создаем группу однотипных потоков:
   const int thrnum = 3;
   for( int i = SIGRTMIN; i - SIGRTMIN < thrnum; i++ ) {
      sigset_t sig;
```

```
      sigemptyset( &sig );
      sigaddset( &sig, i );
      // нам нужно, чтобы главный поток не реагировал:
      sigprocmask( SIG_BLOCK, &sig, NULL );
      if( sigaction( i, &act, NULL ) < 0 )
         perror( "set signal handler: " );
      // для передачи номера сигнала используется
      // трюк с подменой типа параметра:
      pthread_create( NULL, NULL, threadfunc, (void*) ( i ) );
   };
   // начинаем циклическую синхронизацию потоков:
   for( int i = 0; ; i++ ) {
      sleep( 1 );
// посылку сигнала можно (так даже будет корректнее)
// сделать так:
//    union sigval val;
//    val.sival_int = i;
//    sigqueue( getpid(), SIGRTMIN + i % thrnum, val );
// но мы сознательно демонстрируем и приемлемость kill:
      kill( getpid(), SIGRTMIN + i % thrnum );
   };
};
```

В этой программе главный поток циклически по таймеру активизирует поочередно каждый поток. Вот фрагмент вывода работающей программы:

```
SIG = 41; TID = 2
SIG = 42; TID = 3
SIG = 43; TID = 4
SIG = 41; TID = 2
SIG = 42; TID = 3
SIG = 43; TID = 4
SIG = 41; TID = 2
SIG = 42; TID = 3
SIG = 43; TID = 4
```

Часто приходится слышать: «...хотелось бы доставить сигнал всем потокам, уведомить всех потребителей и выполнить функцию реакции в каждом потоке...», и именно в такой последовательности действий понимается модель сигналов в потоках при поверхностном с ней ознакомлении. Иногда это представляется очень интересной возможностью, и мы реализуем такую схему взаимодействия в следующем фрагменте (*файл s10.cc*):

Множественная реакция на сигнал

```
#include <stdio.h>
#include <iostream.h>
#include <signal.h>
#include <unistd.h>
#include <pthread.h>
```

```
#include <sys/neutrino.h>
#include <vector>

static void handler( int signo, siginfo_t* info, void* context ) {
   cout << "SIG = " << signo << "; TID = "
        << pthread_self() << endl;
};

static void endhandler( int signo ) { };

// сигнал, на который реагируют потоки:
const int SIGNUM = SIGRTMIN;
sigset_t sig;
struct threcord {
   int tid;
   bool noblock;
};
static vector<threcord> tharrey; // вектор состояний потоков

void* threadfunc ( void* data ) {
   // блокирование всех прочих сигналов:
   sigset_t sigall;
   sigfillset( &sigall );
   SignalProcmask( 0, 0, SIG_BLOCK, &sigall, NULL );
   // передеспетчеризация для завершения формирования вектора:
   sched_yield();
   tharrey[ (int)data ].noblock =
      ( SignalProcmask( 0, 0, SIG_UNBLOCK, &sig, NULL ) != -1 );
   while( true ) {
      pause();
      tharrey[ (int)data ].noblock =
         !( SignalProcmask( 0, 0, SIG_BLOCK, &sig, NULL ) != 1 );
      bool nolast = false;
      for( vector<threcord>::iterator i = tharrey.begin();
           i != tharrey.end(); i++ )
         if( nolast = i->noblock ) break;
      // последовательная пересылка сигнала следующему потоку
      if( nolast ) kill( getpid(), SIGNUM );
      // ... когда пересылать больше некому -
      // переинициализация масок
      else
         for( vector<threcord>::iterator
              i = tharrey.begin();
              i != tharrey.end(); i++ )
            i->noblock = ( SignalProcmask( 0, i->tid,
                 SIG_UNBLOCK, &sig, NULL ) != -1 );
   };
};

int main() {
   // переопределение реакции ^C в старой манере
   signal( SIGINT, endhandler );
```

```
    // маска блокирования-разблокирования
    sigemptyset( &sig );
    sigaddset( &sig, SIGNUM );
    // блокировка в главном потоке приложения
    sigprocmask( SIG_BLOCK, &sig, NULL );
    cout << "Process " << getpid() << ", waiting fot signal "
         << SIGNUM << endl;
    // установка обработчика (для дочерних потоков)
    struct sigaction act;
    act.sa_mask = sig;
    act.sa_sigaction = handler;
    act.sa_flags = SA_SIGINFO;
    if( sigaction( SIGNUM, &act, NULL ) < 0 )
       perror( "set signal handler: " );
    const int thrnum = 3;
    for( int i = 0; i < thrnum; i++ ) {
       threcord threc = { 0, false };
       pthread_create( &threc.tid, NULL, threadfunc, (void*)i );
       tharrey.push_back( threc );
    };
    pause();
    // сюда мы попадаем после ^C для завершающих операций...
    tharrey.erase( tharrey.begin(), tharrey.end() );
    cout << "Clean vector" << endl;
};
```

Это приложение, в отличие от предыдущих, построено уже с использованием специфики C++, в нем используется контейнерный класс `vector` из библиотеки STL (Standard Template Library). Может быть множество вариаций на подобную тему. Приведенное нами приложение (как одна из вариаций) только подтверждает, что принятая в QNX модель достаточна для описания самых неожиданных потребностей. Логика работы приложения крайне проста: получая сигнал, поток блокирует повторную реакцию на этот сигнал, после чего возбуждает дубликат полученного сигнала от своего имени.

Примечание

Показанное приложение в значительной степени искусственно и неэффективно. Мы приводим его здесь не как образец того, «как нужно делать», а только как иллюстрацию гибкости возможностей, предоставляемых в области параллельного программирования. При некоторой изобретательности можно заставить программу вести себя согласно вашим капризам, какими бы изощренными они ни оказались.

Запускаем полученное приложение:

```
# s10
Process 2089006, waiting fot signal 41
```

После чего с другого терминала пошлем приложению ожидаемый им сигнал, например командой:

```
# kill -41 2089006
```

Посылаем этот сигнал несколько раз (в данном случае 3) и получаем вывод от приложения:

```
SIG = 41; TID = 4
SIG = 41; TID = 2
SIG = 41; TID = 3
SIG = 41; TID = 3
SIG = 41; TID = 4
SIG = 41; TID = 2
SIG = 41; TID = 2
SIG = 41; TID = 3
SIG = 41; TID = 4
^C
Clean vector
```

Видно, что реакция на каждый сигнал возбуждается несколько раз (по числу потоков), каждый раз выполняясь в контексте разного потока (TID). Интересно и изменение порядка активизации потоков от сигнала к сигналу, то есть потоки в очереди ожидающих «перетасовываются» при поступлении каждого сигнала.

Примечание

В приложение добавлена реакция на ^C (сигнал SIGINT):

- начиная с некоторой сложности приложений, их завершению должна обязательно предшествовать некоторая последовательность действий; в данном случае мы условно показываем очистку вектора состояний потоков;
- реакция на SIGINT выполнена в «ненадежной» манере в смешении с моделью очереди сигналов для SIGRTMIN, что показывает возможность смешанного применения всех моделей в рамках одного приложения; все определяется требованиями и вопросами удобства.

Как мы уже видели, тот факт, что обработчик сигнала выполняется в контексте потока, который разблокировал реакцию на этот сигнал (независимо от того, в момент выполнения какого потока приходит сигнал), позволяет реализовать в обработчике сигнала обработку любой сложности в интересах этого потока. Для этого лишь требуется разместить все области данных, запрашиваемые в этой обработке, не в стеке потока (объявленные как локальные переменные потоковой функции), а в области собственных данных потока, которые мы детально рассмотрели ранее. Схематично это можно показать в коде так:

- Положим, нам нужно уведомлять о некоторых событиях N потоков. Будем использовать для этого сигналы SIGRTMIN . . . SIGRTMIN + (N – 1):

```
for( int i = SIGRTMIN; i < SIGRTMIN + N; i++ ) {
   pthread_create( NULL, NULL, threadfunc, (void*)( i ) );
};
```

- При запуске N потоков (из главного потока) потоковые функции, помимо устанавливания своих индивидуальных сигнальных масок (в точности так, как это показано выше в листинге «Чередование потоковых сигналов»), размещают экземпляры собственных потоковых данных:

```
class DataBlock {
   ~ DataBlock( void ) { . . . };
};
static pthread_key_t key;
static pthread_once_t once = PTHREAD_ONCE_INIT;
static void destructor( void* db ) { delete (DataBlock*)db; };
static void once_creator( void ) {
   pthread_key_create( &key, destructor );
};

void* threadfunc ( void* data ) {
   // надлежащим образом маскируем сигналы
   // ...
   // это произойдет только в первом потоке из N
   pthread_once( &once, once_creator );
   DataBlock* pdb = new DataBlock( . . . );
   pthread_setspecific( key, pdb );
   // Теперь поток может пользоваться данными *pdb, как и локальными!
   // цикл ожидания приходящих сигналов:
   while( true ) pause();
};
```

- Все потоки используют один и тот же обработчик для всех сигналов; он выполняет одни и те же действия, но над различными объектами данных. Над каким объектом данных выполнить действие, обработчик «узнает» из контекста потока, в котором он выполняется:

```
static void handler( int signo, siginfo_t* info, void* context ) {
   DataBlock* pdb = (DataBlock*)pthread_getspecific( key );
   // выполняем действия для своего потока . . .
};
```

- Теперь, например из главного потока процесса (главный поток выбран для простоты – источником сигнала может быть произвольный поток, даже не этого процесса), требуемое действие вызывается возбуждением соответствующего сигнала:

```
sigqueue( getpid(), SIGRTMIN + K, val );
```

Это только скелетная схема, но на ее основе можно строить развитые протоколы обработки данных (пример взят из работоспособного приложения).

За пределы POSIX: сигналы в сети

А теперь, «на закуску», посмотрим справочную информацию по системной команде `kill` (послать сигнал). Вы, должно быть, помните, что в QNX есть дополнительная возможность получить справку по любой команде системы, используя команду `# use <имя-команды>`. Более того, вы можете и в любое свое приложение встроить возможность получения интерактивной справки. Как это происходит, описано в [4]. Итак:

```
# use kill
kill - terminate or signal processes (POSIX)

kill [-signal_name|-signal_number] pid ...
kill -l
Options:
 -signal_name   Symbolic name of signal to send
 -signal_number Integer representing a signal type
 -l             List symbolic signal names
 -n node        Kill processes on the specified node.
                (/bin/kill only)
Where:
 Valid signal names are:

SIGNULL   SIGHUP    SIGINT   SIGQUIT    SIGILL     SIGTRAP
SIGIOT    SIGABRT   SIGEMT   SIGFPE     SIGKILL    SIGBUS
SIGSEGV   SIGSYS    SIGPIPE  SIGALRM    SIGTERM    SIGUSR1
SIGUSR2   SIGCHLD   SIGPWR   SIGWINCH   SIGURG     SIGPOLL   SIGSTOP   SIGTSTP
SIGCONT   SIGVTALARM SIGTTIN    SIGTTOU

Note:
 kill is also available as a shell builtin.
```

Здесь нас ожидает сюрприз, который мы выделили в показанном фрагменте жирным шрифтом. И говорит эта строка о том, что в системе QNX сигнал может посылаться процессу, работающему на любом узле сети QNET. И это совершенно естественно, если вспомнить промелькнувшее выше замечание из технической документации, что сигнал в QNX – это пульс, то есть один из видов сообщений микроядра.

Таким образом, системная команда QNX `kill` (именно системная – `/bin/kill`, в отличие от встроенной формы `kill` командных интерпретаторов, которые строго следуют традициям UNIX, как и предупреждает выделенная нами строка) имеет возможность посылать сигналы любому процессу в сети. Тем не менее при рассмотрении прототипов вызовов `kill()` и `sigqueue()` мы не находим и следа параметра, предоставляющего возможность определить удаленный процесс. Тогда каким образом это делает команда `kill`? Совершенно верно: используя вызов native QNX API, который выглядит так (этот вызов, как и многие другие, имеет две формы, вторая из которых является безопасной в многопоточной среде):

```
#include <sys/neutrino.h>
int SignalKill( uint32_t nd, pid_t pid, int tid,
                int signo, int code, int value );
int SignalKill_r( uint32_t nd, pid_t pid, int tid,
                  int signo, int code, int value );
```

где nd – дескриптор сетевого узла QNET, на котором будут разыскиваться pid и tid. Для посылки сигнала локальному процессу (потоку) можно для nd указать 0, но лучше – определенную системой константу ND_LOCAL_NODE.

Примечание

Дескриптор узла в сети QNET – понятие относительное; он может быть получен, например, вызовом netmgr_strtond(). Но и здесь не все так просто:

- Дескриптор, соответствующий, скажем, узлу «host», полученный на узле «A», может иметь значение N, но дескриптор того же узла, полученный на узле «B», будет иметь уже значение M, то есть дескриптор узла – это «дескриптор сетевого узла X, как он видится с сетевого узла Y».
- Тот же дескриптор узла «host» может быть определен как имеющий значение N, но уже через несколько секунд он может «сменить» свое значение на M, то есть значения, полученные netmgr_strtond(), должны использоваться немедленно...

Эти и другие сложности относятся к особенностям программного использования QNET и требуют отдельного обстоятельного обсуждения. Однако они не являются предметом нашего текущего рассмотрения.

pid – PID процесса, которому направляется сигнал. pid может иметь и отрицательное значение, при этом положительное значение (-pid) идентифицирует группу процессов EGID, и сигнал будет отправлен всем процессам группы. При нулевом значении pid сигнал будет отправлен всем процессам группы процесса отправителя.

tid – 0 или TID потока, которому направляется сигнал. При указании tid сигнал будет доставляться только указанному потоку, а при tid = 0 – всем потокам процесса. Дальнейшая судьба сигнала в обоих случаях зависит от маскирования сигнала в потоке, как мы рассматривали ранее.

signo – номер сигнала (с ним неоднократно встречались выше).

code и value – код и значение, ассоциированные с сигналом (их мы тоже встречали при рассмотрении модели сигналов реального времени).

Как и обычно, внешнее различие (для программиста) основной формы SignalKill() и формы, безопасной в многопоточной среде, SignalKill_r() состоит в том, что:

- SignalKill() возвращает –1 в случае ошибки, а код ошибки заносится в errno; любой другой возврат является индикатором успешного выполнения;

- `SignalKill_r()` возвращает `EOK` в случае успеха, а в случае ошибки возвращается отрицательный код ошибки (тот же, который основная форма заносит в `errno`, но со знаком минус).

Возможны следующие коды ошибок, возвращаемые этими вызовами:

`EINVAL` – недопустимое числовое значение `signo`;

`ESRCH` – несуществующий адресат (`pid` или `tid`);

`EPERM` – процесс не имеет достаточных прав для посылки сигнала;

`EAGAIN` – недостаточно ресурсов ядра для выполнения запроса.

Для того чтобы получить работающий пример использования этой возможности, возьмите любой из приводившихся выше примеров, разнесите процессы по сетевым узлам и определите «целеуказание» в процессе-отправителе.

Простейшим примером и демонстрацией удаленной реакции в сети может быть следующая последовательность действий:

- Производим запуск задачи на **удаленном** узле, например:

```
# on -f <host> mqc
```

- После чего, выполнив ряд операций в запущенной программе, прекращаем ее работу по [Ctrl + C] с локального терминала.

Интересно оценить далеко идущие последствия этого «маленького» расширения стандартной POSIX-схемы работы с сигналами:

- На технике «сетевых сигналов» может быть построена целая система уведомлений сетевых составляющих компонент единой программной прикладной системы.
- Именно «уведомлений» (но не синхронизации с наследованием приоритетов, влияющей на общую систему диспетчеризации составляющих частей и т. п.): посылка сигнала является неблокирующей операцией (не требует ответа), а прием сигнала не сопровождается наследованием (или любым изменением) приоритетов.
- Такое «сигнальное» взаимодействие, записанное в формальной POSIX-семантике (но, по сути, осуществляющее механизмы, далеко выходящие за POSIX), может оказаться гораздо проще в записи и понимании, чем при использовании низкоуровневых механизмов обмена сообщениями (пульсами).

4

Примитивы синхронизации

ОС QNX Neutrino предоставляет широкий набор элементов синхронизации выполнения потокоющихся как в рамках одного процесса, так и разных. Это практически полный спектр примитивов, описываемых как базовым стандартом POSIX, так и всеми его расширениями реального времени. Тем не менее при работе со всеми этими примитивами не покидает ощущение, что некоторые из них являются органичными для самой ОС (мьютекс, условная переменная), в то время как другие – достаточно громоздкая надстройка над базовыми механизмами, реализуемая, главным образом, в угоду POSIX.

Примечание

К сожалению, и техническая документация QNX [8], и фундаментальная книга Р. Кертена [1] написаны по одной схеме:[1] все, что касается примитивов синхронизации, введенных более поздними расширениями POSIX (барьеры, жесткая блокировка (sleepon), спинлок, блокировки чтения-записи), описывается детально и сопровождается обстоятельными примерами кода, а вот базовые понятия, такие как `pthread_mutex_t`, `sem_t` (да и `pthread_cond_t`, по существу), описаны лишь качественно, «на пальцах», в иллюстративных рассказах об алгоритме пользования ванной комнатой и кухней (термин bathroom встречается намного чаще, чем `pthread_mutex_t`). Мы попытаемся по возможности компенсировать этот перекос.

Хотелось бы обратить внимание на интересный факт. В POSIX-варианте API QNX представлен большой набор разнообразных средств синхронизации: мьютексы, условные переменные, семафоры, барьеры, блокировки чтения/записи, ждущие блокировки, спинлоки. Однако в родном native API QNX из всего этого многообразия мы видим всего три элемента синхронизации: мьютекс, семафор и условная перемен-

[1] И здесь вопрос не в плагиате. Известно, что Р. Кертен долгое время сотрудничал с QSSL именно по части написания документации, поэтому она во многом следует его манере изложения, хотя более поздняя книга Р. Кертена [1] изложена заметно доходчивее.

ная. И это при том, что условная переменная не является самостоятельным средством синхронизации и применяется как расширение функциональных возможностей мьютекса!

Это означает, что все многообразие средств синхронизации, предоставляемое в POSIX API QNX, строится исключительно с применением этих минимальных средств синхронизации. Действительно, анализ заголовочных файлов системы показывает, что все дополнительные средства синхронизации, появляющиеся в POSIX API, строятся с использованием только мьютекса и условной переменной. Можно сказать, что пара мьютекс–условная переменная с одной стороны и семафор с другой являются независимыми базисами, каждый из которых позволяет построить практически любой, сколь угодно специфический элемент синхронизации, удовлетворяющий потребностям, которые могут возникнуть при построении вашей системы. Мы проиллюстрируем эту мысль позже, при рассмотрении особенностей применения мьютексов.

Независимо от того, какой набор элементов синхронизации окажется предпочтительным для разрабатываемой вами системы, важным является тот факт, что в случае, если предлагаемые системой средства синхронизации по каким-то причинам вас не устраивают, то ничто не мешает разработать собственные средства синхронизации, используя тот или иной базис или даже их комбинацию. И эффективность полученного нового средства синхронизации будет зависеть только от вас.

Мы постараемся проиллюстрировать эту идею примерами использования базовых средств синхронизации в качестве «конструктора» при построении более сложных. Три базовых элемента синхронизации ОС QNX – мьютекс, условная переменная и семафор – реализуются на уровне микроядра системы и создаются вызовом системной функции:

```
SyncTypeCreate( unsigned type, sync_t* sync, const struct _sync_attr_t* attr )
```

Здесь `type` может принимать значения:

`_NTO_SYNC_MUTEX_FREE` – для создания мьютекса;

`_NTO_SYNC_SEM` – для создания семафора;

`_NTO_SYNC_COND` – для создания условной переменной.

Стоит обратить внимание на два важных факта. Во-первых, типы мьютекса (`pthread_mutex_t`), условной переменной (`pthread_cond_t`) и семафора (`sem_t`) являются псевдонимами типа `sync_t`. А во-вторых, типы атрибутов этих объектов также являются псевдонимами одного-единственного типа – `sync_attr_t`, содержащего поле `protocol`, значение которого определяет, каким способом ОС будет варьировать приоритеты во избежание их инверсии.

В соответствии с документацией QNX 6.2.1 это поле присутствует у параметров всех трех базовых объектов синхронизации, но доступно оно только для переопределения мьютексов. По умолчанию (например,

когда вместо `attr` передается `NULL`) поле `protocol` принимает значение `PTHREAD_PRIO_INHERIT`. Это означает, что ОС будет использовать протокол наследования приоритетов для предотвращения инверсии.

Таким образом, все примитивы синхронизации QNX в теории могут учитывать эффект инверсии приоритетов. Однако на практике важны следующие два вопроса: возможно ли в принципе возникновение ситуации инверсии приоритетов для данного типа примитивов, и если да, то актуально ли для данного типа примитивов препятствовать возникновению инверсии (в качестве иллюстрации этого утверждения можно рассмотреть примитив типа барьер).

Тесты [4] показывают, что только для мьютексов наследованием приоритетов (или альтернативными протоколами) однозначно предотвращается возникновение инверсии приоритетов. По-видимому, в первую очередь это связано с тем, что сама по себе инверсия может возникнуть именно в тех случаях, когда из всех элементов синхронизации необходимо использовать именно мьютекс или что-либо из его наследников (например, блокировки чтения-записи).

Семафор (счетный)

Семафор является весьма специфическим (в сравнении с прочими) для ОС QNX средством синхронизации, и хотя функции работы с ним также определяются стандартом POSIX, даже семантика этих функций отличается от всех прочих объектов синхронизации.

Примечание

Функции работы с семафором определяются стандартом POSIX 1003.1 (1993) расширения реального времени, а не стандартом POSIX 1003b (1995), которым определены `pthread_*` и все другие примитивы синхронизации; соответственно функции манипулирования семафорами не начинаются с префикса `pthread_*`.

В классической работе Дейкстры [10] семафор определяется как объект, над которым можно провести две атомарные операции: инкремент и декремент внутреннего счетчика – при условии, что внутренний счетчик не может принимать значение меньше нуля. Если некий поток пытается уменьшить на единицу значение внутреннего счетчика семафора, значение которого уже равно нулю, то этот поток блокируется до тех пор, пока внутренний счетчик семафора не примет значение, равное 1 или больше (посредством воздействия на него других потоков). Разблокированный поток сможет осуществить декремент нового значения.

Принято рассматривать семафоры двух видов: *бинарные*, счетчик которых может принимать только значения 0 либо 1, и *счетные*, или простые, счетчик которых может принимать большие положительные

значения. В QNX 6.2.1 максимальное значение счетчика определяется переменной SEM_VALUE_MAX, значение которой равно 32 767.

Отметим важный момент: семафор является наиболее простым и соответственно наиболее **универсальным** средством синхронизации. И классические решения задач раздельного использования ресурсов, предложенные в теории первоначально Э. Дейкстрой, базируются именно на понятии семафора. Однако в случае систем реального времени применение семафоров для разделения доступа к ресурсу влечет потенциальную опасность возникновения инверсии приоритетов [4], избежать которой никак нельзя.

Причина этого в том, что семафор **по определению** не может знать своего владельца (захватившего его), поскольку у счетного семафора в принципе не может быть владельца, а бинарный семафор, который отличает своего владельца (поток, заблокировавший в данный момент другие потоки на обращении к семафору), уже перестает быть собственно семафором и называется мьютексом, или эксклюзивной блокировкой. По-видимому, именно этим обстоятельством вызван тот факт, что все объекты синхронизации QNX, не реализованные на уровне native API, но предоставляемые на уровне API POSIX, строятся без использования семафоров.

Принципиально схема работы семафора позволяет использовать его для решения максимально широкого круга задач. Приведем только несколько схематичных примеров:

- **Ожидание условия (уведомление).** Часто возникает необходимость остановить (заблокировать) поток до тех пор, пока не наступит некое событие (выполнится условие). Разблокировать остановленный поток в таком случае может только другой, активный к этому времени поток. Предварительно инициализируем семафор нулевым значением:

Поток A	Поток B
`while (!expression)`	`expression = true;`
`sem_wait(&sem);`	`sem_post(&sem);`

  В этом случае поток A ожидает выполнения некоего условия (операция `sem_wait()`), а поток B уведомляет (операция `sem_post()`) о выполнении условия **любые** ожидающие этого условия потоки.

Примечание

В принципе конструкция `while()` здесь не обязательна, можно обойтись и простым `if()`, но проверка выполнения условия и после разблокирования потока будет более строгой формой в многопоточной системе, особенно если выполнения условия ожидают более одного потока.

- **Взаимное исключение.** С помощью семафора можно решить и другую классическую проблему синхронизации – безопасное совместное исполнение кода. Эта проблема возникает, когда из нескольких потоков необходимо провести модификацию общих переменных или обратиться к одному системному ресурсу. В таком случае необходимо установить четкий порядок обращений, не допускающий одновременности исполнения соответствующего участка кода (взаимное исключение). Здесь семафор инициализируется единицей:

Поток A	Поток B
`sem_wait(&sem);`	`sem_wait(&sem);`
`/* код, который нужно защитить от совместного использования. */`	`/* код, который нужно защитить от совместного использования. */`
`sem_post(&sem);`	`sem_post(&sem);`

  В данном случае последовательность доступа потоков к защищаемому участку кода (такой участок кода еще называют **критической секцией**) не играет никакой роли; здесь важно не допустить, чтобы два потока исполняли этот код одновременно.

  Поскольку семафор инициализируется единицей, первый вызов `sem_wait()` не приводит к блокированию потока, а лишь понижает счетчик семафора до нуля. В таком положении семафора любой поток, повторно вызвавший `sem_wait()` на входе в критическую секцию, будет блокирован до тех пор, пока первый вошедший поток не покинет этот участок кода и не вызовет `sem_post()`. После этого **один** из потоков, ожидающих увеличения счетчика семафора, получит управление (это будет один поток, и нельзя заранее сказать, какой именно) и выполнит операцию декремента над счетчиком, вновь заблокировав вход в критическую секцию. Таким образом можно гарантировать строго последовательное выполнение такого кода.

Однако при использовании семафоров для решения задачи взаимного исключения потоков разного приоритета может возникнуть серьезная проблема, известная как инверсия приоритетов. Вопрос инверсии рассматривался в нашей работе [4], и мы не будем здесь подробно останавливаться на этом. В простейшем случае проблема заключается в том, что если в системе присутствуют несколько (3 или более) потоков разного приоритета (высокого, среднего и низкого), использующих общий ресурс, то возможно возникновение ситуации, когда высокоприоритетный поток будет блокирован в ожидании ресурса, ранее захваченного потоком самого низкого приоритета, который в свою очередь вытеснен потоком среднего приоритета, причем такой неразрешимый «клинч» может продолжаться неограниченно долго.

Каким же образом можно предотвратить подобную ситуацию? Для этого необходимо проводить манипуляции с приоритетом потока, входящего в критическую секцию. Однако после выполнения потоком функции `sem_wait()` (при этом счетчик семафора уменьшается до нуля)

и перехода к выполнению кода критической секции уже никак нельзя определить, какой поток заблокировал семафор, и нельзя ничего сделать с его приоритетом.

Для того чтобы система имела возможность влиять на поведение потоков с точки зрения профилактики инверсии приоритетов и взаимных блокировок («мертвых объятий» – deadlock) потоков или других подобных проблем, вызванных взаимным влиянием потоков, необходимо, чтобы объект синхронизации явным образом хранил информацию о том потоке, который его захватил (то есть знал своего хозяина). Семафор такой информации не хранит, и это необходимо помнить при проектировании системы с его использованием. Применение семафора оптимально для случаев слабо связанных и в идеале равноприоритетных потоков. Собственно, для этих случаев семафор как средство синхронизации и разрабатывался [10].

- Частным случаем задачи взаимного исключения является классическая **задача последовательного доступа по типу производитель/потребитель**. Такая ситуация возникает, когда один поток передает другому данные через общую переменную. Пока производитель не «положит» новые данные в эту переменную, потребитель должен простаивать в ожидании.

  Приведем классическое решение этой задачи. В этом случае нам понадобится два семафора (по одному на каждый поток), которые должны инициализироваться следующим образом: тот, который защищает чтение (потребление), инициализируется нулем (блокирует читающий поток), а тот, который защищает запись (производство), – единицей (открывает доступ «писателю» к общему ресурсу).

Поток A	**Поток B**
`sem_wait(&sem_A);`	`sem_wait(&sem_B);`
`/* критическая секция */`	`/* критическая секция */`
`sem_post(&sem_B);`	`sem_post(&sem_A);`

  Положим, поток A является производителем данных, необходимых для работы потока B. Соответственно семафор `sem_A` инициализирован 1, а семафор `sem_B` инициализирован 0. Когда поток B попытается обратиться к общей переменной за данными для работы, он будет блокирован в ожидании результатов работы потока A. Поток A, подготовив необходимые данные, войдет в критическую секцию (поскольку его семафор разблокирован), установит новые данные и, покидая критическую секцию, разблокирует семафор потока B. После этого поток B будет разблокирован и сможет получить новые данные. Обратите внимание, что если данные производятся в цикле (а это обычная ситуация), то поток A не сможет повторно получить доступ к общей переменной до тех пор, пока поток B не закончит чтение этой переменной и не покинет критическую секцию.

Примечание

Описанная схема не является универсальной и хорошо работает для двух потоков. Однако возможна ситуация, когда существует множество потоков потребителей и производителей. В ОС QNX существует специальный примитив синхронизации, называемый «блокировкой чтения/записи» и предназначенный для синхронизации доступа в такой ситуации. Этот примитив предоставляет множественный доступ к критической секции кода со стороны «читателей», поскольку они не изменяют содержимого общих данных, и устанавливает эксклюзивный доступ для «писателей». Этот примитив будет подробнее рассмотрен позже.

- До сих пор мы фактически рассматривали работу семафора в бинарном режиме. Для большого количества задач именно такой режим семафора и является основным. Однако **возможности счетного семафора, позволяющего контролировать количество обращений**, допускают его применение в весьма специфических задачах, где другие средства синхронизации требуют от программиста дополнительной работы.

  Проиллюстрируем работу счетного семафора. Предположим, нам надо синхронизировать работу двух потоков так, чтобы один из них всегда шел «вслед» за другим, то есть выполнял некие циклические операции только после их выполнения первым потоком. Для реализации этой схемы нам понадобится семафор, инициализированный нулевым значением.

Поток A

```
while (true)
{
     /* работа A; */
     sem_post(&sem);
}
```

Поток B

```
while (true)
{
     /* sem_wait(&sem); */
     работа B;
}
```

  Поток A в каждом цикле увеличивает счетчик семафора на 1, а поток B в свою очередь стремится «выбрать слабину» и в каждом цикле уменьшает этот счетчик на единицу до тех пор, пока он не достигнет нуля. Вне зависимости от приоритетов, дисциплины диспетчеризации или любых источников блокировки потоков поток B будет «следовать» за потоком A и выполнять не больше циклов, чем его «ведущий» поток.

Всеми этими примерами мы хотели показать главную особенность семафора: любой поток может менять его состояние (внутренний счетчик), тем самым делая это элемент синхронизации идеальным средством управления порядком выполнения потоков. Семафор является идеальным инструментом для построения собственных средств планирования (диспетчеризации) выполнения потоков вашей системы. Стандарт POSIX предусматривает наличие специальных «именованных» семафоров, к которым может обращаться любой поток, принад-

лежащий любому процессу, выполняющемуся в системе, а в ОС QNX к семафорам может обращаться и любой поток любого процесса во всей сети QNX. Таким образом, семафор позволяет осуществлять планирование выполнения задач даже для вашей распределенной сетевой системы.

Ниже мы приводим краткое описание функций работы как с неименованными, так и с именованными семафорами, реализованными в ОС QNX. Функции работы с семафорами объявлены в заголовочном файле `<semaphore.h>`. Если для работы какой-либо функции необходимы дополнительные заголовочные файлы, они будут указываться явно.

Операции над семафорами

Создание семафора

QNX поддерживает два типа семафоров – неименованные и именованные. Разница между ними заключается в том, что к именованному семафору можно обратиться из любого процесса в системе (или даже по сети QNET с другого сетевого хоста), поскольку такой семафор имеет ассоциированное с ним имя в файловой системе QNX. Необходимо помнить, что именованные семафоры, при прочих равных условиях, медленнее и требуют для своей работы запущенного в системе менеджера очередей сообщений POSIX (mqueue).

Для каждого типа семафоров существует своя группа функций, применение которых не должно смешиваться.

`sem_init()`и `sem_destroy()` – создание и разрушение неименованного семафора. При создании указывается параметр доступа из других потоков и начальное значение счетчика семафора. С неинициализированным семафором никаких операций проводить нельзя (это общее правило справедливо и для всех иных примитивов синхронизации). После разрушения семафора его необходимо повторно инициализировать для использования.

Обе функции возвращают 0 в случае успеха и −1 в случае ошибки. Код ошибки записывается в переменной `errno`. В частности, функция `sem_init()` может сигнализировать о следующих ошибках выполнения:

`EAGAIN` – в данный момент нет ресурсов для инициализации семафора;

`EINVAL` – начальное значение счетчика превышает `SEM_VALUE_MAX`;

`EPERM` – у процесса недостаточно привилегий для инициализации семафора;

`ENOSPC` – ресурсы, необходимые для инициализации, исчерпаны;

`ENOSYS` – функция `sem_init()` не поддерживается реализацией системы.

При вызове функции `sem_destroy()` может регистрироваться только одна ошибка:

EINVAL – неправильный описатель семафора.

sem_open() и sem_close() – открытие и закрытие именованного семафора (если отсутствует ранее созданный семафор с таким именем, то его создание). В вопросе подключения и отключения работа с именованным семафором аналогична работе с обычным файлом. Также для именованных семафоров существует операция sem_unlink(), аналогичная операции unlink() для обычного файла. В функцию sem_open() передается имя семафора, параметры открытия семафора и дополнительные параметры в случае создания семафора.

Операции блокировки

Для семафора определены три модификации операции блокировки:

```
int sem_wait( sem_t* sem )
int sem_trywait( sem_t* sem )
#include <time.h>
int sem_timedwait( sem_t* sem, const struct timespec * abs_timeout )
```

Все эти функции опираются на функцию (native QNX API):

```
int SyncSemWait( sync_t* sync, int try );
```

Функция простого ожидания sem_wait() пытается выполнить декремент счетчика семафора. Если исходное значение счетчика было больше или равно 1, то функция после выполнения операции декремента возвращает управление в вызвавший код. Если же значение внутреннего счетчика семафора равнялось 0, выполнение вызвавшего функцию потока блокируется до тех пор, пока какой-либо другой поток не увеличит значение счетчика семафора. После того как значение счетчика будет увеличено, блокированный поток получает управление (в порядке очереди), выполняет (завершает) декремент счетчика семафора и возвращает управление. Блокированное состояние потока может быть также прервано получением направленного ему сигнала (см. главу 3).

Функция ожидания с проверкой семафора, sem_trywait(), до выполнения операции над семафором проверяет значение счетчика. Если это значение больше нуля, функция выполняет декремент счетчика, а если значение равно нулю – возвращает управление потоку, не блокируясь на ожидании доступности семафора (но захват семафора в этом случае не происходит, о чем извещает код возврата).

Функция ожидания с тайм-аутом, sem_timedwait(), ожидает возможности уменьшить на 1 счетчик семафора (блокируется на ожидании) до момента времени abs_timeout. Это ожидание реализовано с помощью функции TimerTimeout() (native QNX API) с параметром SIGEV_UNBLOCK. Благодаря этой функции может быть прерван ряд состояний блокировки потоков (в том числе блокировки на семафоре, мьютексе и условной переменной).

Операции освобождения

```
int sem_post(sem_t* sem)
```

Эта операция увеличивает на единицу (инкремент) внутренний счетчик семафора, и в этом она диаметрально противоположна операции блокировки. Если перед этим значение счетчика семафора было равно 0, то один из потоков, ожидающих разблокирования семафора, переходит в состояние готовности. Из списка ожидающих потоков система выбирает самый приоритетный, а если таких несколько, то поток, дольше всех ждавший из очереди наиболее приоритетных потоков.

Эта функция имеет свой оригинал в native API QNX:

```
int SyncSemPost( sync_t* sync );
```

Фактически разница между POSIX и QNX API в вариантах этой функции состоит в регистрируемых ею ошибках.

Функция `sem_post()` сообщает о следующих ошибках:

`EINVAL` – неверный дескриптор семафора `sem`;

`ENOSYS` – функция `sem_post()` не поддерживается системой.

В отличие от `sem_post()`, функция `SyncSemPost()` может указывать на ошибки:

`EAGAIN` – недостаточно памяти для создания внутреннего объекта синхронизации;

`EFAULT` – неверный указатель на семафор `sync`;

`EINTR` – выполнение функции прервано сигналом;

`EINVAL` – аргумент `sync` не указывает на инициированный семафор.

Как видим, функция QNX API несколько разнообразнее в плане контроля передаваемых аргументов и результата выполнения функции.

Получение статуса семафора

```
int sem_getvalue( sem_t* sem, int* value );
```

Эта функция используется преимущественно для отладки операций над семафорами. По адресу, указанному в `value`, устанавливается текущее значение счетчика семафора. Поскольку значение счетчика семафора может измениться в любой момент, то значение, которое возвращает эта функция, имеет смысл только непосредственно в точке ее вызова.

Возможные ошибки:

`EINVAL` – неправильный объект семафор `sem`.

Использование семафора

Как уже говорилось выше, семафор является крайне гибким и эффективным средством синхронизации, особенно удобным для построения

собственных средств планирования выполнения потоков. В этом смысле семафор представляет ценность не только как самостоятельное средство синхронизации выполнения потоков, но и как материал для построения специфических средств планирования и синхронизации для конкретных задач. Мы уже говорили, что семафор образует самодостаточный базис, позволяющий строить гораздо более сложные средства синхронизации без привлечения других средств синхронизации. В принципе это так, но нет ничего плохого и в смешанном использовании как семафоров, так и мьютексов для построения собственных средств синхронизации.

Проиллюстрируем все вышесказанное на двух примерах. Сначала мы построим «очередь сообщений», предназначенную для трансляции сообщений графической системы к «медленному» обработчику реакций. Это одно из решений весьма распространенной задачи о предотвращении «зависания» пользовательского интерфейса на период выполнения медленного обработчика. Для решения этой задачи обработчик события оконной системы (например, нажатия кнопки или выбора пункта меню) и функция, которая непосредственно производит требуемые действия (предусмотренные по наступлению указанного события – нажатия кнопки), должны располагаться в разных потоках.

Было бы удобно, если бы при поступлении новых данных от графической системы поток обработки автоматически (неявно) разблокировался и немедленно приступал к обработке, а в периоды отсутствия таких данных – простаивал в блокированном состоянии. Для реализации такой схемы мы построили **синхронизирующую очередь сообщений**, которая использует семафор для уведомления потока обработки о наличии новых данных. В принципе указанная задача сводится к уже упоминавшемуся ранее классу задач о синхронизации производителя и потребителя данных.

