Руководство системного администратора UNIX

2-е издание

Настройка производительности UNIX-систем

СИМВОЛ®

O'REILLY®

Джан-Паоло Д. Мусумеси и Майк Лукидес

System Performance Tuning

Second Edition

Gian-Paolo D. Musumeci,
Mike Loukides

O'REILLY®

Настройка производительности UNIX-систем

Второе издание

Джан-Паоло Д. Мусумеси,
Майк Лукидес

Санкт-Петербург — Москва
2003

Джан-Паоло Д. Мусумеси, Майк Лукидес

Настройка производительности UNIX-систем, 2-е издание

Перевод Ю.Кунивера

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*Ф. Торчинский*
Редактор	*Ю. Кунивер*
Корректор	*С. Беляева*
Верстка	*Н. Гриценко*

Мусумеси Д.-П., Лукидес М.
Настройка производительности UNIX-систем. – Пер. с англ. – СПб: Символ-Плюс, 2003. – 408 с., ил.
ISBN 5-93286-034-0

Книга «Настройка производительности UNIX-систем» отвечает на два важнейших вопроса: как добиться максимального эффекта без покупки дополнительного оборудования и в каких случаях его все же стоит приобрести (больше памяти, более быстрые диски, процессоры и сетевые интерфейсы). Вложение денежных средств – не панацея. Адекватно оценить необходимость обновления и добиться максимальной производительности можно только хорошо представляя работу компьютеров и сетей и понимая распределение нагрузки на системные ресурсы.

Авторы книги оказали неоценимую помощь администраторам, подробно и аргументированно рассказав обо всех тонкостях искусства настройки систем. Полностью обновленное издание ориентировано на Solaris и Linux, но обсуждаемые принципы применимы к любым системам. В книге рассматриваются настройка параметров, управление рабочим процессом, методы измерения производительности, выявление перегруженных и неработоспособных участков сети, добавлен новый материал о дисковых массивах, микропроцессорах и оптимизации программного кода.

ISBN 5-93286-034-0
ISBN 0-596-00284-X (англ)

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 21.10.2003. Формат 70x100 1/16 . Печать офсетная.
Объем 25,5 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Эта книга об искусстве настройки систем, которая необходима для оптимальной производительности прикладных программ. На страницах книги обсуждается выполнение самой настройки и извлечение максимальной пользы из приложений. Здесь рассматриваются базовые алгоритмы, лежащие в основе настройки параметров; их освоение позволит принимать разумные решения в любой операционной среде. Кроме того, затрагивается планирование нагрузки систем – речь пойдет о регулировании системы для решения различных задач.

Для кого написана эта книга

В основном книга предназначена для тех, кто заинтересован в оптимизации производительности своих компьютерных систем. Читатели найдут ясное описание того, как различные компоненты системы взаимодействуют друг с другом и на что следует обращать внимание. Программисты, пишущие или оптимизирующие программы, найдут краткие пояснения к ключам современных компиляторов, а также разбор того, как базовая операционная система управляет запущенными приложениями. Наконец, те читатели, которые просто хотят знать больше о работе компьютеров, найдут на этих страницах интересные разъяснения.

Охват книги

Эта книга в значительной степени ориентирована на операционную среду Solaris (до версии Solaris 8 включительно) и систему Linux. Однако особое значение в книге придается именно Solaris. Для этого есть несколько причин:

- Большинство хранилищ данных работают под управлением Solaris. Такие операционные среды наиболее требовательны к настройке производительности.
- Машины Solaris более других систем сконцентрированы на производительности. Видимо, это объясняется тем, что системы Sun в среднем являются более дорогостоящими, чем их Linux-аналоги.

В результате люди ожидают лучшей производительности, поэтому в этой области в Solaris приложено немало усилий. Если производительность машины Linux недостаточна, то можно купить другую машину и распределить между ними нагрузку – это дешево. Если же Ultra Enterprise 10000 стоимостью несколько миллионов долларов не работает должным образом, а компания каждую минуту теряет из-за этого нетривиальные суммы, то пользователи звонят в Sun Service и требуют ответа.

- Наконец, инструменты для анализа производительности в Solaris *более* серьезные, чем в Linux. Частично это связано с тем, что разработать такие инструменты не так-то просто. А частично с тем, что порой для достижения наилучшей производительности необходимо проводить изменения в аппаратных средствах. Кроме того, Linux – это относительно новая операционная система, разработанная сравнительно небольшим коллективом. В свою очередь, Solaris (и ее предшественница SunOS, давшая уйму обратной связи для процесса разработки Solaris) – это операционная система с богатым прошлым, а Sun – это огромная компания. Возможность тщательного анализа проблемы производительности и принятия грамотных решений по оптимизации в значительной мере основаны на данных, собранных соответствующими инструментами (в противном случае настройка сводится к гаданию на кофейной гуще). Поэтому в конечном итоге настраивать системы Linux чрезвычайно трудно. Такая настройка прямо противоречит одному из основных принципов настройки производительности (см. раздел «Принцип 0: Хорошо понимайте свою операционную среду» главы 1).

Не будем предвзяты в отношении любой операционной системы.[1] Скажем лишь, что серьезная оптимизация производительности в Linux на сегодня является непростой задачей, ибо средств, облегчающих понимание работы Linux, пока не существует.

При запуске Linux совершенно *необходимо* раздобыть и инсталлировать пакет *sysstat*, предоставляющий версии *iostat*, *mpstat* и *sar* (в урезанном варианте). Будучи весьма неполными по сравнению с версиями в Solaris, они по крайней мере помогают получить хоть какую-то информацию. На время написания книги эти программы можно было загрузить с веб-сайта их автора Себастьяна Годарда (Sebastien Godard) *http://perso.wanadoo.fr/sebastien.godard/*.

Как читать эту книгу

Эта книга – и справочник и рассказ. Читатели с опытом анализа и настройки производительности, возможно, загорятся желанием немед-

[1] В интересах полноты представления: автор работал в Sun Microsystems около половины того времени, которое заняло написание этой книги.

ленно перескочить к конкретному разделу (наверное, к тому, где затрагиваются слабые, с их точки зрения, места в системе) и станут рассматривать книгу как справочник. Так поступать ни в коем случае не следует.

Эта книга как рассказ

Для наибольшей пользы лучше сначала прочесть эту книгу как рассказ. Навыки, методы размышления и подходы к задачам – вот самое важное, что можно почерпнуть из текста. Такая информация впитывается на протяжении времени, а не приходит путем запоминания пунктов из списка.

Многие из обсуждаемых тем будут казаться довольно простыми и незамысловатыми. На самом деле они напрямую связаны с основами архитектуры современных компьютеров, а именно с тем, как компьютеры вообще работают. Зачастую эти темы весьма запутанны. Не стоит беспокоиться, если разделы книги придется перечитывать, поскольку для усвоения материала необходимо время.

Эта книга как справочник

Для тех, кто прочел рассказ и получил представление о том, как, словно мозаика, складывается из кусочков производительность, эта книга будет полезным справочником. Кто-то может не прочесть рассказ и, рассматривая одну из таблиц (например, описание параметров памяти ядра в разделе «Управление виртуальной памятью в Linux» главы 4), думать, что он усвоил все подробности. На самом деле это не так – он многое упустил. Прочтение этой книги как рассказа позволит в деталях понимать сложные задачи, связанные с настройкой системы.

Структура книги

Эта книга состоит из девяти глав:

Глава 1 «Введение в настройку производительности» закладывает фундамент для всей книги. В ней освещены основные принципы настройки.

Глава 2 «Управление рабочим процессом» описывает управление производительностью на основе осмысления работы системы и ограничения нагрузки. Кроме того, в ней рассказывается о некоторых распространенных методах измерения производительности.

В главе 3 «Процессоры» обсуждаются архитектура современного процессора, кэширование, многопроцессорные системы и то, как операционная система планирует задачи.

Глава 4 «Память» объясняет, как в компьютере организована подсистема памяти и как она взаимодействует с другими компонентами системы.