```
class event {
/* класс синхронизирующего события, доставляющего
   уведомление о добавлении нового элемента в буфер */
public:
   event() { sem_init( &_block, 0, 0 ); };
   ~event() { sem_destroy( &_block ); };
   void wait() { sem_wait( &_block ); };
   void reset() { sem_post( &_block ); };
private:
   sem_t _block;
};

/* шаблонный класс очереди данных */
template <class T> class CDataQueue {
public:
   CmessageQueue() {};
   ~CmessageQueue() {};
   void push( T _new_data ) {
```

```
        _data_queue.push(_new_data );
        data_event.reset();
    };
    T pop() {
        data_event.wait();
        T res = _data_queue.front();
        _data_queue.pop();
        return res;
    };
private:
    std::queue<T> _data_queue;
    event data_event;
};
```

Принцип работы `CDataQueue` заключается в том, что для хранения вновь поступающих данных используется очередь, что делает практически независимыми потоки производителя и потребителя. Независимыми во всех случаях, кроме пустой очереди. Потребитель должен быть блокирован до тех пор, пока нет данных от производителя. Как только производитель внесет данные в очередь, поток потребителя разблокируется и считает эти данные. Тонкость заключается в том, что поток потребителя блокируется сам при вызове функции `pop()`, а разблокируется из потока производителя при вызове им функции `push()`.

Как видите, в построении специфических средств синхронизации нет ничего сложного, вопреки часто встречающемуся утверждению, что создание средств синхронизации со специфическим поведением неадекватно трудоемко, а простейший код позволяет адаптировать возможности тривиального семафора под конкретную задачу.

А теперь хотелось бы обратить ваше внимание на тот факт, что «безопасным» использованием описанной схемы будет только вариант двух потоков – одного производителя и одного потребителя. Если несколько (более двух) потоков одновременно попробуют выполнить функции `pop()` или `push()`, начнется путаница, и чем это закончится, сказать трудно. По своей логике код обеих функций в многопоточной системе требует эксклюзивного исполнения. Чтобы обеспечить исключительный доступ к этим участкам кода, мы могли бы использовать дополнительный семафор, но есть другой вариант – специальное средство синхронизации, разработанное именно для решения задачи взаимного исключения, – мьютекс.

Мьютекс

Мьютекс (от mutual exclusion – взаимное исключение) – это один из базовых примитивов синхронизации QNX Neutrino. Этот элемент реализуется на уровне микроядра системы и имеет широкий набор атрибутов и настроек. Назначение мьютекса – защита участка кода от совместного выполнения несколькими потоками. Такой участок кода

иногда называют **критической секцией**, и обычно он является областью модификации общих переменных или обращения к разделяемому ресурсу.

Принцип работы мьютекса заключается в следующем: при обращении потока к функции блокировки (захвата) `pthread_mutex_lock()` проверяется, захвачен ли уже мьютекс, и если да, то вызвавший поток блокируется до освобождения критической секции. Если же нет, то объект мьютекс запоминает, какой поток его захватил (то есть владельца) и устанавливает признак, что он захвачен.

Когда действия, которые нельзя производить совместно, закончены, поток должен вызвать функцию разблокировки (освобождения) `pthread_mutex_unlock()`, которая проверяет, действительно ли вызвавший ее поток является тем, который в данный момент владеет мьютексом, и если да, то она разблокирует мьютекс, после чего ОС проводит редиспетчеризацию потоков. Если есть потоки, ожидающие освобождения мьютекса, то один из таких потоков, имеющий наивысший приоритет, переводится из состояния блокирования в состояние готовности и захватывает мьютекс.

В QNX Neutrino 6.2.1 мьютекс имеет наибольшие возможности по тонкой настройке своих параметров среди всех иных элементов синхронизации. В связи с этим поведение мьютекса очень сильно зависит от того, какие значения вы присвоите его атрибутам.

Как видите, главное отличие мьютекса от семафора заключается в том, что он хранит информацию о потоке, исполняющем код критической секции. Отсюда и важнейшие свойства мьютекса. Мьютекс нельзя разблокировать из другого потока. Если поток захватил мьютекс, то только он может его «отпустить». Используя информацию о владельце (tid потока), система может изменять в нужное время приоритет владельца для разрешения проблемы инверсии приоритетов. Наконец, зная идентификатор потока, мьютекс может выделить ситуацию, когда поток, уже захвативший мьютекс, пытается захватить его повторно (одна из разновидностей deadlock – мертвой блокировки, когда не существует ни одного потока, способного отпустить мьютекс и разблокировать потоки, ожидающие освобождения этого мьютекса).

В ОС QNX возможен вариант работы мьютекса, не предусмотренный стандартом POSIX, – **рекурсивный мьютекс**. В этом режиме поток, владеющий мьютексом при повторном его захвате, не блокируется. Мьютекс только отмечает в своем внутреннем счетчике, сколько раз он был захвачен, и разблокируется только после равного количества освобождений (естественно, тем же потоком).

Все объявления относительно мьютексов находятся в заголовочном файле `<pthread.h>`, и программный код, их использующий, должен включать директиву:

```
#include <pthread.h>
```

Параметры мьютекса

Параметры мьютекса хранятся в структуре pthread_mutexattr_t, которая определена типом sync_attr_t. Эта структура должна быть создана и определена до инициализации мьютекса, после чего может быть переопределена и использована для других объектов типа мьютекс.

Инициализация параметров

```
int pthread_mutexattr_init( const pthread_mutexattr_t* attr );
```

Функция инициализирует структуру атрибутов мьютекса, на которую указывает параметр attr. Тип данных pthread_mutexattr_t определен в файле <pthread.h> (производный от типа sync_attr_t, который в свою очередь определен в файле <target_nto.h>) следующим образом:

```
struct _sync_attr_t {
   int  protocol;
   int  flags;
   int  prioceiling;
   int  clockid;    /* только для condvar */
   int  reserved[ 4 ];
};
```

После инициализации значения по умолчанию могут быть прочитаны или изменены группой функций, приведенной ниже.

Установка граничного приоритета

```
int pthread_mutexattr_setprioceiling (
                          pthread_mutexattr_t* attr, int prioceiling );
int pthread_mutexattr_getprioceiling (
                       const pthread_mutexattr_t* attr, int* prioceiling );
```

Эти функции записывают/читают значение приоритета, которое будет присваиваться потоку, захватившему мьютекс, если поле protocol структуры pthread_mutexattr_t установлено в значение PTHREAD_PRIO_PROTECT.

Этот параметр используется для реализации протокола граничного приоритета (Priority Ceiling Protocol), предлагающего альтернативный вариант защиты от инверсии приоритетов там, где наследование приоритетов может быть неэффективно или нежелательно.

Примечание

Как известно, классический случай инверсии приоритетов может возникать, когда более двух потоков разного приоритета конкурируют и разделяют один общий ресурс. В этой ситуации возможно такое стечение обстоятельств, когда низкоприоритетный поток захватывает ресурс, но позже вытесняется потоком среднего приоритета, а поток с высоким приоритетом блокируется в ожидании освобождения ресурса, захваченного низкоприоритетным потоком. Эта ситуация детально описана в соответствующей главе [4], где приводится сравнительный анализ пове-

дения различных ОС в этой ситуации. Классическим решением этой проблемы является протокол наследования приоритетов, когда ОС распознает ситуацию подобного блокирования и автоматически повышает приоритет вытесненного потока (низкого приоритета) до уровня наивысшего приоритета из числа потоков, ожидающих освобождения ресурса. В результате поток с низким приоритетом не вытесняется, как надлежало бы в соответствии с его изначальным приоритетом, а быстро проходит критическую секцию (освобождая ее), после чего его приоритет возвращается на исходный уровень.

В ряде случаев наследование приоритетов может оказаться не самым оптимальным решением. Примером здесь может служить ситуация, когда один высокоприоритетный поток разделяет много ресурсов с низкоприоритетными потоками – по одному ресурсу на каждый поток. В такой ситуации может возникнуть положение, когда много низкоприоритетных потоков (вытесненных) выстроятся перед высокоприоритетным. Но тогда длинный ряд последовательных операций вытеснения («поштучно») и наследования приоритетов блокирующих потоков может привести к тому, что не хватит времени до окончания критического срока выполнения потока высокого приоритета, пока ОС будет анализировать ситуацию и последовательно проводить протокол наследования приоритетов.

Именно для таких ситуаций и предназначен протокол граничного приоритета. В соответствии с этим протоколом примитив синхронизации (в нашем рассмотрении это мьютекс) наделяется собственным фиксированным приоритетом, а приоритет любого потока, захватившего этот мьютекс, поднимается до предустановленного граничного уровня приоритета мьютекса.

Определение протокола защиты от инверсии приоритетов

```
int pthread_mutexattr_setprotocol(
                              pthread_mutexattr_t* attr, int protocol );
int pthread_mutexattr_getprotocol(
                              pthread_mutexattr_t* attr, int* protocol );
```

Эти функции устанавливают/считывают протокол, который реализуется мьютексом для защиты от инверсии приоритетов. Переменная `protocol` может принимать следующие значения:

`PTHREAD_PRIO_INHERIT` (значение по умолчанию) – определяет, что для воспрепятствования возникновению инверсии приоритетов будет использоваться протокол наследования приоритетов.

`PTHREAD_PRIO_PROTECT` – любой поток, захвативший мьютекс и созданный с таким параметром, будет устанавливать фиксированный уровень приоритета в соответствии со значением поля `prioceiling`, возвращаемого функцией `pthread_mutexattr_getprioceiling()`. Таким образом, установка этого значения в качестве протокола мьютекса приводит

к реализации протокола граничного приоритета для защиты от инверсии приоритетов.

Внешний доступ

```
int pthread_mutexattr_setpshared(
          pthread_mutexattr_t* attr, int pshared );
int pthread_mutexattr_getpshared(
          const pthread_mutexattr_t* attr, int* pshared );
```

Эти функции устанавливают/считывают внутреннее поле атрибутной записи мьютекса, определяющее, возможен ли доступ к мьютексу из потоков, запущенных вне процесса, в котором был создан и инициализирован мьютекс. Параметр `pshared` может принимать следующие значения:

`PTHREAD_PROCESS_SHARED` – любой поток из любого процесса в системе, который может получить доступ к синхронизирующему объекту (для этого придется использовать какой-либо из методов IPC, возможно shared memory), может использовать его по назначению.

`PTHREAD_PROCESS_PRIVATE` (значение по умолчанию) – мьютекс может использоваться только потоками, порожденными в том же процессе, где был инициализирован мьютекс. В документации сказано: попытка захвата мьютекса с таким значением параметра доступа к потокам из «чужого» процесса приведет к неопределенному результату. На практике же функция захвата возвращает управление в любом случае, независимо от того, был ли уже захвачен мьютекс другим потоком или нет (как будто происходит нормальный захват). Состояние мьютекса при этом никак не меняется.

Разрешение рекурсивного захвата

```
int pthread_mutexattr_setrecursive(
                          pthread_mutexattr_t* attr, int recursive );
int pthread_mutexattr_getrecursive(
                         const pthread_mutexattr_t* attr, int* recursive );
```

Функции устанавливают/считывают в атрибутной записи мьютекса признак, определяющий, может ли поток, ранее захвативший мьютекс (его владелец), захватить его еще раз (естественно, что любой другой поток захватить такой мьютекс уже не может и он будет заблокирован). Режим реализован для возможности рекурсивного вызова процедур в потоке. Необходимо помнить, что при рекурсивном захвате мьютекс должен быть освобожден столько раз, сколько раз он был захвачен. Параметр `recursive` может принимать следующие значения:

`PTHREAD_RECURSIVE_ENABLE` – разрешает рекурсивный захват мьютекса;

`PTHREAD_RECURSIVE_DISABLE` (значение по умолчанию) – запрещает рекурсивный захват мьютекса. В результате при попытке захвата мьютекса

потоком, который им уже владеет, вызов pthread_mutex_lock() не приведет к захвату мьютекса и вернет значение EDEADLK.

Определение типа мьютекса

```
int pthread_mutexattr_settype(
                        pthread_mutexattr_t* attr, int type );
int pthread_mutexattr_gettype(
                        const pthread_mutexattr_t* attr, int* type );
```

В версиях QNX 6.2.1 и 6.3 предусматривается создание мьютексов следующих типов:

- PTHREAD_MUTEX_NORMAL – для этого типа не проводится контроль «мертвой блокировки» (deadlock) в ситуации, когда поток, захвативший мьютекс, пытается захватить его повторно. Поэтому при попытке повторного захвата такого мьютекса тем же потоком этот поток будет безусловно блокирован (то есть он попадает в «мертвую блокировку», а это во всех случаях аварийная ситуация в выполнении приложения); такой мьютекс уже некому разблокировать (мьютекс может разблокироваться только своим владельцем). Попытка освободить (unlock) мьютекс такого типа, захваченный другим потоком, или освободить незахваченный мютекс ни к чему не приводит, при этом не возвращается ошибка выполнения.
- PTHREAD_MUTEX_ERRORCHECK – включается контроль ошибок. В этом режиме регистрируются следующие ситуации:
  - попытка повторного захвата мьютекса тем же потоком;
  - попытка освобождения мьютекса, захваченного другим потоком;
  - освобождение свободного мьютекса.
- PTHREAD_MUTEX_RECURSIVE – мьютекс, допускающий рекурсивный захват. Поток, пытающийся захватить мьютекс, уже захваченный в этом потоке, сможет это сделать, при этом количество захватов будет учитываться при освобождении мьютекса. Другой поток сможет захватить такой мьютекс только тогда, когда он будет освобожден столько же раз, сколько был захвачен. Если поток пытается освободить мьютекс, захваченный другим потоком, или свободный мьютекс, то будет возвращено сообщение об ошибке (регистрируются ошибки, предусмотренные предыдущим типом, за исключением повторного захвата, который является для рекурсивного мьютекса штатным действием).

Примечание

Обратите внимание, что разрешение рекурсивного захвата мьютекса необходимо проводить установкой двух параметров (type и recursive).

- PTHREAD_MUTEX_DEFAULT (значение по умолчанию) – попытка рекурсивного захвата мьютекса, освобождения мьютекса, захваченного дру-

гим потоком, или освобождения уже свободного мьютекса ни к чему не приводит и не возвращает ошибку выполнения.

Освобождение параметров

```
int pthread_mutexattr_destroy( pthread_mutexattr_t* attr );
```

Вызов разрушает ранее применявшийся объект – атрибутную запись мьютекса, после чего она уже не может более использоваться для инициализации мьютекса без предварительного выполнения вызова `pthread_mutexattr_init()`.

На этом обсуждение атрибутов заканчивается, и мы переходим непосредственно к функциям работы с мьютексом.

Операции над мьютексом

Инициализация мьютекса

```
int pthread_mutex_init( pthread_mutex_t* mutex,
                        const pthread_mutexattr_t* attr );
```

Структура данных `pthread_mutex_t` определена в файле `<pthread.h>` (производный тип от типа `sync_t`, который в свою очередь определен в файле `<target_nto.h>`) и имеет следующий вид:

```
struct _sync_t {
   /* Счетчик для рекурсивного мьютекса или семафора */
   int      count;
   /* TID потока - имеет смысл и применяется только для мьютексов */
   unsigned owner;
}
```

Функция `pthread_mutex_init()` инициализирует переданный объект мьютекс в соответствии со значением переданных атрибутов. Если вместо `attr` передать `NULL`, то мьютекс будет создан в соответствии со значениями атрибутов по умолчанию. В native QNX API эта функция реализуется вызовом `SyncTypeCreate()`. `SyncTypeCreate()` – единая функция для создания всех базовых объектов синхронизации QNX Neutrino.

Вместо прямого вызова функции `pthread_mutex_init()` для начальной инициализации статических мьютексов (глобальных на уровне файла кода или пространства имен namespase либо явно описанных с квалификатором `static`) можно воспользоваться двумя макросами `PTHREAD_MUTEX_INITIALIZER` и `PTHREAD_RMUTEX_INITIALIZER`:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_RMUTEX_INITIALIZER;
```

Первый из них создает мьютекс в соответствии со значениями атрибутов по умолчанию, а второй – мьютекс с разрешенным рекурсивным захватом.

Операции с граничным приоритетом

Большинство параметров мьютекса не могут быть изменены после его создания. Но не все. В процессе работы с мьютексом может быть изменено значение приоритета, которое система использует для реализации протокола граничного приоритета с целью предотвращения инверсии приоритетов:

```
int pthread_mutex_setprioceiling( pthread_mutex_t* mutex,
                                  int prioceiling, int* old_ceiling );
int pthread_mutex_getprioceiling( const pthread_mutex_t* mutex,
                                  int* prioceiling );
```

Функция `pthread_mutex_setprioceiling()` захватывает мьютекс (или блокируется, пока мьютекс не будет освобожден, и уже тогда захватывает его) и изменяет установленную для него величину граничного приоритета, после чего освобождает мьютекс для использования другими потоками. После изменения значения граничного приоритета предыдущее значение возвращается в `old_ceiling`.

Функция возвращает следующие значения:

`EOK` — успешное завершение;

`EINVAL` — указанный в вызове мьютекс не существует или указанный приоритет выходит за диапазон допустимых значений;

`EPERM` — поток, вызвавший функцию, не имеет прав на изменение граничного приоритета указанного мьютекса.

Захват мьютекса

Захват мьютекса может производиться тремя разными функциями, в основе которых лежит функция из native QNX API `SyncMutexLock()`.

Простой захват

```
int pthread_mutex_lock( pthread_mutex_t* mutex );
```

Функция захватывает мьютекс, на который ссылается `mutex`. Если мьютекс уже захвачен другим потоком, то вызвавший поток блокируется до освобождения мьютекса и после этого захватывает его. Только после этого функция `pthread_mutex_lock()` возвращает управление. Если захватить мьютекс пытается поток, который им уже владеет, то поведение функции `pthread_mutex_lock()` будет зависеть от значений атрибутов мьютекса, указанных при его создании. QNX предоставляет возможность рекурсивного захвата мьютекса при соответствующих настройках атрибутов (см. выше раздел «Параметры мьютекса»). При создании мьютекса с параметрами по умолчанию попытка повторного захвата мьютекса ни к чему не приводит. Если включен режим контроля ошибок и отключен рекурсивный захват мьютекса, функция `pthread_mutex_lock()` возвращает `EDEADLK` при попытке повторного захвата мьютекса тем же потоком.

Функция pthread_mutex_lock() может возвращать следующие значения:

EOK – успешное завершение;

EAGAIN – недостаточно системных ресурсов для захвата мьютекса;

EDEADLK – вызывающий поток уже владеет мьютексом и мьютекс не поддерживает рекурсивный захват (режим контроля ошибок);

EINVAL – некорректное значение параметра mutex.

Попытка захвата

```
int pthread_mutex_trylock( pthread_mutex_t* mutex );
```

Функция проверяет, свободен ли мьютекс mutex, и если да, то она захватывает его. В противном случае функция возвращает значение EBUSY.

Возвращаемые значения:

EOK – успешное завершение;

EAGAIN – недостаточно системных ресурсов для захвата мьютекса;

EBUSY – мьютекс mutex уже захвачен;

EINVAL – некорректное значение параметра mutex.

Захват с установкой времени ожидания

```
#include <pthread.h>
#include <time.h>
int pthread_mutex_timedlock( pthread_mutex_t* mutex,
                             const struct timespec* abs_timeout );
```

Функция проверяет, свободен ли мьютекс (mutex), и если да, то поток, в котором вызвана функция, захватывает этот мьютекс. Если мьютекс уже захвачен, вызвавший поток блокируется до освобождения мьютекса либо до наступления времени, указанного в аргументе abs_timeout. Если это время уже наступило, поток не блокируется вообще, но захват все-таки произойдет, если мьютекс свободен.

Наступление времени определяется по часам REALTIME_CLOCK, когда значение часов оказывается равным или большим значения, указанного в abs_timeout. Тип данных timespec определен в файле <time.h>.

Если мьютекс создан с атрибутом протокола PRIO_INHERIT, то после выхода потока из блокировки на мьютексе по тайм-ауту приоритет владельца мьютекса подвергается пересмотру в соответствии с приоритетами потоков, оставшихся в очереди на захват мьютекса.

Возвращаемые значения:

EOK – успешное завершение;

EAGAIN – недостаточно системных ресурсов для захвата мьютекса;

EDEADLK – вызывающий поток уже владеет мьютексом, который не поддерживает рекурсивный захват (режим контроля ошибок);

EINVAL – мьютекс использует протокол граничного приоритета для предотвращения инверсии (атрибут protocol установлен в значение PTHREAD_PRIO_PROTECT), но приоритет вызвавшего потока выше граничного приоритета, присвоенного мьютексу; поток должен быть блокирован (мьютекс не свободен), а значение поля abs_timeout, показывающее количество наносекунд, меньше нуля или больше 1000 миллионов; переменная, на которую указывает mutex, не является инициированным объектом – мьютексом.

ETIMEDOUT – мьютекс не может быть захвачен, поскольку указанный тайм-аут истек.

Освобождение мьютекса

```
int pthread_mutex_unlock( pthread_mutex_t* mutex );
```

Функция pthread_mutex_unlock() освобождает мьютекс, на который ссылается переменная mutex. Вызвавший поток должен быть владельцем мьютекса. Если есть потоки, блокированные в ожидании освобождения мьютекса, то поток с наивысшим приоритетом (или при равных приоритетах дольше всех ждавший) выходит из блокированного состояния и становится владельцем мьютекса.

Для мьютексов, разрешающих рекурсивный захват, функция освобождения должна вызываться столько же раз, сколько и функция захвата.

Возвращаемые значения:

EOK – успешное завершение;

EINVAL – переменная, на которую указывает mutex, не является инициированным объектом – мьютексом;

EPERM – вызвавший поток не является владельцем мьютекса.

Разрушение объекта мьютекс

```
int pthread_mutex_destroy( pthread_mutex_t* mutex );
```

Вызов разрушает объект мьютекс, на который указывает переменная mutex. После чего эта переменная не может быть использована без предварительного вызова pthread_mutex_init().

Возвращаемые значения:

EOK – успешное завершение;

EBUSY – мьютекс захвачен и не может быть разрушен до освобождения;

EINVAL – переменная, на которую указывает mutex, не является инициированным объектом – мьютексом.

Операции, не поддерживаемые POSIX

В native QNX API есть ряд функций работы с мьютексом, которые не определены POSIX-стандартом, однако они могут оказаться весьма по-

лезными. Поскольку тип POSIX-мьютекса порождается от sync_t, то вполне возможно использование комбинации функций, определенных POSIX, и «родных» native-функций QNX. Однако необходимо помнить, что в таком случае ни о какой межсистемной совместимости говорить уже не приходится.

Восстановление «мертвого» мьютекса

```
#include <sys/neutrino.h>
int SyncMutexRevive( sync_t* sync );
int SyncMutexRevive_r( sync_t* sync );
```

Эти функции[1] предназначены для восстановления мьютекса, который находится в состоянии блокирования DEAD. Мьютекс попадает в состояние DEAD, когда память, использованная при захвате мьютекса, освобождается. Такое может произойти, например, когда умирает поток, захвативший мьютекс, расположенный в разделяемой памяти. В результате вызова вызвавший поток становится владельцем мьютекса, и его счетчик захватов устанавливается в 1 для рекурсивного мьютекса.

Ошибки выполнения функции:

EFAULT – ошибка при обращении к указанным в аргументах переменным;

EINVAL – указанный объект синхронизации не существует или не находится в состоянии DEAD;

ETIMEDOUT – отмена вызова по тайм-ауту ядра (устанавливается вызовом TimerTimeout()).

Установка уведомления о «смерти» мьютекса

Определить состояние мьютекса как DEAD можно с помощью функции SyncMutexEvent(), которая определяет событие, связанное со «смертью» мьютекса.

```
#include <sys/neutrino.h>
int SyncMutexEvent( sync_t* sync, struct sigevent* event );
int SyncMutexEvent_r( sync_t* sync, struct sigevent* event );
```

Данная функция предназначена для установки обработчика ситуации, когда мьютекс попадает в состояние DEAD (то есть перераспределяется память, из которой произошел захват мьютекса). Захватить мьютекс, оказавшийся в состоянии DEAD, можно далее с помощью вызова функции SyncMutexRevive().

---

[1] Функции SyncMutexRevive() и SyncMutexRevive_r(), из которых вторая является потокобезопасной формой первой, как это описывалось выше, отличаются между собой только способом уведомления об ошибке: первая форма возвращает –1 в случае ошибки и устанавливает errno, а вторая непосредственно возвращает код ошибки.

Ошибки выполнения функции:

EAGAIN – в данный момент ядро не имеет ресурсов для обработки запроса;

EFAULT – ошибка произошла при попытке обращения к sync;

EINVAL – объект синхронизации, на который указывает sync, не существует.

Пример применения мьютекса

Модернизируем наш пример из раздела, посвященного использованию семафора для случая множества потоков источников и приемников данных. Проблема заключается в том, что когда несколько потоков одновременно попытаются вызвать функцию push() или pop(), может произойти сильная путаница, поэтому код этих функций должен исполняться эксклюзивно, только одним потоком. Решить эту проблему можно двумя способами: воспользоваться бинарным семафором или мьютексом. Мы решили применить именно мьютекс и ниже расскажем причину, по которой мы здесь смешали в одной конструкции эти два элемента синхронизации.

```
/* Шаблонный класс очереди данных */
template <class T> class CdataQueue {
public:
   CmessageQueue() { pthread_mutex_init( &_mutex, NULL ); };
   ~CmessageQueue() { pthread_mutex_destroy( &_mutex );};
   void push( T _new_data ) {
      pthread_mutex_lock( &_mutex );
      data_queue.push(_new_data );
      data_event.reset();
      pthread_mutex_unlock( &_mutex );
   };
   T pop() {
      data_event.wait();
      pthread_mutex_lock( &_mutex );
      T res = data_queue.front();
      data_queue.pop();
      pthread_mutex_unlock( &_mutex );
      return res;
   };
private:
   std::queue<T> data_queue;
   event data_event;
   pthread_mutex_t _mutex;
};
```

На первый взгляд задача очевидна: надо не допустить одновременного исполнения двух участков кода. Почему же не воспользоваться семафором, как мы описывали, когда рассказывали о способах его применения? Дело в том, что мы хотели получить универсальное средство передачи данных между потоками, не зависящее от допущений о при-

оритетах потоков и степени их зависимости. Когда мы строим систему реального времени, вопрос взаимного неявного влияния разных потоков на выполнение друг друга становится очень важным. Мы уже неоднократно упоминали эффект инверсии приоритетов и те способы, которыми можно ее избежать, используя мьютекс для защиты эксклюзивно используемого кода.

Сравнение и эффективность

В этом месте временно прервем наше последовательное повествование: мы закончили рассмотрение двух наиболее известных, важных и применяемых примитивов синхронизации – семафора и мьютекса. Теперь сделаем короткую остановку и проведем их взаимное сравнение, а также попробуем на примерах оценить затраты процессорной производительности, требуемые этими механизмами.

Дело в том, что на первый взгляд эти два механизма в высшей степени подобны, особенно если речь заходит о бинарном семафоре, принимающем значения счетчика 0 либо 1. Настолько подобны, что и в обсуждениях, и даже в неспециальной литературе можно встретить утверждения, что это «одно и то же». Сейчас мы увидим, что эти два сходных механизма различаются всем: и затратами процессорного времени на их обслуживание, и целями и задачами, которые они призваны решать, и временем жизни... Начнем с простой оценки затрат процессорного времени на обслуживание каждого из механизмов, после чего остальные различия станут нам намного понятнее.

Для проведения таких оценок используем уже применявшуюся нами схему «симметричных» тестов. Почему именно их? Да, здесь нам не требуются в явном виде обменные операции потоков, но воспользуемся «симметричными» тестами просто в силу минимальных переделок того, что уже было написано ранее. Итак, первый вариант теста для мьютекса (*файл sy20m.cc*):

Скоростные показатели мьютекса

```
unsigned long N = 1000;
static pthread_barrier_t bstart;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static bool debug = false;
static char* str;
static volatile int ind = 0;

void* threadfunc ( void* data ) {
   pthread_barrier_wait( &bstart );
   unsigned long i = 0;
   char tid[ 8 ];
   sprintf( tid, "%d", pthread_self() );
   uint64_t t = 0, t1;
   while( i++ != N ) {
```

```
      t1 = ClockCycles();
      pthread_mutex_lock( &mutex );
      if( debug ) str[ ind++ ] = *tid;
      pthread_mutex_unlock( &mutex );
      t += ClockCycles() - t1;
      sched_yield();
   };
   cout << pthread_self() << "\t: cycles - "
        << t << "; on mutex - " << t / N << endl;
   return NULL;
};

int main( int argc, char *argv[] ) {
   int opt, val;
   while ( ( opt = getopt( argc, argv, "n:v" ) ) != -1 ) {
      switch( opt ) {
         case 'n' :
            if( sscanf( optarg, "%i", &val ) != 1 )
               cout << "parse command line error" << endl,
               exit( EXIT_FAILURE );
            if( val > 0 ) N = val;
            break;
         case 'v' :
            debug = true;
            break;
         default :
            exit( EXIT_FAILURE );
       }
   };
   if( debug ) str = new char [ 2 * N + 1 ];
   const int T = 2;
   pthread_t tid[ T ];
   if( pthread_barrier_init( &bstart, NULL, T ) != EOK )
      perror( "barrier init" ), exit( EXIT_FAILURE );
   for( int i = 0; i < T; i++ )
      if( pthread_create( tid + i, NULL, threadfunc, NULL ) != EOK )
         perror( "thread create" ), exit( EXIT_FAILURE );
   for( int i = 0; i < T; i++ )
      pthread_join( tid[ i ], NULL );
   if( debug ) {
      str[ ind ] = '\0';
      cout << str << endl;
      delete [] str;
   };
   exit( EXIT_SUCCESS );
};
```

Результаты выполнения этого теста:

```
# sy20m -n100000
3       : cycles - 14644442; on mutex - 146
2       : cycles - 14614219; on mutex - 146
```

```
# sy20m -n1000000
3       : cycles - 146505047; on mutex - 146
2       : cycles - 146388673; on mutex - 146
```

Модифицируем программу, используя вместо мьютекса неименованный бинарный семафор. Для того чтобы не загромождать текст практически тем же кодом, перечислим только необходимые при этом изменения (*файл sy20s.cc*):

1. Вместо мьютекса объявляем неименованный семафор, а статическая инициализация мьютекса заменяется на оператор (в теле главной программы) динамической инициализации семафора с присвоением ему начального значения 1:

```
static sem_t sem;
. . .
if( sem_init( &sem, 0, 1 ) != 0 )
   perror( "semaphore init" ), exit( EXIT_FAILURE );
```

2. Функция потока принимает вид:

```
void* threadfunc ( void* data ) {
   . . .
   while( i++ != N ) {
      t1 = ClockCycles();
      sem_wait( &sem );
      if( debug ) str[ ind++ ] = *tid;
      sem_post( &sem );
      t += ClockCycles() - t1;
      sched_yield();
   };
   . . .
};
```

В результате исполнения на этот раз мы получим:

```
# sy20s -n100000
3       : cycles - 87048886; on semaphore - 870
2       : cycles - 87077787; on semaphore - 870
# sy20s -n1000000
3       : cycles - 869638168; on semaphore - 869
2       : cycles - 868725494; on semaphore - 868
```

Делаем последнюю модификацию в этой группе тестов, теперь используем специфику именованного семафора (*файл sy20n.cc*):

1. Вместо оператора динамической инициализации неименованного семафора мы теперь должны создать именованный семафор:

```
static sem_t* sem;
. . .
const char semname[] = "/duble";
if( ( sem = sem_open( semname, O_CREAT, S_IRWXO, 1 ) ) == SEM_FAILED )
   perror( "semaphore init" ), exit( EXIT_FAILURE );
```

Примечание

Последний оператор заслуживает отдельного комментария. Техническая документация утверждает, что функция sem_open(), нормально возвращающая указатель созданного дескриптора семафора типа sem_t, в случае ошибки возвращает –1 (так было записано и в самых ранних редакциях POSIX). Но использование конструкции вида:

```
if( sem_open(. . .) == -1 )
```

просто вызовет синтаксическую ошибку (несоответствие типов) и не пройдет компиляцию! Естественнее было бы для такой функции возвращать NULL в случае ошибки, но... так определено в POSIX. Кроме того, во многих реализациях UNIX определяется константа:

```
#define SEM_FAILED( ( sem_t* )( -1 ) )
```

В документации QNX она нигде не упоминается, но, как мы видим, она определена, и все прекрасно работает!