Глава 5 «Диски» представляет диски в простейшем виде: как функционируют жесткие диски и что можно предпринять в отношении некоторых связанных с ними ограничений.

Глава 6 «Дисковые массивы» в каком-то смысле является продолжением предыдущей главы. В ней обсуждается методика объединения многочисленных дисков в одно логическое устройство.

Глава 7 «Сети» посвящена производительности сети: от физического уровня до протоколов связи, таких как стек TCP/IP, и систем совместного использования файлов Samba и NFS.

Глава 8 «Оптимизация кода» – это лаконичное введение в методику создания быстродействующего программного кода, а также сводка по работе с современными компиляторами.

Глава 9 «Первоочередная настройка» – это краткое изложение некоторых важных пунктов, обсуждаемых на протяжении всей книги. Она представляет собой справочник для быстрого устранения неполадок и конфигурирования системы.

Соглашения по оформлению

В книге приняты следующие соглашения:

`Моноширинный шрифт`

Используется для примеров исходного кода и фрагментов исходного кода в самом тексте, включая имена переменных и функций. Моноширинный шрифт также применяется для отображения результатов, выданных компьютером, и содержимого файлов.

`Моноширинный полужирный`

Применяется для команд, набираемых пользователем.

Курсив

Используется для названий команд, имен каталогов и файлов. Также применяется для выделения новых терминов.

Полужирный

Применяется для выделения векторов (в математическом смысле) и ключей команд.

Комментарии и вопросы

Вся информация, приведенная в книге, была по возможности протестирована и проверена. В то же время какие-то детали могли измениться, а опечатки или ошибки быть обнаружены. О таких данных, а так-

же о своих предложениях по поводу будущих изданий можно сообщать по адресу:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800)998-9938 (для США или Канады)
(707)829-0515 (международный/местный)
(707)829-0104 (факс)

Кроме того, можно посылать сообщения по электронной почте. Чтобы попасть в список рассылки или заказать каталог, присылайте письмо по адресу

info@oreilly.com

Чтобы задать технические вопросы или прислать комментарии к книге, присылайте письмо по адресу

bookquestions@oreilly.com

Существует сайт, где приведены примеры из книги, список ошибок и планы на будущее. Эта страница доступна по адресу

http://www.oreilly.com/catalog/spt2/

За более подробной информацией об этой и других книгах можно обратиться на сайт издательства по адресу

http://www.oreilly.com

Благодарности Джан-Паоло Мусумеси

Я в долгу перед многими людьми за их помощь и поддержку во время написания этой книги.

Прежде всего, мне хочется поблагодарить моего редактора, Майка Лукидеса (Mike Loukides). Срок сдачи книги отодвигался много раз, и его терпение, замечательные советы и оптимизм оказались жизненно важными для выхода книги. Также я очень признателен моей организационной команде в Университете штата Иллинойс, особенно Моне Хит (Mona Heath) и Эду Кролу (Ed Krol), и в Sun Microsystems – доктору Кенг-Тай Ко (Keng-Tai Ko) и Мирославу Кливански (Miroslav Klivansky). Их поддержка, терпение и понимание были бесценны.

В ходе работы над книгой мои друзья оказывали мне постоянную поддержку. Они давали технические консультации и понимали фразу «Нет, я не приду, я должен работать над книгой», которая неожиданно вошла в мой лексикон на два года. Мне хочется особенно поблагодарить Кейт Вессел (Keith Wessel), Криса Вехнера (Kris Wehner), Майди Аббас (Majdi Abbas), Деб Флигор (Deb Fligor), Джея Крейбича (Jay Kreibich), Адриана Кокрофта (Adrian Cockcroft), Ричарда Макдугала

(Richard McDougall), Элизабет Парсел (Elizabeth Purcell), Тауфун Косаглу (Tayfun Kocaglu), Пола Стронга (Paul Strong), Фила Страччино (Phil Stracchino), Кристофа Харпера (Christof Harper) и Кена МакИнниса (Ken MacInnis). Порядок здесь не важен. Доктор Санья Пател (Sanjay Patel) был очень любезен допустить учащегося на свой курс по архитектуре компьютеров в Университете штата Иллинойс без прохождения предварительных курсов. Его поддержка была неоценима. Роза Платт (Rose Platt), Ненси Вебер (Nancy Weber) и доктор Роберт Кан (Robert Cahn) открыли для меня свои двери. Благодаря им я мог долгое время сидеть и писать в надежном уединении. Кейт Секор (Kate Secor) помогла мне сохранить рассудок, когда я писал последнюю часть этой книги. Она шутит, что была моим «хорошим рабочим тотемом». Когда Кейт была рядом, работа шла значительно быстрее.

Трудно переоценить вклад моей семьи, особенно моих родителей, докторов Диану и Антонио Мусумеси. Присутствие людей, которые вырастили тебя, порою очень обнадеживает. Особенно если они оба написали книги – не так уж это и трудно, правда?[1] Мои братья Доменико и Уолтер не дали мне оторваться от реального мира. Мой дедушка Уолтер напомнил мне, что бывали времена и без компьютеров. Домашний пес Гизмо был постоянным источником забав (очень трудно сосредоточиться, когда 15-футовый бордер-терьер носится по дому со всей скоростью, на какую способен; кроме того, иногда полезно *не* сосредотачиваться ни на чем).

Я рад посвятить эту книгу моей бабушке, Кэтлин Миколайтис.

Благодарности Майка Лукидеса

Прежде всего, я хочу поблагодарить Джан-Паоло Д. Мусумеси (Gian-Paolo D. Musumeci). Я бы никогда не сделал то, что сделал он. Подготовка второго издания книги – это действительно его заслуга. Я горжусь первым изданием, но должен признаться, что в основном это была журналистика. Я общался с группой системных экспертов и записывал их опыт. Это были Дуг Гилмор (Doug Gilmore), Крис Райленд (Chris Ryland), Тен Бронсон (Tan Bronson) и другие эксперты из сообщества Multiflow и не только. (Мало кто знает, что эта книга зарождалась как часть комплекта документации Multiflow TRACE/UNIX, хотя Multiflow прекратил свое существование еще до публикации.) В то время как я – просто журналист, Джан-Паоло является первоклассным специалистом. Он привнес в книгу огромные знания и опыт, которых у меня никогда не было. И как бы я ни гордился первым изданием, второе издание значительно лучше.

[1] Ответ: *очень трудно*.

- *Введение в архитектуру компьютера*
- *Принципы настройки производительности*
- *Настройка статической производительности*
- *Заключение*

1

Введение в настройку производительности

Тысяча испытаний во владениях зла
ожидает того, кто покушается на основы.

Henry David Thoreau, 1854

Пожалуй, стремление двигаться быстрее есть сущность человеческой эволюции. Всего триста лет назад наивысшая скорость движения, которую можно было себе представить, составляла несколько десятков миль в час – на борту быстроходного парусного судна со свежим ветром за спиной. Теперь пришло время расширить наши воззрения на максимально достижимую скорость в пределах земной атмосферы – возможно, пятнадцать тысяч миль в час, вдвое больше скорости звука. Таков потолок для штатских лиц, не имеющих доступа к новейшим военным самолетам. Путешествие под парусом из Лондона в Нью-Йорк ранее занимало три недели. Сегодня такое расстояние можно преодолеть всего за два с половиной часа, попивая при этом шампанское.

Врожденное человеческое стремление быть мобильнее выражается во многом: микроволновые печи позволяют быстро приготовить обед, мощные автомобили и мотоциклы несутся на бешеной скорости, электронная почта позволяет общаться со скоростью мысли. Но что происходит, когда почтовый сервер перегружен, если все мы одновременно подключаемся к серверу, чтобы проверить, не пришло ли что-нибудь, пока мы спали? Или когда система снабжения компании, продающей микроволновые печи, способна осилить лишь половину нагрузки, или когда система CAD инженера-механика работает столь медленно, что двигатель для новой модели года, разрабатывающийся с ее помощью, не будет готов в срок?

Все это – задачи, подводящие к необходимости настройки производительности: адаптация скорости компьютерной системы к требованиям по скорости, предъявляемым реальным миром.

Иногда эти задачи менее заметны, иногда более. Необходимо тщательно анализировать изменения в системе, приведшие к тому, что система стала работать недопустимо медленно. Такие воздействия могут как скрываться внутри, например избыточная нагрузка на систему, так и появляться извне. К примеру, незначительное (или полное) изменение алгоритма управления ресурсами в новой редакции операционной системы, то есть нечто чрезвычайно важное для программного обеспечения. Решения порою бывают мгновенными (всего-то выверить одну опцию), а иногда медленными и болезненными (недели анализа, консультаций с производителями и внимательной перепланировки инфраструктуры).

В начале работы над книгой все казалось незатейливым: взять замечательное, но устаревшее первое издание книги Майка Лукидеса (Mike Loukides) и доработать ее с учетом знаний о современных компьютерных системах. Но чем дальше, тем более становилось ясно, что на самом деле эта книга даст намного больше, чем просто описание настройки производительности. В действительности она охватывает две отдельные темы:

Настройка производительности

Искусство повышения производительности для конкретного набора приложений (также известное как «выдавливание крови из камня»).

Планирование нагрузки

Размышления о том, какое оборудование приобрести для решения задач (также известные как «предсказание будущего»).

Эти темы опираются на теорию архитектуры компьютера. Эта книга посвящена не разработке приложений; скорее она фокусирует внимание на операционной системе, базовых аппаратных средствах и их взаимодействии.

Для большинства системных администраторов компьютер представляет собой черный ящик. Это весьма разумно для многих задач: в конце концов, чтобы конфигурировать и обслуживать почтовый сервер, естественно, нет необходимости понимать, как операционная система управляет свободной памятью. Однако при настройке производительности, которая, по существу, зиждется на знании базовых аппаратных средств и того, как они абстрагированы, подлинное понимание поведения системы приходит с обстоятельными сведениями о внутренней работе машины. В этой главе мы кратко обсудим некоторые наиболее важные понятия архитектуры компьютера, а затем обратимся к основным принципам настройки производительности.

Введение в архитектуру компьютера

Полное обсуждение архитектуры компьютера выходит за рамки этой книги. Однако время от времени вопросы архитектуры будут рассматриваться, чтобы при обсуждении системы исходить из первооснов. Интересующимся этим предметом можно посоветовать обратиться к замечательным руководствам по этой теме. Возможно, наиболее читаемыми являются руководства Джона Хеннесси (John Hennessy) «Computer Organization and Design: The Hardware/Software Interface» (Организация и проектирование компьютера: интерфейсы программных и аппаратных средств) и Давида Паттерсона (David Patterson) «Computer Architecture: A Quantitative Approach» (Архитектура компьютера: количественный подход). Обе книги изданы Морганом Кауфманом (Morgan Kaufmann).

Этот раздел будет посвящен двум важнейшим общим концепциям архитектуры: базовым средствам, с помощью которых мы подходим к задаче (уровни представления), и основной модели, на базе которой созданы компьютеры.

Уровни представления

Сведем задачу к обзору того, что присуще работе компьютера: от логических схем, дающих решение важнейшей задачи – как создавать компьютеры общего назначения, до нескольких миллионов бит двоичного кода. По мере прохождения этих этапов задача будет преобразована в более простую (по крайней мере с точки зрения компьютера). Назовем эти шаги *уровнями представления*.

Программное обеспечение: алгоритмы и языки

Перед лицом задачи, предполагающей помощь компьютера, в первую очередь разрабатывается *алгоритм* с целью свести задачу к заданию. Проще говоря, алгоритм – это итеративная последовательность инструкций для выполнения конкретной задачи – например, как должным образом сортировать почту клерку, изучающему и распределяющему входящую корреспонденцию.

Далее программист должен преобразовать этот алгоритм в программу, написанную на *языке*.[1] Как правило, это язык высокого уровня, такой как C или Perl, хотя это может быть и язык низкого уровня, такой как ассемблер. Наличие языка делает жизнь легче: структура и грамматика языков высокого уровня позволяют легко писать сложные программы. Программы, написанные на языке высокого уровня (обычно пере-

[1] Было высказано предположение, что математики – это устройства для преобразования кофе в теоремы. Если это правда, то, видимо, программисты – это устройства для преобразования кофеина и алгоритмов в исходный код.

носимые между различными системами), затем преобразуются компилятором в команды нижнего уровня – в соответствии с архитектурой конкретной системы. Эти команды определяются структурой системы команд.

Структура системы команд

Структура системы команд (*Instruction Set Architecture*, ISA) – это основной язык микропроцессора: он определяет базовые, неделимые команды, которые могут быть выполнены. ISA служит интерфейсом между программными и аппаратными средствами. Примеры структуры системы команд включают IA-32, используемую в процессорах Intel и AMD, MIPS, реализованную в R-серии микропроцессоров Silicon Graphics/MIPS (пр. R12000), и систему SPARC V9, применяемую в сериях UltraSPARC компании Sun Microsystems.

Аппаратура: микроархитектура, схемы и устройства

На этом уровне происходит погружение в электронику и схемотехнику. Сначала функциональные блоки микроархитектуры и эффективность разработки. Затем, ниже уровня микроархитектуры, реализация функциональных блоков на основе разработки схем: здесь становятся реальными проблемы электрических взаимных наводок. Полное обсуждение уровня аппаратных средств не входит в замысел книги; настройка различных микропроцессоров далеко не всегда возможна.

Модель фон Неймана

Модель фон Неймана (von Neumann) служила базовой моделью при проектировании всех современных компьютерных систем: она дает каркас, на который можно повесить абстракции и наполнить их «плотью», сформированной средствами уровней представления.[1]

Модель состоит из четырех ключевых компонентов:

- *Система памяти*, которая хранит как команды, так и данные. Известна как *система с хранимой программой*. Доступ к этой памяти осуществляется с помощью *регистра адреса* (*memory address register*, MAR), куда подсистема памяти помещает адрес ячейки памяти, и *регистра данных* (*memory data register*, MDR), куда она помещает данные из ячейки с указанным адресом. Более подробно подсистема памяти обсуждается в главе 4.
- По крайней мере один *блок обработки данных*, наиболее известный как *арифметико-логическое устройство* (ALU). Эти блоки чаще

[1] Отличная книга с более полным описанием модели фон Неймана – Уильям Аспрей (William Aspray) «John von Neumann and the Origins of Modern Computing» (Джон фон Нейман и истоки современной вычислительной техники), издательство MIT Press.

называют *центральными процессорами* (CPU).[1] Этот блок отвечает за выполнение всех команд. Процессор также имеет небольшой объем памяти, называемый *набором регистров*. Обсуждение процессоров будет более подробным.

- *Блок управления*, отвечающий за операции между компонентами модели. Включает в себя *счетчик команд*, содержащий следующую команду для загрузки, и *регистр команд*, в котором находится текущая команда. Особенности модели управления выходят за рамки этой книги.
- Системе необходим энергонезависимый способ хранения данных, а также выдачи их пользователю и принятия входных данных. Это прерогатива подсистемы *ввода-вывода* (*I/O*). Книга главным образом касается дисковых накопителей как механизмов для ввода-вывода. Они будут обсуждаться в главе 5, а вопросы сетевого ввода-вывода – в главе 7.

Несмотря на существенный прогресс в компьютерной технике за последние шестьдесят лет, устройство компьютеров по-прежнему умещается в этих рамках. Это очень весомое утверждение: несмотря на то, что компьютеры стали мощнее, а их нынешнее применение было невозможно даже представить в конце второй мировой войны, основные идеи, заложенные фон Нейманом и его коллегами, пригодны и сегодня.