2. Функция потока принимает вид (теперь sem, в отличие от предшествующего случая, ведь теперь это уже указатель на переменную типа sem_t):

```
void* threadfunc ( void* data ) {
   . . .
   while( i++ != N ) {
      t1 = ClockCycles();
      sem_wait( sem );
      if( debug ) str[ ind++ ] = *tid;
      sem_post( sem );
      t += ClockCycles() - t1;
      sched_yield();
   };
   . . .
};
```

3. Теперь особое внимание необходимо уделить не только созданию, но и ликвидации именованного семафора (такой семафор имеет время жизни ядра системы и будет продолжать свое существование и после завершения нашего приложения):

```
sem_close( sem );
sem_unlink( semname );
```

Запустим полученное приложение при таком значении -n, которое обеспечит достаточное время его работы. Прежде чем обсуждать полученные результаты, посмотрим отображение семафора на пространство файловых имен системы во время работы приложения:

```
# ls -l /dev/sem
total 1
n------r-x  1 root      root             1 Feb 10 18:56 duble
```

А теперь и результаты работы программы:

```
# nice -n-19 sy20n -n100000
3       : cycles - 1453746002; on semaphore - 14537
2       : cycles - 1454203573; on semaphore - 14542
```

Наконец, мы можем обратиться к количественному анализу полученных цифр:

- Примитивы – мьютекс, неименованный и именованный семафоры, – кажущиеся на первый взгляд сходными, требуют для своего обслуживания в эквивалентных условиях принципиально различных затрат, величины которых радикально отличаются: 140 – 870 – 14500 процессорных циклов соответственно, что соотносится как 1:6,2:104.
- Так же радикально отличаются и их характеристики доступа: изнутри процесса (или даже только из того потока, который уже владеет мьютексом), из внешнего процесса, из процесса, работающего на совершенно другом сетевом узле... То, что мы уже рассматривали как «характеристики времени жизни» объектов, принадлежит различным категориям: процесса (process-persistent), ядра (kernel-persistent) или файловой системы (filesystem-persistent).
- У захваченного мьютекса всегда есть поток-владелец, и только он может освободить его в дальнейшем. Именно поэтому мьютекс может использоваться для синхронизации потоков, но только синхронизации в смысле **разграничения временной последовательности доступа** к фрагменту кода – к тому, что часто называют критической секцией кода. Функциональность семафора значительно выше: при возможности (почти всегда) его применения в том контексте, в котором используется и мьютекс (только нужно ли это делать?), он может применяться и для синхронизации потоков в смысле **координации последовательности** их взаимодействия в качестве элемента, управляющего порядком выполнения. Покажем это на примере. Для этого незначительно трансформируем код предыдущего теста для семафора (*файл sy21.cc*):

Синхронизация потоков семафорами

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
#include <iostream.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <semaphore.h>

unsigned long N = 1000;
unsigned int T = 2;
```

```
static sem_t* sem;
static bool debug = false;
static char* str;                    // строка диагностики
static volatile int ind = 0;
uint64_t *t;

void* threadfunc ( void* data ) {
   unsigned long i = 0;
   char tid[ 8 ];
   sprintf( tid, "%X", pthread_self() );
   // временная метка начала во всех потоках устанавливается
   // на время достижения этой точки в последнем (активном) потоке
   if( (int)data == T - 1 ) {
      uint64_t c =  ClockCycles();
      for( int i = 0; i < T; i++ ) t[ i ] = c;
   };
   // рабочий цикл переключений за счет синхронизации
   while( i++ < N ) {
      sem_wait( sem + (int)data );
      if( debug ) str[ ind++ ] = *tid;
      sem_post( sem + ( (int)data + 1 ) % T );
   };
   t[ (int)data ] = ClockCycles() - t[ (int)data ];
   return NULL;
};

int main( int argc, char *argv[] ) {
   int opt, val;
   while ( ( opt = getopt( argc, argv, "n:t:v" ) ) != -1 ) {
      switch( opt ) {
         case 'n' :
            if( sscanf( optarg, "%i", &val ) != 1 )
               cout << "parse command line error" << endl,
               exit( EXIT_FAILURE );
            if( val > 0 ) N = val;
            break;
         case 't' :
            if( sscanf( optarg, "%i", &val ) != 1 )
               cout << "parse command line error" << endl,
               exit( EXIT_FAILURE );
            if( val > 0 ) T = val;
            break;
         case 'v' :
            debug = true;
            break;
         default :
            exit( EXIT_FAILURE );
       }
   };
   if( debug ) str = new char[ T * N + 1 ];
   pthread_t* tid = new pthread_t[ T ];
```

```
    sem = new sem_t[ T ];
    t = new uint64_t[ T ];
    for( int i = 0; i < T; i++ ) {
       // все потоки, кроме последнего, будут заблокированы
       // на своих семафорах сразу же после старта
       if( sem_init( sem + i, 0, ( i == ( T - 1 ) ) ? 1 : 0 ) )
          perror( "semaphore init" ), exit( EXIT_FAILURE );
       if( pthread_create( tid + i, NULL,
                           threadfunc, (void*)i ) != EOK )
          perror( "thread create error" ), exit( EXIT_FAILURE );
    };
    for( int i = 0; i < T; i++ )
       pthread_join( tid[ i ], NULL );
    for( int i = 0; i < T; i++ ) sem_destroy( sem + i );
    delete [] sem;
    for( int i = 0; i < T; i++ )
       cout << tid[ i ] << "\t: cycles - " << t[ i ]
            << ";\ton semaphore - " << t[ i ] / T / N
            << endl;
    delete [] tid;
    delete [] t;
    if( debug ) {
       str[ ind ] = '\0';
       cout << str << endl;
       delete [] str;
    };
    exit( EXIT_SUCCESS );
};
```

Логически приложение изменилось следующим образом:

- Теперь у нас может быть не 2 идентичных (симметричных) потока, а произвольное их количество (ключ `-t` при запуске приложения).
- Потоки синхронизируются не на одном семафоре – введен массив семафоров по числу потоков: каждый поток блокируется на «своем» семафоре, но разблокирует его (после очередного выполнения своего фрагмента) семафор заблокированного «соседа».
- Теперь нам нет нужды использовать барьер для одновременного старта всех созданных потоков: семафоры всех создаваемых потоков инициализируются нулевым значением; стартующий поток тут же блокируется на своем семафоре, и только последний из запущенных выполняется, не блокируясь на семафоре.
- Из кода исключены какие бы то ни было средства принудительной передачи управления (`sched_yield()`) – все управление логикой ветвления осуществляется только состояниями семафоров.

Посмотрим, что у нас получилось. Запускаем приложение с диагностическим выводом идентификаторов потоков (ключ `-v`; он у нас был в тестах и ранее, только мы о нем не упоминали):

```
# nice -n-19 sy21 -n20 -t12 -v
```

```
2       : cycles - 664874;      on semaphore - 2770
3       : cycles - 649150;      on semaphore - 2704
4       : cycles - 638906;      on semaphore - 2662
5       : cycles - 622987;      on semaphore - 2595
6       : cycles - 611781;      on semaphore - 2549
7       : cycles - 594515;      on semaphore - 2477
8       : cycles - 571003;      on semaphore - 2379
9       : cycles - 552834;      on semaphore - 2303
10      : cycles - 536817;      on semaphore - 2236
11      : cycles - 519357;      on semaphore - 2163
12      : cycles - 500388;      on semaphore - 2084
13      : cycles - 296633;      on semaphore - 1235
D23456789ABCD23456789ABCD23456789ABCD23456789ABCD23456789ABCD23456789ABCD234
56789ABCD23456789ABCD23456789ABCD23456789ABCD23456789ABCD23456789ABCD2345678
9ABCD23456789ABCD23456789ABCD23456789ABCD23456789ABCD23456789ABCD23456789ABC
D23456789ABC
```

В строке диагностики (внизу) хорошо видно регулярное чередование выполняющихся потоков, причем оно начинается с индекса последнего созданного потока (13 в показанном примере). Теперь проделаем то же самое, но на большой выборке и с отключенной диагностикой, чтобы получить «чистое» время контекстных переключений потоков, не искаженное затратами на операции формирования диагностики (результаты для нескольких различных размерностей задачи при разном количестве потоков):

```
# nice -n-19 sy21 -n100000 -t12
2       : cycles - 1509597589;  on semaphore - 1257
3       : cycles - 1509581545;  on semaphore - 1257
4       : cycles - 1509570283;  on semaphore - 1257
5       : cycles - 1509552472;  on semaphore - 1257
6       : cycles - 1509537934;  on semaphore - 1257
7       : cycles - 1509519299;  on semaphore - 1257
8       : cycles - 1509502312;  on semaphore - 1257
9       : cycles - 1509482667;  on semaphore - 1257
10      : cycles - 1509466343;  on semaphore - 1257
11      : cycles - 1509449264;  on semaphore - 1257
12      : cycles - 1509431112;  on semaphore - 1257
13      : cycles - 1509222808;  on semaphore - 1257

# nice -n-19 sy21 -n100000 -t7
2       : cycles - 859768389;   on semaphore - 1228
3       : cycles - 859756956;   on semaphore - 1228
4       : cycles - 859745649;   on semaphore - 1228
5       : cycles - 859736698;   on semaphore - 1228
6       : cycles - 859724685;   on semaphore - 1228
7       : cycles - 859707720;   on semaphore - 1228
8       : cycles - 859554045;   on semaphore - 1227

# nice -n-19 sy21 -n50000 -t13
2       : cycles - 832789852;   on semaphore - 1281
```

```
3       : cycles - 832813231;   on semaphore - 1281
4       : cycles - 832835011;   on semaphore - 1281
5       : cycles - 832851360;   on semaphore - 1281
6       : cycles - 832868482;   on semaphore - 1281
7       : cycles - 832884308;   on semaphore - 1281
8       : cycles - 832900935;   on semaphore - 1281
9       : cycles - 832916093;   on semaphore - 1281
10      : cycles - 832931944;   on semaphore - 1281
11      : cycles - 832946479;   on semaphore - 1281
12      : cycles - 832962202;   on semaphore - 1281
13      : cycles - 832976433;   on semaphore - 1281
14      : cycles - 832782465;   on semaphore - 1281

# nice -n-19 sy21 -n50000 -t17
2       : cycles - 1142879872;  on semaphore - 1344
3       : cycles - 1142906138;  on semaphore - 1344
4       : cycles - 1142927650;  on semaphore - 1344
5       : cycles - 1142943675;  on semaphore - 1344
6       : cycles - 1142959582;  on semaphore - 1344
7       : cycles - 1142974919;  on semaphore - 1344
8       : cycles - 1142991068;  on semaphore - 1344
9       : cycles - 1143005896;  on semaphore - 1344
10      : cycles - 1143021518;  on semaphore - 1344
11      : cycles - 1143036136;  on semaphore - 1344
12      : cycles - 1143053448;  on semaphore - 1344
13      : cycles - 1143068415;  on semaphore - 1344
14      : cycles - 1143083676;  on semaphore - 1344
15      : cycles - 1143098361;  on semaphore - 1344
16      : cycles - 1143114009;  on semaphore - 1344
17      : cycles - 1143128525;  on semaphore - 1344
18      : cycles - 1142872665;  on semaphore - 1344
```

Есть некоторая корреляция времени переключения контекста с размером выборки и количеством обрабатывающих потоков, но она в широком диапазоне этих параметров не превышает 8%. В данном приложении эта численная величина включает в себя: блокирование на семафоре, переключение на контекст другого потока и разблокирование семафора. Если вспомнить, что раньше мы получали оценки для принудительного (посредством `sched_yield()`) переключения контекста потоков в 375 процессорных циклов, а для захвата-освобождения семафора – порядка 870, то эти цифры хорошо согласуются с полученными сейчас результатами.

Рассматриваемые примитивы служат принципиально различным целям. Мьютекс, как уже было сказано ранее, предназначен в первую очередь для регламентации доступа к участкам программного кода. Семафоры же больше предназначены для регламентации порядка доступа к определенным объектам данных. Классическими задачами этого класса являются задачи «производитель – потребитель», когда M производителей создают некоторые объекты данных (читая эти дан-

ные с реальных внешних устройств, или создавая их как результат только внутренних вычислений, или любым другим способом), а N потребителей независимо берут произведенные объекты данных на последующую обработку.

Это настолько общий и часто встречающийся класс задач, что покажем для него простейший «скелет» в виде отдельного приложения, в котором отслеживание порядка доступа потребителей будет осуществлять счетный семафор (*файл sy22.cc*). Для простоты понимания приложение сделано как трансформация кода предшествующей группы тестов. В качестве имитации производства объекта данных, как и в качестве его обработки потребителем, используется пассивная пауза (`delay()`) на случайную величину (производство и обработка объектов данных в коде не показаны, так как это не относится к существу рассматриваемого – нас интересуют процессы синхронизации этих операций, а не сами операции).

Кроме основной нашей цели это приложение дополнительно демонстрирует:

- Практическое использование принудительного завершения (отмены) потоков «извне» с управлением состоянием завершаемости потоков и расстановкой точек отмены, о чем мы уже говорили ранее.
- Использование атомарных (непрерываемых) операций (например, `atomic_add_value()`), о которых мы будем говорить чуть позже.
- Использование реентерабельных форм функций стандартной библиотеки, безопасных в многопоточной среде (`rand_r()` вместо `rand()`).

Один производитель – T потребителей

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <semaphore.h>
#include <atomic.h>

const int D = 10;
unsigned int T = 2;
static sem_t sem;
pthread_t* tid;

void* writer ( void* data ) {
   unsigned long i = (int)(data); // общий размер выборки
   unsigned int s = 1;
   while( i-- > 0 ) {
      delay( (long)rand_r( &s ) * D / RAND_MAX + 1 );
      sem_post( &sem );  // объект данных произведен
   };
```

```
   for( i = 0; i < T; i++ ) pthread_cancel( tid[ i + 1 ] );
   return NULL;
};

static char *str;  // строка результирующей диагностики
static volatile unsigned ind = 0;

void* reader ( void* ) {
   char tid[ 8 ];
   sprintf( tid, "%X", pthread_self() );
   unsigned int s = rand();
   pthread_setcanceltype( PTHREAD_CANCEL_DEFERRED, NULL );
   while( true ) {
      sem_wait( &sem );              // получен объект данных
      str[ atomic_add_value( &ind, 1 ) ] = *tid;
      pthread_testcancel();
      delay( (long)rand_r( &s ) * D * T / RAND_MAX + 1 );
   };
   return NULL;
};

int main( int argc, char *argv[] ) {
   unsigned long N = 1000;
   int opt, val;
   while ( ( opt = getopt( argc, argv, "n:t:" ) ) != -1 ) {
      switch( opt ) {
         case 'n' :
            if( sscanf( optarg, "%i", &val ) != 1 )
               cout << "parse command line error" << endl,
               exit( EXIT_FAILURE );
            if( val > 0 ) N = val;
            break;
         case 't' :
            if( sscanf( optarg, "%i", &val ) != 1 )
               cout << "parse command line error" << endl,
               exit( EXIT_FAILURE );
            if( val > 0 ) T = val;
            break;
         default :
            exit( EXIT_FAILURE );
      }
   };
   str = new char[ N + 1 ];
   tid = new pthread_t[ T + 1 ];
   if( sem_init( &sem, 0, 0 ) )
      perror( "semaphore init" ), exit( EXIT_FAILURE );
   if( pthread_create( tid, NULL, writer, (void*)N ) != EOK )
      perror( "writer create error" ), exit( EXIT_FAILURE );
   for( int i = 0; i < T; i++ )
      if( pthread_create( tid + i + 1, NULL, reader, NULL ) != EOK )
         perror( "reader create error" ), exit( EXIT_FAILURE );
```

```
    for( int i = 0; i < T; i++ )
       pthread_join( tid[ i ], NULL );
    sem_destroy( &sem );
    delete [] tid;
    str[ ind ] = '\0';
    cout << str << endl;
    delete [] str;
    exit( EXIT_SUCCESS );
};
```

Вот как выглядит результат выполнения этой программы (во избежание внесения дополнительного синхронизма в качестве общего числа циклов «производства» и числа потоков потребителей выбраны взаимно простые числа):

```
# sy22 -n200 -t13
3456789ABCDEF7936A8547E39DCB45F67A59B84D37EC64F395B6AEF78B9DF34CB53B86A5FEDF
975B3A8EC46FB8AD954736FA78C3ED46F7B594EC7B83AC6F9D4BCE569A73F86BCAD74C536EB7
9F5C8DA5B463EFBC7D937AEC85FDE456BCAF69DE7F385CA6
```

Хорошо видно, как строго последовательный поначалу порядок доступа потребителей к объектам данных десинхронизируется и становится хаотическим: каждый освободившийся потребитель приступает к работе над следующим объектом данных, как только тот становится доступен.

Атомарные операции

Атомарные операции не относятся к элементам синхронизации параллельных ветвей программы. Но им следует уделить внимание по двум причинам. Во-первых, атомарные операции – это простое и эффективное средство, позволяющее во многих случаях избежать использования механизмов синхронизации. А во-вторых, атомарные операции зачастую выпадают из рассмотрения из-за их двойственного положения: при обсуждении параллелизма и синхронизации они не рассматриваются, потому что не являются элементами синхронизации, а при обсуждении последовательных программ не рассматриваются потому, что здесь в них просто нет необходимости.

Атомарные операции – это операции, для которых гарантируется их непрерываемость даже при выполнении на симметричных мультипроцессорных платформах. Выполнение атомарных операций не прерывается даже асинхронными аппаратными прерываниями. Таким образом, эта группа операций является также и безопасной в многопоточном окружении.

Действительно, наиболее часто примитивы синхронизации применяются для создания критической секции кода с целью предотвращения возможности одновременного воздействия на объекты данных со стороны нескольких параллельно развивающихся ветвей программы.

При одновременной работе с данными из различных потоков состояние данных после такого воздействия должно считаться «неопределенным», при этом последствия могут быть более тяжкими, чем просто некорректное состояние данных – структура сложных объектов может быть просто разрушена.

В многопоточной среде элементарные и привычные операции могут таить в себе опасности. Действительно, простейший оператор вида:

```
i = i + 1;
```

содержит в себе опасность, если этот оператор записан в функции потока, выполняемой несколькими экземплярами потоков (совершенно типичный случай). Не менее опасен, но менее очевиден по внешнему виду и оператор:

```
i += 1;
```

Даже операторы инкремента и декремента (`++i` и `--i`), которые в системе команд практически всех типов процессоров выполняются как **атомарные** и которые являются основой для реализации семафорных операций, в симметричной мультипроцессорной архитектуре перестают быть безопасными. Хуже того, привычные программисту операции стандартной библиотеки и просто синтаксические конструкции языка становятся небезопасными в многопоточной среде. Вот еще два примера:

1. Оператор копирования нетипизированного блока памяти, безбоязненно используемый десятилетиями:

```
void* memcpy( void* dst, const void* src, size_t length );
```

2. Операторы присваивания, инициализации или сравнения структурированных объектов данных:

```
struct X {
   X( const X& y ) { . . . };
   friend bool operator==( const X& f, const X& s ) { . . . };
   // оператор присваивания мы не переопределяем, используется
   // присваивание по умолчанию – побайтовое копирование
}
. . .
X    A;
. . .
X B( A );              // потенциальная ошибка
. . .
B = A;                 // потенциальная ошибка
if( A == B ) { . . . };// потенциальная ошибка
```

Примечание

Обратите внимание, что все объекты данных, для которых могут наблюдаться обсуждаемые эффекты, должны быть доступны вне потока, то есть быть глобальными с точки зрения видимости в потоке.

Именно для безопасного манипулирования данными в параллельной среде QNX API и вводятся атомарные операции. Десять атомарных функций делятся на две симметричные группы по виду своего именования и логике функционирования. Все атомарные операции осуществляются только над одним типом данных `unsigned int`, но, как будет показано далее, это не такое уж и сильное ограничение. Сам объект, над которым осуществляется атомарная операция (типа `unsigned int`), – это самая обычная переменная целочисленного типа, только описанная с квалификатором `volatile`.

Помимо атомарных операций над этой переменной могут выполняться любые другие действия, которые можно считать безопасными в многопоточной среде: инициализация, присваивание значений, сравнения. Более того, при выходе программы за область возможного многопоточного доступа к этой переменной она может далее использоваться любым традиционным и привычным образом.

Важно также отметить, что термин «атомарность» относится не к особым свойствам некоторого объекта данных, а к ограниченному ряду операций, которые можно безопасно выполнять над этим объектом в многопоточной среде.

Общий вид прототипов каждой из двух групп атомарных операций следующий:

```
void atomic_*( volatile unsigned *D , unsigned S );

unsigned atomic_*_value( volatile unsigned *D, unsigned S );
```

где вместо `*` должно стоять имя одной из пяти операций (таким алгоритмом и обеспечивается 10 различных атомарных функций):

`add` – добавить численное значение к операнду;

`sub` – вычесть численное значение из операнда;

`clr` – очистить **биты** в значении операнда (выполняется побитовая операция `(*D) &= ~S`);

`set` – установить **биты** в значении операнда (выполняется побитовая операция `(*D) |= S`);

`toggle` – инвертировать **биты** в значении операнда (выполняется побитовая операция `(*D) ^= S`);

`D` – именно тот объект, над которым осуществляется атомарная операция;

`S` – второй операнд осуществляемой операции.

Две формы атомарных функций для каждой операции отличаются тем, что первая из них выполняет операцию без возврата значения, а вторая возвращает значение, которое операнд D имел до выполнения операции (т. е. прежнее значение, как это делают, например, префиксные операции инкремента ++D и декремента --D, в отличие от постфиксных D++ и D--).

Зачем нужны две формы для операции? Техническая документация QNX утверждает, что вторая форма может выполняться дольше. Справедливость этого утверждения и насколько дольше выполняется вторая форма, мы скоро увидим на примерах.

Итак, у нас есть 10 функций для выполнения пяти атомарных операций:

```
atomic_add()          atomic_add_value()
atomic_sub()          atomic_sub_value()
atomic_clr()          atomic_clr_value()
atomic_set()          atomic_set_value()
atomic_toggle()       atomic_toggle_value()
```

Как используются атомарные операции? Обычно для предотвращения одновременного изменения некоторого счетчика индекса мы вынуждены создавать критическую секцию, обозначая ее, скажем, операциями над мьютексом. В частности, в следующем примере нам необходимо из различных потоков последовательно дописывать некоторые байтовые результаты в единый буфер:

```
// глобальные описания, доступные всем потокам
const unsigned int N = . . .
uint8_t buf[ N ];
// индекс текущей позиции записи
unsigned int ind = 0;
// общий мьютекс, доступный каждому из потоков
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
. . .
// выполняется в каждом из потоков:
uint8_t res[ M ];             // результат некоторой операции
unsigned int how = . . .      // реальная длина этого результата
pthread_mutex_lock( &mutex );
memcpy( (void*)buf + ind, (void*)res, how );
ind += how;
pthread_mutex_unlock( &mutex );
```

Используя атомарные операции, мы можем этот процесс записать так (все глобальные описания остаются неизменными):

```
// глобальные описания, доступные всем потокам
. . .
// индекс текущей позиции записи
volatile unsigned int ind = 0;
. . .
```

```
// выполняется в каждом из потоков:
uint8_t res[ M ];              // результат некоторой операции
unsigned int how = . . .       // реальная длина этого результата
memcpy( (void*)buf + atomic_add_value( ind, how ), (void*)res, how );
```

Или даже так:

```
// глобальные описания, доступные всем потокам
. . .
// указатель текущей позиции записи:
volatile unsigned int ind = (unsigned int)buf;
. . .
// выполняется в каждом из потоков:
memcpy( (void*)atomic_add_value( ind, how ),
        (void*)res, how );
```

В последнем случае это, конечно, трюкачество, построенное на том, что в 32-разрядной архитектуре представления указателя и переменной `unsigned int` совпадают, но это «работающее трюкачество» и работающее иногда весьма эффективно.

Техника применения атомарных операций оказывается крайне удобной, например, при осуществлении вывода, часто диагностического, из различных потоков. Положим, нам нужно в каждом из многих потоков выводить диагностическую строку при достижении ими определенной точки исполнения:

```
cout << "Это вывод потока " << pthread_self () << endl;
```

Но так делать нельзя: при таком решении у нас появляются 2 проблемы, одна из которых очевидна, а другая – не очень:

1. Выводы разных потоков могут «смешиваться»; более того, за счет буферизации вывода некоторые «рваные» фрагменты мы будем наблюдать дважды. Одним словом, наш вывод окажется полностью «нечитабельным».
2. При осуществлении вывода в контексте потока почти наверняка в процессе вывода будут выполняться системные вызовы, которые потребуют новой диспетчеризации и приведут к вытеснению исходного потока. При этом порядок передачи управления от потока к потоку при наличии отладочного вывода будет отличаться от порядка при его отсутствии. А это, наверное, не совсем то, что мы ожидали. В результате при наличии отладочного вывода мы можем наблюдать совсем не ту картину, для изучения которой этот вывод, собственно, и предназначен.

Эти эффекты не связаны с какой-то конкретной формой вывода, такой как вывод в поток, показанный выше; точно так же будет вести себя и традиционный вызов `printf(. . .)`.

Проблема очень просто решается, если вместо непосредственного вывода из потоков последовательно сбрасывать все сообщения в проме-

жуточный буфер, который будет выводиться в те периоды времени программы, когда запись в него не производится:

```
const int N = . . ., M = . . .;
char buf[ N ];
volatile unsigned ind = 0;
. . .
// а вот это производится из каждого потока
char tbuf[ M ];
sprintf( tbuf, "Это вывод потока %X", pthread_self() );
strcpy( buf + atomic_add_value( ind, strlen( tbuf ) ), tbuf );
```

И наконец, последнее: есть ли смысл во введении этого дополнительного механизма, если всегда существует альтернативная форма организации такой же защиты доступа посредством критической секции (например, при использовании мьютекса)? Сравним (*файл a1.cc*) временные затраты при многократном изменении значения переменной для случаев атомарных операций и критической секции на базе мьютекса (мы берем именно мьютекс, потому что из всех примитивов синхронизации он самый низкоуровневый и быстрый):

Сравнение мьютекса и двух форм вызова атомарной операции

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <unistd.h>
#include <inttypes.h>
#include <pthread.h>
#include <sys/neutrino.h>
#include <atomic.h>

int main( int argc, char *argv[] ) {
   uint64_t N = 100000;
   bool atom = false, value = false;
   int opt, val;
   while ( ( opt = getopt( argc, argv, "n:av" ) ) != -1 ) {
      switch( opt ) {
         case 'n' :          // количество повторений
            if( sscanf( optarg, "%i", &val ) != 1 )
               cout << "parse command line error" << endl,
               exit( EXIT_FAILURE );
            if( val > 0 ) N = val;
            break;
         // использовать атомарные операции
         case 'a' :
            atom = true;
            break;
         // использовать форму, возвращающую значение
         case 'v' :
            value = true;
```

```
            break;
         default :
            exit( EXIT_FAILURE );
       }
    };
    // замеряется количество процессорных циклов для каждого случая
    uint64_t i = N, t = ClockCycles();
    volatile unsigned ind = 0;
    if( !atom ) {
       pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
       while( i-- ) {
          pthread_mutex_lock( &mutex );
          ind++;
          pthread_mutex_unlock( &mutex );
       }
    }
    else if( value )
       while( i-- ) atomic_add_value( &ind, 1 );
    else while( i-- ) atomic_add( &ind, 1 );
    t = ClockCycles() - t;
    cout << "all cycles - " << t << "; on operation - "
         << t / N << endl;
    exit( EXIT_SUCCESS );
};
```

Вот результат при использовании критической секции:

```
# nice -n-19 a1 -n10000000
all cycles - 1120872156; on operation - 112
```

Результат с применением атомарной операции, не возвращающей значения:

```
# nice -n-19 a1 -n10000000 -a
all cycles - 391018203; on operation - 39
```

Результат с применением атомарной операции, возвращающей значение (обещанная разница составляет порядка 10%):

```
# nice -n-19 a1 -n10000000 -a -v
all cycles - 441158981; on operation - 44
```

Условная переменная

Одним из важнейших принципов использования мьютексов является максимальное сокращение размеров критической секции, то есть участка, который потоки должны проходить последовательно. Однако зачастую возникает необходимость ожидания выполнения некоторого условия внутри критической секции.

Реализация подобного ожидания «в лоб» привела бы к тому, что все потоки, разделяющие данную критическую секцию, были бы вынуждены ждать выполнения условия для каждого из них. При «правильной» реализации ожидания поток должен освобождать мьютекс на время ожидания и вновь захватывать его, когда ожидаемое условие выполняется. Специально для этого случая стандартом POSIX предусмотрены **условные переменные**. QNX Neutrino реализует условные переменные как на уровне вызовов микроядра в своем native API, так и в соответствии со стандартом POSIX.

Отметим, что дополнение мьютекса условной переменной делает их комбинацию универсальным базисом, на котором могут быть построены любые сколь угодно сложные в своем поведении объекты синхронизации.

Фактически совместное использование мьютекса и условной переменной создает специфический комбинированный объект синхронизации, который может иметь принципиально более широкое применение, чем отдельно взятый мьютекс. Тем не менее поведение этого объекта синхронизации не столь просто и далеко не очевидно. Рассмотрим его более подробно.

На рис. 4.1 приведена блок-схема операций, выполняемых потоком при использовании мьютекса и условной переменной для синхронизации. Линиями отделены операции, выполняющиеся «внутри» функций, указанных справа. Обратите внимание, что наиболее сложная логика соответствует вызову функции ожидания на условной переменной.

Проблема в первую очередь заключается в том, что внутри критической секции, отмеченной вызовами функций `pthread_mutex_lock()` и `pthread_mutex_unlock()`, не может находиться более одного потока в единый момент времени. Следовательно, даже если поток, блокированный на условной переменной, и получит `pthread_cond_signal()` или `pthread_cond_broadcast()`, он не сможет немедленно продолжить свое выполнение, если внутри критической секции уже находится другой поток. В этом случае разблокированный (на условной переменной) поток изменяет свой статус с «блокированного на условной переменной» (в котором он находился до этого) на статус «блокированного на мьютексе» и пребывает в нем до тех пор, пока текущий владелец не освободит мьютекс.

Все функции операций над условной переменной и ее атрибутами реализованы в заголовочном файле `<pthread.h>`. Если для вызова функции необходимы дополнительные заголовочные файлы, это будет указано особо.

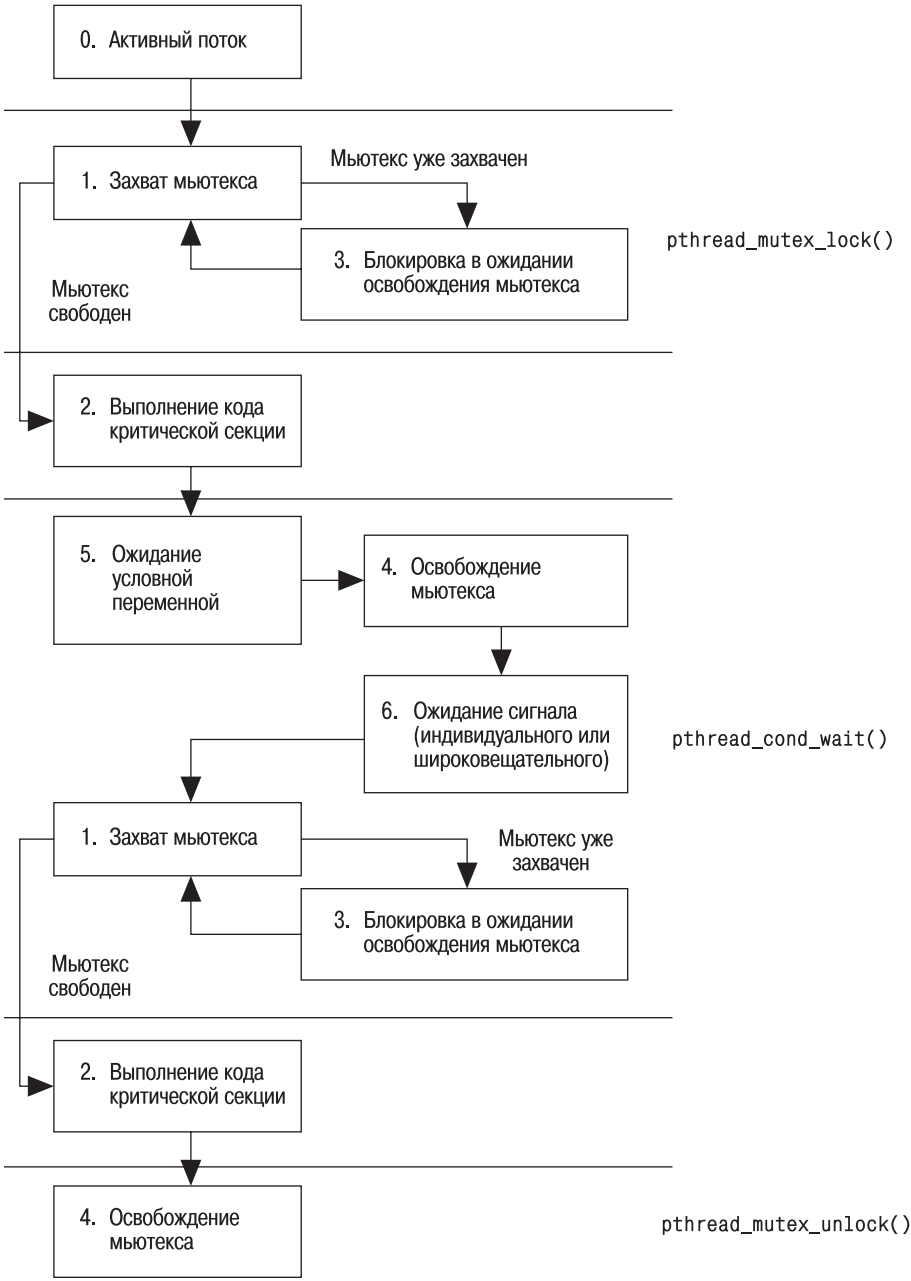

Рис. 4.1. *Схема действий потока при выполнении синхронизации с применением пары мьютекс–условная переменная (обратите внимание, что операции при участии мютекса (1, 2, 3) выполняются дважды)*

Операции над условной переменной

Параметры условной переменной

Инициализация параметров

```
int pthread_condattr_init( pthread_condattr_t* attr );
```

Функция инициализирует структуру атрибутов условной переменной, на которую указывает параметр attr. Структура данных pthread_condattr_t определена в файле <pthread.h> и является производной от типа syncattr_t, описанного в разделе «Параметры мьютекса». При инициализации атрибуты устанавливаются в значения по умолчанию.

Возвращаемые значения:

EOK – успешное завершение;

ENOMEM – недостаточно памяти для инициализации атрибутов условной переменной attr.

Для условной переменной возможна модификация значительно меньшего числа параметров, чем для мьютекса. Следующие функции описывают доступ к этим параметрам.

Параметр доступа

```
int pthread_condattr_setpshared(
                            pthread_condattr_t* attr, int pshared );
int pthread_condattr_getpshared(
                            const pthread_condattr_t* attr, int* pshared );
```

Функции устанавливают/считывают, возможен ли доступ к условной переменной из потоков, порожденных в других процессах. Параметр pshared может принимать следующие значения:

- PTHREAD_PROCESS_SHARED – любой поток, имеющий доступ к области памяти, в которой расположена условная переменная, может ее использовать.
- PTHREAD_PROCESS_PRIVATE (значение по умолчанию) – только поток, созданный в контексте того же процесса, в котором находится условная переменная, может ее использовать. Если поток из другого процесса попытается использовать условную переменную, созданную с параметром PTHREAD_PROCESS_PRIVATE, результат будет неопределен.

Возвращаемые значения:

EOK – успешное завершение;

EINVAL – атрибуты условной переменной, на которые указывает attr, или новое значение, на которое ссылается pshared, не определены.

Параметры тайм-аута

```
int pthread_condattr_setclock(
                        pthread_condattr_t* attr, clockid_t id );
int pthread_condattr_getclock(
                        const pthread_condattr_t* attr, clockid_t* id );
```

Функции устанавливают/считывают, каким способом (т. е. на основании какого счетчика) вычисляется значение тайм-аута при вызовах `pthread_cond_timedwait()`. Этот параметр в QNX Neutrino 6.2 не является обязательным и введен в соответствии со стандартом POSIX 1003.1j (draft). На практике можно устанавливать только значение `REALTIME_CLOCK` в качестве параметра `id`; это же значение является значением по умолчанию.

Возвращаемые значения:

`EOK` – успешное завершение;

`EINVAL` – неверный аргумент `attr`.

Разрушение блока параметров

```
int pthread_condattr_destroy( pthread_condattr_t* attr );
```

Функция разрушает блок параметров условной переменной, на которые указывает `attr`, после чего он уже не может использоваться без повторной инициализации.

На практике разрушение параметров объекта синхронизации не имеет особого смысла. Вы всегда можете переопределить атрибуты, содержащиеся в переменной `attr`, для инициализации других условных переменных.

Возвращаемые значения:

`EOK` – успешное завершение;

`EINVAL` – неверный аргумент `attr`.

Инициализация условной переменной

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init( pthread_cond_t* cond, pthread_condattr_t* attr );
```

Инициализирует условную переменную `cond` со значениями, установленными атрибутами `attr`. Вместо прямого вызова функции `pthread_cond_init()` для начальной инициализации статических условных переменных (глобальных на уровне файла кода или пространства имен namespase либо явно описанных с квалификатором `static`) можно воспользоваться макросом `PTHREAD_COND_INITIALIZER`.

Возвращаемые значения:

`EOK` – успешное завершение;

`EAGAIN` – нет свободных системных объектов синхронизации;

EBUSY – переменная cond уже инициализирована и не разрушалась;

EFAULT – ошибка доступа ядра к объектам cond или attr;

EINVAL – неправильное значение переменной cond.