Кэши и иерархия памяти

Как будет позднее рассмотрено в разделе «Принципы настройки производительности» этой главы, один из принципов настройки производительности гласит: без компромиссов *не* обойтись. Это было выяснено первопроходцами в этой области, а идеальное решение не найдено и по сию пору. При проектировании системы памяти часто приходится выбирать между ценой, скоростью и емкостью. (Физические параметры, скажем, теплоотвод, также играют свою роль, но в рамках этого обсуждения они скрыты за другими переменными.) Конечно, можно предложить чрезвычайно большие, чрезвычайно быстрые системы, например суперкомпьютер Cray 1S, который использовал очень быстрое статическое ОЗУ (RAM) исключительно для памяти.[2] Но это решение не всегда можно повторить для других компьютерных устройств.

Итак, проблема состоит в том, что емкость запоминающих устройств обратно пропорциональна производительности, особенно в отношении наивысшего соотношения цена–производительность. Современный

[1] В современных реализациях под термином «CPU» подразумевается как сам центральный процессор, так и блок управления.

[2] Трудности из-за тепловыделения, касающиеся памяти, были первейшей причиной предусмотреть в системе водоохлаждение. Кроме того, при типичной конфигурации стоимость подсистемы памяти составляет около трех четвертей стоимости машины.

микропроцессор может иметь время цикла, измеряемое в долях наносекунды, в то время как доступ к оперативной памяти вполне может быть в пятьдесят раз медленнее.

Чтобы разрешить эти трудности, возьмем на вооружение так называемую *иерархию памяти*. Она основана на создании пирамиды участков памяти (рис. 1.1). Вверху пирамиды располагаются очень маленькие, чрезвычайно быстрые участки памяти. Ниже представлены более медленные участки, зато соответственно большего размера. В основании пирамиды находится библиотека на лентах: много терабайт, но доступ к запрашиваемой информации может занимать минуты.

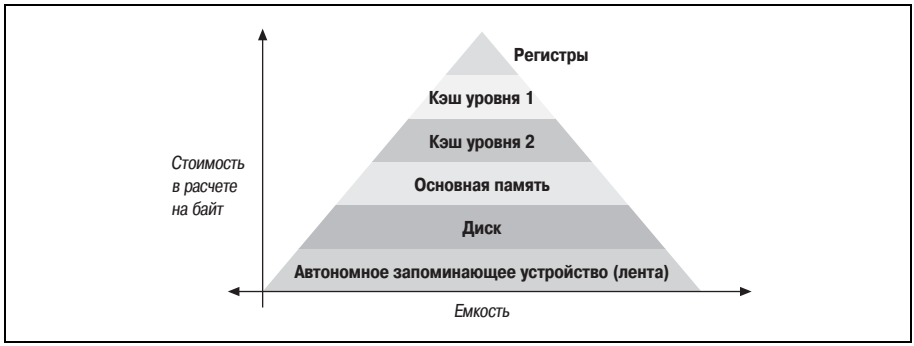


Рис. 1.1. *Иерархия памяти*

С точки зрения микропроцессора оперативная память очень медленная. Все, что связано с обращением к оперативной памяти, неудачно – если только мы не адресуемся к оперативной памяти вместо того, чтобы обратиться к еще более медленному носителю (такому как диск).

Назначение пирамиды – кэшировать наиболее часто используемые данные и команды на более высоких уровнях. Например, если необходимо снова и снова обращаться к одному и тому же файлу на ленте, то было бы неплохо хранить временную копию на следующем по быстродействию уровне хранения (диск). Таким же образом, учитывая существенное премущество в производительности, можно хранить файл в основной памяти, если к нему идет частое обращение на диске.

Выгоды 64-разрядной архитектуры

Компании, производящие аппаратные и программные средства компьютеров, часто считают необходимым упомянуть размер адресного пространства их систем (обычно 32 или 64 бит). За последние пять лет скачок от 32-разрядных к 64-разрядным микропроцессорам и операционным системам вызвал много крикливой рекламы со стороны отделов сбыта. Истина состоит в том, что хотя в определенных случаях

64-разрядная архитектура работает значительно быстрее 32-разрядной, в основном их производительность сопоставима.

Что понимается под 64 разрядами?

Количество «разрядов» имеет отношение к ширине шины данных. Однако на самом деле все зависит от контекста. Например, можно говорить о 16-разрядной шине данных (скажем, UltraSCSI). Это означает, что такое соединение позволяет передавать 16 бит информации за единицу времени. При условии одинаковости всего остального это означает в два раза более быстрое взаимодействие, нежели в случае с 8-разрядной шиной.

«Битовость» запоминающей системы определяется тем, сколько адресных линий используется при передаче адреса памяти. Например, при наличии 8-разрядной шины и необходимости доступа к 19-му участку памяти применяются соответствующие адресные линии (1, 2 и 5 – исходя из 19 в двоичной системе (00010011); там, где стоит единица, активизируется адресная линия). Заметим, однако, что на 8-битную адресацию налагается ограничение 64 (2^8) адреса в памяти. 32-разрядные системы, следовательно, ограничены 4 294 967 296 (2^{32}) позициями. Так как обычно извлекать информацию из памяти можно однобайтными блоками, это означает, что системе может быть непосредственно доступно не более 4 Гбайт памяти. Переход на 64-разрядные операционные системы и аппаратные средства знаменует, что максимальный объем адресуемой памяти составляет около 16 петабайт (16 777 216 Гбайт), что, вероятно, вполне достаточно для обозримого будущего.

К сожалению, на практике не все так просто. 32-разрядная система SPARC допускает применение более чем 4 Гбайт установленной памяти, но в Solaris ни один отдельный процесс не может использовать более 4 Гбайт. Это вызвано тем, что аппаратура, отвечающая за управление памятью, действительно использует 44-битную адресную схему, но операционная система Solaris отводит одному процессу объем памяти, адресуемый всего лишь 16 бит.

Последствия для производительности

Переход от 32- к 64-разрядной архитектуре к тому же увеличил емкость основной памяти и объема памяти, выделяемой отдельному процессу. Обычный вопрос: что от этого выиграли приложения? Вот несколько видов приложений, выигравших от большего адресного пространства:

- Приложения, ранее не способные применять для решения задач наиболее эффективные с точки зрения времени алгоритмы, поскольку эти алгоритмы были ориентированы на память более 4 Гбайт.
- Приложения, в которых кэширование большого объема данных критически важно, а потому чем больше памяти доступно процессу, тем больше данных может быть кэшировано.

- Приложения в системах, где память – узкое место, вследствие непомерного к ней обращения (много маленьких процессов). Заметим, что в системах SPARC это не было проблемой: каждый процесс мог видеть только 4 Гбайт, но памяти могло быть инсталлировано намного больше.

Вообще говоря, больше всех от 64-разрядных систем выиграли машины для высокопроизводительных вычислений и корпоративных баз данных. Для средних настольных рабочих станций 32 разрядов вполне достаточно.

К сожалению, переход на 64-разрядные системы также означал, что базовые операционные системы и системные вызовы нужно было модифицировать. Порой это приводило к снижению производительности (например, при работе с указателями приходится иметь дело с большим количеством данных). Это означает, что при запуске в 64-разрядном режиме возможно небольшое снижение производительности.

Принципы настройки производительности

В этой книге представлено несколько хорошо зарекомендовавших себя практических правил. Как говорится в давнем техническом бюллетене, выпущенном IBM, такие правила исходят от людей, которые живут за городом и не отягощены производственным опытом.[1]

Помня об этом, основы настройки производительности системы можно свести к пяти принципам: хорошо понимайте свою операционную среду, нет ничего по-настоящему бесплатного, пропускную способность и время ожидания не меряют одной меркой (т. к. по сути это разные вещи), ресурсы не должны быть перегружены, а эксперименты следует проводить внимательно.

Принцип 0: следует хорошо понимать свою операционную среду

При недостаточно хорошем понимании своей операционной среды возникающие задачи, скорее всего, не решить. Вот почему концептуальному материалу в этом руководстве придается большое значение. Даже если изменятся отдельные детали реализации алгоритма или же сменится сам алгоритм, теоретические знания о том, что представляет собой задача и как подходить к ее решению, останутся действенными. Понимать задачу – значит обладать гораздо большим потенциалом по сравнению с ситуацией, когда известно решение, но нет понимания того, откуда возникла проблема.

Конечно, в какой-то мере сложные случаи лежат вне компетенции среднего системного администратора: настройка производительности

[1] «MVS Performance Management», GG22-9351-00.

сети на обсуждаемом здесь уровне не требует углубленных знаний того, как реализован стек TCP/IP на уровне модулей и вызовов функций. Заинтересованным в деталях работы различных вариантов операционной системы UNIX следует обратиться к нескольким замечательным руководствам: «Solaris Kernel Internals» (Внутренности ядра Solaris) Ричарда Мак-Дугалла (Richard McDougall) и Джеймса Мауро (James Mauro), «The Design of the UNIX Operating System» (Реализация операционной системы UNIX) Мауриса Баха (Maurice J. Bach) и «Operating Systems, Design and Implementation» (Операционные системы: проектирование и реализация) Эндрю Таненбаума (Andrew Tanenbaum) и Эндрю Вудфулла (Andrew Woodfull). Все книги изданы Prentice Hall.

Конечно, важнейшая ссылка – это сам исходный код. На время написания книги исходный код для всех детально описанных в этой книге операционных систем (Solaris и Linux) является бесплатным и доступным.

Принцип 1: БСНБ!

БСНБ означает, что Бесплатного Сыра Не Бывает.[1] По существу, настройка производительности представляет собой нахождение компромисса между различными характеристиками. Обычно это список из трех желательных свойств, из которых можно выбрать только два.[2]

Один пример приходит из настройки сетевого уровня TCP, где алгоритм Нагла (Nagle) приносит в жертву время ожидания или время, требуемое для доставки одиночного пакета, в обмен на повышение пропускной способности, или объем данных, которые можно фактически протолкнуть по проводам. (Алгоритм Нагла обсуждается более детально в разделе «Алгоритм Нагла» главы 7.)

Этот принцип часто обязывает делать реальный, значимый и трудный выбор.

Принцип 2: пропускная способность против времени ожидания

Во многих отношениях системные администраторы, оценивающие компьютерные системы, зачастую подобны юнцам, оценивающим автомобили. К сожалению, в обоих случаях существует определенный набор показателей и стремление найти наивысшее значение для наиболее «важного» показателя. Обычно это «величина пропускной способности» для компьютеров и «лошадиная сила» для автомобилей.

---

[1] Наши извинения Роберту Хайнлайну (Robert A. Heinlein) «The Moon Is A Harsh Mistress» (Луна – суровая хозяйка).

[2] Мы выполняем ваш заказ быстро, качественно и недорого. Выбирайте любые два пункта. – *Примеч. науч. ред.*

Шаги, которые готовы делать люди, чтобы выжать максимум лошадиных сил из своих четырехколесных транспортных средств, нередко отчасти смехотворны. Незначительное изменение ракурса обнаруживает, что в жемчужине производительности существуют и другие грани. Много усилий тратится на оптимизацию отдельных параметров, что, вполне вероятно, не является настоящим подспорьем. Проиллюстрируем это примитивным сопоставлением (пожалуйста, помните, что речь идет только о сравнении производительности). Транспорт A дает около 250 лошадиных сил, в то время как мощность транспорта B – около 95. Кто-то готов предположить, что транспорт A на практике демонстрирует существенно «лучшую» производительность, нежели транспорт B. Проницательный читатель может спросить: «А какой вес у этих транспортных средств?» Вес транспорта A составляет около 3600 фунтов, в то время как транспорт B весит около 450. Теперь видно, что транспорт B на самом деле значительно быстрее (разгон с нуля до 100 км в час примерно за три с половиной секунды против неторопливого транспорта A с пятью с половиной секундами). Однако если сравнивать настоящую скорость движения по переполненной 101-й магистрали на полуострове Сан-Франциско, транспорт B победит, и даже с большим перевесом, потому что мотоциклам в Калифорнии позволено ездить между полосами движения (между рядами машин).[1]

Возможно, в этом мире компромиссов более всего пренебрегают временем ожидания. К примеру, представим вымышленное приложение – почтовый сервер. Назовем его СуперПуперПочта. Рекламные материалы этого приложения обещают, что сервер СуперПуперПочта способен обрабатывать свыше миллиона писем в час. Это может показаться вполне рациональным: такая скорость значительно выше скорости, требуемой большинством компаний. Другими словами, пропускная способность в целом хорошая. А что если взглянуть на производительность этого почтового сервера с другой стороны и спросить, как долго обрабатывается одно письмо? После нескольких наводящих вопросов к отделу сбыта СуперПуперПочты выясняется, что полчаса. Это выглядит противоречиво: как будто программа может обрабатывать самое большее два сообщения в час. Однако оказывается, что изнутри сервер СуперПуперПочта основан на последовательности приемников и перемещает письма к следующему приемнику, когда текущий приемник полностью заполнен. В этом примере, несмотря на приемлемую пропускную способность, время ожидания *ужасно*. Кто захочет послать письмо человеку, сидящему в соседней комнате, если доставка займет полчаса?

---

[1] Для любознательных. Траспорт A – Audi S4 седан (2,7 литра с двойным турбонаддувом V6). Транспорт B – мотоцикл Honda VFR800FI Interceptor (четырехклапанный атмосферный двигатель 781 см³). Оба вида транспорта – модели 2000 года.– *Примеч. науч. ред.*

Принцип 3: не перегружайте ресурсы

Каждому, кто управлял автомобилем на запруженной магистрали, знакома ситуация с ориентированием в незнакомой местности: знак «нижний предел скорости» выглядит злой шуткой! Очевидно, существует много факторов, относящихся к развитию сообщения между штатами, особенно один, тяжело воспринимаемый пригородными участниками движения: пиковая нагрузка существенно отличается от средней, с финансированием дорог всегда проблема и т. д. Более того, добавление еще одной полосы к магистрали (предположим, что место и финансирование это позволяют) обычно приводит к временному закрытию по крайней мере одной активной полосы шоссе. Это неизменно расстраивает «пригородных» еще больше. Стремление достичь «достаточной» пропускной способности всегда присутствует, но из-за препятствий с его реализацией цель достигается не сразу.

Теоретически подход расти «вширь» предпочтителен, когда обвал случился или неотвратим. Обычно это разумный вариант: зачем заботиться о дополнительной пропускной способности, если существующая используется не в полной мере? К сожалению, бывают случаи, когда полная загрузка не оптимальна. Это справедливо для компьютеров, и все же люди часто нагружают свои системы на 100% еще до размышлений о модернизации.

Перегрузка – опасное дело. Общее полезное правило гласит: нагрузка не должна превышать 70% от максимальной в любой момент времени. Это дает запас прочности перед снижением производительности.

Принцип 4: при проведении экспериментов необходима внимательность

Объясним этот принцип на простом примере передачи файла размером 130 Мбайт в сети Gigabit Ethernet по протоколу *ftp*.

```
% pwd
/home/jqpublic
% ls -l bigfile
-rw-------   1 jqpublic staff    134217728 Jul 10 20:18 bigfile
% ftp franklin
Connected to franklin.
220 franklin FTP server (SunOS 5.8) ready.
Name (franklin:jqpublic): jqpublic
331 Password required for jqpublic.
Password: <secret>
230 User jqpublic logged in.
ftp> bin
200 Type set to I.
ftp> prompt
Interactive mode off.
ftp> mput bigfile
```

```
200 PORT command successful.
150 Binary data connection for bigfile (192.168.254.2,34788).
226 Transfer complete.
local: bigfile remote: bigfile
134217728 bytes sent in 13 seconds (9781.08 Kbytes/s)
ftp> bye
221 Goodbye.
```

Читатели, обратившие внимание на производительность, вероятно, будут очень расстроены: ведь можно было ожидать скорость передачи порядка 120 Мбит/с! Между тем, достигнута лишь *десятая* часть этой скорости. Что же, в конце концов, случилось? Существует ли параметр, который не был должным образом установлен? Неисправна сетевая карта? Можно потратить много времени в поисках ответа на эти вопросы. Истина же состоит в том, что в рассмотрение принималась скорость, с которой можно считывать данные из */home*,[1] или скорость, с которой удаленный хост мог бы принимать данные.[2] Сетевой уровень здесь не является узким местом. Это что-то совсем иное: CPU, диски, операционная система и т. д.