Ожидание условия

Простое ожидание

```
int pthread_cond_wait( pthread_cond_t* cond, pthread_mutex_t* mutex );
```

Вызов функции блокирует вызвавший поток на условной переменной cond и разблокирует мьютекс mutex. Поток блокируется до тех пор, пока другой поток не вызовет функцию разблокирования на условной переменной cond (pthread_cond_signal() или pthread_cond_broadcast()). Мьютекс mutex должен быть захвачен потоком до вызова функции. Поток, блокированный на условной переменной, может быть разблокирован также приходом сигнала или вызовом завершения потока. В любом случае при разблокировании потока и выходе из функции ожидания поток вновь захватывает мьютекс mutex.

Не следует использовать условную переменную с мьютексом, у которого разрешен рекурсивный захват.

Возвращаемые значения:

EOK – успешное завершение ожидания либо ожидание прервано сигналом;

EAGAIN – недостаток системных ресурсов для реализации ожидания на условной переменной;

EFAULT – произошла ошибка при попытке обращения к указателям cond или mutex;

EINVAL – возвращается в следующих ситуациях:

- не инициализированы переменные, на которые указывают cond или mutex;
- попытка использования переменной, на которую указывает cond, для нескольких мьютексов;
- вызвавший поток не владеет указанным мьютексом.

Ожидание с тайм-аутом

```
#include <time.h>
int pthread_cond_timedwait( pthread_cond_t* cond,
                            pthread_mutex_t* mutex,
                            const struct timespec* abstime );
```

Поведение функции идентично варианту обычного ожидания, за исключением того, что ожидание может завершиться также при наступлении времени, переданного параметром abstime.

Следует помнить, что после наступления времени тайм-аута управление совсем не обязательно вернется к вызвавшему потоку. После наступления этого времени функция переведет поток из состояния блокирования на условной переменной в состояние готовности и предпримет попытку захвата мьютекса. Если мьютекс в это время захвачен другим потоком, вызвавший поток перейдет в состояние блокирования на мьютексе.

Возвращаемые значения:

`EOK` – успешное завершение ожидания либо ожидание прервано сигналом;

`EAGAIN` – недостаток системных ресурсов для реализации ожидания на условной переменной;

`EFAULT` – произошла ошибка при попытке обращения к указателям `cond` или `mutex`;

`EINVAL` – возвращается в следующих ситуациях:

- не инициализированы переменные, на которые указывают `cond` или `mutex`;
- попытка использования переменной, на которую указывает `cond`, для нескольких мьютексов;
- вызвавший поток не владеет указанным мьютексом.

`ETIMEDOUT` – завершение функции по наступлению времени, указанного в `abstime`.

Выполнение условия

Штатным способом разблокирования потока, блокированного на условной переменной, является вызов функции, сигнализирующей о выполнении условия. В native API это функция `SyncCondvarSignal()`, которая имеет две POSIX-обертки: `pthread_cond_signal()` и `pthread_cond_broadcast()`. Разница между ними заключается в том, что первая пробуждает только один, самый приоритетный поток из ждущих выполнения условия, а вторая пробуждает все потоки, ожидающие выполнения условия.

Однако необходимо помнить про специфику ожидания внутри критической секции: вызов функции `pthread_cond_broadcast()` только переведет ожидающие потоки из состояния блокирования на условной переменной в состояние блокировки на мьютексе, поскольку мьютекс сможет захватить только самый приоритетный поток.

Нештатным способом завершения ожидания на условной переменной может быть приход немаскированного сигнала UNIX. Если для данного сигнала определен обработчик, он выполнится без захвата мьютекса, а попытка захвата будет произведена уже после его завершения.

Выполнение условия для единичного потока

```
int pthread_cond_signal( pthread_cond_t* cond );
```

Функция переводит в состояние готовности самый приоритетный поток из блокированных на условной переменной cond, после чего поток предпринимает попытку захвата своего мьютекса. Если есть несколько потоков с равным (и высшим) приоритетом, блокированных на условной переменной, то разблокируется тот поток, который ожидал дольше остальных.

Возвращаемые значения:

EOK – успешное завершение;

EFAULT – произошла ошибка при попытке обращения к указателям cond или mutex;

EINVAL – не инициализирована переменная, на которую указывает cond.

Выполнение условия для всех ожидающих потоков

```
int pthread_cond_broadcast( pthread_cond_t* cond );
```

Вызов функции разблокирует все потоки, блокированные на условной переменной cond. Потоки разблокируются в порядке приоритетов. Для потоков равного приоритета разблокирование проводится в порядке FIFO.

Возвращаемые значения:

EOK – успешное завершение;

EFAULT – произошла ошибка при попытке обращения к указателям cond или mutex;

EINVAL – не инициализирована переменная, на которую указывает cond.

Разрушение условной переменной

```
int pthread_cond_destroy( pthread_cond_t* cond );
```

Вызов функции деинициализирует условную переменную cond. Для дальнейшего использования условной переменной, на которую ссылается cond, ее необходимо инициализировать вызовом pthread_cond_init(). Функция может использоваться для изменения параметров условной переменной.

Возвращаемые значения:

EOK – успешное завершение;

EBUSY – в данный момент другой поток блокирован на условной переменной cond;

EINVAL – не инициализирована переменная cond.

Ждущая блокировка

QNX предоставляет упрощенный вариант использования условной переменной для блокирования (остановки) потока при помощи интерфейса так называемой *ждущей блокировки* (*sleepon*). Для использования этого механизма не нужно явно создавать никаких объектов синхронизации, за вас это делает ОС. Внешне ждущие блокировки выглядят как набор функций ожидания и освобождения, при этом последовательность действий в принципе аналогична использованию мьютексов и условных переменных.

За этим интерфейсом на самом деле скрывается один мьютекс и несколько дополнительных условных переменных. Использование функций ожидания должно проходить внутри участка кода, отмеченного вызовами блокирования и разблокирования мьютекса, ассоциированного со ждущей блокировкой. Одним из основных недостатков ждущей блокировки является то, что для всех потоков и всех ключей ожидания используется один общий мьютекс. ОС не может никоим образом отслеживать взаимные блокировки потоков при использовании ждущих блокировок. В целом поведение этого средства синхронизации идентично бинарным семафорам, но оно требует дополнительных операций блокирования мьютекса.

Все функции для работы со ждущими блокировками объявлены в файле `<pthread.h>`.

Операции со ждущей блокировкой

Захват и освобождение ждущей блокировки

Вызов функций ожидания может производиться только внутри блока захвата и освобождения ждущей блокировки:

```
int pthread_sleepon_lock( void );
int pthread_sleepon_unlock( void );
```

Функция захвата `pthread_sleepon_lock()` возвращает следующие значения:

`EOK` — успешное выполнение;

`EDEADLK` — попытка повторного захвата мьютекса;

`EAGAIN` — может возникнуть при первом вызове в процессе, если системе не хватает ресурсов для создания внутреннего мьютекса.

Функция освобождения `pthread_sleepon_unlock()` возвращает значения:

`EOK` — успешное выполнение;

`EPERM` — вызвавший поток не является владельцем внутреннего мьютекса.

Функции ожидания

Ожидание выполнения условия для ждущей блокировки может выполняться в двух вариантах: простое ожидание и ожидание с установкой тайм-аута.

```
int pthread_sleepon_wait( const volatile void* addr );
int pthread_sleepon_timedwait( const volatile void* addr, uint64_t nsec );
```

При вызове функций ожидания необходимо указать ключ `addr` (произвольный адрес в памяти). Если этот адрес указывается впервые, для данного вызова создается новая условная переменная. Поток освобождает захваченный внутренний мьютекс и переходит в состояние блокировки на условной переменной.

Ожидание завершения потока

Ожидание родительским потоком завершения одного или нескольких порожденных им «присоединенных» потоков (на вызове `pthread_join()`) – это простейший и эффективный вариант синхронизации потоков, не требующий для своей реализации каких-либо дополнительных синхронизирующих примитивов. Ранее мы уже детально рассматривали процесс порождения и ожидания завершения потоков, сейчас же лишь коротко вернемся к этому вопросу с иной точки зрения – с позиции синхронизации. В простейшем случае общая схема такой синхронизации всегда одинакова и описывается подобной структурой кода:

```
void* threadfunc ( void* data ) {
   ...
   return NULL;
};


...
// здесь создается нужное количество (N) потоков:
pthread_t tid[ N ];
for( int i = 0; i < N; i++ )
   pthread_create( tid + i, NULL, threadfunc, NULL );
...
// а вот здесь ожидается завершение всех потоков!
for( int i = 0; i < N; i++ )
   pthread_join( tid + i, NULL );
```

При первом знакомстве с подобным шаблоном кода пугает то обстоятельство, что предписан такой же порядок ожидания завершения потоков, как и при их создании. И это при том, что порядок их завершения может быть совершенно произвольным. Но представленный шаблон верен: если некоторый ожидаемый в текущем цикле поток `j` «задерживается», а мы заблокированы именно в ожидании `tid[ j ]`, то после завершения этого ожидаемого потока, которое когда-то все-таки наступит, мы «мгновенно» пробегаем все последующие `i`, для которых

соответствующие tid[i] уже завершились ранее. Так что представленный шаблон корректен и широко используется на практике.

Примечание

В такой схеме потоки могут возвратить в точку ожидания (и зачастую делают это) результат своего выполнения. В представленном шаблоне мы не стали показывать возврат значений, чтобы не загромождать код. Возврат результата подробно рассматривался ранее, когда речь шла о завершении потоков.

Показанная схема синхронизации на завершении потоков не является примитивом синхронизации и не требует использования таковых, но она выводит нас на еще один тип примитивов – барьер.

Барьер

Барьер как раз и предназначен для разрешения выше обозначенной проблемы – ожидания условия достижения несколькими заданными потоками точки синхронизации. Достигнув этой точки, потоки освобождаются «одновременно» и уже с этой точки продолжают свое независимое развитие. «Классическая» схема использования барьера (именно в таком качестве он чаще всего и используется), неоднократно приводимая в описаниях, выглядит так (мы уже много раз использовали ее в примерах кода):

```
static pthread_barrier_t bfinish;

void* threadfunc ( void* data ) {
   // ... потоки что-то делают ...
   pthread_barrier_wait( &bfinish );
   return NULL;
};

int main( int argc, char *argv[] ) {
   ...
   int N = ... ; // будем создавать N идентичных потоков
   if( pthread_barrier_init( &bfinish, NULL, N + 1 ) != EOK )
      perror( "barrier init" ), exit( EXIT_FAILURE );
   for( int i = 0; i < N; i++ )
      if( pthread_create( NULL, NULL, threadfunc, NULL ) != EOK )
         perror( "thread create" ), exit( EXIT_FAILURE );
   pthread_barrier_wait( &bfinish );
   ...
};
```

Очевидно, что по функциональности эта схема мало отличается от ожидания завершения потоков на pthread_join(), описанного выше. Однако есть различия в организации: если ранее мы просто ожидали полного завершения дочерних потоков, то в данной схеме мы ожидаем достижения ими специально созданной точки синхронизации. Еще од-

но отличие состоит в том, что схема синхронизации с ожиданием завершения на pthread_join() приемлема только для «присоединенных» потоков, тогда как схема на pthread_barrier_wait() может применяться и к «отсоединенным», автономным потокам.

Но если бы различие двух схем только на том и заканчивалось, то, возможно, нецелесообразно было бы вводить новый механизм барьеров. Однако техника использования барьеров шире, она может быть использована, например, когда нужно, чтобы, напротив, последовательно создаваемые потоки (в цикле порождающего потока) стартовали на исполнение «одновременно» (особенно это характерно тогда, когда дочерние потоки создаются с более высоким приоритетом, чем порождающий):

```
static pthread_barrier_t bstart;

void* threadfunc ( void* data ) {
   // все потоки после создания должны "застрять" на входном барьере,
   // чтобы потом одновременно "сорваться" в исполнение...
   pthread_barrier_wait( &bstart );
   // ... выполнение ...
   return NULL;
};

int main( int argc, char *argv[] ) {
   ...
   int N = ... ;   // будем создавать N идентичных потоков
   if( pthread_barrier_init ( &bstart, NULL, N ) != EOK )
      perror( "barrier init" ), exit( EXIT_FAILURE );
   for( int i = 0; i < nthr; i++ ) {
      if( pthread_create( NULL, NULL,
                          threadfunc, NULL ) != EOK )
         perror( "thread create" ), exit( EXIT_FAILURE );
   };
   ...
};
```

Обратите внимание на параметр количества ожидающих на барьере потоков при его инициализации: здесь он на единицу меньше.

Применение барьеров подробно описано в литературе [1], поэтому мы не будем специально останавливаться на этом элементе синхронизации, тем более что это один из наиболее простых в применении элементов.

По непонятным причинам документация QNX [8] причисляет барьеры к элементам синхронизации ядра, однако никаких средств native API QNX, предназначенных для работы с барьерами, документация не описывает, а заголовочный файл <pthread.h> так описывает тип pthread_barrier_t:

```
typedef struct {
   unsigned int     barrier;
```

```
    unsigned int     count;
    pthread_mutex_t  lock;
    pthread_cond_t   bcond;
} pthread_barrier_t;
```

Выводы можно сделать самостоятельно.

Также несколько загадочно выглядит тот факт, что согласно документации QNX 6.2.1 все функции работы с барьером и его атрибутами описаны в заголовочном файле <pthread.h>, за исключением двух функций pthread_barrier_wait() и pthread_barrierattr_setpshared(), о которых говорится, что они описаны в файле <sync.h>! Но если заглянуть в заголовочные файлы, то выясняется, что можно спокойно использовать для абсолютно всех функций работы с барьером либо заголовочный файл <pthread.h>, либо <sync.h>.

Операции с барьерами

Параметры барьера

Следующие функции инициализируют и разрушают блок параметров барьера:

```
int pthread_barrierattr_init( pthread_barrierattr_t* attr );
int pthread_barrierattr_destroy( pthread_barrierattr_t* attr );
```

Функция инициализации возвращает следующие значения:

EOK – успешное выполнение;

ENOMEM – недостаточно памяти для инициализации атрибутов барьера.

Функция разрушения атрибутов объекта возвращает значения:

EOK – успешное выполнение;

EINVAL – передан неверный объект атрибутов барьера attr.

Параметры барьера описываются типом pthread_barrierattr_t и аналогично другим типам синхронизации используются в момент инициализации элемента синхронизации. Фактически единственным атрибутом барьера является модификатор доступа к барьеру из других процессов:

```
int pthread_barrierattr_setpshared(
                        pthread_barrierattr_t* attr, int pshared );
int pthread_barrierattr_getpshared(
                        const pthread_barrierattr_t* attr, int* pshared );
```

По умолчанию атрибуты барьера запрещают доступ к элементу синхронизации из других процессов.

Обе функции могут возвращать следующие значения:

EOK – успешное выполнение;

EINVAL – одно или оба из переданных функции значений неверны.

Инициализация и разрушение барьера

```
int pthread_barrier_init( pthread_barrier_t* barrier,
                          const pthread_barrierattr_t* attr,
                          unsigned int count );
```

Функция инициализирует объект синхронизации типа барьер, после чего его можно использовать. В атрибутах барьера устанавливается (или запрещается) возможность доступа к барьеру из других процессов. По умолчанию такой доступ запрещен. Для того чтобы изменить возможность доступа к созданному ранее барьеру, его необходимо разрушить, установить соответствующий атрибут и инициализировать барьер повторно. Параметр `count` показывает, какое количество потоков будет ожидать на барьере до их освобождения.

Возвращаемые значения:

`EOK` – успешное выполнение;

`EAGAIN` – системе не хватает ресурсов для инициализации барьера;

`EBUSY` – попытка инициализации уже инициализированного барьера;

`EFAULT` – сбой произошел при обращении ядра к аргументам;

`EINVAL` – `attr` указывает на неинициализированное значение атрибутов.

```
int pthread_barrier_destroy( pthread_barrier_t* barrier );
```

Функция разрушает барьер, после чего соответствующий элемент синхронизации `barrier` не может использоваться без повторной его инициализации.

Возвращаемые значения:

`EOK` – успешное выполнение;

`EBUSY` – в настоящее время есть потоки, блокированные на барьере;

`EINVAL` – неинициализированный объект `barrier`.

Ожидание на барьере

Функция ожидания (синхронизации) на барьере:

```
int pthread_barrier_wait( pthread_barrier_t* barrier );
```

Вызов этой функции приводит к блокированию потока до тех пор, пока на барьере не накопится количество заблокированных потоков, определенное ранее при вызове функции `pthread_barrier_int()`.

Необходимо с особой осторожностью относиться к использованию барьеров для остановки и синхронизации потоков разных приоритетов. Поскольку потоки, ожидающие у барьера, находятся в блокировке на условной переменной (внутренней), то система никак не отслеживает эффекты, связанные с возможной инверсией приоритетов.

Если поток, блокированный на барьере, получает сигнал, для которого определен обработчик, то обработчик сигнала выполняется, но по завершении его выполнения поток вновь блокируется до выполнения условия барьера.

Документация QNX утверждает, что нельзя заранее сказать, какой поток будет освобожден первым, когда заданное количество потоков достигнет барьера. Учитывая, что при реализации операций над потоками использовалась комбинация мьютекса с условной переменной (как видно из приведенного выше определения, типа `pthread_barrier_t`), освобождение блокированных потоков будет проходить по принципам, определенным для потоков, блокированных на условной переменной и получающих «широковещательное» (broadcast), или одновременное, освобождение. Документация QNX утверждает, что в таком случае первым будет «отпущен» тот поток, который ждал дольше других.

Функция ожидания на барьере возвращает значения:

`BARRIER_SERIAL_THREAD` — для первого из потоков, достигшего барьера и блокированного на нем;

`0` — для всех последующих потоков;

`EINVAL` — при ошибке выполнения, которая может возникнуть только по некорректности инициализации барьера.

Блокировки чтения/записи

Блокировка чтения/записи является специфическим механизмом, отличающимся от рассмотренных выше. Специфика состоит в следующем:

- Данный тип блокировки даже по POSIX является альтернативным. Часто этот тип блокировки может реализовываться как надстройка над уже существующими базовыми примитивами. У. Стивенс [2] показывает один из вариантов такой «оберточной» реализации и приводит ссылки на еще несколько вариантов, в том числе и на предложенный стандартом IEEE 1996.
- Функциональность, обеспечиваемая блокировкой чтения/записи, может быть предоставлена и простым использованием других базовых механизмов, однако реализация с блокировкой чтения/записи может быть намного эффективнее по производительности приложения (как мы увидим позже).

Целесообразность привлечения блокировки чтения/записи возникает тогда, когда все множество потоков может быть отчетливо разделено на две группы: потоки, только считывающие защищаемые блокировкой данные (читатели), и потоки, модифицирующие эти данные (писатели). Такая ситуация традиционно возникает в области работы с объемными, сложно структурированными данными.

При использовании блокировки чтения/записи, в отличие от других ранее рассмотренных механизмов, вводится различие между получением такой блокировки для считывания и записи. При этом действуют следующие правила:

- Любое количество потоков может заблокировать ресурс для считывания, если ни один поток не заблокировал его по записи.
- Наличие множественных блокировок по чтению не препятствует (не блокирует) ни одному из потоков-читателей выполнять свои операции с ресурсом.
- Блокировка по записи может быть установлена, только если ни один поток не блокирует ресурс ни по чтению, ни по записи.
- Блокировка по записи запрещает дальнейшую блокировку ресурса (блокирует запрашивающий процесс) и по чтению, и по записи.

Функции работы с блокировками чтения/записи объявлены, как и большинство примитивов синхронизации, в заголовочном файле `<pthread.h>`.

Операции с блокировками чтения/записи

Инициализация объекта блокировки

```
int pthread_rwlock_init( pthread_rwlock_t* rwl,
                         const pthread_rwlockattr_t* attr );
int pthread_rwlock_destroy( pthread_rwlock_t* rwl );
```

Вызов функций инициализирует/разрушает блокировку чтения/записи. При инициализации блокировки ей передается структура параметров блокировки `pthread_rwlockattr_t`, в которой могут устанавливаться параметры доступа к объекту из других процессов.

Вместо прямого вызова функции `pthread_rwlock_init()` для начальной инициализации статических блокировок чтения/записи (глобальных на уровне файла кода или пространства имен namespase либо явно описанных с квалификатором `static`) можно воспользоваться макросом `PTHREAD_RWLOCK_INITIALIZER`.

Захват блокировки чтения/записи

В связи со спецификой применения блокировок чтения/записи этот объект синхронизации имеет две группы функций захвата, позволяющих по-разному регулировать доступ к защищаемому участку кода.

К первой группе относятся функции, допускающие рекурсивный захват объекта синхронизации и совместное использование участка кода. Это так называемые функции *блокировки по чтению*.

Вторая группа функций допускает только эксклюзивный захват объекта синхронизации и к выполнению защищаемого кода допускается только один поток. Это функции *блокировки по записи*.

Функции блокировки по чтению

```
int pthread_rwlock_rdlock( pthread_rwlock_t* rwl );
int pthread_rwlock_tryrdlock( pthread_rwlock_t* rwl );
int pthread_rwlock_timedrdlock( pthread_rwlock_t* rwlock,
                                const struct timespec* abs );
```

Эта группа функций позволяет множественный захват объекта синхронизации и одновременное исполнение защищаемого участка кода, с которым ассоциируется переменная `rwl`. Данная группа функций предназначена для обеспечения одновременного чтения данных, разделяемых несколькими потоками. Выполнение захвата с использованием этой группы функций будет блокировано, если объект синхронизации, на который указывает `rwl`, уже захвачен одной из функций эксклюзивного исполнения кода (блокировки по записи).

Первая функция в группе позволяет выполнить простой захват объекта синхронизации. При рекурсивном захвате с помощью любой из указанных функций необходимо помнить, что освобождение должно производиться столько же раз, сколько и захват.

Функция `pthread_rwlock_rdlock()` может возвращать следующие значения:

`EOK` – успешное выполнение;

`EAGAIN` – при первом использовании статически инициированной блокировки чтения/записи (`PTHREAD_RWLOCK_INITIALIZER`) недостаточно системных ресурсов для инициализации блокировки чтения/записи;

`EDEADLK` – вызывающий поток уже провел эксклюзивный захват (блокировку по записи) объекта синхронизации, на который ссылается `rwl`, и повторный захват на чтение привел бы к полному («мертвому») блокированию потока;

`EFAULT` – сбой при обращении ядра к `rwl`;

`EINVAL` – `rwl` указывает на неверный объект блокировки чтения/записи.

Вторая из перечисленных функций проверяет, не находится ли соответствующий объект синхронизации в эксклюзивном использовании, и если это так, она возвращает соответствующее значение без последующего ожидания освобождения объекта синхронизации.

Функция `pthread_rwlock_tryrdlock()` возвращает следующие значения:

`EOK` – успешное выполнение;

`EAGAIN` – при первом использовании статически инициированной блокировки чтения/записи (`PTHREAD_RWLOCK_INITIALIZER`) недостаточно системных ресурсов для инициализации блокировки чтения/записи;

`EBUSY` – блокировка чтения/записи блокирована по записи (собственно, это и есть основной мотив использования именно этого вызова);

EDEADLK – вызывающий поток уже провел эксклюзивный захват (блокировку по записи) объекта синхронизации, на который ссылается rwl;

EFAULT – сбой при обращении ядра к rwl;

EINVAL – rwl указывает на неверный объект блокировки чтения/записи.

Третья функция из группы предусматривает завершение блокировки ожидания освобождения объекта синхронизации при возникновении блокировки по записи или по истечении заданного тайм-аута ожидания.

Функция pthread_rwlock_timedrdlock() может возвращать следующие коды ошибки:

EAGAIN – система не может захватить блокировку по чтению, поскольку достигнуто максимальное число блокировок чтения для данного объекта;[1]

EDEADLK – вызывающий поток уже является владельцем указанного объекта синхронизации; он захватил его, используя блокировку по записи, и повторная блокировка по чтению привела бы к полному блокированию потока;

EINVAL – неверный параметр вызова: либо rwl указывает на неинициализированный объект блокировки чтения/записи, либо время тайм-аута abs задано меньше нуля или равно или выше предельного значения 1000 миллионов;

ETIMEDOUT – не удалось захватить блокировку до истечения срока тайм-аута.

Функции блокировки по записи

```
int pthread_rwlock_wrlock( pthread_rwlock_t* rwl );
int pthread_rwlock_trywrlock( pthread_rwlock_t* rwl );
int pthread_rwlock_timedwrlock( pthread_rwlock_t* rwlock,
                                const struct timespec* abs_timeout );
```

Функции этой группы предназначены для эксклюзивного захвата объекта синхронизации и использования его для блокирования по записи. Блокировка по записи, в отличие от блокировки по чтению, не допускает совместного исполнения защищаемого участка кода (ни пишущими, ни читающими потоками). Так же как в группе функций блокировки по чтению, в этой группе присутствуют функции простого захвата, попытки захвата и захвата с тайм-аутом ожидания освобождения.

Функция pthread_rwlock_wrlock() возвращает следующие значения:

EOK – успешное выполнение;

---

[1] Забавно! Речь идет об исчерпании количества рекурсивных захватов для внутреннего мьютекса. Здесь есть некоторое несоответствие с другими объектами синхронизации.

EAGAIN – при первом использовании статически инициированной блокировки чтения/записи (PTHREAD_RWLOCK_INITIALIZER) недостаточно системных ресурсов для инициализации блокировки чтения/записи;

EDEADLK – вызывающий поток уже является владельцем блокировки в эксклюзивном режиме;

EFAULT – сбой при обращении ядра к rwl;

EINVAL – rwl указывает на неверный объект блокировки чтения/записи.

Функция pthread_rwlock_trywrlock() возвращает значения:

EOK – успешное выполнение;

EAGAIN – при первом использовании статически инициированной блокировки чтения/записи (PTHREAD_RWLOCK_INITIALIZER) недостаточно системных ресурсов для инициализации блокировки чтения/записи;

EBUSY – блокировка уже захвачена в режиме чтения или записи;

EDEADLK – вызывающий поток уже является владельцем блокировки в эксклюзивном режиме;

EFAULT – сбой при обращении ядра к rwl;

EINVAL – rwl указывает на неверный объект блокировки чтения/записи.

Функция pthread_rwlock_timedwrlock() возвращает значения:

EOK – успешное выполнение;

EAGAIN – система не может захватить блокировку по записи, поскольку достигнуто максимальное число блокировок по записи для данного объекта;

EDEADLK – вызывающий поток уже является владельцем блокировки в эксклюзивном режиме;

EINVAL – неверный параметр вызова: либо rwl указывает на неинициализированный объект блокировки чтения/записи, либо время тайм-аута abs задано меньше нуля или равно или выше предельного значения 1000 миллионов;

ETIMEDOUT – не удалось захватить блокировку до истечения заданного срока тайм-аута.

Освобождение блокировки

```
int pthread_rwlock_unlock( pthread_rwlock_t* rwl );
```

Функция освобождает захваченный любым образом объект блокировки чтения/записи. Если объект был захвачен в режиме множественного использования (блокировки по чтению), то количество его освобождений должно равняться количеству захватов.

Возвращаемые значения:

EOK – успешное завершение;

EAGAIN – при первом использовании статически инициированной блокировки чтения/записи (PTHREAD_RWLOCK_INITIALIZER) недостаточно системных ресурсов для инициализации блокировки чтения/записи;

EFAULT – ядро не смогло обратиться к объекту rwl;

EINVAL – объект rwl указывает на неверно инициированный объект блокировки чтения/записи;

EPERM – нет потоков, захвативших объект rwl в режиме чтения или записи, или вызывающий поток не владеет блокировкой в режиме записи.

Использование блокировок чтения/записи

Построим приложение, использующее блокировку чтения/записи (*файл sy10.cc*):

Эффективность блокировки чтения/записи

```
#include <sys/syspage.h>
#include <sys/neutrino.h>
#include <list>

// сколь угодно сложные элементы внутренней базы данных
// приложения; в примере мы используем их простейший вид:
typedef int element;

// база данных приложения - динамический список элементов
class dbase - public list<element> {
   static const int READ_DELAY = 1, WRITE_DELAY = 2;
public:
   // операция "добавить в базу данных"
   void add( const element& e ) {
      int pos = size() * rand() / RAND_MAX;
      list<element>::iterator p = begin();
      for( int i = 0; i < pos; i++ ) p++;
      insert( p, e );
      delay( WRITE_DELAY );
   };
   // операция "найти в базе данных"
   int pos( const element& e ) {
      int n = 0;
      for( list<element>::iterator i = begin();
           i != end(); i++, n++ )
         if( *i == e ) { delay( READ_DELAY ); break; };
      if( n == size() )  n = -1;
      return n;
   };
} data;

inline element erand( unsigned long n ) {
   return (element)( ( n * rand() ) / RAND_MAX );
};
```

```
inline bool wrand( double p ) {
   return (double)rand() / (double)RAND_MAX < p;
};

int main( int argc, char *argv[] ) {
   // общее число обращений приложения к базе данных
   static unsigned long n = 1000;
   // вероятность обновления базы данных при обращении
   static double p = .1;
   unsigned long m;
   if( argc > 1 && ( m = atoll( argv[ 1 ] ) ) != 0 ) n = m;
   double q;
   if( argc > 2 && ( q = atof( argv[ 2 ] ) ) != 0 ) p = q;
   // начальное заполнение базы данных
   for( int i = 0; i < n; i++ ) data.add( erand( n ) );
   // тактовая частота процессора (для измерения времени)
   const uint64_t cps = SYSPAGE_ENTRY( qtime )->cycles_per_sec;
   // последовательность n обращений к базе данных,
   // из них p*n требуют обновления списка, а остальные
   // (1-p)*n требуют только выборки данных:
   uint64_t t = ClockCycles();
   for( int i = 0; i < n; i++ ) {
      element e = erand( n );
      if( !wrand( p ) ) data.pos( e );
      else data.add( e );
   };
   t = ( ( ClockCycles() - t ) * 1000000000 ) / cps;;
   cout << "evaluation time: " << (double)t / 1.E9
        << " sec." << endl;
   return EXIT_SUCCESS;
};
```

Перед нами простейшая последовательная программа, которая массированно читает свою базу данных и изредка ее модифицирует. Для выполнения реальных операций чтения/записи данных программе необходимо выполнять некоторые достаточно продолжительные операции. В приведенном коде эти операции имитируются задержками `delay( WRITE_DELAY )` и `delay( READ_DELAY )`.

Возникает совершенно естественное желание преобразовать последовательные запросы к данным в параллельные (*файл sy11.cc*). Для этого преобразуем структуру списка элементов, с которым работаем:

```
class dbase : public list<element> {
   static const int READ_DELAY = 1, WRITE_DELAY = 2;
   pthread_mutex_t loc;
public:
   dbase( void ) { pthread_mutex_init( &loc, NULL ); };
   ~dbase( void ) { pthread_mutex_destroy( &loc ); };
   void add( const element& e ) {
      pthread_mutex_lock( &loc );
      int pos = size() * rand() / RAND_MAX;
```

```
      list<element>::iterator p = begin();
      for( int i = 0; i < pos; i++ ) p++;
      insert( p, e );
      delay( WRITE_DELAY );
      pthread_mutex_unlock( &loc );
   };
   int pos( const element& e ) {
      int n = 0;
      pthread_mutex_lock( &loc );
      for( list<element>::iterator i = begin();
           i != end(); i++, n++ )
         if( *i == e ) { delay( READ_DELAY ); break; };
      pthread_mutex_unlock( &loc );
      if( n == size() )  n = -1;
      return n;
   };
} data;
```

А в вызывающей программе цикл запросов к данным преобразуем в:

```
pthread_t *h = new pthread_t[ n ];
uint64_t t = ClockCycles();
for( int i = 0; i < n; i++ ) {
   element e = erand( n );
   pthread_create( h + i, NULL, wrand( p ) ? add : pos, (void*)e );
};
for( int i = 0; i < n; i++ )
   pthread_join( h[ i ], NULL );
t = ( ( ClockCycles() - t ) * 1000000000 ) / cps;
delete h;
```

А используемые этим фрагментом функции потоков определим как:

```
static void* add( void* par ) { data.add( (element)par ); };
static void* pos( void* par ) { data.pos( (element)par ); };
```

Совершенно естественно, что список элементов, из которого мы извлекаем данные (и куда изредка помещаем новые), должен защищаться как при модификации, так и при считывании (во избежание их одновременной модификации «со стороны»). Понятно, что в представленном решении мы чересчур перестраховались: во время считывания мы должны защищаться от потенциальной одновременной модификации, но нет необходимости защищать структуру данных от параллельного считывания. Поэтому переопределим структуру данных (*файл sy12.cc*), используя блокировку чтения/записи, оставив все прочее без изменений:

```
class dbase : public list<element> {
   static const int READ_DELAY = 1, WRITE_DELAY = 2;
   pthread_rwlock_t loc;
public:
   dbase( void ) { pthread_rwlock_init( &loc, NULL ); };
```

```
    ~dbase( void ) { pthread_rwlock_destroy( &loc ); };
    void add( const element& e ) {
       pthread_rwlock_wrlock( &loc );
       int pos = size() * rand() / RAND_MAX;
       list<element>::iterator p = begin();
       for( int i = 0; i < pos; i++ ) p++;
       insert( p, e );
       delay( WRITE_DELAY );
       pthread_rwlock_unlock( &loc );
    };
    int pos( const element& e ) {
       int n = 0;
       pthread_rwlock_rdlock( &loc );
       for( list<element>::iterator i = begin();
            i != end(); i++, n++ )
          if( *i == e ) { delay( READ_DELAY ); break; };
       pthread_rwlock_unlock( &loc );
       if( n == size() )  n = -1;
       return n;
    };
} data;
```

А теперь пришло время сравнить варианты:

```
# nice -n-19 sy10 500 .2
evaluation time: 1.2296 sec.
# nice -n-19 sy11 500 .2
evaluation time: 1.24973 sec.
# nice -n-19 sy12 500 .2
evaluation time: 0.440904 sec.
```

При «жесткой» блокировке мы не получаем никакого выигрыша за счет параллельного выполнения запросов к данным, а при использовании блокировки чтения/записи – 3-кратный выигрыш. Проделаем то же самое, но в условиях гораздо меньшей интенсивности обновления данных относительно общего потока запросов:

```
# nice -n-19 sy10 500 .02
evaluation time: 0.989699 sec.
# nice -n-19 sy11 500 .02
evaluation time: 0.98391 sec.
# nice -n-19 sy12 500 .02
evaluation time: 0.0863443 sec.
```

Выигрыш становится более чем 10-кратным.

Показанные примеры (*sy10.cc*, *sy11.cc*, *sy12.cc*) в высшей степени условны: картина происходящего будет существенно другой при замене пассивного ожидания (`delay()`) на активные процессорные операции над данными, но общие тенденции сохраняются.

Спинлок

Спинлок, или «крутящаяся» блокировка, предназначен исключительно для применения в системах SMP (Symmetrical Multi-Processing), то есть в многопроцессорных системах. Поведение спинлока практически идентично классическому мьютексу, за единственным исключением – ожидающий поток не блокируется и не вытесняется. Не забывайте, речь идет о многопроцессорной системе! Основным назначением спинлока является задержка выполнения обработчиков прерываний, и предназначены они для исключения временных потерь, связанных с переключением контекстов.

Функции работы с «крутящейся» блокировкой объявлены в заголовочном файле <pthread.h>. Самих функций немного, и они имеют минимальные возможности по настройке. Спинлок не поддерживает тайм-ауты. Появление этого элемента синхронизации в QNX Neutrino связано с требованиями стандарта POSIX 1003.1j (draft).

Операции со спинлоком

Инициализация и разрушение спинлока

```
int pthread_spin_init( pthread_spinlock_t* spinner, int pshared );
```

Функция инициализирует объект синхронизации спинлока блокировки, на который указывает аргумент spinner, и устанавливает для него параметр доступа из других процессов в соответствии со значением переменной pshared. Эта переменная может принимать следующие значения:

- PTHREAD_PROCESS_SHARED – с объектом спинлок может оперировать поток любого процесса, имеющего доступ к памяти, в которой распределен объект спинлок;
- PTHREAD_PROCESS_PRIVATE – доступ к объекту синхронизации возможен только для потоков процесса, из адресного пространства которого была распределена память объекта синхронизации.