Большое внимание в этой книге уделяется объяснению, *как* и *почему* работают системы, – чтобы в ходе проводимых экспериментов оценивалось именно то, что планировалось оценить.

Существует масса слухов, касающихся анализа производительности. Единственная возможность быть объективным – это понимать суть дела, проводить тесты и накапливать данные.

Вывод из этого примера таков: думать, думать и думать при экспериментах по оценке производительности. При измерениях много чего значимого не лежит на поверхности. А потому следует измерять все, имеющее малейшее отношение к тому, что на самом деле необходимо измерить. Если необходимо протестировать что-либо, важно подобрать соответствующие инструменты, специально предназначенные для диагностики этого компонента. И даже в этом случае нужно быть внимательным: очень легко обжечься.

Настройка статической производительности

В значительной степени эта книга посвящена настройке производительности систем в динамическом режиме. Внимание почти полностью отдано производительности системы, работающей в напряженных

[1] Речь идет об измерении скорости, которое автоматически выполняется программой *ftp* при передаче данных. – *Примеч. науч. ред.*

[2] Что не обязательно равнозначно «с какой скоростью эти данные могут быть записаны на диск», хотя такое и *может* быть.

условиях. В то же время существует и другой подход к настройке производительности, основанный на *статических* (то есть не зависящих от нагрузки) факторах. Влияние таких факторов способно снизить производительность системы вне зависимости от нагрузки. И не всегда это связано с проблемой соперничества из-за ресурсов.

Безусловно, наиболее причастными к статической производительности можно считать службы имен – средства, с помощью которых извлекается информация об объекте. Примеры служб имен: NIS+, LDAP и DNS.

Симптомы часто бывают расплывчаты: медленный вход в систему, застывшие окна веб-броузера, блокировка нового окна при начальной загрузке или зависший экран регистрации X или CDE. Вот несколько пунктов, расположенных в примерном порядке вероятности, на которые следует обратить внимание:

/etc/nsswitch.conf

Этот файл читается один раз для каждого процесса, использующего службу имен, поэтому для того, чтобы изменения возымели эффект, может понадобиться перезагрузка.

/etc/resolv.conf

Правильно ли указаны сервер имен и домен? Неправильные или чересчур длинные (со многими подкомпонентами) определения доменов могут быть причиной того, что DNS будет выдавать множество запросов. Серверы имен должны быть отсортированы по времени задержки с ответом на запросы.

Запущен ли демон кэширования для службы имен (name service cache daemon, nscd)?

Этот процесс кэширует информацию от службы имен, существенно повышая производительность. Демон *nscd* является штатным для Solaris и доступен для Linux. Исторически *nscd* винили во многих грехах. Со временем от большинства из них избавились, и сейчас этот демон можно запускать. Одно исключение – режим поиска неисправностей, когда *nscd* может скрывать глубинные проблемы службы имен.

Другой возможный источник неприятностей – сетевая файловая система (Network File System, NFS). Важно не допускать вложенного монтирования. Например, пусть рабочая станция монтирует три каталога с трех различных серверов NFS:

```
alpha:/home
bravo:/home/projects
delta:/home/projects/system-performance-tuning-2nd-ed
```

Чтение файла */home/projects/system-performance-tuning-2nd-ed/README* потребует вовлечения всех трех серверов NFS и приведет к чрезмерному трафику. Иметь для всего отдельные точки монтирования было бы намного эффективнее.

Настройка параметров ядра

По ходу книги мы будем предлагать настроить конкретные переменные ядра. Никогда не выполняйте настройку ядра на рабочей системе. Воспроизведите рабочую ситуацию в контролируемых лабораторных условиях, где и проводите эксперименты по настройке. Даже если необходимо произвести изменения на рабочей системе, тщательно и *всесторонне* протестируйте сделанные изменения.

В Solaris параметры ядра можно настраивать «на лету» с помощью *adb* или устанавливать их при начальной загрузке. В качестве примера будем использовать `maxpgio` (более подробную информацию о `maxpgio` можно почерпнуть из раздела «Свободный список» главы 4). Прежде всего, нужно знать, какое ядро будет загружаться (64- или 32-разрядное). Это можно узнать, выполнив *isainfo -v*. Вот пример для 32-разрядной системы:

```
% isainfo -v
32-bit sparc applications
```

И для 64-разрядной системы:

```
% isainfo -v
64-bit sparcv9 applications
32-bit sparc applications
```

Чтобы проверить значение переменной ядра, запускайте *adb* в режиме анализа ядра (*-k*). Чтобы посмотреть значение переменной на 32-разрядной системе, наберите имя переменной, а после нее `/D`; на 64-разрядной системе после имени переменной наберите `/E`. Если после имени переменной указан неправильный ключ, выходные данные будут бессмысленными. Вот пример для 32-разрядной системы:

```
# adb -k
physmem 3bd6
maxpgio/D
maxpgio:
maxpgio:        40
```

И для 64-разрядной системы:

```
# adb -k
physmem 1f102
maxpgio/E
maxpgio:
maxpgio:        40
```

Если необходимо изменить значение переменной, это можно сделать сразу. Выйдите из *adb*, набрав <Ctrl>+<D>, и запустите *adb -kw*. Чтобы изменить переменную, необходимо снова указать соответствующий ключ: `/W` для 32-разрядных систем и `/Z` для 64-раз-

рядных. В конце строки следует набрать `0t` и то десятичное значение, которое должно быть присвоено переменной.

Например, присвоим параметру `maxpgio` значение 100 на 32-разрядной системе:

```
# adb -kw
physmem 3bd6
maxpgio/W 0t100
maxpgio:        0x28            =       0x64
maxpgio/D
maxpgio:
maxpgio:        100
```

На 64-разрядной системе:

```
# adb -kw
physmem 1f102
maxpgio/Z 0t100
maxpgio:        28              =       64
maxpgio/E
maxpgio:
maxpgio:        100
```

Это изменение не сохранится после перезагрузки. Чтобы сохранить эту установку, внесем изменения в */etc/system*:

```
*
* change kernel variable at boot-time: 08-11-2001 by gdm
set maxpgio=100
```

В Linux, в зависимости от того, была переменная выставлена в дереве */proc* или нет, есть два пути: либо редактировать исходный код ядра и собирать ядро заново, либо изменять соответствующий файл в */proc*. Редактировать файлы в *proc* можно напрямую – скажем, изменим в Linux 2.2 значение параметра `min percent`, отвечающего за минимальный процент памяти системы, доступной для кэширования. Этот параметр можно найти в файле */proc/sys/vm/buffermem*. Формат этого файла: `min_percent max_percent borrow_percent`. Значение `min_percent` можно изменить непосредственно:

```
# cat /proc/sys/vm/buffermem
2 10 60
# echo "5 10 60" > /proc/sys/vm/buffermem
# cat /proc/sys/vm/buffermem
5 10 60
```

Эти изменения не сохранятся после перезагрузки, но эти строки можно поместить в стартовый сценарий, чтобы выполнять настройку автоматически.

И снова – будьте внимательны и *всегда* тестируйте сделанные изменения!