В случае успешного завершения функция возвращает нулевое значение, в противном случае – один из следующих кодов ошибок:

AGAIN – системе не хватает ресурсов для инициализации блокировки;

EBUSY – объект крутящейся блокировки, на который указывает spinner, уже инициирован;

EINVAL – некорректный объект spinner;

ENOMEM – система не имеет достаточного количества свободной памяти для создания нового объекта.

```
int pthread_spin_destroy( pthread_spinlock_t* spinner );
```

Функция деинициализирует объект крутящейся блокировки. После деинициализации для последующего применения объекта он должен быть вновь инициализирован. Обратите внимание, результат функции не определен, если поток в данный момент крутится на блокировке, на которую указывает spinner, либо если объект spinner не был инициализирован.

Возвращаемые значения:

EOK – успешное выполнение;

EBUSY – блокировка используется другим потоком и не может быть разрушена;

EINVAL – некорректный объект spinner.

Захват и освобождение спинлока

```
int pthread_spin_lock( pthread_spinlock_t* spinner );
int pthread_spin_trylock( pthread_spinlock_t* spinner );
```

Это функции захвата и попытки захвата крутящейся блокировки соответственно. Как и для мьютекса, если объект spinner в момент захвата свободен, то поток, вызвавший одну из этих функций, становится владельцем крутящейся блокировки. Если spinner уже захвачен другим потоком, то в случае вызова второй из рассматриваемых функций управление возвращается немедленно, а в случае простого захвата (первая функция) вызвавший поток «крутится», то есть остается активным, но не возвращает управления до тех пор, пока объект синхронизации не освободится.

Попытка повторного захвата крутящейся блокировки из того же потока приводит к мертвой блокировке.

Функции возвращают следующие параметры:

EOK – успешное выполнение;

EAGAIN – недостаточно ресурсов системы для захвата spinner;

EDEADLK – вызвавший поток уже владеет spinner;

EINVAL – spinner – неверный объект типа pthread_spinlock_t;

EBUSY – объект захвачен другим потоком (для pthread_spin_trylock()).

```
int pthread_spin_unlock( pthread_spinlock_t* spinner );
```

Вызов этой функции освобождает объект крутящейся блокировки, на который указывает аргумент spinner.

Функция может возвращать значения:

EOK – успешное выполнение;

EINVAL – неверный объект spinner;

EPERM – вызывающий поток не является владельцем крутящейся блокировки.

5

Специфические механизмы QNX

Операционная система QNX изнутри вся построена на клиент-серверных принципах, которые вытекают из микроядерной архитектуры и обмена сообщениями микроядра. Мы не могли обойти вниманием эти механизмы, поскольку они предоставляют огромный арсенал возможностей, однако их обстоятельное описание потребовало бы отдельной книги (полное описание см. в технической документации QNX по системной архитектуре). Более того, лучшая книга по обмену сообщениями микроядра уже, пожалуй, написана и переведена на русский язык [1]. В дополнение ко всему приложение «Организация обмена сообщениями», написанное В. Зайцевым и ранее не публиковавшееся, содержит обстоятельный анализ этого механизма.

Поэтому в главе мы лишь кратко рассмотрим вопросы параллелизма и синхронизации, присущие самой микроядерной архитектуре системы.

Обмен сообщениями микроядра

Модель обмена сообщениями – это тот фундамент, на котором стоит архитектура любой микроядерной ОС, как на трех китах: SEND – RECEIVE – REPLY. Обмен сообщениями микроядра построен на трех группах вызовов native API QNX (рис. 5.1):

1. **Принять сообщение**. Процесс[1], являющийся сервером некоторой услуги, выполняет вызов группы `MsgReceive*()`[2], фактически сооб-

---

[1] Естественно, поток, один из потоков процесса, но оригинальные описания логики обмена сообщениями сформулированы в терминологии процессов, и мы не станем отходить от этой традиции. Это обусловлено, скорее, преемственностью изложений с предыдущими реализациями ОС QNX – 4.X и более ранними, т. к. логика функционирования обмена сообщениями остается практически неизменной на протяжении более 20 лет развития линии QNX.

[2] Как и функции `spawn*()` и `exec*()`, API обмена сообщениями предоставляет целые группы вызовов, различающихся суффиксами имен и форматами входных параметров.

щая этим о готовности обслуживать запрос клиента, и переходит при этом в блокированное состояние со статусом RECEIVE, ожидая прихода клиентского запроса.

2. **Послать сообщение.** Клиентский процесс запрашивает эту услугу, посылая сообщение вызовом `MsgSend*()`, и переходит в блокированное состояние со статусом SEND. Переход осуществляется обычно на очень короткое время, пока сервер не примет его сообщение и не начнет обработку. Как только сервер принимает посланное сообщение, он разблокируется и меняет статус с RECEIVE на READY. Сервер начинает обработку полученного сообщения, а статус блокировки клиентского процесса меняется на REPLY.

3. **Ответить на полученное сообщение.** Завершив обработку полученного на предыдущем шаге сообщения, сервер выполняет вызов группы `MsgReply*()` для передачи запрошенного результата ожидающему клиенту. После этого вызова клиент, блокированный на вызове `MsgSend*()` со статусом REPLY, разблокируется (переходит в состояние READY). После выполнения `MsgReply*()` сервер также переходит в состояние READY. Однако чаще всего сервер снова входит в блокированное состояние на вызове `MsgReceive*()`, поскольку его работа организована как бесконечный цикл.

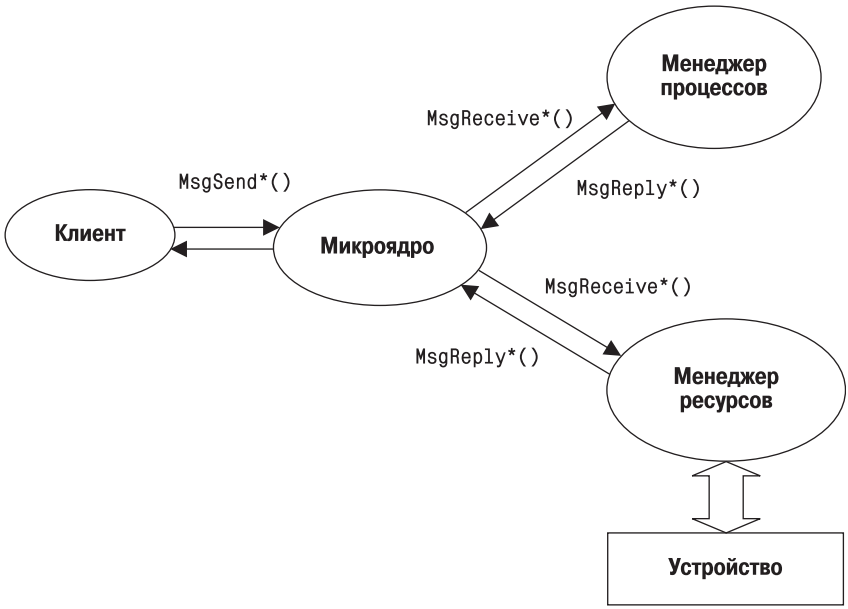

***Рис. 5.1.** Обмен сообщениями микроядра и менеджер ресурсов*

Уже из этого поверхностного описания понятно, что передача сообщений микроядра – это не только средство взаимодействия процессов

с обменом данными, но и крайне гибкая система синхронизации всех участников взаимодействия.

Могут возникнуть вопросы: Это один из низкоуровневых механизмов (существуют ли другие нативные механизмы?), на которых базируется ОС QNX? Какое это может иметь отношение к взаимодействиям на уровне POSIX API? Самое прямое! Все традиционные вызовы POSIX (`open()`, `read()`, ... и все другие) реализованы в ОС QNX как обмен сообщениями, который только «камуфлируется» под стандарты техникой использования менеджеров ресурсов, о которых разговор еще впереди.

Технология обмена сообщениями микроядра хорошо описана [1] и требует для своего понимания и освоения тщательного изучения. В этой же главе, посвященной совершенно другим предметам, мы не будем детально описывать эту технологию.

Остановимся только на одном обстоятельстве: адресат получателя, которому направляется каждое сообщение, определяется при начальном установлении идентификатора соединения (coid – connect ID) вызовом:

```
#include <sys/neutrino.h>
int ConnectAttach( int nd, pid_t pid, int chid,
                   unsigned index, int flags );
```

Адрес назначения (сервера) в этом вызове определяется триадой { ND / PID / CHID }, где:

`nd` – идентификатор сетевого узла. Мы не станем углубляться в идентификацию сетевых узлов сети QNET. Возьмем на заметку лишь тот факт, что обмен сообщениями с одинаковой легкостью осуществляется как с процессом на локальном узле (`nd` = 0), так и на любом другом сетевом узле.

`pid` – PID процесса-сервера, с которым производится соединение.

`chid` – идентификатор канала, который открыл процесс с указанным PID, выполнив предварительно `ChannelCreate()`, и к которому устанавливается соединение вызовом `ConnectAttach()`.

Выше мы неоднократно отмечали, что с процессом как с пассивной субстанцией, вообще говоря, невозможно обмениваться сообщениями. Хотя в адресной триаде обмена фигурирует именно PID процесса! Это обстоятельство не меняет положения вещей: именно адресная компонента CHID и определяет тот поток (часто это может быть главный поток приложения), с которым будет осуществляться обмен сообщениями, а PID определяет то адресное пространство процесса, в которое направляется сообщение, адресованное CHID.

Детальнее это выглядит так: в коде сервера именно тот поток, который выполнит `MsgReceive*( chid, ... )`, и будет заблокирован в ожидании запроса от клиента `MsgSend*()`. Аналогично и в коде клиента вся последовательность выполнения блокировок, обозначенная выше, будет относиться именно к **потоку**, выполняющему последовательные операции:

```
coid = ConnectAttach( ..., chid, ... );
MsgSend*( coid, ... );
```

Содержимое двух предыдущих абзацев ни одной буквой не противоречит и не отменяет положения традиционного изложения [1] технологии обмена сообщениями микроядра. Тогда зачем же мы даем именно такую формулировку? Для того чтобы акцентировать внимание на том, что все блокированные состояния и их освобождение имеют смысл относительно потоков (и только потоков!), которые выполняют последовательность операций `MsgSend*()` – `MsgReceive*()` – `MsgReply*()` **(даже если это единственный поток – главный поток приложения, и тогда мы говорим о блокировании процессов). Проиллюстрируем сказанное следующим приложением (*файл n1.cc*):**

Обмен сообщениями и взаимные блокировки

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <inttypes.h>
#include <errno.h>
#include <iostream.h>
#include <pthread.h>
#include <signal.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>

static const int TEMP = 500;  // темп выполнения приложения
static int numclient = 1;     // число потоков клиентов

// многопотоковая версия вывода диагностики в поток:
iostream& operator <<( iostream& c, char* s ) {
   static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
   pthread_mutex_lock ( &mutex );
   c << s << flush;
   pthread_mutex_unlock ( &mutex );
   return c;
};

static uint64_t tb;     // временная отметка начала приложения

// временная отметка точки вызова:
inline uint64_t mark( void ) {
   // частота процессора:
   const static uint64_t cps =
      SYSPAGE_ENTRY ( qtime )->cycles_per_sec;
   return ( ClockCycles () - tb ) * 1000 / cps;
};

const int MSGLEN = 80;
```

```
// потоковая функция сервера:
void* server ( void* chid ) {
   int rcvid;
   char message[ MSGLEN ];
   while( true ) {
      rcvid = MsgReceive( (int)chid, message, MSGLEN, NULL );
      sprintf( message + strlen( message ), "[%07llu] ... ", mark() );
      delay( TEMP );              // имитация обслуживания
      sprintf( message + strlen( message ), "[%07llu]->", mark() );
      MsgReply ( rcvid, EOK, message, strlen( message ) + 1 );
   };
   return NULL;
};

// потоковая функция клиента:
void* client ( void* data ) {
   while( true ) {
      char message[ MSGLEN ];
      sprintf( message, "%d:\t[%07llu]->", pthread_self (), mark() );
      MsgSend( (int)data, message, strlen( message ) + 1, message, MSGLEN );
      sprintf( message + strlen( message ), "[%07llu]", mark() );
      cout << message << endl;
      static unsigned int seed = 0;
      delay( numclient*
      ( ( (long)rand_r( &seed ) * TEMP / RAND_MAX ) + TEMP )
           );          // имитация вычислений...
   };
   return NULL;
};

int main( int argc, char** argv ) {
   // 1-й параметр - число потоков клиентов:
   if( argc > 1 && atoi( argv[ 1 ] ) > 0 )
       numclient = atoi( argv[ 1 ] );
   tb = ClockCycles();
   int chid = ChannelCreate( 0 );
   if( pthread_create( NULL, NULL, server, (void*)chid ) != EOK )
      perror( "server create" ), exit( EXIT_FAILURE );
   for( int i = 0; i < numclient; i++ )
      if( pthread_create( NULL, NULL, client,
         (void*)ConnectAttach( 0, 0, chid, _NTO_SIDE_CHANNEL, 0 )
        ) != EOK )
         perror( "client create" ), exit( EXIT_FAILURE );
   sigpause ( SIGINT );
   return EXIT_SUCCESS;
};
```

Все происходит в рамках единого процесса:

- **Создается единый поток сервера, ожидающий сообщений от клиентов и отвечающий на них.**

- Создается N потоков клиентов (задается параметром командной строки запуска приложения), которые будут обращаться к серверу.
- К одному каналу сервера устанавливается N соединений от клиентов.
- Канал прослушивания для сервера и идентификаторы соединений для клиентов сознательно создаются в главном потоке (т. е. вне потоков, которые их будут использовать); их значения поступают в потоки (сервера и клиентов) как параметры потоковых функций (трюк с подменой целочисленных значений на указатели мы рассматривали ранее).
- Сообщение продвигается от клиента к серверу и обратно к клиенту; в ходе пересылки объем сообщения нарастает: оно образуется конкатенацией полей, добавляемых последовательно клиентом, сервером и снова клиентом.
- В результате полного цикла обмена сообщением в теле самого сообщения формируется текст, содержащий 5 последовательных полей – идентификатор потока клиента (обращающегося с сообщением) и 4 абсолютные временные метки (в миллисекундах): передачи сообщения клиентом, приема сообщения сервером (начало обработки), ответа на сообщение сервером (конец обработки), приема ответа клиентом.

Запустим полученное приложение, например, так:

```
# n1 5
```

И прежде чем обсуждать результаты его работы, понаблюдаем состояния (блокировки) его потоков командой `pidin` (с другого терминала, естественно). Вот несколько «снимков» состояний, здесь можно наблюдать весь спектр блокированных состояний, о которых говорилось выше:

```
5546027   1 ./n1              10r SIGSUSPEND
5546027   2 ./n1              10r NANOSLEEP
5546027   3 ./n1              10r NANOSLEEP
5546027   4 ./n1              10r SEND        5546027
5546027   5 ./n1              10r REPLY       5546027
5546027   6 ./n1              10r NANOSLEEP
5546027   7 ./n1              10r NANOSLEEP

5730347   1 ./n1              10r SIGSUSPEND
5730347   2 ./n1              10r RECEIVE     1
5730347   3 ./n1              10r NANOSLEEP
5730347   4 ./n1              10r NANOSLEEP
5730347   5 ./n1              10r NANOSLEEP
5730347   6 ./n1              10r NANOSLEEP
5730347   7 ./n1              10r NANOSLEEP
```

А теперь рассмотрим результаты выполнения (на меньшем числе потоков клиентов, которое легче анализировать):

```
# n1 3
3:      [0000000]->[0000000] ... [0000501]->[0000501]
4:      [0000000]->[0000501] ... [0001003]->[0001003]
5:      [0000000]->[0001003] ... [0001505]->[0001505]
3:      [0002003]->[0002003] ... [0002504]->[0002505]
5:      [0003462]->[0003462] ... [0003964]->[0003964]
4:      [0003485]->[0003964] ... [0004466]->[0004466]
3:      [0005017]->[0005017] ... [0005518]->[0005518]
5:      [0005624]->[0005624] ... [0006126]->[0006126]
4:      [0006741]->[0006741] ... [0007243]->[0007243]
...
```

Видно, как 3 клиента отправляют сообщения одновременно ([0000000]), поток сервера (TID = 2) немедленно получает сообщение ([0000000], 1-я строка), отправленное клиентом с TID = 3, два других сообщения (от клиентов с TID = 4 и 5) помещаются системой в очередь обслуживания (строки 2 и 3). После завершения обслуживания запроса от TID = 3 и ответа ([0000501]) поток сервера получает (извлекается из очереди ранее отправленное сообщение) сообщение от TID = 4 и так далее.

Еще содержательнее для интерпретации становится картина для большего числа потоков клиентов (здесь очередь ожидающих запросов становится гораздо длиннее, а ее поведение трудно предсказуемым – почти каждый запрос ожидает обслуживания), но эти результаты требуют намного более тщательного разбора для их осмысления:

```
# n1 10
3:      [0000000]->[0000000] ... [0000501]->[0000501]
4:      [0000000]->[0000501] ... [0001003]->[0001003]
5:      [0000000]->[0001003] ... [0001505]->[0001505]
6:      [0000000]->[0001505] ... [0002007]->[0002007]
7:      [0000000]->[0002007] ... [0002508]->[0002508]
8:      [0000000]->[0002508] ... [0003010]->[0003010]
9:      [0000000]->[0003010] ... [0003512]->[0003512]
10:     [0000000]->[0003512] ... [0004014]->[0004014]
11:     [0000000]->[0004014] ... [0004516]->[0004516]
12:     [0000000]->[0004516] ... [0005017]->[0005018]
3:      [0005501]->[0005501] ... [0006003]->[0006003]
5:      [0008024]->[0008024] ... [0008526]->[0008526]
7:      [0008038]->[0008526] ... [0009028]->[0009028]
4:      [0009273]->[0009273] ... [0009775]->[0009775]
6:      [0010377]->[0010377] ... [0010878]->[0010878]
8:      [0010590]->[0010878] ... [0011380]->[0011380]
9:      [0010952]->[0011380] ... [0011882]->[0011882]
12:     [0011297]->[0011882] ... [0012384]->[0012384]
11:     [0011356]->[0012384] ... [0012886]->[0012886]
10:     [0012024]->[0012886] ... [0013387]->[0013388]
3:      [0012874]->[0013388] ... [0013889]->[0013889]
7:      [0014888]->[0014888] ... [0015390]->[0015390]
4:      [0016254]->[0016254] ... [0016756]->[0016756]
5:      [0017646]->[0017646] ... [0018148]->[0018148]
```

```
6:      [0019088]->[0019088] ... [0019590]->[0019590]
11:     [0020206]->[0020206] ... [0020708]->[0020708]
8:      [0020320]->[0020708] ... [0021210]->[0021210]
10:     [0021078]->[0021210] ... [0021712]->[0021712]
12:     [0021384]->[0021712] ... [0022213]->[0022213]
7:      [0021630]->[0022213] ... [0022715]->[0022715]
9:      [0021811]->[0022715] ... [0023217]->[0023217]
3:      [0022009]->[0023217] ... [0023719]->[0023719]
```

Динамический пул потоков

Динамический пул потоков не является каким-то специфическим механизмом, продиктованным именно микроядерной архитектурой QNX. Это удачная искусственная конструкция, все определения которой размещены в файле <sys/dispatch.h>. Удивительно не то, что в составе API QNX имеется такой механизм, а то, что подобные инструменты отсутствуют в других ОС.

В предыдущих примерах кода мы неоднократно создавали наборы потоков для тех или иных целей, но всем им было присуще одно: общее количество потоков в них было фиксированным на момент создания. Это и были **статические** пулы потоков, разделяющих между собой работу приложения. Архитекторы QNX идут чуть дальше: они предоставляют инструментарий для создания пулов **однотипных** (с общей функцией потока) потоков, в которых конкретное число потоков может увеличиваться или уменьшаться синхронно с изменением нагрузки на приложение. Именно своим **динамическим** составом эта конструкция и отличается.

Динамический пул потоков нужен разработчикам QNX в первую очередь как инструмент построения многопоточных менеджеров ресурсов – основы построения сервисов ОС QNX. Но и помимо этой цели динамический пул потоков представляет собой мощнейшее средство для конструирования параллельных механизмов обработки.

Проиллюстрируем применение динамического пула потоков примером программного кода, который был нами описан в книге [4] в главе «Сервер TCP/IP... много серверов хороших и разных». По сути, это ретранслирующий TCP/IP-сервер, но сейчас это для нас неважно:

Сервер на базе динамического пула потоков

```
#include <pthread.h>
#include <sys/dispatch.h>

static int ls;          // прослушивающий TCP-сокет

THREAD_POOL_PARAM_T* alloc( THREAD_POOL_HANDLE_T* h ) {
   return (THREAD_POOL_PARAM_T*)h;
};
```

```
// функция блокирования пула потоков
THREAD_POOL_PARAM_T* block( THREAD_POOL_PARAM_T* p ) {
   int rs = accept( ls, NULL, NULL );
   if( rs < 0 ) errx( "accept error" );
   return(THREAD_POOL_PARAM_T*)rs;
};

int handler( THREAD_POOL_PARAM_T* p ) {
   retrans( (int)p );
   close( (int)p );
   delay( 250 );
   cout << pthread_self() << flush;
   return 0;
};

int main( int argc, char* argv[] ) {
   // создать TCP-сокет на порт
   ls = getsocket( THREAD_POOL_PORT );
   // создание атрибутной записи пула потоков:
   thread_pool_attr_t attr;
   memset( &attr, 0, sizeof( thread_pool_attr_t ) );
   // заполнение блока атрибутов пула:
   /* - min число блокированных потоков в пуле */
   attr.lo_water = 3;
   /* - max число блокированных потоков в пуле */
   attr.hi_water = 7;
   /* - инкремент шага создания потоков       */
   attr.increment = 2;
   attr.maximum = 9;
   /* - общий предел числа потоков в пуле      */
   attr.handle = dispatch_create();
   attr.context_alloc = alloc;
   attr.block_func = block;
   attr.handler_func = handler;
   // фактическое создание пула потоков:
   void* tpp = thread_pool_create( &attr, POOL_FLAG_USE_SELF );
   if( tpp == NULL ) errx( "create pool" );
   // начало функционирования пула потоков:
   thread_pool_start( tpp );
   // ... выполнение никогда не дойдет до этой точки!
   exit( EXIT_SUCCESS );
};
```

Примечание

В примере используются, но не определены две функции, которые не столь существенны для понимания примера с точки зрения функционирования пула:

- `errx()` – реакция на ошибку выполнения с выводом сообщения и последующим аварийным завершением;
- `retrans()` – прием сообщения с присоединенного TCP-сокета с последующей ретрансляцией полученного содержимого в него же.

Итак, первая особенность пула потоков в том, что мы построили многопоточный сервер, почти не прописывая собственного кода, – большую часть рутинной работы за нас сделала библиотека пула.

Приведем описание логики работы пула потоков и показанного примера на самом качественном, простейшем уровне:

- Первоначально (при запуске пула потоков в работу вызовом `thread_pool_start()`) создается `attr.lo_water` потоков («нижняя ватерлиния» числа блокированных потоков).
- При создании любого потока (как в процессе начального, так и в процессе последующего создания) вызывается функция `attr.context_alloc()` (в контексте созданного потока).
- По завершении функция вызывает блокирующую функцию потока `attr.block_func()`, на которой созданный поток ожидает события активизации (в показанном примере событие активизации – это установление соединения новым клиентом по возврату из `accept()`).
- Блокирующая функция после наступления события активизации переведет поток в состояние READY и вызовет в контексте этого потока функцию обработчика `attr.handler_func()`.
- Если после предыдущего шага число оставшихся заблокированных потоков станет ниже `attr.lo_water`, механизм пула создаст дополнительно `attr.increment` потоков и «доведет» их до блокирующей функции.
- Активизированный поток производит всю обработку, предписанную функцией потока, и после выполнения потоковой функции будет опять переведен в блокированное состояние в функции блокирования...
- ...но перед переводом потока вновь в блокированное состояние проверяется, не будет ли при этом превышено число блокированных потоков `attr.hi_water` («верхняя ватерлиния»), и если это имеет место, то поток вместо перевода в блокированное состояние самоуничтожается.
- Все проверки числа потоков производятся для того, чтобы общее число потоков пула (т. е. число активизированных потоков вместе с блокированными) не превышало общее ограничение `attr.maximum`.

Разобрав общую логику функционирования пула потоков, можно теперь детальнее рассмотреть отдельные шаги всего процесса:

1. Прежде чем создавать пул потоков, мы должны создать атрибутную запись, определяющую все поведение пула. Атрибутная запись описана так (`<sys/dispatch.h>`):

```
typedef struct _thread_pool_attr {
   THREAD_POOL_HANDLE_T* handle;
   THREAD_POOL_PARAM_T*
           (*block_func)(THREAD_POOL_PARAM_T* ctp);
```

```
    void    (*unblock_func)(THREAD_POOL_PARAM_T* ctp);
    int     (*handler_func)(THREAD_POOL_PARAM_T* ctp);
    THREAD_POOL_PARAM_T*
            (*context_alloc)(THREAD_POOL_HANDLE_T* handle);
    void    (*context_free)(THREAD_POOL_PARAM_T* ctp);
    pthread_attr_t*       attr;
    unsigned short        lo_water;
    unsigned short        increment;
    unsigned short        hi_water;
    unsigned short        maximum;
    unsigned              reserved[8];
} thread_pool_attr_t;
```

Дескриптор создаваемого пула потоков `handle`, посредством которого мы будем ссылаться на пул, является просто синонимом типа `dispatch_t`:

```
#ifndef THREAD_POOL_HANDLE_T
  #define THREAD_POOL_HANDLE_T   dispatch_t
#endif
```

Атрибуты потоков, которые будут работать в составе пула, определяются полем `attr` типа `pthread_attr_t` (эту структуру мы детально рассматривали ранее при обсуждении создания единичных потоков).

Численные параметры пула определяют:

`lo_water` — «нижняя ватерлиния», минимальное число потоков пула, находящихся в блокированном состоянии (в ожидании активизации). Если в результате некоторого события один из ожидающих потоков переходит в состояние активной обработки и число оставшихся блокированных потоков становится меньше `lo_water`, создается дополнительно `increment` потоков, которые переводятся в блокированное состояние.

`hi_water` — максимальное число потоков, которые допустимо иметь в блокированном состоянии. Если после завершения обработки некоторым потоком число заблокированных потоков становится больше `hi_water`, то этот поток уничтожается.

`maximum` — общая верхняя граница числа потоков пула (активизированных и заблокированных). Даже если число заблокированных потоков (в пике активности) станет ниже `lo_water`, но общее число потоков уже достигнет `maximum`, то новые потоки для пула создаваться не будут.

Функциональные параметры пула определяют:

`context_alloc()` и `context_free()` — функции создания и уничтожения контекста потока, которые вызываются при создании и уничтожении каждого потока пула. Функция создания контекста потока ответственна за индивидуальные настройки создаваемого потока. Она возвращает «указатель на контекст» типа `THREAD_POOL_PARAM_T`. Однако системе такой тип неизвестен:

```
#ifndef THREAD_POOL_PARAM_T
  #define THREAD_POOL_PARAM_T   void
#endif
```

В качестве контекста может использоваться любой пользовательский тип, и он будет передаваться последовательно в качестве параметра (ctp) во все последующие функции обслуживания потока.

block_func() – функция блокирования, которая вызывается в потоке сразу же после context_alloc() или после очередного этапа выполнения потоком функции обработчика handler_func(). Функция блокирования получает и возвращает далее обработчику (возможно, после модификации) структуру контекста (в приведенном выше примере контекстом является int – значение присоединенного TCP-сокета).

handler_func() – это, собственно, и есть аналог потоковой функции, в которой выполняется вся полезная работа потока. Функция вызывается библиотекой после выхода потока из блокирующей функции block_func(), при этом функция-обработчик handler_func() получит параметр контекста, возвращенный block_func().

Примечание

В текущей реализации handler_func() должна возвращать 0; все другие значения зарезервированы для дальнейших расширений. Аналогично определенная в атрибутной записи функция unblock_func() зарезервирована для дальнейших расширений, и вместо ее адреса следует устанавливать NULL.

2. После создания атрибутной записи пула, определяющей всю функциональность его дальнейшего поведения, можно приступать к непосредственному созданию пула потоков:

```
thread_pool_t* thread_pool_create(
                        thread_pool_attr_t* attr, unsigned flags );
```

где attr – подробно рассмотренная (и созданная) ранее атрибутная запись пула;

flags – флаг, определяющий поведение вызывающего потока после последующего вызова thread_pool_start(). В документации описано два возможных значения флага:

- POOL_FLAG_EXIT_SELF – после старта пула поток, вызвавший thread_pool_start() (часто это главный поток приложения), завершается;
- POOL_FLAG_USE_SELF – после старта пула поток, вызвавший thread_pool_start(), включается в пул в качестве одного из его потоков.

И в том и в другом случае в типовом фрагменте (как и в показанном выше примере):

```
thread_pool_start( tpp );
exit( EXIT_SUCCESS );
```

управление никогда не дойдет до выполнения exit(). Но существует еще третье допустимое значение, прямо не указанное в документации, но мельком упоминаемое в других местах документации:

- 0 – после старта пула поток, вызвавший thread_pool_start(), продолжает свое естественное выполнение.

Например, некоторый фрагмент кода мог бы выглядеть так:

```
thread_pool_attr_t att   // ...
thread_pool_t *tpp = thread_pool_create( &attr, 0 );
thread_pool_start( tpp );
wyile( true ){
  // выполнять некоторую отличную от пула работу
};
exit( EXIT_SUCCESS );
```

Как уже понятно из описаний, thread_pool_create() возвращает указатель на управляющую структуру пула потоков, которая позже будет передана thread_pool_start(). Если создание пула завершилось неудачей, то результатом выполнения будет NULL, а в errno будет установлен код ошибки (документацией предусмотрен только один код ошибки: ENOMEM – недостаточно памяти для размещения структур данных).

Примечание

Управляющая структура пула потоков описана так:

```
typedef struct _thread_pool   thread_pool_t;
struct _thread_pool {
   thread_pool_attr_t       pool_attr;
   unsigned                 created;
   unsigned                 waiting;
   unsigned                 flags;
   unsigned                 reserved[3];
};
```

3. Последний шаг в процедуре запуска пула потоков:

```
int thread_pool_start( void* pool );
```

где pool – это указатель, возвращаемый thread_pool_create().[1]

При успешном завершении (которого почти никогда не происходит, за исключением значения флага 0; об этом см. выше) функция возвращает EOK, в противном случае (что происходит гораздо чаще) – значение -1.

[1] Вы спросите, почему указатель, возвращаемый thread_pool_create(), имеет тип thread_pool_t*, а получающий его параметр thread_pool_start() определен как void*? Это или неаккуратность разработчиков QNX, или глубокая сермяжная правда, которую мы пока не понимаем.

4. Другие, относящиеся к библиотеке динамического пула потоков функции, которые целесообразно посмотреть в документации QNX (но которые в силу различных обстоятельств используются гораздо реже):

```
int thread_pool_destroy( thread_pool_t* pool );

int thread_pool_control( thread_pool_t* pool,
                         thread_pool_attr_t* attr,
                         _Uint16t lower, _Uint16t upper,
                         unsigned flags );

int thread_pool_limits( thread_pool_t* pool,
                        int lowater, int hiwater,
                        int maximum, int increment,
                        unsigned flags );
```

Менеджеры ресурсов

QNX вводит технику программирования, которая единообразно проходит сквозь всю систему.[1] Идея техники менеджеров ресурсов столь же проста, сколь и остроумна:

- Вся система построена на тщательно проработанной в теории (и редко используемой при построении реальных ОС) концепции – коммутации сообщений. Ядро (точнее «микроядро») операционной системы при таком подходе выступает в качестве компактного коммутатора сообщений между взаимодействующими программными компонентами. При этом взаимодействующие компоненты выступают в качестве клиента, запрашивающего услугу (ресурс), и сервера, обеспечивающего эту услугу (обслуживающего ресурс).
- Большинство системных вызовов API (в том числе все «привычные» POSIX-вызовы: `open()`, `read()`, `write()`, `seek()`, `close()`...) реально посылаются обслуживающему данный ресурс сервису (например, в файловую систему типа FAT32 – `fs-dos`) в виде сообщений уровня микроядра. Код сообщения при этом определяет тип операции (например, `open()`), а последующее тело сообщения – конкретные параметры запроса, зависящие от типа операции (параметры запроса пакуются в тело сообщения).
- Раз эта схема столь универсальна, единообразна и не зависит от конкретной природы ресурса, на котором обеспечивается обслуживание, то разработчики QNX предоставляют некоторый шаблон сер-

[1] Эта техника возникла не «сразу» и не случайно: ее идеология практически сложилась за почти 20 лет развития системы QNX, но не была представлена в виде формальных механизмов. В QNX 6.X оставалось только придать ей формальный вид и обеспечить ее поддержание написанием специальных библиотек.

вера, в котором на месте обработчиков стандартных POSIX-запросов находятся пустые программные заглушки. Этот шаблон и служит базовым элементом построения разнообразных серверов услуг, называемых при выполнении в такой технике «менеджерами ресурса».

- При запуске программа менеджера ресурса регистрирует свое имя (точнее имя управляемого ею ресурса) в пространстве имен файловой системы QNX (обычно в каталоге `/dev`, но это может быть любое место файловой системы). Теперь можно обращаться с запросами к данному менеджеру так же, как и к любому реальному файлу в файловой системе.
- Программисту, пишущему свой драйвер услуги, ресурса, устройства или псевдоустройства, остается только переопределить программное наполнение тех программных заглушек, которые ответственны за интересующие его вызовы (например, `open()`, `read()`, `close()`), никак не затрагивая вызовы, не обеспечиваемые этим ресурсом (например, `write()`, `seek()` и др.).

В наши цели не входит детальное обсуждение техники написания менеджеров ресурсов (этому посвящено специальное исчерпывающее руководство в составе технической документации QNX объемом более 80 страниц[1]). Поэтому, как и ранее с динамическим пулом потоков, начнем с примера. Приведем простейший код менеджера ресурса, который использовался нами для тестирования наследования приоритетов в QNX (*файл prior.cc*):

Однопоточный менеджер ресурса

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

// обработчик запроса от клиента read(),
// возвращающий текущий приоритет обслуживания
static int prior_read( resmgr_context_t *ctp, io_read_t *msg,
                       RESMGR_OCB_T *ocb ) {
   static bool odd = true;
   int status = iofunc_read_verify( ctp, msg, ocb, NULL );
   if( status != EOK ) return status;
   if( msg->i.xtype & _IO_XTYPE_MASK != _IO_XTYPE_NONE )
```

[1] В Интернете доступен прекрасный перевод этого документа, выполненный Владимиром Зайцевым и отредактированный к публикации Сергеем Малышевым: *http://qnxclub.net/files/articles/resmgr/resmgr.pdf.gz*.

```
      return ENOSYS;
   if( odd ) {
      struct sched_param param;
      sched_getparam( 0, &param );
      static char rbuf[ 4 ];
      sprintf( rbuf, "%d\n", param.sched_curpriority );
      MsgReply( ctp->rcvid, strlen( rbuf ) + 1,
                     rbuf, strlen( rbuf ) + 1 );
   }
   else  MsgReply( ctp->rcvid, EOK, NULL, 0 );
   odd = !odd;
   return  _RESMGR_NOREPLY;
};

// главная программа запуска менеджера
main( int argc, char **argv ) {
   resmgr_attr_t        resmgr_attr;
   dispatch_t           *dpp;
   dispatch_context_t   *ctp;
   int                  id;
   // инициализация интерфейса диспетчеризации
   if( ( dpp = dispatch_create() ) == NULL )
      perror( "allocate dispatch" ), exit( EXIT_FAILURE );
   // инициализация атрибутов менеджера
   memset( &resmgr_attr, 0, sizeof resmgr_attr );
   resmgr_attr.nparts_max = 1;
   resmgr_attr.msg_max_size = 2048;
   // инициализация таблиц функций обработчиков
   static resmgr_connect_funcs_t  connect_funcs;
   static resmgr_io_funcs_t       io_funcs;
   iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
                     _RESMGR_IO_NFUNCS, &io_funcs );
   // здесь нами дописан всего один обработчик - операции read,
   // все остальное делается менеджером по умолчанию!
   io_funcs.read = prior_read;
   // инициализация атрибутной структуры, используемой
   // устройством:
   static iofunc_attr_t  attr;
   iofunc_attr_init( &attr, S_IFNAM | 0666, 0, 0 );
   // здесь создается путевое имя для менеджера
   id = resmgr_attach( dpp, &resmgr_attr, "/dev/prior",
                       _FTYPE_ANY, 0, &connect_funcs,
                       &io_funcs, &attr );
   if( id == -1 )
      perror( "attach name" ), exit( EXIT_FAILURE );
   ctp = dispatch_context_alloc( dpp );
   // старт менеджера как бесконечный цикл ожидания
   // поступающих сообщений для диспетчеризации:
   while( true ) {
      if( ( ctp = dispatch_block( ctp ) ) == NULL )
         perror( "block error" ), exit( EXIT_FAILURE );
```

```
        dispatch_handler( ctp );
    }
};
```

Здесь использован простейший однопоточный шаблон написания менеджера. Менеджер отрабатывает только одну команду `read()` (т. е. отрабатывает нестандартно; в целевом коде все остальные команды, например `open()`, он отрабатывает по умолчанию). По команде `read()` менеджер: а) возвращает в виде текстовой строки, завершающейся переводом строки, текущий приоритет (помните, что в QNX приоритеты «плавают»?), на котором он обрабатывает запрос, и б) делает это через один запрос, в оставшиеся разы создавая на всякий случай (почему «на всякий», сейчас станет понятно) ситуацию EOF (конца файла). Выполним несколько команд:

```
# prior &
# ls -l /dev/pr*
nrw-rw-rw- 1 root root 0 Dec 18 17:13 /dev/prior
```

Все соответствует нашим ожиданиям: менеджер ресурса запущен, он зарегистрировал в пространстве имен свое имя `/dev/prior`, по которому мы можем к нему обращаться. Теперь выполним обращения к нашему... «устройству». Для этого мы сознательно не станем пользоваться каким-либо специальным клиентом, запрашивающим наш созданный сервис, а воспользуемся самыми заурядными командами UNIX, которые ничего не подозревают о существовании нового сервиса:

```
# cat /dev/prior
10
# nice -n-5 cat /dev/prior
15
# nice -n-19 cat /dev/prior
29
```

Вот здесь и проявляется исключительная мощь техники написания менеджера ресурса: созданная минимальными средствами серверная служба «камуфлирует» специфичный QNX-механизм передачи сообщений микроядра под стандартные POSIX-запросы к файловой системе (`open()`, `read()` и т. д.), и стандартные команды UNIX «не видят» отличий новой серверной службы от стандартных файлов (устройств) UNIX. Вот для достижения такой полной совместимости с «привычками» команд UNIX и созданы «на всякий случай» те особенности формата, возвращаемого запросами `read()`, о которых упоминалось выше.

Теперь разработка, например драйвера некоторого специфичного устройства, перемещается из области шаманства «системного программиста» в область деятельности проблемного программиста, да и выполняется привычными высокоуровневыми инструментальными средствами, например C++.

Примечание

Пользуясь случаем, именно здесь уместно на примере созданного менеджера ресурсов продемонстрировать гибкость микроядерной архитектуры и техники менеджера ресурса, а заодно убедиться, что наследование приоритетов (критически важное свойство для систем реального времени) сохраняется при запросе к удаленному менеджеру ресурса, запущенному на другом узле сети (имя узла – rtp):

```
# on -frtp prior &
# ls -l /net/rtp/dev/pr*
nrw-rw-rw- 1 root root 0 Dec 18 17:09 /net/rtp/dev/prior
# nice -n-5 cat /net/rtp/dev/prior
15
# nice -n-19 cat /net/rtp/dev/prior
29
```

Многопоточный менеджер

Следующим шагом развития техники менеджера ресурсов является многопоточный менеджер. Фактически это объединение техники менеджера ресурсов с динамическим пулом потоков, рассмотренным выше.

Реальный работающий многопоточный менеджер с сопутствующим ему обстоятельным обсуждением приводился нами в книге [4] в главе «Драйверы». Мы не станем полностью приводить здесь этот достаточно объемный текст, поскольку он отличается от ранее показанного однопоточного менеджера только несколькими строками после вот этого оператора регистрации префикса имени менеджера:

```
// здесь создается путевое имя для менеджера
id = resmgr_attach( dpp, &resmgr_attr, "/dev/prior",
     _FTYPE_ANY, 0, &connect_funcs, &io_funcs, &attr );
if( id == -1 )
   perror( "attach name" ), exit( EXIT_FAILURE );
```

Вот те несколько строк, которые, собственно, и превращают однопоточный менеджер в многопоточный:

```
...
thread_pool_attr_t  pool_attr;
memset( &pool_attr, 0, sizeof pool_attr );
pool_attr.handle = dpp;
// это всегда остается так... :
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func = dispatch_block;
pool_attr.handler_func = dispatch_handler;
pool_attr.context_free = dispatch_context_free;
// численные параметры пула:
pool_attr.lo_water = 2;
pool_attr.hi_water = 6;
pool_attr.increment = 1;
pool_attr.maximum = 50;
```

```
    thread_pool_t  *tpp;
    // флаг создания пула, который может принимать значения:
    // POOL_FLAG_EXIT_SELF, POOL_FLAG_USE_SELF или,
    // наконец, 0 и который определяет, что будет
    // происходить дальше с вызывающим потоком...
    if( ( tpp = thread_pool_create( &pool_attr, POOL_FLAG_EXIT_SELF ) )
         == NULL )
       perror( "create pool" ), exit( EXIT_FAILURE );
    thread_pool_start( tpp );
    ...
};
```

Но всю эту последовательность действий мы уже видели ранее при описании динамического пула потоков, и какого-то специфического отношения к созданию именно менеджера ресурса она не имеет.

Вот такими элементарными манипуляциями мы превращаем менеджер ресурса (практически любой менеджер!) в многопоточный. С другой стороны, простота трансформации одной формы в другую подсказывает простое и эффективное решение: вначале всегда пишите однопоточный менеджер, поскольку в отладке и понимании он намного проще, и только потом при необходимости трансформируйте его в многопоточный.

Множественные каналы

Техника написания менеджеров ресурсов в QNX открывает перспективу для простого и ясного написания драйверов системы без необходимости «залезать» в специфические низкоуровневые детали. Тем не менее в описаниях технологии создания менеджеров ресурсов есть один аспект, который имеет непосредственное отношение к синхронизации параллельных ветвей, и нельзя сказать, что этот вопрос не освещен в технической документации, однако его составляющие детали «размазаны» по документации, и общую картину приходится восстанавливать.

Суть вопроса в следующем. Писать менеджер ресурсов как системный драйвер некоторого специфического аппаратного устройства – это удел единиц (на каждое устройство – по одному разработчику! ... шутка), но менеджер ресурсов – это прекрасная альтернатива для описания чисто программных «псевдоустройств». Например, это могла бы быть некоторая оконная GUI-подсистема, в которой `open()` создает прорисовку окна на экране, `write()` вписывает некоторый текст в окно, а `read()` считывает из окна текст, вводимый пользователем (подобная конструкция описывалась нами в главе «Драйверы» [4]). Таким решением мы с минимальными затратами придаем POSIX-функциональность своим совершенно неожиданным программным подсистемам.

Однако для «истинных драйверов» запросы `open()` – `read()` – `write()` должны, как правило, быть последовательными (право, бессмысленно

пытаться писать и читать один файл одновременно из двух потоков)... Это обусловливается тем, что в конечном итоге все функции-обработчики операций менеджера ресурса выходят на единичный экземпляр оборудования, которое должно физически отработать переданный ему запрос.

Гораздо свободнее может себя чувствовать разработчик драйвера псевдоустройства (программной модели): здесь каждый запрос `open()` (будь то из одного последовательного потока, различных потоков процесса или даже из потоков, принадлежащих разным процессам) может порождать новый экземпляр псевдоустройства. Возвращаемый им файловый дескриптор (в QNX это дескриптор соединения) станет ссылаться на порожденный экземпляр, а вызовы `read()` – `write()`, оперирующие с различным дескриптором, будут направляться соответствующим различным экземплярам. (Понятно, что такой параллелизм операций может обеспечить только многопоточный менеджер ресурса, но нужно еще «заставить» его сделать это.)

Это настолько часто используемая модель, что она заслуживает отдельного рассмотрения. Дополнительную сложность создает то обстоятельство, что мы, как уже отмечалось, договорились писать программный код на C++, а здесь нам предстоит переопределять из своего кода определения в заголовочных файлах менеджера ресурсов, не нарушая их C-синтаксис.

Ниже показан текст простейшего многопоточного менеджера (исключены даже самые необходимые проверки), ретранслирующего по нескольким каналам независимо получаемые текстовые строки (строки кода, принципиальные для обеспечения параллельности и многоканальности, выделены жирным шрифтом):

Подмена стандартного Open Control Block

```
// предшествующие общие строки #include не показаны...
// это переопределение нужно для исключения предупреждений
// компилятора: 'THREAD_POOL_PARAM_T' redefined
#define THREAD_POOL_PARAM_T dispatch_context_t
#include <sys/dispatch.h>
// следующее переопределение принципиально важно:
// оно предписывает вместо стандартного блока OCB (open control block),
// создаваемого вызовом клиента open() и соответствующего его файловому
// дескриптору, использовать собственную структуру данных.
// Эта структура должна быть производной от стандартной
// iofunc_ocb_t, а определение должно предшествовать
// включению <sys/iofunc.h>.
#define IOFUNC_OCB_T struct ownocb
#include <sys/iofunc.h>

class ownocb : public iofunc_ocb_t  {
   static const int BUFSIZE = 1024;
public:
```

```
   char *buf;
   ownocb( void ) { buf = new char[  BUFSIZE ]; };
   ~ownocb( void ) { delete buf; };
};

IOFUNC_OCB_T *ownocb_calloc( resmgr_context_t *ctp, IOFUNC_ATTR_T *device ) {
   return new ownocb;
};
void ownocb_free( IOFUNC_OCB_T *o ) { delete o; };
iofunc_funcs_t ownocb_funcs = {
   _IOFUNC_NFUNCS, ownocb_calloc, ownocb_free
};
iofunc_mount_t mountpoint = { 0, 0, 0, 0, &ownocb_funcs };

// Вместо умалчиваемой операции iofunc_lock_ocb_default(),
// вызываемой перед началом обработки запросов чтения/записи
// и блокирующей атрибутную запись, мы предписываем вызывать
// "пустую" операцию и не блокировать атрибутную запись,
// чем обеспечиваем параллелизм.
static int nolock( resmgr_context_t *ctp, void *v, IOFUNC_OCB_T *ocb ) {
   return EOK;
};

// обработчик запроса чтения
static int line_read( resmgr_context_t *ctp,
                      io_read_t *msg,
                      IOFUNC_OCB_T *ocb ) {
   if( strlen( ocb->buf ) != 0 ) {
      MsgReply( ctp->rcvid, strlen( ocb->buf ) + 1,
                ocb->buf, strlen( ocb->buf ) + 1 );
      strcpy( ocb->buf, "" );
   }
   else MsgReply( ctp->rcvid, EOK, NULL, 0 );
   return  _RESMGR_NOREPLY;
};

// обработчик запроса записи
static int line_write( resmgr_context_t *ctp,
                       io_write_t *msg,
                       IOFUNC_OCB_T *ocb ) {
   resmgr_msgread( ctp, ocb->buf,
                   msg->i.nbytes, sizeof( msg->i ) );
   _IO_SET_WRITE_NBYTES( ctp, msg->i.nbytes );
   return  EOK;
};

// имя, под которым регистрируется менеджер:
const char sResName[ _POSIX_PATH_MAX + 1 ] = "/dev/wmng";

// старт менеджера ресурса
static void StartResMng( void ) {
   dispatch_t*  dpp;
```

```
    if( ( dpp = dispatch_create() ) == NULL )
       perror( "dispatch create" ), exit( EXIT_FAILURE );
    resmgr_attr_t  resmgr_attr;
    memset( &resmgr_attr, 0, sizeof resmgr_attr );
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;
    // статичность 3-х последующих описаний принципиально важна!
    // (также они могут быть сделаны глобальными переменными файла):
    static resmgr_connect_funcs_t  connect_funcs;
    static resmgr_io_funcs_t  io_funcs;
    static iofunc_attr_t  attr;
    iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
                      _RESMGR_IO_NFUNCS, &io_funcs );
    // переопределение обработчиков по умолчанию
    io_funcs.read = line_read;
    io_funcs.write = line_write;
    io_funcs.lock_ocb = nolock;
    iofunc_attr_init( &attr, S_IFNAM | 0666, NULL, NULL );
    // через это поле осуществляется связь с новой
    // структурой OCB:
    attr.mount = &mountpoint;
    if( resmgr_attach( dpp, &resmgr_attr, sResName,
                       _FTYPE_ANY, 0, &connect_funcs,
                        &io_funcs, &attr ) == -1 )
       perror( "name attach" ), exit( EXIT_FAILURE );

    // создание пула потоков (многопоточность)
    thread_pool_attr_t pool_attr;
    memset( &pool_attr, 0, sizeof pool_attr );
    pool_attr.handle = dpp;
    pool_attr.context_alloc = dispatch_context_alloc;
    pool_attr.block_func = dispatch_block;
    pool_attr.handler_func = dispatch_handler;
    pool_attr.context_free = dispatch_context_free;
    pool_attr.lo_water = 2;
    pool_attr.hi_water = 6;
    pool_attr.increment = 1;
    pool_attr.maximum = 50;
    thread_pool_t*  tpp;
    if( ( tpp = thread_pool_create( &pool_attr,
                          POOL_FLAG_EXIT_SELF ) ) == NULL )
       perror( "pool create" ), exit( EXIT_FAILURE );
    thread_pool_start( tpp );
    // ... к этой точке return управление уже никогда не подойдет...
};

int main( int argc, char *argv[] ) {
    // проверка, не загружен ли ранее экземпляр менеджера,
    // 2 экземпляра нам ни к чему...
    char sDirName[ _POSIX_NAME_MAX + 1 ];
    int nDirLen = strrchr( (const char*)sResName, '/' ) - (char*)sResName;
```

```
    strncpy( sDirName, sResName, nDirLen );
    sDirName[ nDirLen ] = '\0';
    DIR *dirp = opendir( sDirName );
    if( dirp == NULL )
       perror( "directory not found" ), exit( EXIT_FAILURE );
    struct dirent *direntp;
    while( true ) {
       if( ( direntp = readdir( dirp ) ) == NULL ) break;
       if( strcmp( direntp->d_name, strrchr( sResName, '/' ) + 1 ) ==  0 )
          cout << "second copy of manager" << endl,
          exit( EXIT_FAILURE );
    };
    closedir( dirp );
    // старт менеджера
    StartResMng();
    // ... к этой точке мы уже никогда не подойдем...
    exit( EXIT_SUCCESS );
};
```

В отличие от типового и привычного шаблона многопоточного менеджера, мы проделали здесь дополнительно следующее:

- Определили собственную структуру OCB, новый экземпляр которой должен создаваться для каждого нового подключающегося клиента:

```
class ownocb : public iofunc_ocb_t  { ... };
```

- Переопределили описание структуры OCB, используемое библиотеками менеджера ресурсов:

```
#define IOFUNC_OCB_T struct ownocb
```

- Заполняя атрибутную запись устройства:

```
attr.mount = &mountpoint;
```

мы к точке монтирования «привязываем» функции создания и уничтожения вновь определенной структуры OCB (по умолчанию библиотека менеджера станет размещать только стандартный OCB):

```
iofunc_funcs_t ownocb_funcs = {
   _IOFUNC_NFUNCS, ownocb_calloc, ownocb_free
};
iofunc_mount_t mountpoint = { 0, 0, 0, 0, &ownocb_funcs };
```

(_IOFUNC_NFUNCS – это просто константа, определяющая число функций и равная 2.)

- Определяем собственные функции размещения и уничтожения структуры OCB с прототипами:

```
IOFUNC_OCB_T* ownocb_calloc( resmgr_context_t*, IOFUNC_ATTR_T* );
void ownocb_free( IOFUNC_OCB_T *o );
```

В нашем случае это: а) интерфейс из С-понятия «создать–удалить», в C++ – «конструктор–деструктор» и б) именно здесь создается

и инициализируется сколь угодно сложная структура экземпляра OCB.

- В функциях обработки запросов клиента (операций менеджера) мы позже будем в качестве 3-го параметра вызова обработчика получать указатель именно того экземпляра, для которого требуется выполнить операцию, например:

```
int read( resmgr_context_t*, io_read_t*, IOFUNC_OCB_T* ) {...}
```

Дополнительно мы проделываем еще один трюк, запрещая менеджеру блокировать атрибутную запись устройства при выполнении операций (что он делает по умолчанию; для реальных устройств это резонно, но для программного псевдоустройства это, как правило, не является необходимым). Для этого:

- В таблице операций ввода/вывода переназначаем функцию-обработчик операции блокирования атрибутной записи:

```
io_funcs.lock_ocb = nolock;
```

- В качестве такого обработчика предлагаем «пустую» операцию:

```
static int nolock( resmgr_context_t*, void*, IOFUNC_OCB_T* ) {
   return EOK;
};
```

Запустим менеджер и проверим, как происходит его установка в системе:

```
/dev # ls -l /dev/w*
nrw-rw-rw-  1 root      root       0 Nov 09 23:17 /dev/wmng
```

Теперь подготовим простейший клиент:

```
void main( int argc, char *argv[] ) {
   char sResName[ _POSIX_PATH_MAX + 1 ] = "/dev/wmng";
   if( argc > 1 ) strcpy( sResName, argv[ 1 ] );
   int df = open( sResName,  O_RDWR | O_NONBLOCK );
   if( df < 0 )
      perror( "device open" ), exit( EXIT_FAILURE );
   cout << "open " << sResName
        << " , desc. =  " << df << endl;
   char ibuf[ 2048 ], obuf[ 2048 ];
   int r, w;
   while( true ) {
      if( ( r = read( df, obuf, sizeof( obuf ) ) ) < 0 )
         break;
      cout << '#' << obuf << endl;
      cout << '>' << flush;
      cin >> ibuf;
      if( ( w = write( df, ibuf, strlen( ibuf ) + 1 ) ) <= 0 ) break;
   };
   if( r < 0 ) perror( "read error" );
   if( w <= 0 ) perror( "write error" );
```

```
    exit( EXIT_FAILURE );
};
```

Запустим одновременно 2 экземпляра клиента (их, собственно, может быть сколь угодно много) и убедимся, что каждый из клиентов работает со своей отдельной копией структур данных внутри процесса менеджера ресурса:

```
# wmclient
open /dev/wmng , desc. =  3
#
>1234
#1234
>54321
#54321
>

# wmclient
open /dev/wmng , desc. =  3
#
>qwerty
#qwerty
>asdf
#asdf
>
```

Отчетливо видно, что каждый клиент с получением своего файлового дескриптора (реально это дескриптор соединения) получает и свой экземпляр данных.

Полную параллельность и независимость обращений (например, возможность выполнения `read()` в то время, когда менеджер занят выполнением `read()` от другого клиента) к данному псевдоустройству отследить сложнее. Для этого в код обработчиков операций чтения/записи следует внести ощутимую задержку (например, `sleep()` или `delay()`) и воздействовать достаточно плотным потоком запросов со стороны нескольких клиентов. Такие эксперименты показывают полную независимость операций по разным файловым дескрипторам, что обеспечивается переопределением обработчика по умолчанию – `iofunc_lock_ocb_default()`.

Сообщения или менеджер?

Этот вопрос возникает (должен возникать!) у каждого, кто приступает к разработке реального проекта, особенно если функциональность проекта распределяется между несколькими автономными процессами. Такая структуризация и вовсе не привычна разработчикам, приходящим из мира Windows. Для UNIX создание проектов, в которых порождается несколько процессов, такая структуризация уже гораздо органичнее, но и там это чаще всего лишь клонирование образа единого сер-

верного процесса посредством `fork()`. QNX предоставляет возможность идти еще дальше в построении приложений, представленных (разделенных) как группа разнородных взаимодействующих **процессов**:

- Уже полученные нами ранее тестовые результаты времени диспетчеризации и переключений контекстов (пусть даже они и сделаны бегло, только в качестве оценочных ориентиров) показывают, что представления приложения в качестве единого, монолитного процесса или процесса, содержащего группу потоков, либо просто разбиение приложения на группу процессов по производительности если и не эквивалентны, то крайне близки. Этот фактор не должен быть определяющим, и при структурировании приложения следует руководствоваться целесообразностью и удобством.
- Процессы QNX сохраняют все качества таковых и в UNIX вообще: они являются изолированными сущностями, которые взаимодействуют, если это необходимо, используя достаточно тяжеловесные (расточительные) механизмы IPC. Собственно, в этом и ценность процессов с их изолированными адресными пространствами – это механизм обеспечения высокой надежности и живучести приложений. Но QNX, не сужая спектр общепринятых IPC-механизмов, привносит совершенно новый «слой» инструментария взаимодействия – обмен сообщениями микроядра. При этом «проницаемость» процессов как отдельных клеток живого организма становится много выше, нисколько не снижая их «защищенности».

Но у нас есть две принципиально различные альтернативы для выражения этого «слоя» взаимодействий в своем программном коде: базовый механизм обмена сообщениями (низкоуровневая техника, известная еще со времен QNX 4.X) и механизм менеджера ресурса. Делать выбор между ними приходится на этапе раннего эскизного проектирования системы, поскольку перестраивать систему с одного механизма на другой в ходе развития – достаточно трудоемкий процесс, который может потребовать пересмотра и архитектурных основ развиваемого проекта.

Поэтому, приступая к проектированию нового проекта, мы должны априорно, до начала фактической разработки, отчетливо представлять, что выигрываем и что проигрываем, используя тот или иной механизм реализации обмена сообщениями.

Две стороны единого механизма

При рассмотрении базовой для QNX (собственно, для всех микроядерных ОС) техники обмена сообщениями в сравнении с технологией написания менеджеров ресурсов не покидает ощущение поразительной схожести происходящих в обоих случаях процессов. В этом нет ничего удивительного, поскольку инструментарий менеджеров ресурсов – это только система внешних «оберток» над базовым механизмом обмена сообщениями.

Для эффективного применения той или иной альтернативной технологии мы должны иметь возможность проанализировать многие сравнительные показатели выбираемого инструментария: простота, гибкость, эффективность, трудоемкость реализации, возможности внесения изменений при развитии проекта и на этапе его последующего сопровождения. Этим мы и займемся в оставшейся части главы.

Простота и трудоемкость

Механизм прямого обмена сообщениями крайне просто выражается в программном коде. Когда достигнута полная ясность в значениях адресных параметров обмена, необходимо всего лишь несколько операторов, чтобы заставить все это «крутиться».

Со стороны сервера, например, это выглядит так:

```
int chid = ChannelCreate( 0 );
. . .
while( true ) {
   struct _msg_info info;
   int rcvid = MsgReceive( chid, &bufin, sizeof( bufin ), &info );
   if( rcvid < 0 ) exit( EXIT_FAILURE );
   if( MsgReply( rcvid, EOK, &bufou, sizeof( bufou ) < 0 )
      exit( EXIT_FAILURE );
   };
```

Со стороны клиента:

```
int coid = ConnectAttach( node, pid, chid, _NTO_SIDE_CHANNEL, 0 );
if( coid < 0 ) exit( EXIT_FAILURE );
. . .
while( . . . )
   if( MsgSend( coid,
                &bufou, sizeof( bufou ),
                &bufin, sizeof( bufin ) ) == -1 )
         exit( EXIT_FAILURE );
};
```

Код для реализации того же обмена, но организованного как менеджер ресурса, будет как минимум в несколько раз объемнее (образцы менеджеров мы уже видели ранее по тексту). Кроме того, по большей части он будет состоять из заполнения полей некоторых внутренних структур, используемых библиотеками менеджера ресурсов или пула потоков. На первый поверхностный взгляд такой код маловразумителен.

С другой стороны, весь достаточно объемный код **любого** менеджера ресурса – это очередное повторение одного и того же общего шаблона для написания менеджеров. При некоторых минимальных навыках написание самых замысловатых менеджеров ресурсов становится совершенно рутинным занятием, не превышающим по трудоемкости написание простого обмена сообщениями. Большим подспорьем здесь является наличие в комплекте технической документации QNX огромно-

го (более 80 страниц) раздела, исчерпывающе описывающего технику создания менеджеров ресурсов; по качеству и скрупулезности изложения это одна из лучших частей всей технической документации.

Гибкость и мобильность

При установлении соединения техника простого обмена сообщениями в качестве адресата сообщений (сервера) использует «магическую тройку» (триплет [1]) параметров ND, PID и CHID, где:

- `ND` – дескриптор сетевого узла, на котором работает интересующая нас программа-сервер (узел, на который надо отсылать сообщение);
- `PID` – PID процесса этой программы на своем сетевом узле (кому отсылать сообщение);
- `CHID` – номер канала, который открыла эта программа для приема сообщений данного вида.

В этой адресации, пожалуй, и кроется самая главная причина негибкости механизма обмена сообщениями. Дескриптор сетевого узла `nd`, значение которого, кроме того, способно самопроизвольно изменяться в сети с течением времени, мы можем установить по сетевому имени интересующего нас хоста, используя `netmgr_strtond()`. (Это действие по своей сути избыточное, дополнительный уровень косвенности, так как первичным идентификатором узла для **пользователя приложения** является его имя, а не дескриптор.)

Гораздо хуже дело обстоит с `pid` и `chid`, особенно для процесса, выполняющегося на удаленном сетевом узле. Не существует в общем виде **прямого** способа установить PID удаленного процесса, а тем более номер канала, который открыл этот процесс для обмена (или вообще не открывал, если мы, например, ошиблись в определении его PID). И тогда на помощь приходят некоторые искусственные приемы, построенные либо на использовании некоторых иерархических (родительский–дочерний) соотношений процессов клиента и сервера, либо на системах совершенно условных договоренностей (произвольных и варьирующихся от случая к случаю).

Р. Кертен [1] отмечает, что существует множество способов нахождения этой адресной триады, и перечисляет некоторые из них:

1. Открыть файл с известным именем и сохранить в нем ND/PID/CHID...

2. Использовать для объявления идентификаторов ND/PID/CHID глобальные переменные программы...

3. Занять часть пространства имен путей и стать администратором ресурсов.

Не вдаваясь в подробный анализ (вы это можете сделать сами), отметим, что 1-й способ – крайне искусственный и негибкий (особенно в сетевой среде), 2-й – крайне ограничен и применим лишь к узкому кругу

задач, а 3-й способ подводит нас к применению совсем другой, альтернативной технологии с используемыми ею принципами адресации.

Несколько, безусловно, интересных и заслуживающих внимания вариаций на тему техники обмена сообщениями предлагает В. Зайцев в приложении, которое следует за данной главой.

Пользуясь случаем, внесем и мы свою лепту в «копилку» способов вычисления адресной триады и увязывания клиента с соответствующим сервером. В тех нечастых случаях, когда требуется обеспечить максимально возможную плотность потока обмена (об этом см. далее), а информационный канал желательно создать на базе именно прямого обмена сообщениями, мы предлагаем оформлять серверный процесс одновременно и как специальный менеджер ресурса, и как канал прямого обмена сообщениями. При этом клиент, пользуясь адресацией пути к менеджеру, запрашивает его по `read()` или `devctl()`, на которые менеджер возвращает свой PID и открытый для прямого обмена дополнительный CHID. На этом функции менеджера заканчиваются, а весь информационный обмен далее идет обменом сообщений через указанный канал. Полный текст такого сервера будет показан в примере позже.

Теперь обратимся к технологии менеджера ресурсов. В этой технике менеджер регистрирует в пространстве имен (в файловой системе) уникальное имя, по которому клиенты, заинтересованные в его ресурсе, будут адресоваться к менеджеру. Идея не нова для мира UNIX (каталоги файловой системы `/proc` или `/dev`, как правило, вообще не содержат реальных файлов), и она находит все более последовательное расширение в новых разработках операционных систем, отталкивающихся от UNIX, например Plan9 или Inferno.

Техника менеджера ресурса вводит дополнительный уровень разрешения имен, который реализуется через **менеджер процессов** `procnto` (как это происходит, подробно и на примерах описывается в [1]). Происходящее при выполнении POSIX-оператора:

```
int fd = open( /net/host/dev/srv, O_RDONLY );
```

по внутреннему содержанию в точности соответствует тому, что происходит в процессе организации обмена сообщениями при выполнении:

```
int coid = ConnectAttach( node, pid, chid, 0, 0 );
```

и может даже при определенных обстоятельствах возвратить то же значение и уж по крайней мере **всегда** возвращает значение той же природы, хотя мы и говорим по привычке в первом случае «файловый дескриптор», а во втором – «идентификатор соединения». Здесь отчетливо видна подмена адресной триады `node`, `pid` и `chid` именем пути `/net/host/dev/srv`.

Модель адресации менеджера ресурса в QNX, конечно, намного более универсальна, гибка и мобильна, нежели модель прямого обмена сообщениями. Например, можно написать сервер, который при запуске

воспринимал бы **полное** имя, под которым он будет регистрироваться в пространстве имен, например (пусть даже некоторые варианты и сомнительны в своей осмысленности):

```
# server -n /dev/srv
# server -n /proc/srv
# server -n /fs/srv
```

Можно запустить несколько экземпляров такого сервера, возможно модифицированных использованием других ключей запуска:

```
# server -n /dev/srv1
# server -n /dev/srv2
```

Наконец, можно сделать это не только на своем локальном узле сети, но и на других сетевых узлах:

```
# on -f host1 server -n /dev/srv1
# on -f host1 server -n /dev/srv2
# on -f host2 server -n /dev/srv1
# on -f host2 server -n /dev/srv1
```

Теперь, если наш клиент выполнен так, что позволяет при запуске указать имя сервера, который он должен использовать, мы можем применить такой клиент для работы с самыми различными экземплярами серверов, где бы они ни находились в сети, например:

```
# client -s /dev/srv1
# client -s /net/host2/dev/srv1
```

Полный исходный код такой реализации будет показан в примере, к рассмотрению которого мы перейдем после завершения этого раздела.

В чем еще состоит различие, которое можно отнести к категории гибкости механизмов?

В краткой схеме, показанной кодом предыдущего раздела, вызовом:

```
MsgSend( coid, &bufou, sizeof( bufou ), &bufin, sizeof( bufin ) )
```

может быть послано сообщение **произвольной** (в пределах абсолютных ограничений) длины (`sizeof( bufou )`). Это сообщение (с информацией о его фактической длине) будет принято сервером, который в свою очередь может ответить сообщением произвольной длины, которое и будет доставлено клиенту в ответ на оператор `MsgSend()`.

При обмене с менеджером ресурсов, в силу необходимости приведения клиентских запросов в «прокрустово ложе» POSIX, картина принципиально другая: каждый запрос может оперировать только с данными той длины, которая предопределена стандартом.

1. Команды группы `read()` могут передать в направлении сервера только код команды, уточненный параметрами (например, длина запрашиваемых данных), но не данные. В ответ сервер может пере-

дать клиенту данные **произвольной** длины. Обмен данными **однонаправленный**, в направлении от сервера к клиенту.

2. Команды группы `write()` могут передать от клиента к серверу данные **произвольной** длины, но в ответ сервер может возвратить только код результата – число байт, фактически успешно полученных в результате операции. Обмен данными **однонаправленный**, в направлении от клиенту к серверу.

3. Команда `devctl()`, использующаяся обычно для организации канала управления (но это не обязательно), в зависимости от кода команды может передавать данные либо к серверу (подобно `write()`), либо от сервера (подобно `read()`), либо в обоих направлениях за один обмен. Таким образом, этой командой может быть организован **двунаправленный** обмен. Вообще говоря, принято считать, что по `devctl()` передаются данные **фиксированной** длины: длина передаваемого блока данных определяется непосредственно кодом команды. Но это не является серьезным ограничением: мы можем динамически формировать код команды перед обменом исходя из объема данных, подлежащих передаче (как это будет показано в примере следующего раздела). Такой трюк позволяет организовать обмен данными **произвольной** длины. Ограничение здесь состоит в другом: объемы данных, передаваемые по `devctl()` в обоих направлениях, должны быть равны! А это, согласитесь, не совсем то, что мы видели при простом обмене сообщениями.

4. Наконец, последним вариантом обмена с менеджером ресурса является обмен «сырыми», неформатированными сообщениями. Но это уже вариация простого обмена сообщениями, а как ее реализовать в коде, показано в приложении В. Зайцева.

С другой стороны, такая повышенная гибкость простого обмена сообщениями в отношении размеров передаваемых данных – предмет потенциальных ошибок, в то время как регламентируемое POSIX поведение обменных функций несет в себе дополнительный контроль корректности.

Эффективность реализации

Если техника менеджеров ресурсов – это только надстройка над базовым механизмом обмена сообщениями, то возникает совершенно естественный вопрос: какова же плата за использование этого производительного и «комфортного» механизма?

Для анализа «скоростных» характеристик альтернативных механизмов обмена сообщениями создадим группу приложений (клиентские и сервер, *файлы cli.cc*, *clr.cc* и *srv.cc*), а чтобы отдельно не выписывать определения, используемые приложениями, вынесем их в отдельный файл определений (*файл common.h*).

Общие определения проекта

```
const char VERSION[] = "vers. 1.03";

// имя, под которым будет регистрироваться в пространстве
// имен наш тестовый менеджер ресурса
static const char DEVNAME[ _POSIX_PATH_MAX ] = "/dev/srr";

// "базовая часть" команды devctl(), конкретный код команды будет
// формироваться динамически на основе этой части, но исходя
// из фактической длины блока передаваемых данных
const unsigned int DCMD_CMD = 1,
    DCMD_SRR = _POSIX_DEVDIR_TOFROM + ( _DCMD_NET << 8 ) + DCMD_CMD;

// структура ответов менеджера ресурса по запросу read()
struct result {
   pid_t    pid;
   int      chid;
   uint64_t cps;
   result( void ) {
      pid = getpid();
      // если уж возвращать, то и служебную информацию ;)
      cps = SYSPAGE_ENTRY( qtime )->cycles_per_sec;
   };
};

// завершение с извещением кода причины
inline void exit( const char *msg ) {
   cout << '\r';
   perror( msg );
   exit( EXIT_FAILURE );
};
```

В этой части каких-либо комментариев заслуживает разве что структура `result`. Наш сервер устроен «наоборот»: информационный обмен данными он осуществляет по запросу `devctl()`, запрос `read()` нам «не нужен», и мы используем его только для возврата информации (PID + CHID) об автономном канале обмена сообщениями. Заодно мы передаем в поле `cps` этой структуры значение тактовой частоты процессора сервера, что позволяет знать, «с кем мы имеем дело» при экспериментах в сети.

Теперь мы вполне готовы написать код сервера. Этот сервер (*файл srv.cc*), в отличие от традиционных, имеет два независимых канала подключения: как менеджер ресурсов и как сервер простого обмена сообщениями. Как менеджер он по запросу `read()` возвращает адресные компоненты (PID, CHID) для обмена сообщениями (ND клиент должен восстановить самостоятельно). По запросу `devctl()`, а также по запросу по автономному каналу обмена сообщениями сервер просто ретранслирует обратно полученный от клиента блок данных (в каком-то смысле

по обоим каналам обмена наш сервер является «зеркалом», отражающим данные).