Проверка всякой всячины

Одинаковые IP-адреса в сети, а также другие ошибки в конфигурировании интерфейсов хоста могут быть источниками неполадок. Необходимо обеспечить жесткий контроль над адресацией IP. Иногда повреждаются кабели и возникают ошибки. В этом случае следует оценить частоту ошибок в работе интерфейса с помощью команды *netstat -i*.[1]

Иногда бывают неработоспособны процессоры. Это почти всегда вызывает аварийное завершение работы. Однако в мощных системах Sun (например, E3500-E6500 и E10000) система автоматически перезагрузится, попытается изолировать отказавший элемент и возобновит работу. В результате может показаться, что система перегрузилась якобы самовольно, «исключив» несколько процессоров, отказ которых повлек перезагрузку. Хорошее правило – применять команду *psrinfo*, дабы убедиться, что после загрузки работают все процессоры. Причем неважно, по какой причине произошла перезагрузка.

В аппаратном обеспечении возникает немало осложнений, которые могут привести к проблемам статической производительности. Если есть подозрение на неработоспособность, обычно намного легче для психики просто заменить поврежденный элемент. Если это невозможно, готовьтесь к длительному и напряженному процессу поиска неисправностей. Почти всегда полезно держать перед собой белый лист бумаги, чтобы набросать варианты конфигурации и отметить, при каких из них система работает, а при каких нет.

Заключение

Решение любой сложной задачи, как правило, требует определенной подготовки и хорошего понимания базовых принципов. В этой главе обсуждалось, что представляет собой настройка производительности, а также основные идеи в разрезе архитектуры компьютера, важные вопросы 64-разрядной среды, приемы настройки статической производительности и некоторые фундаментальные принципы производительности. Хотя очень трудно говорить об эффективности, скажем, учебных классов в подготовке пилотов, есть надежда, что данная глава дала представление о действующих лицах дальнейшего повествования.

Несколько слов об упражнениях, особенно уместных при намерении прочесть книгу от корки до корки. Вернитесь назад и перечитайте пять принципов настройки производительности (см. раздел «Принципы настройки производительности» ранее в этой главе), отложите эту книгу на час и подумайте, о каких принципах идет речь и что из них

[1] Такие проблемы часто возникают с коаксиальным кабелем (10base2); их очень трудно выявить, и это угнетает. Необходимо приобрести кабель хорошего качества и тщательно его промаркировать.

следует. Это очень широкие, общие постулаты, выходящие за рамки того контекста, в котором они были поданы. Ответственность за их применение ложится на читателя – и это в некотором отношении самый трудный момент, – хотя надеемся, что остальная часть книги будет действенным путеводителем. Приступая к анализу задачи, задайте себе несколько вопросов. Понимаю ли я, что случилось? Если нет, то какие эксперименты необходимо провести, чтобы подтвердить свои догадки? Не ищу ли я или мои клиенты «бесплатный сыр»? На какие компромиссы я готов пойти, чтобы добиться желаемой производительности? Не перегружаю ли я ресурсы? Использую ли я в оценке производительности ту систему мер, которую разработал? Те ли это показатели, которые я имею в виду?

Это сложные вопросы, и неважно, насколько простыми они могут казаться. Это невинно выглядящие лазейки, ведущие во тьму, в запутанные лабиринты. Нет ничего необычного встретить восемь или десять экспертов по производительности, собравшихся обсудить, что же случилось с системой. Они выдвигают теории и проводят эксперименты, чтобы подтвердить свои догадки или опровергнуть. Пять приведенных принципов – это главное, что следует вынести из книги. Их свет не даст заблудиться.

2

- *Определение параметров рабочего процесса*
- *Регулирование рабочей нагрузки*
- *Оценка производительности*
- *Заключение*

Управление рабочим процессом

Чем точнее определена координата, тем менее точно в это мгновение известно количество движения, и наоборот.

Werner von Heisenberg, 1927

Тема «управление рабочим процессом» – скользкая. Ее можно толковать по-разному. В этой главе рассматриваются практические средства, призванные обеспечить нулевой принцип настройки производительности: понимание того, что представляет собой операционная среда. Это сердцевина настройки динамической производительности. Остальная часть книги только улучшает понимание возможных вариантов операционной среды.

Как уже упомянуто, в основном речь пойдет об анализе динамической производительности. Оцениваемая система меняется на наших глазах. В каком-то смысле это подобно наблюдению за прудом. Представим ручей, который вливается в этот пруд. Какое влияние он оказывает на жизнь в пруду? Что произойдет, когда бобры построят на нем свою плотину? Когда ребята обнаружат этот пруд и станут кидать в него камни? Или когда кто-нибудь развеет на его середине пепел давних любовных посланий?

Чем дальше, тем сложнее. Будем руководствоваться непоколебимым принципом физики. Принцип неопределенности Гейзенберга (Heisengerg) гласит: как бы мы ни были аккуратны при проведении измерения, мы всегда возмущаем систему, и часть информации остается вне поля нашего зрения. Однако эти возмущения можно свести к минимуму. По ходу этой главы будет рассмотрено, насколько существенны такие возмущения при оценке производительности.

Такое положение дел может ввести в уныние. Мало того, что эксперименты затрагивают нечто, постоянно изменяющееся под влиянием непонятных факторов, которые не поддаются контролю. Ко всему этому сами измерения, проводимые в системе, возбуждают в ней дополнительные изменения! Тогда стоит ли беспокоиться об измерениях? Ответ прост. Без тщательных, систематических измерений практически невозможно принимать обоснованные аналитические решения. Дело в том, что в этом случае трудно представить, почему существует проблема или что произошло. Всегда лучше что-либо знать о ситуации. Если не стремиться получить о ней четкое представление, то есть все шансы уподобиться администраторам системы культа Карго.[1] И сидеть за столами с деревянными клавиатурами и терминалами[2], ожидая чародеев, которые прилетят на блестящих серебряных птицах. И за несколько сотен долларов в час будут решать все проблемы. Это не самое лучшее решение для всех, кроме чародеев.

В этой главе управление рабочим процессом разбито на две части: определение параметров рабочего процесса и его регулирование. Такой порядок важен: регулирование, проводимое наобум, вносит настоящую неразбериху. Многие консультанты накопили целые состояния, разбирая завалы после таких действий. Не нужно падать духом, думая о сложности этой темы. Существует немало инструментов, помогающих понять, а затем и ограничить нагрузку, возлагаемую на систему. Многие из них просты в применении. Далее речь пойдет о программах оценки – принятых способах измерения производительности систем с нагрузкой разной величины и характера. Эту главу можно рассматривать как учебник по превращению дерева культа Карго в кремний двадцать первого века.

Определение параметров рабочего процесса

В некотором смысле эксперты по производительности – это светила в высях компьютерного мира.[3] Каждый раз, когда какой-нибудь застенчивый проситель приходит к ним со своей бедой, гуру какое-то время

---

[1] Термин «культ Карго» (cargo cult) возвращает к событиям, происходившим после второй мировой войны. Аборигены с островов Тихого океана строили деревянные взлетно-посадочные полосы и макеты самолетов. Они надеялись, что к ним снова прилетят самолеты с грузами (cargo), которые приземлялись на островах во время войны. – *Примеч. перев.*

[2] Хотя часто хочется, чтобы пейджер и сотовый телефон были деревянными. Тогда они не смогут звонить.

[3] Продолжим эту аналогию. Порою представляется, что эксперты по производительности баз данных обитают в космическом пространстве. Тому есть три серьезных доказательства. Говорить с ними до смешного дорого. Вы или они должны проделать большой путь для аудиенции. Они говорят на странном языке, который трудно понять.

неспешно копается в проблеме, а затем отсылает просителя, требуя большей информации. Определение параметров – это сбор максимально возможного объема данных о системе. Цель этого процесса – выявить модели и тенденции в системе. Такие модели жизненно важны, когда производительность стремительно падает. Можно собрать воедино все данные о нарушениях в работе системы и исходя из них определить, что послужило их причиной. Это весьма похоже на изучение кильватера проплывающего корабля – что это было за судно и каким курсом оно плывет.