Сервер запросов

```
result data;
//-------------------------------------------------------------
// реализация обработчика read():
 static int readfunc( resmgr_context_t *ctp,
                      io_read_t *msg,
                      iofunc_ocb_t *ocb ) {
   int sts = iofunc_read_verify( ctp, msg, ocb, NULL );
   if( sts != EOK ) return sts;
   // возвращать одни и те же статические данные…
   MsgReply( ctp->rcvid, sizeof( result ),
             &data, sizeof( result ) );
   return  _RESMGR_NOREPLY;
};
//-------------------------------------------------------------
// реализация обработчика devctl:
static int devctlfunc( resmgr_context_t *ctp,
                       io_devctl_t *msg,
                       iofunc_ocb_t *ocb ) {
   int sts = iofunc_devctl_default( ctp, msg, ocb );
   if( sts != _RESMGR_DEFAULT ) return sts;
   // разбор динамически создаваемого кода dvctl(),
   // извлечение из него длины принятого блока
   unsigned int nbytes = ( msg->i.dcmd -  DCMD_SRR ) >> 16;
   msg->o.nbytes = nbytes;
   // и тут же ретрансляция блока назад
   return _RESMGR_PTR( ctp, &msg->i, sizeof( msg->i ) + nbytes );
};
//-------------------------------------------------------------
// установка однопоточного менеджера, выполняемая
// в отдельном потоке
static void* install( void* data ) {
   dispatch_t  *dpp;
   if( ( dpp = dispatch_create() ) == NULL )
      exit( "dispatch allocate" );
   resmgr_attr_t  resmgr_attr;
   memset( &resmgr_attr, 0, sizeof( resmgr_attr ) );
   resmgr_attr.nparts_max = 1;
   resmgr_attr.msg_max_size = 2048;
   static resmgr_connect_funcs_t   connect_funcs;
   static resmgr_io_funcs_t             io_funcs;
   iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
                     _RESMGR_IO_NFUNCS, &io_funcs );
   // определяем обработку, отличную от обработки по умолчанию,
   // только для двух команд: read() & devctl()
   io_funcs.read = &readfunc;
   io_funcs.devctl = &devctlfunc;
```

```
   static iofunc_attr_t  attr;
   iofunc_attr_init( &attr, S_IFNAM | 0666, 0, 0 );
   // связываем менеджер с его префиксным именем
   if( resmgr_attach( dpp, &resmgr_attr, DEVNAME,
                      _FTYPE_ANY, 0, &connect_funcs,
                      &io_funcs, &attr ) == -1 )
      exit( "prefix attach" );
   dispatch_context_t  *ctp = dispatch_context_alloc( dpp );
   while( true ) {
      if( ( ctp = dispatch_block( ctp ) ) == NULL )
         exit( "block error" );
      dispatch_handler( ctp );
   };
};

// размер буфера для обмена сообщениями;
// этого нам хватит с большим запасом и надолго ;)
const int blk = 100000;
// обработчик низкоуровневых сообщений,
// также работающий в отдельном потоке
void* msginout( void* c ) {
   static uint8_t bufin[ blk ];
   struct _msg_info info;
   while( true ) {
      int rcvid = MsgReceive( data.chid, &bufin, blk, &info );
      if( rcvid < 0 ) exit( "message receive" );
      if( MsgReply( rcvid, EOK, &bufin, info.msglen ) < 0 )
         exit( "message reply" );
   };
};
//-------------------------------------------------------
// "пустой" обработчик реакции на ^C (сигнал SIGINT)
inline static void empty( int signo ) { };
//-------------------------------------------------------
// главная программа, которая все это "хозяйство" установит
// и будет безропотно ждать завершения по ^C ;)
int main( int argc, char *argv[] ) {
   cout << "SRR server: " << VERSION << endl;
   // открывается менеджер ресурса . . .
   int fd = open( DEVNAME, O_RDONLY );
   // если менеджер открылся, то это нам не нужно -
   // дубликаты не создавать!
   if( fd > 0 )
      close( fd ),
      cout << "already in use: " << DEVNAME << endl,
      exit( EXIT_FAILURE );
   // перехватываем реакцию ^C:
   cout << ". . . . . waiting  ^C. . . . . ." << flush;
   signal( SIGINT, empty );
   // создается канал для обмена низкоуровневыми сообщениями
   data.chid = ChannelCreate( 0 );
```

```
    // и запускается отдельным потоком ретранслятор с этого канала
    if( pthread_create( NULL, NULL, msginout, NULL ) != EOK )
        exit( "message thread" );
    // запускается менеджер ресурса
    if( pthread_create( NULL, NULL, install, NULL ) != EOK )
        exit( "manager thread" );
    // . . . все! Мы свое дело сделали и ожидаем ^C . . .
    pause();
    cout << "\rFinalisation...                    " << endl;
    // . . . очистка, завершение . . .
    ChannelDestroy( data.chid );
    return EXIT_SUCCESS;
};
```

Первая клиентская программа (*файл cli.cc*) посылает серверу блок данных указанной длины (длина может изменяться в широких пределах указанием при запуске ключа `-b`) и ожидает от него ретрансляции, после чего замеряет время ответа от сервера. Этот процесс повторяется многократно (ключ `-m`).

Первый клиентский процесс

```
#include "common.h"

static uint64_t *tim;
static int num = 10;

// вывод результатов с оценкой статистики: среднее, С.К.О. . .
static void outtim( void ) {
    double m = 0., s = 0.;
    for( int i = 0; i < num; i++ ) {
        double d = (double)tim[ i ];
        m += d; s += d * d;
    };
    m /= num;
    s = sqrt( s / num - m * m );
    cout << '\t' << (uint64_t)floor( m + .5 ) << "\t~"
         << (uint64_t)floor( s + .5 ) << "\t{"
         << (uint64_t)floor( s / m * 100 + .5 ) << "%}"
         << endl;
};

int main( int argc, char **argv ) {
    cout << "SRR client: " << VERSION << endl;
    int opt, val;
    unsigned int blk = 100;
    char PATH[ _POSIX_PATH_MAX ] = "";
    while ( ( opt = getopt( argc, argv, "n:b:m:" ) ) != -1 )
    {
        switch( opt ) {
            case 'n' :          // имя хоста сервера
                strcpy( PATH, "/net/" );
                strcat( PATH, optarg );
```

```
            break;
         case 'b' :          // размер блока обмена (байт)
            if( sscanf( optarg, "%i", &blk ) != 1 )
               exit( "parse command line failed" );
            break;
         case 'm' :          // число повторений таких блоков
            if( sscanf( optarg, "%i", &num ) != 1 )
               exit( "parse command line failed" );
            break;
         default :
            exit( EXIT_FAILURE );
      }
};
// "составить" полное имя менеджера
strcat( PATH, DEVNAME );
cout << "server path: " << PATH << ", block size = "
     << blk << " bytes, repeat = " << num << endl;
// при инициализации мы сразу получаем скорость процессора клиента
result data;
cout << "CPU speed [c.p.s.]: client = " << data.cps;
// пытаемся подключиться к серверу-менеджеру
int fd = open( PATH, O_RDONLY );
if( fd < 0 ) exit( "server not found" );
// читаем его параметры
if( read( fd, &data, sizeof( result ) ) == -1 )
   exit( "parameter block read" );
cout << ", server = " << data.cps << endl;
tim = new uint64_t[ num ];
uint64_t tim2;
uint8_t *bufin = new uint8_t[ blk ],
        *bufou = new uint8_t[ blk ];
// определяем дескриптор сетевого узла
int32_t node = netmgr_strtond( PATH, NULL );
// это интересное место: если в имени нет сетевого префикса пути,
// но это имя удается открыть, то это локальный хост!
if( node == -1 && fd > 0 && errno == ENOENT )
   node = ND_LOCAL_NODE;
// по адресным данным, полученным ранее по read(), создаем канал
// для прямого обмена сообщениями с тем же процессом:
int coid = ConnectAttach( node, data.pid, data.chid,
                          _NTO_SIDE_CHANNEL, 0 );
if( coid < 0 ) exit( "connect to message channel" );
cout << "- message exchange:" << flush;
// обмен по каналу низкоуровневых сообщений:
for( int i = 0; i < num; i++ ) {
   tim[ i ] = ClockCycles();
   if( MsgSend( coid, bufou, blk, bufin, blk ) == -1 )
      exit( "exchange data with channel" );
   tim[ i ] = ClockCycles() - tim[ i ];
};
outtim();
```

```
    ConnectDetach( coid );
    // повторяем в точности тот же обмен, но по запросу devctl():
    unsigned int DCTL = ( blk<<16 ) + DCMD_SRR;
    cout << "- manager exchange:" << flush;
    for( int i = 0; i < num; i++ ) {
       tim[ i ] = ClockCycles();
       if( devctl( fd, DCTL, bufou, blk, NULL ) != EOK )
          exit( "DEVCTL error" );
       tim[ i ] = ClockCycles() - tim[ i ];
    };
    outtim();
    close( fd );
    delete [] bufin, delete [] bufou;
    delete [] tim;
    return EXIT_SUCCESS;
};
```

Смотрим локальные результаты исполнения и оценки, которые дает нам эта клиентская программа (знаком «~» отмечено С.К.О. предшествующего ему в выводе значения измеренной средней величины, после чего в скобках – процентное отношение этого С.К.О. к измеряемой величине):

```
# nice -n-19 cli -b1 -m1000
SRR client: vers. 1.03
server path: /dev/srr, block size = 1 bytes, repeat = 1000
CPU speed [c.p.s.]: client = 534639500, server = 534639500
- message exchange:     2693    ~168    {6%}
- manager exchange:     6579    ~357    {5%}
# nice -n-19 cli -b10 -m1000
SRR client: vers. 1.03
server path: /dev/srr, block size = 10 bytes, repeat = 1000
CPU speed [c.p.s.]: client = 534639500, server = 534639500
- message exchange:     2726    ~258    {9%}
- manager exchange:     6725    ~378    {6%}
# nice -n-19 cli -b100 -m1000
SRR client: vers. 1.03
server path: /dev/srr, block size = 100 bytes, repeat = 1000
CPU speed [c.p.s.]: client = 534639500, server = 534639500
- message exchange:     3351    ~190    {6%}
- manager exchange:     7055    ~414    {6%}
# nice -n-19 cli -b1000 -m1000
SRR client: vers. 1.03
server path: /dev/srr, block size = 1000 bytes, repeat = 1000
CPU speed [c.p.s.]: client = 534639500, server = 534639500
- message exchange:     3912    ~369    {9%}
- manager exchange:     8312    ~4024   {48%}
# nice -n-19 cli -b4000 -m1000
SRR client: vers. 1.03
server path: /dev/srr, block size = 4000 bytes, repeat = 1000
CPU speed [c.p.s.]: client = 534639500, server = 534639500
```

```
- message exchange:     5393    ~518    {10%}
- manager exchange:     10666   ~770    {7%}
# nice -n-19 cli -b6000 -m1000
SRR client: vers. 1.03
server path: /dev/srr, block size = 6000 bytes, repeat = 1000
CPU speed [c.p.s.]: client = 534639500, server = 534639500
- message exchange:     7373    ~612    {8%}
- manager exchange:     12423   ~995    {8%}
# nice -n-19 cli -b10000 -m1000
SRR client: vers. 1.03
server path: /dev/srr, block size = 10000 bytes, repeat = 1000
CPU speed [c.p.s.]: client = 534639500, server = 534639500
- message exchange:     14365   ~953    {7%}
- manager exchange:     16018   ~5399   {34%}
```

Это дает нам следующую информацию:

- Обмен с сервером, работающим на **локальном хосте**, происходит синхронно: клиент, переслав запрос серверу, блокируется в ожидании ответа от него. В этих условиях мы загружаем процессор на 100% совместной активностью клиента и сервера.
- Обмен в **эквивалентных** условиях с сервером, работающим как менеджер ресурса, требует (в сравнении с прямым обменом сообщениями) в 1,12–2,44 раз большее количество процессорных циклов на свое обслуживание, или, в относительных единицах, максимально достижимая производительность менеджера меньше на 12–144%.
- Самые неблагоприятные (144%) значения относятся к случаю обмена короткими сообщениями (1–10 байт); достаточно ощутимое (~2) значение этого соотношения сохраняется до размеров передаваемых блоков данных, равных 8–10 Кбайт.
- Накладные расходы на передачу единичного байта информации недопустимо велики (2693 циклов на байт при обмене сообщениями и 6579 циклов на байт – для менеджера) при организации обмена короткими сообщениями. С ростом объема данных, передаваемых за один цикл обмена, этот показатель очень резко падает (на блоках по 100 байт уже 33,5 и 70 соответственно, т. е. 2 порядка). Для систем с интенсивными потоками обмена необходимо стремиться максимально блокировать передаваемые данные и минимизировать число актов обмена.

Теперь выполним то же самое, но при обмене с сервером, локализованным на удаленном хосте сети (мы используем низкоскоростную сеть 10 Мбит/сек, на которой все эффекты более наглядны):

```
# nice -n-19 cli -nrtp -b1 -m500
SRR client: vers. 1.03
server path: /net/rtp/dev/srr, block size = 1 bytes, repeat = 500
CPU speed [c.p.s.]: client = 534639500, server = 451163200
- message exchange:     671017  ~391587 {58%}
- manager exchange:     712181  ~394844 {55%}
```

```
# nice -n-19 cli -nrtp -b10 -m500
SRR client: vers. 1.03
server path: /net/rtp/dev/srr, block size = 10 bytes, repeat = 500
CPU speed [c.p.s.]: client = 534639500, server = 451163200
- message exchange:     642456  ~380313 {59%}
- manager exchange:     743094  ~423717 {57%}
# nice -n-19 cli -nrtp -b100 -m500
SRR client: vers. 1.03
server path: /net/rtp/dev/srr, block size = 100 bytes, repeat = 500
CPU speed [c.p.s.]: client = 534639500, server = 451163200
- message exchange:     878686  ~432230 {49%}
- manager exchange:     907474  ~420140 {46%}
# nice -n-19 cli -nrtp -b1000 -m500
SRR client: vers. 1.03
server path: /net/rtp/dev/srr, block size = 1000 bytes, repeat = 500
CPU speed [c.p.s.]: client = 534639500, server = 451163200
- message exchange:     2064542 ~358333 {17%}
- manager exchange:     2113638 ~372487 {18%}
# nice -n-19 cli -nrtp -b3000 -m200
SRR client: vers. 1.03
server path: /net/rtp/dev/srr, block size = 3000 bytes, repeat = 200
CPU speed [c.p.s.]: client = 534639500, server = 451163200
- message exchange:     4134249 ~418168 {10%}
- manager exchange:     4181481 ~418139 {10%}
# nice -n-19 cli -nrtp -b5000 -m200
SRR client: vers. 1.03
server path: /net/rtp/dev/srr, block size = 5000 bytes, repeat = 200
CPU speed [c.p.s.]: client = 534639500, server = 451163200
- message exchange:     5805056 ~252663 {4%}
- manager exchange:     5825837 ~229120 {4%}
# nice -n-19 cli -nrtp -b8000 -m200
SRR client: vers. 1.03
server path: /net/rtp/dev/srr, block size = 8000 bytes, repeat = 200
CPU speed [c.p.s.]: client = 534639500, server = 451163200
- message exchange:     8741090 ~446299 {5%}
- manager exchange:     8788642 ~427459 {5%}
# nice -n-19 cli -nrtp -b10000 -m200
SRR client: vers. 1.03
server path: /net/rtp/dev/srr, block size = 10000 bytes, repeat = 200
CPU speed [c.p.s.]: client = 534639500, server = 451163200
- message exchange:     8971296 ~451857 {5%}
- manager exchange:     9731224 ~301409 {3%}
```

В этом варианте основной компонент задержки вносится передачей данных по физическому каналу; разница между реализациями обмена сообщениями и менеджера ресурсов в значительной степени нивелирована.

Наш второй клиент (*файл clr.cc*), неизменно работающий с тем же сервером, весьма похож на предыдущий, но он массированно «гонит» поток данных на сервер, пользуясь только одним из механизмов (ключ `-d`)

до прекращения его выполнения пользователем по ^C. Результат его работы – средняя плотность потока информации за весь интервал работы.

Второй клиентский процесс

```
#include "common.h"

static bool conti = true;
// завершение процесса по сигналу пользователя (SIGINT - ^C)
inline static void trap( int signo ) { conti = false; };

int main( int argc, char **argv ) {
   cout << "SRR repeater: " << VERSION << endl;
   int opt, val;
   unsigned int blk = 100;
   char PATH[ _POSIX_PATH_MAX ] = "";
   bool lowlvl = true;
   while ( ( opt = getopt( argc, argv, "n:b:d" ) ) != -1 )
   {
      switch( opt ) {
         case 'n' :              // имя сетевого узла
            strcpy( PATH, "/net/" );
            strcat( PATH, optarg );
            break;
         case 'b' :              // размер блока данных
            if( sscanf( optarg, "%i", &blk ) != 1 )
            exit( "parse command line failed" );
            break;
         case 'd' :              // обмен сообщениями
            lowlvl = false;
            break;
         default :
            exit( EXIT_FAILURE );
       }
   };
   strcat( PATH, DEVNAME );
   cout << "server path: " << PATH
        << ", block size = " << blk << " bytes" << endl;
   // при инициализации мы сразу получаем скорость процессора клиента
   result data;
   cout << "CPU speed [c.p.s.]: client = " << data.cps;
   uint64_t cps = data.cps;
   // пытаемся подключиться к серверу-менеджеру
   int fd = open( PATH, O_RDONLY );
   if( fd < 0 ) exit( "server not found" );
   // читаем его параметры
   if( read( fd, &data, sizeof( result ) ) == -1 )
      exit( "parameter block read" );
   cout << ", server = " << data.cps << endl;
   // определяем дескриптор сетевого узла
   int32_t node = netmgr_strtond( PATH, NULL );
   if( node == -1 && fd > 0 && errno == ENOENT )
```

```
        node = ND_LOCAL_NODE;
    // по адресным данным, полученным ранее по read(), создаем
    // канал для прямого обмена сообщениями с тем же процессом
    int coid = ConnectAttach( node, data.pid, data.chid, _NTO_SIDE_CHANNEL, 0 );
    if( coid < 0 ) exit( "connect to message channel" );
    // динамически готовим код команды devctl():
    unsigned int DCTL = ( blk << 16 ) + DCMD_SRR;
    cout << ". . . . . waiting  ^C. . . . . ." << flush;
    // устанавливается реакция на пользовательский ^C
    signal( SIGINT, trap );
    uint64_t num = 0;
    uint8_t *bufin = new uint8_t[ blk ],
            *bufou = new uint8_t[ blk ];
    uint64_t tim = ClockCycles();
    // в зависимости от выбранного механизма передаем с его помощью данные
    if( lowlvl )
       while( true ) {
          if( MsgSend( coid, bufou, blk, bufin, blk ) == -1 )
             exit( "exchange data with channel" );
          num++;
          if( !conti ) break;
       }
    else {
       while( true ) {
          if( devctl( fd, DCTL, bufou, blk, NULL ) != EOK )
             exit( "DEVCTL error" );
          num++;
          if( !conti ) break;
       };
    };
    tim = ClockCycles() - tim;
    cout << '\r' << ( lowlvl ? "message exchange:" : "manager exchange:" )
         << " number = " << num << "; stream = "
         << (double)num * blk / ( (double)tim / (double)cps ) / 1E6 * 8
         << " Mbit/sec" << endl;
    ConnectDetach( coid );
    close( fd );
    delete [] bufin, delete [] bufou;
    return EXIT_SUCCESS;
};
```

В результате мы получаем оценки максимальной плотности потока обмена, достижимые в выбранных (при помощи ключей) условиях на данном процессоре:

```
# clr -b1
SRR repeater: vers. 1.03
server path: /dev/srr, block size = 1 bytes
CPU speed [c.p.s.]: client = 534639500, server = 534639500
message exchange: number = 906400; stream = 1.54088 Mbit/sec
# clr -b1 -d
SRR repeater: vers. 1.03
```

```
server path: /dev/srr, block size = 1 bytes
CPU speed [c.p.s.]: client = 534639500, server = 534639500
manager exchange: number = 335725; stream = 0.617311 Mbit/sec
# clr -b10
SRR repeater: vers. 1.03
server path: /dev/srr, block size = 10 bytes
CPU speed [c.p.s.]: client = 534639500, server = 534639500
message exchange: number = 1119211; stream = 15.0758 Mbit/sec
# clr -b10 -d
SRR repeater: vers. 1.03
server path: /dev/srr, block size = 10 bytes
CPU speed [c.p.s.]: client = 534639500, server = 534639500
manager exchange: number = 316948; stream = 6.1421 Mbit/sec
# clr -b100
SRR repeater: vers. 1.03
server path: /dev/srr, block size = 100 bytes
CPU speed [c.p.s.]: client = 534639500, server = 534639500
message exchange: number = 729460; stream = 122.617 Mbit/sec
# clr -b100 -d
SRR repeater: vers. 1.03
server path: /dev/srr, block size = 100 bytes
CPU speed [c.p.s.]: client = 534639500, server = 534639500
manager exchange: number = 318435; stream = 57.3215 Mbit/sec
# clr -b1000
SRR repeater: vers. 1.03
server path: /dev/srr, block size = 1000 bytes
CPU speed [c.p.s.]: client = 534639500, server = 534639500
message exchange: number = 823535; stream = 1054.65 Mbit/sec
# clr -b1000 -d
SRR repeater: vers. 1.03
server path: /dev/srr, block size = 1000 bytes
CPU speed [c.p.s.]: client = 534639500, server = 534639500
manager exchange: number = 367712; stream = 493.455 Mbit/sec
# clr -b10000
SRR repeater: vers. 1.03
server path: /dev/srr, block size = 10000 bytes
CPU speed [c.p.s.]: client = 534639500, server = 534639500
message exchange: number = 196479; stream = 2861.27 Mbit/sec
# clr -b10000 -d
SRR repeater: vers. 1.03
server path: /dev/srr, block size = 10000 bytes
CPU speed [c.p.s.]: client = 534639500, server = 534639500
manager exchange: number = 141593; stream = 2487.18 Mbit/sec
```

Цифры достаточно интересные, для того чтобы рассмотреть их детальнее:

- При непрерывном потоке обмена очень короткими сообщениями (1 байт) плотность информационного потока падает до смехотворно низкой величины – 192 Кбайт/сек для обмена сообщениями и 77 Кбайт/сек для обмена с менеджером ресурса.

- При размере блока данных, передаваемого за один обмен, порядка нескольких килобайт разница скоростей информационных потоков для обмена сообщениями и менеджера ресурса практически нивелируется.
- При промежуточных размерах блока данных (от нескольких десятков до сот байт) обмен сообщениями обеспечивает плотность информационного потока до двух раз выше.

Естественно, поскольку мы рассматриваем чисто программные реализации обмена, абсолютные численные значения будут прямо пропорционально зависеть от скорости процессора (представленные результаты соответствуют процессору 533 Мгц). На рис. 5.2 показана динамика загрузки процессора при работе тестовых приложений для случая локального размещения клиента и сервера. Хорошо видно, что в периоды выполнения программы `clr` загрузка процессора подскакивает до 100% – совместной активностью клиент и сервер забирают весь ресурс процессора.

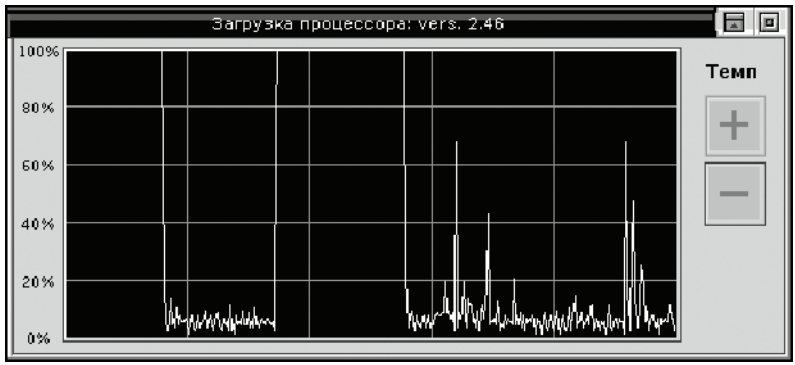

Рис. 5.2. *Динамика загрузки процессора при локальном взаимодействии клиента с сервером*

Далее посмотрим выполнение той же пары приложений, но уже при разнесении их между раздельными узлами сети:

```
# clr -nrtp -b1
SRR repeater: vers. 1.03
server path: /net/rtp/dev/srr, block size = 1 bytes
CPU speed [c.p.s.]: client = 534639500, server = 451163200
message exchange: number = 5049; stream = 0.00670981 Mbit/sec
# clr -nrtp -b1 -d
SRR repeater: vers. 1.03
server path: /net/rtp/dev/srr, block size = 1 bytes
CPU speed [c.p.s.]: client = 534639500, server = 451163200
manager exchange: number = 4824; stream = 0.00598806 Mbit/sec
# clr -nrtp -b10
SRR repeater: vers. 1.03
server path: /net/rtp/dev/srr, block size = 10 bytes
```

```
CPU speed [c.p.s.]: client = 534639500, server = 451163200
message exchange: number = 3885; stream = 0.0651842 Mbit/sec
# clr -nrtp -b10 -d
SRR repeater: vers. 1.03
server path: /net/rtp/dev/srr, block size = 10 bytes
CPU speed [c.p.s.]: client = 534639500, server = 451163200
manager exchange: number = 3102; stream = 0.0557978 Mbit/sec
# clr -nrtp -b100
```

При взаимодействии клиента с сервером по сети в тех же условиях, что и на рис. 5.2, клиент уже не загружает процессор более чем на 50% (рис. 5.3). Если организовать обмен клиента с сервером в несколько потоков (2–3), то при максимальной загрузке процессора можно увеличить плотность потока еще вдвое.

Рис. 5.3. *Загрузка процессора клиента при сетевом взаимодействии клиента с сервером*

```
SRR repeater: vers. 1.03
server path: /net/rtp/dev/srr, block size = 100 bytes
CPU speed [c.p.s.]: client = 534639500, server = 451163200
message exchange: number = 3347; stream = 0.507917 Mbit/sec

# clr -nrtp -b100 -d
SRR repeater: vers. 1.03
server path: /net/rtp/dev/srr, block size = 100 bytes
CPU speed [c.p.s.]: client = 534639500, server = 451163200
manager exchange: number = 2167; stream = 0.480264 Mbit/sec
# clr -nrtp -b1000
SRR repeater: vers. 1.03
server path: /net/rtp/dev/srr, block size = 1000 bytes
CPU speed [c.p.s.]: client = 534639500, server = 451163200
message exchange: number = 1400; stream = 2.0555 Mbit/sec
# clr -nrtp -b1000 -d
SRR repeater: vers. 1.03
server path: /net/rtp/dev/srr, block size = 1000 bytes
CPU speed [c.p.s.]: client = 534639500, server = 451163200
manager exchange: number = 1626; stream = 2.00553 Mbit/sec
```

```
# clr -nrtp -b10000
SRR repeater: vers. 1.03
server path: /net/rtp/dev/srr, block size = 10000 bytes
CPU speed [c.p.s.]: client = 534639500, server = 451163200
message exchange: number = 366; stream = 4.73793 Mbit/sec
# clr -nrtp -b10000 -d
SRR repeater: vers. 1.03
server path: /net/rtp/dev/srr, block size = 10000 bytes
CPU speed [c.p.s.]: client = 534639500, server = 451163200
manager exchange: number = 440; stream = 4.39515 Mbit/sec
```

При взаимодействии по сети разница между реализациями обмена сообщениями и менеджера ресурсов не так заметна. Это и понятно: плотность потока обмена начинает ограничиваться в первую очередь задержками физической среды передачи.

Обратите внимание, что при больших блоках передаваемых данных (10 Кбайт) скорость информационного канала ($4.395 - 4.738 * 2$, учитывая что ретрансляция ведется в двух направлениях) сильно приближается к физической пропускной способности канала (10 Мбит/сек, как уже отмечалось выше), что попутно говорит о весьма высокой эффективности реализации обмена протоколами сети QNET.

Что же в итоге?

В итоге, имеющие место споры приверженцев организации обмена сообщениями и сторонников написания менеджеров ресурсов оказываются бессмысленными. В системах, обслуживающих максимально плотные потоки непрерывной входной информации (классическая постановка задачи для телефонных коммутаторов), реализация обмена сообщениями может оказаться заметно продуктивнее. С другой стороны, в системах с эпизодическим обслуживанием запросов (радиолокационные системы, системы управления технологическим оборудованием) реализация менеджера ресурса может привести к тому, что система станет намного более простой и гибкой в эксплуатации.

Два альтернативных пути не являются «взаимоисключающими», хотя это и реализации единого базового механизма. Они настолько далеко «разошлись» друг от друга, что приобрели индивидуальные, не воспроизводимые альтернативным способом черты. Более того, они могут кооперироваться в рамках даже одного процесса, как это было сделано в показанном ранее примере. Принятию того или иного решения должен предшествовать детальный анализ требований решаемой задачи.

Приложение

Организация обмена сообщениями

Владимир Зайцев

Обмен сообщениями (message passing) является основой архитектуры ОС QNX, на которой строится значительная часть служебных функций системы. Несмотря на свою «элементарность», он является удобным (и в силу своей «нативности» чрезвычайно эффективным!) механизмом для непосредственной организации взаимодействия между процессами. Особый же шарм этого механизма заключается в том, что вместе с передачей данных как таковой можно естественным образом (на основе блокировок Send/Receive/Reply) организовать синхронизацию взаимодействующих процессов.

И хотя в QNX 6 появилось такое мощное средство для организации обмена данными, как менеджер ресурсов, а также имеется богатый набор средств синхронизации, способный удовлетворить любого программиста, имеющего опыт работы в POSIX-совместимых системах, механизм обмена сообщениями по-прежнему остается привлекательным средством, используемым непосредственно в разработке ПО. Особенно отчетливо это проявляется в среде разработчиков, мигрирующих с предыдущих версий ОС QNX, и вряд ли может быть объяснено только их, программистов, консерватизмом.

Вместе с тем переход от QNX 4 к QNX 6 вызвал изменения в реализации механизма обмена сообщениями и, как следствие, API-функций. Причиной этого стал переход от однопоточных к многопоточным процессам, при этом обмен сообщениями стал осуществляться не между процессами[1], а между потоками. Соответственно изменился и «адресат» сообщения. В QNX 4 в этом качестве выступал процесс и его можно было однозначно определить по его идентификатору – действитель-

[1] Строго говоря, в однопоточных процессах QNX 4 обмен тоже организовывался между потоками, просто там разделение понятий «процесс» и «поток» не имело смысла и поэтому всюду использовался только термин «процесс».

ному (при работе на одном узле) или идентификатору виртуального канала («virtual circuit») при межузловых сообщениях. Таким образом, для того чтобы передать сообщение (функцией из семейства `Send*()`) в адрес некоего сервера, процессу-клиенту достаточно было «знать» этот идентификатор. Получал он его, как правило, либо от «родителя» искомого процесса, либо через сервис глобального пространства имен (`qnx_name_attach()` и `qnx_name_locate()`).

Теперь же, в QNX 6, в качестве «адресата» сообщения стал выступать идентификатор соединения (`coid`), и именно он требуется при вызове функций семейства `MsgSend*()`. Для создания же соединения с сервером клиенту необходимо «знать» триаду соединения: идентификатор этого процесса-сервера (`pid`), дескриптор узла (`nd`), на котором сервер выполняется, и идентификатор созданного сервером канала (`chid`).

Вторым «возмущением», привнесенным QNX 6 в привычную и сложившуюся технику разработок, явился переход от идентификаторов узлов (`nid`), которые являлись уникальными в пределах сети и однозначно определяли каждый узел, к дескрипторам узлов (`nd`), которые уникальны только в пределах каждого данного узла, но не в сети. Уникальность в пределах сети теперь должны обеспечивать символьные имена узлов.

В этом приложении предпринята попытка обрисовать специфику, характерную для организации обмена сообщениями в QNX 6, особенно проявляющуюся при межузловом обмене, и поделиться практическими решениями, учитывающими эту специфику.

Организация обмена сообщениями на основе «семейных» процессов

Рассмотрим, как можно организовать обмен сообщениями между потоками, принадлежащими процессам, связанным «родственными узами». Для простоты изложения, чтобы в дальнейшем не формулировать «поток, принадлежащий процессу», будем рассматривать однопоточные процессы и говорить (в традициях QNX 4) «обмен сообщениями между процессами».

Итак, пусть некий *родительский процесс* порождает на другом узле *дочерний процесс*. Под порождением будем подразумевать «запуск с узла», то есть запуск процесса, выполняемый утилитой `on` с опцией `-f`. Для порождения используем функцию `spawn()`:

```
char* args[ ] = { "/net/904-3/home/ZZZ/bin/TestChild", NULL };
...
spawn( "/home/ZZZ/bin/TestChild", 0, NULL, &InhProc, args, NULL );
```

Рассмотрим вначале проблемы, стоящие перед дочерним процессом при его желании связаться с родительским. Как уже указывалось, для создания соединения ему необходимо «знать» триаду соединения.

Пусть процессу-клиенту известно символьное имя узла, на котором функционирует искомый адресат. (При описываемой здесь «семейной» архитектуре разрабатываемой системы символьные имена обычно жестко определены и поэтому зачастую просто записываются в конфигурационном файле, откуда процессы всегда могут получить символьное имя нужного узла.) Для того чтобы получить дескриптор узла по известному имени, можно вызвать функцию `netmgr_strtond()`, которая преобразует имя узла в его дескриптор.

Однако вызов этой функции дочерним процессом приводит к неожиданному (по крайней мере, так было со мной...) на первый взгляд результату: функция возвращает дескриптор узла «с точки зрения» родительского процесса! Иными словами, нулевой дескриптор приписывается не узлу, на котором «живет» дочерний процесс, а узлу с родительским процессом. Другие дескрипторы тоже соотносятся с именами узлов так, как это выполняется на узле с родителем – порожденный процесс, унаследовав от родителя текущую рабочую и корневую директории, тем самым остался, если можно так выразиться, душой на своей исторической родине.

В [4] (глава Дмитрия Алексеева «Утилита on») этот вопрос был достаточно хорошо освещен и было сказано, что для решения проблемы следует перед порождением процесса вызвать функцию `chroot()` с именем узла, на котором процесс будет порожден (и при необходимости «обратным» вызовом `chroot()` с именем узла процесса-родителя после вызова `spawn()`). Это позволяет порожденному процессу обрести новую корневую директорию на том узле, где он выполняется. И тогда вызовы `netmgr_strond()` будут возвращать дескрипторы узлов именно с точки зрения того узла, на котором функционирует порожденный процесс.

Далее, если дочерний процесс запущен на удаленном узле, то вызванная им функция `getppid()` в качестве родительского процесса возвращает совсем даже не идентификатор «фактического» родителя, а идентификатор процесса `io-net`, что, может быть, формально и верно, но по существу это издевательство (особенно для «мигрантов» с QNX 4, где они получали идентификатор виртуального канала и обращались с ним как с обычным идентификатором процесса). Итак, для того чтобы порожденный процесс знал свое «отчество», проще всего родителю передать свой идентификатор дочернему процессу при его рождении в параметре списка `argv[]`.

Я бы рекомендовал также не уповать на то, что дескрипторы каналов, создаваемые процессами-адресатами отправляемых сообщений, всегда будут равны 1. По ряду причин это далеко не всегда так. И для надежности лучше просто передавать этот дескриптор в аргументах при порождении процесса.

Таким образом, родительский процесс при порождении дочернего процесса должен передать ему в списке аргументов свой идентификатор и дескриптор созданного канала и косвенно, посредством вызова `chroot()`,

имя узла, на котором дочерний процесс запускается. После этого порожденный процесс способен правильно собрать триаду, необходимую для выполнения MsgSend().

Теперь обсудим проблемы, стоящие перед родительским процессом. Если мы хотим отсылать сообщения с родительского процесса на порожденный, то два из трех членов триады родительский процесс может легко получить: дескриптор узла – с помощью функции netmgr_strtond(), а идентификатор порожденного процесса возвращается функцией spawn(). Но вот с дескриптором канала опять появляется риск «не угадать». Кроме того, если родитель породит дочерний процесс и немедленно после этого попытается подсоединиться к каналу, который должен создать этот процесс, то, вероятнее всего, функция ConnectAttach() вернет -1, поскольку порожденный процесс еще не успеет к тому времени создать канал. Значит, понадобится цикл на N попыток с паузой в ожидании открытия.

Не проще ли тогда просто выполнить синхронизацию? То есть родительскому процессу дождаться сообщения от дочернего процесса, которое будет означать, что порожденный процесс выполнил все необходимые действия по своему развертыванию и в частности создал канал. И теперь совершенно естественно передать в этом синхронизирующем сообщении дескриптор созданного канала. После принятия сообщения родительский процесс имеет все необходимые ему данные для выполнения функции отсылки сообщения MsgSend().

При подобной иерархической структуре системы по типу «родитель–ребенок» общение между порожденными процессами, если таковое требуется, обеспечивается с помощью родительского процесса. Породив один из процессов и получив от него дескриптор канала, родительский процесс может при порождении еще одного процесса передать ему полную триаду «старшего» дочернего процесса, позволяющую новому процессу установить с ним соединение.

Ниже приводится образец кода, реализующего этот подход. Обратите внимание на значение аргумента index, задаваемое в вызовах функции ConnectAttach() равным _NTO_SIDE_CHANNEL. В примерах из [1], книги, безусловно, основополагающей для любого программиста под QNX 6, для упрощения изложения это значение устанавливается в 0. Однако значение, равное _NTO_SIDE_CHANNEL, гарантирует, что возвращаемое функцией значение идентификатора соединения будет взято не из того же пространства, из которого выделяются файловые дескрипторы; в противном случае возникают проблемы, достаточно определенно обрисованные в описании функции ConnectAttach(), приведенном в технической документации QNX.

Пример кода родительского процесса

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <process.h>
#include <sys/neutrino.h>
#include <sys/netmgr.h>
#include <spawn.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>
#include <locale.h>

int main( int argc, char **argv ) {
   int nid;        // Дескриптор удаленного узла
   int PChanid;    // Идентификатор созданного канала
   int CChanid;    // Идентификатор канала, созданного
                   // порожденным процессом на удаленном узле
   int coid;       // Идентификатор связи с порожденным
                   // процессом по созданному им каналу
   int rcvid;      // Идентификатор отправителя полученного
                   // сообщения
   int ErrCode;    // Код ошибки
   char *args[ ] = {
      "/net/904-3/home/ZZZ/BIN/TestChild",
      "pid данного процесса",
      "идентификатор канала",
      NULL
   };
   char BufName[ 100 ], Bufpid[ 12 ],
        Bufchanid[ 12 ], RecBuffer[ 100 ];
   char SendBuf[ 100 ] = "привет, сынок!";
   pid_t procid, childid;
   struct inheritance Inhproc;

   setlocale( LC_CTYPE, "C_TRADITIONAL" );

   if( ( PChanid = ChannelCreate( 0 ) ) == -1 )
      printf( "Родитель: странно, но не удалось "
              "создать канал\n" );
      else printf( "Родитель: канал PChanid = %i создан\n",
                   Pchanid );

   strcpy( BufName, "Bed-Test" );
   // Передаем порожденному процессу свой pid...
   args[ 1 ] = itoa( procid = getpid(), Bufpid, 10 );

   //... и дескриптор канала
   args[2] = itoa( PChanid, Bufchanid, 10 );

   InhProc.flags = SPAWN_SETND | SPAWN_NOZOMBIE;
```

```
if( ( nid = netmgr_strtond( BufName, NULL ) ) == -1 ) {
   printf( "Родитель: отсутствует %s\n", BufName );
   return( -1 );
}
else printf( "Родитель: найден узел %s, его nid = %i\n",
             BufName, nid );

InhProc.nd = nid;

sprintf( BufName, "/net/Bed-Test/" );
chroot( BufName );

errno = 0;
childid = spawn( args[0], 0, NULL &InhProc, args, NULL );
ErrCode = errno;

sprintf( BufName, "/net/904-3/" );
chroot( BufName );

if( childid= = -1 )
   printf( "Родитель: не удалось породить процесс,"
           " errno = %i\n", ErrCode );
   else
      printf( "Родитель: мой id = %i,"
              "порожденный процесс имеет id = %i\n",
              procid, childid );

if( ( rcvid = MsgReceive( PChanid, RecBuffer,
                          100, NULL ) ) == -1 )
   printf( "Родитель: от дитятки не удалось"
           " получить сообщение\n" );
else {
   printf( "Родитель: от дитятки получено"
           " сообщение:\"%s\"\n", RecBuffer );
   CChanid = atoi( RecBuffer );
   strcpy( RecBuffer, "спасибо, сынок!" );
   if( MsgReply( rcvid, EOK, RecBuffer, 100 ) == -1 )
      printf( "Родитель: почему-то не удалось "
              "ответить сыночку. Ау, где ты?\n");
}

if( ( coid = ConnectAttach( nid, childid, Cchanid,
                    _NTO_SIDE_CHANNEL, 0 ) ) == -1 ) {
   printf( "Родитель: странно, но не смог установить"
           " канал связи с ребенком:"
           "nid = %i childid = %i CChanid = %i\n",
           nid, childid, Cchanid );
   return( -1 );
}

printf( "Родитель: установил связь coid = %i с"
        " ребенком\n", coid );
```

```
   errno = 0;
   if( MsgSend( coid, SendBuf, 100, SendBuf, 100 ) == -1 )
      printf( "Родитель: на MsgSend получил errno = %i\n",
              errno );
   else
      printf( "Родитель: получен отклик на MsgSend ()"
              ": \%s\\n", SendBuf );

   printf( "Родитель: позвольте откланяться\n" );

   ChannelDestroy( Pchanid );
   ConnectDetach( Cchanid );
   return( 0 );
}
```

Пример кода порожденного процесса

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <process.h>
#include <sys/netmgr.h>
#include <sys/neutrino.h>
#include <errno.h>
#include <locale.h>

int main( int argc, char **argv ) {
   int nid;            // Дескриптор текущего узла
   int Cchanid;        // Идентификатор созданного канала
   int coid;           // Идентификатор связи с родителем
                       // по созданному им каналу
pid_t Parpid;          // Идентификатор родительского процесса
   int rcvid;          // Идентификатор отправителя
                       // полученного сообщения
   char BufName[ 100 ];
   char SendBuf[ 100 ], RecBuf[ 100 ];

   setlocale( LC_CTYPE, "C-TRADITIONAL" );

   if( ( CChanid = ChannelCreate( 0 ) ) == -1 )
      printf( "Ребенок: странно, но не удалось создать"
              " канал\n" );
   else
      printf( "Ребенок: канал CChanid = %i создан\n",
              CChanid );

   Parpid = atoi( argv [1] );
   printf( "Ребенок сообщает: он жив благодаря папане"
           " Parpid = %i\n", Parpid );

   strcpy( BufName, "904-3" );
```

```
    if( ( nid = netmgr_strtond( BufName, NULL ) ) == -1 )
       printf( "Ребенок: узел \"%s\" не найден!\n",
               BufName );
    else
       printf( "Ребенок: узел \"%s\" найден, его nid = %i\n",
               BufName, nid );

    if( ( coid = ConnectAttach( nid, Parpid,
       atoi( argv [2] ), _NTO_SIDE_CHANNEL, 0 ) ) == -1 ) {
       printf( "Ребенок: странно, но дитя не смогло"
               " установить канал связи с папаней\n" );
       return( -1 );
    }
    printf( "Ребенок: установил связь coid = %i с процессом"
            " Parpid = %i на узле %i\n", coid, Parpid, nid );

    // Вот здесь хорошее место, чтобы выполнить все действия,
    // необходимые для развертывания данного процесса

    itoa( CChanid, SendBuf, 10 );
    errno = 0;
    if( MsgSend( coid, SendBuf, 100, SendBuf, 100 ) == -1 )
       printf( "Ребенок: на MsgSend () к отцу получил"
               " errno = %i\n", errno );
    else
       printf( "Ребенок: на MsgSend () получен отклик"
               " от родителя:\"%s\"\n", SendBuf );

    rcvid = MsgReceive( CChanid, RecBuf, 100, NULL );
    printf( "Ребенок: от папани получено сообщение:"
            " \"%s\"\n", RecBuf );

    strcpy( RecBuf, "я здесь, папаня!" );
    if( MsgReply( rcvid, EOK, RecBuf, 100 ) == -1 )
       printf( "Ребенок: почему-то не удалось ответить"
               " папаше. Ау, где ты?\n" );

    printf( "Ребенок: дитятко работу закончило\n" );

    ChannelDestroy( CChanid );
    ConnectDetach( coid );
    return( 0 );
}
```

Обмен сообщениями на основе менеджера ресурсов

Описанный выше способ построения функционирующей в сети системы процессов может быть реализован далеко не всегда. Зачастую клиенту не известна полная триада, позволяющая ему создать соединение

с сервером. Вспомним, что в QNX 4, где для создания связи с другим процессом был необходим его идентификатор, существовала служба пространства имен, обеспечиваемая сервером службы `nameloc`. Сервер объявлял свое имя в пространстве имен с помощью функции `qnx_name_attach()`, а затем клиент, вызвав функцию `qnx_name_locate()`, получал от системы идентификатор сервера, по которому мог далее с ним общаться.

Разработчики QNX 6 настоятельно рекомендуют вместо использования службы имен выполнять сервер в виде менеджера ресурсов, причем настолько настоятельно, что до версии 6.3 аналог этой службы – менеджер службы глобальных имен `gns` – функционировал только локально. И надо признать, что мощь и изящество менеджера ресурсов являются очень убедительным подкреплением этих рекомендаций.

При использовании механизма менеджера ресурсов процесс, выступающий в качестве сервера, регистрирует свой так называемый префикс путевого имени файла в пространстве файловых имен, после чего другие процессы (клиенты) могут открывать это имя как файл, используя стандартную библиотечную функцию `open()`. Получив в результате выполнения этой функции дескриптор файла, они затем могут обращаться к серверу, используя стандартные библиотечные функции С, такие как `read()`, `write()` и т. д.

Однако важным (по крайней мере, для программистов, не желающих отказываться от такого привычного и эффективного механизма передачи данных, как обмен сообщениями) является тот факт, что этот дескриптор на самом деле является не чем иным, как идентификатором соединения. И поэтому к серверу можно обращаться не только через высокоуровневые функции работы с файлами, но и с помощью элементарных функций обмена сообщениями `MsgSend*()` (элементарных, напомню, в том смысле, что в действительности все стандартные высокоуровневые функции работы с файлами реализованы через функции обмена сообщениями).

Вместе с тем следует учитывать, что менеджер ресурсов поставляется для программиста фактически в готовом виде – как шкаф, то есть уже имеются все отделения, полки, ножки, дверцы, и задача разработчика – лишь заполнить его своим содержимым. Однако «навесить» на него свою полочку уже невозможно. Иными словами, при передаче сообщений с использованием менеджера ресурсов необходимо применять уже имеющиеся средства менеджера ресурсов, благо их вполне достаточно.

Самым очевидным и наиболее простым способом передачи сообщений к серверу является инкапсуляция сообщений в «сообщения управления устройством» – сообщения типа `devctl()`. Однако этот способ имеет существенный недостаток, заключающийся в том, что при взаимном обмене данными между сервером и клиентом, что является более общим и достаточно частым случаем, мы вынуждены передавать в обоих направлениях буферы одинаковой длины. Это объясняется тем, что

функция devctl() имеет только один параметр для размеров обоих буферов. Поэтому в качестве универсального средства передачи сообщений применение этой функции выглядит непривлекательным.

К радости разработчиков, менеджер ресурсов предлагает функцию приватных сообщений io_msg() для сообщений типа _IO_MSG. Менеджер способен их обрабатывать после соответствующей «настройки», заключающейся в подключении диапазона сообщений, интерпретируемых как приватные (допустимые значения должны быть больше 0x1ff – диапазона, резервируемого за системой). При этом сервер в состоянии как сразу «отпустить» Reply-блокированного клиента, так и оставить его в этом состоянии до нужного момента.

Ниже приводится код процесса-клиента и процесса-сервера. Последний представляет собой стандартный менеджер ресурсов – в таком виде, в каком он, так сказать, поставляется разработчику. Единственная шляпа, помещаемая в этот шкаф, – это обработчик приватных сообщений. Здесь вы и должны поместить специфический код обработки принятого сообщения.

В остальном все достаточно тривиально. Более подробно о том, как писать менеджеры ресурсов, можно прочитать в главе «Writing a Resource Manager» технической документации QNX, а также в книгах [1] и [4] (глава Олега Цилюрика «Драйверы»).

Пример обмена сообщениями с помощью менеджера ресурсов

Код файла заголовков

```
#define NET_OPER       "/net/904-3"
#define NET_REG        "/net/Bed-Test"
// Максимальная длина обычного стандартного сообщения:
#define MESSIZE_MAX   100
// Максимальная длина инвентаризационного имени процесса
#define PROC_NAME_MAX 100

struct IdLabel_t {         // Структура, содержащая:
int id;                    // - инвентаризационную метку процесса
char name [PROC_NAME_MAX]; // - инвентаризационное имя процесса
} IdLabel [ ] = {
/* диапазон, выделенный Группе # 1: от 0x5000 до 0x50ff */
0x5001, "пробный менеджер ресурсов",
0x5002, "первый тестовый клиент для менеджера ресурсов",
0x5003, "второй тестовый клиент для менеджера ресурсов",
0x5004, "третий тестовый клиент для менеджера ресурсов",
0x50ff, "четвертый тестовый клиент для менеджера ресурсов"

/* диапазон, выделенный Группе # 2: от 0x5100 до 0x51ff */
/* диапазон, выделенный Группе # 3: от 0x5200 до 0x52ff */
};
```

```
char Anonimus [] = "чуждый процесс";
int ALLNUM_MYPROC = sizeof( IdLabel ) /
                    sizeof( IdLabel[ 0 ] );
```

Код процесса-клиента

Как было сказано, клиент открывает файл (функция `open()`), после чего использует `MsgSend()`, отсылая сообщения и получая ответы.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <locale.h>
#include <string.h>
#include "/home/ZZZ/TESTS/MR/MessTest.h"

int main() {
   int fdRM;  // Дескриптор соединения с менеджером ресурсов
   char BufferSend[ MESSIZE_MAX ],
        BufferReply[ MESSIZE_MAX ];

   setlocale( LC_CTYPE, "C-TRADITIONAL" );

   if( fdRM = open( strcat( strcpy( BufferSend, NET_REG ),
             "/dev/MESSTEST/RM" ), O_RDWR ) ) == -1 ) ) {
      printf( "Клиент не нашел имени менеджера!\n");
      fflush( stdout );
      return( -1 );
   }

   /* Заполнение заголовка - первых 4-х байт сообщения,
      содержащего инвентаризационную метку данного процесса
      (описаны в "IRL32.h") */
   ( (int *) ( BufferSend ) ) [0] = 0x5002;

   /* Заполнение сообщения */
   strcpy( BufferSend + 4,
           "Так вот ты какой, Менеджер Ресурсов!" );

   if( MsgSend( fdRM, BufferSend, 100,
                BufferReply, 100 ) = = -1 )
      printf( "Клиенту не удалось передать сообщение\n" );
   else
      printf( "Клиент передал сообщение и получил <%s>\n",
              BufferReply );
   fflush( stdout );
   close( fdRM );
   return( 0 );
}
```

Код процесса-сервера (менеджера ресурсов)

Для запуска сервера на удаленном узле выполните с терминала команду:

```
# on -f /net/Bed-Test /net/904-3/home/ZZZ/BIN/TestMGR
```

где `Bed-Test` — имя удаленного узла, `904-3` — имя локального узла, `/home/ZZZ/BIN/TestMGR` — путь к исполняемому файлу.

Вначале сервер выполняет действия по своей инициализации, специфические для данного процесса. Если они завершились успешно, т. е. сервер готов обслуживать клиентов, он инициализирует себя как администратор устройства (функции `dispatch_create()`, `memset( &resmgr_attr, ... )`, `iofune_func_init()`, `iofune_attr_init()`, `resmgr_attach()`, `message_attach()`, `dispatch_context_alloc()`), при этом на том узле, где запущен менеджер, появляется файл `/dev/MESSTEST/RM`. После этого, если все прошло успешно, сервер выходит на бесконечную петлю приема сообщений.

Прием сообщений осуществляется функцией `dispatch_block()`, блокирующей процесс-сервер на ожидании сообщений. При получении сообщения оно передается функции `dispatch_handler()`, которая производит разборку сообщения. Если это сообщение относится к известным разборщику, оно направляется к соответствующей функции обработки, принимаемой по умолчанию.

Так, в частности, обрабатываются сообщения на открытие ресурса (пересылаемое клиентом при вызове им функции `open()`), на отсоединение и закрытие ресурса (отсылаются клиентом при вызове им функции `close()`), на чтение или запись (если клиент вызовет функции `read()` или `write()`) и ряд других. Кроме того, разборщику известны сообщения, заголовок которых содержит «инвентаризационную метку», попадающую в диапазон, указанный при вызове функции присоединения приватных сообщений `message_attach()`. В этом случае сообщение передается для дальнейшей обработки функции-обработчику приватных сообщений (в нашем примере это функция `PrivatHandler()`).

При рассмотрении функции обработки приватных сообщений `PrivatHandler()` следует обратить внимание, что, хотя в этой функции и предусмотрено освобождение клиента с Reply-блокировки, она возвращает не `_RESMGR_NOREPLY`, как можно было бы ожидать, а значение `0`, что указывает библиотеке менеджера ресурсов на то, что отвечать Reply-сообщением клиенту уже нет необходимости. Это объясняется тем, что обработчик приватных сообщений сам выполняет Reply-сообщение, и это заложено в нем изначально. В этом состоит важное отличие этого обработчика от всех прочих (взгляните на код обработчика `prior_read()` в разделе «Менеджеры ресурсов» главы 5).

Еще одна тонкость: при работе с приватными сообщениями в процессе-менеджере необходимо использовать функции диспетчеризации `dispatch_*()` (`dispatch_block()`, `dispatch_handler()` и т. д.), а не функции менеджера ресурсов `resmgr_*()` (`resmgr_block()`, `resmgr_handler()` и т. д.).

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <devctl.h>
#include <locale.h>
#include "/home/ZZZ/TESTS/MR/MessTest.h"

int PrivatHandler( message_context_t *ctp, int code,
                   unsigned flags, void* handle );
char* IdLabelParse( int id );

// Таблица функций связи
static resmgr_connect_funcs_t connect_funcs;
// Таблица функций ввода/вывода
static resmgr_io_funcs_t io_funcs;
// Структура атрибутов устройства
static iofunc_attr_t attr;

main( int args, char **argv ) {
resmgr_attr_t resmgr_attr; // Структура атрибутов менеджера ресурсов
dispatch_t *dpp;           // Указатель на структуру диспетчеризации,
                           // содержит идентификатор канала.
dispatch_context_t *ctp;   // Контекстная структура; содержит буфер сообщений,
                           // буфер векторов ввода/вывода
int id;
int result;
char BufferRec[ 100 ];
int rcvid;

setlocale( LC_CTYPE, "C-TRADITIONAL" );

/* Здесь должны выполняться необходимые действия
   по инициализации конкретного сервера */

/* Считаем, что все необходимое теперь выполнено... */

/* Инициализация интерфейса диспетчеризации */
if( ( dpp = dispatch_create() ) == NULL ) {
   printf( "%s: невозможно разместить обработчик"
           " диспетчеризации.\n", argv[0] );
   return EXIT_FAILURE;
}
/* В результате по адресу dpp создана структура диспетчеризации */

/* Инициализация атрибутов менеджера ресурсов */
memset( &resmgr_attr, 0, sizeof resmgr_attr );
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = MESSIZE_MAX;
```

```
/* Задаем число доступных структур векторов ввода/вывода
   (IOV) = 1.
   Задаем максимальный размер буфера получения равным
   MESSIZE_MAX.
   В результате инициализируются атрибуты менеджера ресурсов */

/* Инициализация функций обработки сообщений */
iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
                  _RESMGR_IO_NFUNCS, &io_funcs );
/* В результате заполняются две таблицы (структуры), задающие функции
обработки для двух специальных типов сообщений: таблица функций связи
и таблица функций ввода/вывода. В соответствующих местах размещаются
принимаемые по умолчанию функции iofunc_*_default()... Своими не заменяем -
нет необходимости. */

/* Инициализация используемой устройством структуры атрибутов */
iofunc_attr_init( &attr, S_IFNAM | 0666, 0, 0 );
attr.nbytes = MESSIZE_MAX + 1;
/* В результате инициализируется структура атрибутов,
   используемая устройством:
   S_IFNAM указывает, что тип устройства - Special named file;
   побитовые флаги определяют права доступа;
   число байт в ресурсе задается равным размеру буфера. */

/* Прикрепление имени устройства */
if( ( id = resmgr_attach( dpp, &resmgr_attr,
           "/dev/MESSTEST/RM", _FTYPE_ANY, 0,
           &connect_funcs, &io_funcs, &attr ) ) == -1 ) {
   printf( "%s: невозможно прикрепить имя менеджера"
           " ресурсов.\n", argv[ 0 ] );
   return EXIT_FAILURE;
}

/* Ключевое действие: мы регистрируем на нашем узле имя /dev/MESSTEST/RM
   dpp и resmgr_attr - инициализированные выше структуры;
   /dev/MESSTEST/RM - ассоциированное с устройством имя;
   _FTYPE_ANY - определяет тип открытия устройства (в данном
   случае допускается любой тип запроса открытия);
   равный нулю флаг разборки пути имени файла определяет,
   что запрос - только по имени /dev/MESSTEST/RM;
   &connect_funcs - заданные выше подпрограммы связи;
   &io_funcs - заданные выше подпрограммы ввода/вывода;
   attr - инициализированная выше структура атрибутов устройства;
   Подключаем диапазон сообщений, которые должны рассматриваться
   как приватные, с передачей их обработчику для таких
   сообщений - PrivatHandler() */
if( message_attach( dpp, NULL, 0x5000, 0x5fff,
                    &PrivatHandler, NULL ) == -1) {
   printf( "невозможно подключить данный "
           "диапазон приватных сообщений\n" );
   return (EXIT_FAILURE);
}
```

```
/* Размещение контекстной структуры */
ctp = dispatch_context_alloc(dpp);
/* Размер буфера сообщений, содержащегося в этой структуре, равно как и число
векторов ввода/вывода, также содержащихся в этой структуре, установлены при
инициализации структуры атрибутов менеджера ресурсов */

/* Запуск петли сообщений менеджера ресурсов */
while( 1 ) {
   // ожидание прихода сообщений
   if( ( ctp = dispatch_block( ctp ) ) == NULL ) {
      printf( "ошибка блока\n" );
      return EXIT_FAILURE;
   }
   printf( "Менеджер ресурсов получил сообщение"
   " длиной %i байт\n", ctp->resmgr_context.info.msglen );

   result = dispatch_handler( ctp );
   // сообщение раскодируется, и на основании заданных таблиц функций связи
   // и ввода/вывода вызывается соответствующая функция обработки сообщения
   if( result )
      printf( "Менеджер ресурсов не смог обработать"
              " сообщение result = %i\n", result );
}

/*************************************************************************
Обработчик приватных сообщений, то есть сообщений, заголовок которых
укладывается в диапазон, указанный при вызове функции message_attach()
*************************************************************************/
int PrivatHandler( message_context_t* ctp, int code,
                   unsigned flags, void* handle) {
   char Buffer [MESSIZE_MAX];
   printf( "получено приватное сообщение тип %x от"
           " \"%s\"\n", code, IdLabelParse( code ) );
   printf( "Вот это сообщение: <<%s>>\n",
           (char *)( ctp->msg ) + 4 );
   strcpy( Buffer, "Клиенту: да, я такой" );
   MsgReply( ctp->rcvid, EOK, Buffer, sizeof( Buffer ) );
   return( 0 );
}

/*************************************************************************
Функция пользовательской библиотеки, определяющая инвентаризационное
имя процесса по его инвентаризационной метке
*************************************************************************/
char* IdLabelParse( int id ) {
   struct IdLabel_t Inventory;
   int i = 0;
   while( IdLabel[ i ].id ! = id && i < ALLNUM_MYPROC ) i++;
   if( i == ALLNUM_MYPROC ) return Anonimus;
   else return( IdLabel[ i ].name );
}
```

Использование менеджера службы глобальных имен

Начиная с QNX версии 6.3 сервис глобальных имен, обеспечиваемый GNS-менеджером службы (утилитой `gns`), действует в сети. Используя этот сервис, нет необходимости организовывать программу как полноценный менеджер ресурсов, при этом приложение-сервер может объявлять свою службу, а приложения-клиенты могут отыскивать и использовать службы через QNET-сеть без знания таких частностей, как, например, где эта служба располагается и кто ее обеспечивает. Подробно о сервисе глобальных имен см. в [4].

Для того чтобы развернуть этот сервис, необходимо в режиме сервера запустить менеджер службы глобальных имен на том узле, где должно работать наше приложение-сервер. В режиме сервера GNS-менеджер выступает в роли некой центральной базы данных, хранящей объявленные службы, и обрабатывает запросы на поиск и установление связи с ними. На узле же, где располагается клиент, запускаем менеджер в режиме клиента, при этом он передает запросы объявления, поиска и установки связи между локальным (то есть расположенным на этом же узле) приложением-клиентом и сервером (серверами) `gns`.

Серверный узел:

```
# gns -s
```

Клиентский узел (узлы):

```
# gns -c
```

В результате на узлах, где запущены службы глобальных имен, появятся имена `/dev/name/global` и `/dev/name/local`. Каждый узел, на котором запущен gns-клиент или сервер, в одной и той же сети имеет один и тот же вид пространства имен на `/dev/name/global`. Каждый узел имеет локальное пространство имен `/dev/name/local`, являющееся локальным для данной машины и отличающееся от локального пространства имен на другой машине. (Кстати, помимо имен `global` и `local` под `/dev/name/` появится еще имя `gns_server` или `gns_local` — имя, под которым регистрируется сам GNS-менеджер.)

Существует несколько функций API, относящихся к службе глобальных имен: `name_attach()`, `name_open()` и `name_close()`. Программисты, знакомые с QNX 4, сразу «узнают» в них аналоги известных им функций `qnx_name_attach()`, `qnx_name_open()` и `qnx_name_close()`. Приложения используют эти функции для объявления имени службы, связи со службой и отсоединения от службы.

Итак, чтобы объявить свое имя глобально в сети, приложение-сервер должно на узле, где в режиме сервера функционирует менеджер службы глобальных имен, объявить свою службу, выполнив вызов:

```
if( !( NameServer = name_attach( NULL, "MyService",
                                 NAME_FLAG_ATTACH_GLOBAL ) ) )
   return EXIT_FAILURE;
```

Флаг NAME_FLAG_ATTACH_GLOBAL указывает, что приложение-сервер объявляет свое имя глобально – в сети. Приложение, которое может подсоединить службу глобально, должно иметь право доступа root. После выполнения этого вызова в директории /dev/name/global появится подсоединенное имя MyService (если бы третий аргумент вызова был установлен в ноль, это имя оказалось бы подсоединенным к /dev/name/local и было бы доступно только локально).

Регистрируя имя в пространстве глобальных имен, функция name_attach() создает канал, идентификатор которого она возвращает в составе структуры NameServer. Отметим, что этот канал создается с определенными установленными флагами, задающими соответствующие действия системе:

- _NTO_CHF_UNBLOCK – доставлять владельцу канала импульс с кодом _PULSE_CODE_UNBLOCK и значением rcvid каждый раз, когда Reply-блокированный клиент попытается разблокироваться (скажем, по получению сигнала или по таймеру);
- _NTO_CHF_DISCONNECT – доставлять владельцу канала импульс с кодом _PULSE_CODE_DISCONNECT, когда от процесса отсоединились все установленные соединения клиента (клиент выполнил name_close() на каждый свой name_open() к имени сервера либо вообще умер);
- _NTO_CHF_COID_DISCONNECT – доставлять владельцу канала импульс с кодом _PULSE_CODE_COIDDEATH и значением coid (идентификатора соединения) для каждого соединения по этому каналу, когда канал закрывается.

Теперь, после создания канала, сервер может становиться на прием сообщений от клиентов:

```
rcvid = MsgReceive( NameServer->chid, &MsgBuf, sizeof MsgBuf );
```

Однако может так случиться, что клиент пошлет не непосредственное сообщение для сервера, а выполнит, скажем, чтение, что, по сути, тоже является отосланным сообщением. Поэтому при получении сообщений необходимо производить их «фильтрацию»:

```
if( MsgBuf.hdr.type >= _IO_BASE &&
    Buffer.hdr.type <= _IO_MAX ) {
   MsgError( rcvid, ENOSYS );
   continue;
}
```

Получив от клиента некое предопределенное сообщение, сервер сбрасывает флаг flagWork и выходит из петли ожидания сообщений, тем самым завершая свою работу.

С учетом этих деталей и организован нижеописанный сервер.

Код процесса-сервера, использующего службу глобальных имен

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/dispatch.h>

/* На сервер могут приходить и импульсы. Как минимум. */
typedef struct _pulse msg_header_t;

/* Структура сообщения состоит из заголовка и буфера наших данных */
typedef struct _MsgBuf {
   msg_header_t  hdr;
   char*        Buffer;
} MsgBuf_t;

int main() {
   name_attach_t* NameServer;
   MsgBuf_t MsgBuf;
   int rcvid;
   char BufReply[ 100 ];
   int flagWork = 1;

/* Создаем глобальное имя /dev/name/global/MyService */
   if( !( NameServer = name_attach( NULL, "MyService",
                                    NAME_FLAG_ATTACH_GLOBAL ) ) )
      return EXIT_FAILURE;

/* Становимся на петлю получения сообщений */
   while( flgWork ) {

      if( ( rcvid = MsgReceive( NameServer->chid, &MsgBuf,
                        sizeof MsgBuf, NULL ) ) == -1 ) {
         printf( "Ошибка при получении сервером MyService "
                 "сообщения от клиента\n" );
         fflush( stdout );
         break;
      }

      if( !rcvid ) {    // Получен импульс
        switch( MsgBuf.hdr.code ) {
            case _PULSE_CODE_DISCONNECT:
/* Поскольку для канала установлен флаг _NTO_CHF_DISCONNECT, ядро
   автоматически не освобождает связи, установленные клиентом ранее.
   Сервер должен выполнить это со своей стороны сам, "сознательно"
   удалив маршрут от себя обратно к клиенту. */
               ConnectDetach( MsgBuf.hdr.scoid );
               break;
            case _PULSE_CODE_UNBLOCK:
/* Клиент пытается разблокироваться, не дождавшись ответа по Reply.
   Надо выполнить какие-то действия, чтобы корректно (для себя)
```

```
            обработать эту ситуацию, и все-таки отпустить этого клиента -
            ему ведь надо! При этом импульсе в MsgBuf.hdr.value приходит rcvid */
                        MsgReply( MsgBuf.hdr.value.sival_int,
                                  EAGAIN, NULL, 0 );
                        break;
                    default: break;
                }
                continue;
        // вновь уходим на петлю приема сообщений
            }

        /* Полученное сообщение находится в диапазоне системных
            сообщений ввода/вывода. Не обрабатываем.  */
            if( MsgBuf.hdr.type >= _IO_BASE &&
                MsgBuf.hdr.type <= _IO_MAX ) {
                MsgError( rcvid, ENOSYS );
                continue;
            }

        /* А вот это - сообщение для сервера. Обрабатываем. */
            if( MsgBuf.hdr.type <= 0x50001 ||
                MsgBuf.hdr.type >= 0x500ff ) {
                printf( "Сервер получил сообщение неизвестно от"
                        " кого с меткой %#x\n", MsgBuf.hdr.type );
                strcpy( BufReply, "а кто это???" );
            }
            else {
                printf( "Сервер получил сообщение: \"%s\"\n",
                        MsgBuf.Buffer );
                strcpy( BufReply, "а, это ты, клиент" );
            }
            MsgReply( rcvid, EOK,
                      BufReply, strlen( BufReply ) + 1 );
        }
        // Конец петли получения сообщений

        /* Отсоединяемся от службы глобальных имен */
        name_detach( NameServer, 0 );
        return EXIT_SUCCESS;
    }
```

Приложения-клиенты, которым надо использовать глобальную службу имен, могут использовать функцию API `name_open()`. В случае, когда служба обеспечивается несколькими серверами (интересная и очень полезная возможность, заключающаяся в том, что несколько серверов вправе на различных узлах сети объявить одну и ту же службу), правила связи со службой имен таковы:

- Если поставщик службы имеется на том же узле, что и приложение, запросившее службу, менеджер пытается связать приложение прежде всего с локальным поставщиком. Если связь успешна, при-

ложение общается со своим поставщиком локально, что обеспечивает лучшую производительность.

- Если локальный поставщик отсутствует или по каким-то причинам отказывает в поставке службы, менеджер пытается связать приложение с другими поставщиками. Если имеется несколько удаленных поставщиков, то порядок, в котором производятся попытки установить с ними связь (т. е. кто получит связь первым), не определен.

Код процесса-клиента, использующего службу глобальных имен

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/dispatch.h>

/* На сервер могут приходить и импульсы. Как минимум. */
typedef struct _pulse msg_header_t;

/* Структура сообщения состоит из заголовка и буфера наших данных */
typedef struct _MsgBuf {
   msg_header_t hdr;
   char*       Buffer;
} MsgBuf_t;

int main() {
   MsgBuf_t MsgBuf;
   int fd;
   char BufReply[ 100 ];

   if( ( fd = name_open( “MyService”,
            NAME_FLAG_ATTACH_GLOBAL ) ) == -1 ) {
      printf( “Клиенту не удалось присоединиться к”
            “ сервису\n” );
      fflush( stdout );
      return EXIT_FAILURE;
   }

// Инвентаризационная метка данного клиента
   MsgBuf.hdr.type = 0x50001;
   MsgBuf.hdr.subtype = 0x00;

   strcpy( MsgBuf.Buffer, “Здравствуй, дорогой сервер!” );
   if( MsgSend( fd, &MsgBuf, sizeof MsgBuf,
              BufReply, sizeof BufReply ) == -1 ) {
      printf( “Клиент имеет проблемы с передачей сообщений”
            “ серверу\n” );
      fflush( stdout );
      name_close( fd );
      return EXIT_FAILURE;
```

```
    }
    printf( "Клиент получил от сервера такой ответ: "
            "\"%s\"\n", BufReply );
    name_close( fd );
    return EXIT_SUCCESS;
}
```

Тем, кто уже использовал функции работы со службой глобальных имен в предыдущей реализации ОС QNX 6.2 (где, как указывалось выше, она уже существовала, но могла функционировать только локально), следует обратить внимание, что в поведении этих функций появились небольшие изменения.

Раньше, когда приложение-клиент использовало вызов функции `name_open()` для связи с сервером, сервер об этом не знал. Теперь это изменено: серверу фактически отсылается сообщение `_IO_CONNECT / _IO_OPEN`. Кроме того, изменено приложение-сервер, чтобы иметь возможность обрабатывать приход сообщения `_IO_CONNECT`.

Заключение

Задача проведения сравнительного анализа рассмотренных нами методов организации обмена сообщениями с точки зрения их быстродействия или, скажем, объема исполняемого кода не показалась мне актуальной. Полагаю, что благодаря высокой эффективности механизма как такового различные его реализации не будут кардинально отличаться. А вот с выбором «правильных» критериев есть проблемы, по крайней мере, у меня.

Конечно, первый из рассмотренных методов является самым «спартанским» и поэтому наиболее эффективным по действию, однако и наиболее «хрупким», требующим внимательного обустройства, а также мало приемлемым в больших системах. Лично мне больше импонирует второй метод, но боюсь, что это просто личные пристрастия (возможно, нравится потому, что лентяю не может не доставлять удовольствия тот факт, что предлагается уже готовый прототип менеджера ресурсов, и самому надо делать совсем немного...). Выбирайте то, что больше всего нравится вам и более всего подходит для вашей конкретной задачи.

Литература

1. Роб Кертен «Введение в QNX/Neutrino 2». – СПб.: Петрополис, 2001. – 478 с.
2. Уильям Стивенс «UNIX: взаимодействие процессов». – СПб.: Питер, 2002. – 576 с.
3. Уильям Стивенс «UNIX: разработка сетевых приложений». – СПб.: Питер, 2003. – 1086 с.
4. Алексеев, Видревич, Волков, Горошко, Горчак, Жавнис, Сошин, Цилюрик, Чиликин «Практика работы с QNX». – М.: КомБук, 2004. – 432 с.
5. Кэйт Хэвиленд, Дайна Грей, Бен Салама «Системное программирование в UNIX». – М.: ДМК Пресс, 2000. – 368 с.
6. Теренс Чан «Системное программирование на C++ для UNIX». – К.: Издательская группа BHV, 1997. – 592 с.
7. Андрей Робачевский «Операционная система UNIX». – СПб.: BHV – Санкт-Петербург, 1997. – 528 с.
8. «QNX Neutrino Realtime Operating System. System Architecture». – QNX Software Systems Ltd, 2002.
9. Йон Снейдер «Эффективное программирование TCP/IP». – СПб.: Питер, 2001. – 320 с.
10. Э. Дейкстра «Взаимодействие последовательных процессов», сборник «Языки программирования» под ред. Ф. Женюи. – М.: Мир, 1972.
11. Стивен Янг «Алгоритмические языки реального времени. Конструирование и разработка». – М.: Мир, 1985. – 400 с.
12. Марк Митчелл, Джеффри Оулдем, Алекс Самьюэл «Программирование для Linux. Профессиональный подход». – М.: Издательский дом «Вильямс», 2002. – 288 с.
13. C. L. Liu and J.W. Layland «Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment», J. CM, Vol. 20, No. 1, Jan. 1973, pp. 46–61.
14. The Open Group Base Specifications Issue 6, IEEE Std 1003.1-2001. General Information. Copyright 2001 The IEEE and The Open Group.

Алфавитный указатель

Q

R

S

T

U

V

W