Можно провести аналогию между управлением рабочим процессом и финансовыми транзакциями. Первый раз авторы прочли об этом в замечательной книге Адриана Кокрофта (Adrian Cockcroft) «Настройка и производительность Sun»(Sun Perofrmance and Tuning), изданной Prentice Hall. Идея состоит в том, что управление рабочим процессом в компьютерных системах аналогично управлению ведомством с бюджетом, персоналом и прочим, которое выполняет какую-то задачу.

По существу, возможны три исхода:

1. Если не существует плана и эффективного контроля за персоналом, то сотрудники становятся неуправляемыми, захватывая столько бюджета, сколько могут, чтобы обеспечить своим собственным проектам хорошее финансирование. Некоторые сотрудники в этом отнюдь не преуспевают, в то время как другие берут на себя «расследование ситуации» в Мауи.[1] Проект в целом заканчивается полной неразберихой (так называемая «стартовая модель»).
2. Не в меру старательный управленческий персонал создает бюрократические нагромождения: как планировать, оценивать планирование и перепланировать. Эта бюрократическая прослойка съедает весь бюджет. Она мешает тем, кто делает реальную работу, требуя от них ежедневные сводки. Административные издержки практически не позволяют рационально расходовать средства. Работа заканчивается неразберихой, потому что к моменту, когда все должно быть закончено, работа только началась (так называемая «правительственная модель»).
3. В идеальном случае руководители хорошо обдумывают и контролируют финансирование. При этом аппетиты персонала ограничены, но каждый обладает достаточными средствами, чтобы выполнить свою работу. Бюрократические издержки сведены к минимуму, и сводки запрашиваются нечасто. Работа выполнена в срок и в рамках бюджета (так называемая «модель сказочной страны»).

Аналогия с управлением производительностью довольно близкая. Если хотите, роль бюджета играют компьютерные показатели: циклы процессора, скорость дискового ввода-вывода, пропускная способность сети и прочее. Необходимо управлять этими ресурсами, чтобы не ока-

[1] Один из Гавайских островов, фешенебельный курорт. – *Примеч. перев.*

заться в силках стартовой модели, где получение любых ресурсов сводится к их грабежу. Однако если управлять ими с излишним рвением, то это будет похоже на правительственную модель – когда никто ничего не делает, так как слишком много времени уходит на канцелярскую работу. Поэтому следует применять сбалансированный подход: модель сказочной страны. К сожалению, название модели вполне оправдано. Прийти к ней очень трудно, а придерживаться ненамного легче.

Первый шаг в управлении производительностью – наметить ориентиры, которыми будем руководствоваться. Следует выбрать конкретные цели и задать критерии производительности, по которым можно многократно оценивать работу системы. Кроме того, необходимо определить текущее состояние системы. Впоследствии эта информация даст возможность управлять ресурсами и распределять их, улучшая работу пользователей.

Как дисциплина, управление производительностью зародилось десятилетия назад в мире мэйнфреймов, где все потрясающе дорого. Воплощение решений о настройке, основанное на понимании реальной ситуации, обычно окупалось. Фактическое снижение себестоимости часто бывало достаточно большим. Сейчас, когда стоимость вычислительной техники сильно упала, становится сложнее обосновывать время и усилия, затрачиваемые на диагностику и настройку. Приходится опираться на неявные преимущества анализа: проведенная наобум модернизация не оправдывает себя в течение нескольких месяцев, в то время как внимательное изучение системы и ее рациональная модернизация способны существенно улучшить производительность. Прибыль, полученная за эти несколько месяцев благодаря увеличению производительности, может быть намного выше стоимости анализа.

Существуют три важных инструмента, которые можно применить для получения более глубокого представления о поведении системы. Это простые, довольно известные команды по оценке производительности, а также учет процессов и возможности автоматического сбора данных утилиты *sar*. В этой главе кратко коснемся анализа конфигурации сети.

Простые команды

Такие простые утилиты, как *iostat*, *vmstat* и *mpstat*, всем известны. Это базовые инструменты для анализа производительности, которые могут предоставить много полезной информации о том, что происходит в системе. Рассмотрим один из быстрых способов собрать данные о функционировании системы. Он заключается в установке интервала, скажем, в несколько минут, с которым эти утилиты будут запускаться, и перенаправлении их вывода в файл. Возникает затруднение – как отслеживать точное время каждого конкретного сбора данных. Такую заботу может взять на себя следующий сценарий Perl:

```
#!/usr/bin/perl
while (<>) { print localtime() . ":  $_"; }
```

Вот результаты этого скрипта, запущенного в системе Linux:

Пример 2.1. vmstat в системе Linux

```
# vmstat 5 | chrononome.pl
Sat Jun 30 00:37:28 2001:  procs                  memory      swap      io     system      cpu
Sat Jun 30 00:37:28 2001: r  b  w   swpd   free  buff  cache  si  so   bi   bo   in   cs us sy id
Sat Jun 30 00:37:28 2001: 1  0  0   5472 26680  8420 177908   0   0   10   37   63   37  5  6 14
Sat Jun 30 00:37:33 2001: 0  0  0   5472 26576  8420 177908   0   0    0   20  163   37  1  0 99
Sat Jun 30 00:37:38 2001: 0  1  0   5472 26540  8420 177916   0   0    0   12  148   41  1  1 98
Sat Jun 30 00:37:43 2001: 0  0  0   5472 26904  8420 177920   0   0    2    5  141   43  1  2 98
Sat Jun 30 00:37:48 2001: 0  0  0   5472 26904  8420 177920   0   0    0    1  129   39  2  1 98
Sat Jun 30 00:37:53 2001: 0  0  0   5472 26904  8420 177920   0   0    0    0  124   36  0  1 99
Sat Jun 30 00:37:58 2001: 0  0  0   5472 26904  8420 177920   0   0    0   10  143   35  1  0 99
```

Некоторые администраторы запускают эти команды с очень коротким интервалом с помощью записи в *crontab*. Это действенный подход, но, с точки зрения авторов, не всегда оправданный. Все зависит от того, что необходимо измерить. Если запустить *vmstat* с интервалом 1800 секунд, то можно ожидать, что каждая строка будет сообщать о средней активности виртуальной памяти за последние полчаса. Однако если запускать *vmstat* с интервалом в две секунды каждые полчаса не из *cron*, то будут получены срезы производительности на получасовых отметках. Полезны и те, и другие данные, но, скорее всего, данные с большим интервалом предпочтительнее, по крайней мере вначале. Если же в данных обнаружатся пробелы, всегда можно запустить еще одну копию этой утилиты с интервалом, позволяющим поймать «горы и долины» активности. Увеличение интервала сбора данных не позволяет зафиксировать долговременные крены в производительности. Как правило, приходится выполнять много сопоставлений на большом объеме данных.

Учет процессов

Учет процессов – это метод, с помощью которого система собирает информацию о каждом запущенном процессе. Эти данные говорят об использовании процессора, активности дискового ввода-вывода, потреблении памяти и других полезных «лакомых кусочках». Вероятно, наибольшая польза от учета процессов заключается в том, что он позволяет выявить наиболее типичную нагрузку, испытываемую системой.

Часть системных администраторов обходят стороной учет процессов из-за боязни высоких накладных расходов. Но сбор отчетных данных по сути не оказывает сильного воздействия на систему: ядро всегда собирает статистические данные, поэтому дополнительные усилия связаны лишь с записью 40 байт в протокол учета. В то же время скрипты, обрабатывающие подобные файлы протоколов, могут работать довольно долго. Поэтому лучше не запускать их в часы пик.

Включение учета процессов

Запуск учета процессов очень прост. В Solaris надо убедиться, что дополнительные пакеты, позволяющие вести учет, установлены. Это па-

кеты SUNWaccu и SUNWaccr. В Solaris 8 эти пакеты находятся на втором инсталляционном диске.

Первый шаг – создать символические ссылки на скрипты запуска и останова:

```
# ln /etc/init.d/acct /etc/rc0.d/K22acct
# ln /etc/init.d/acct /etc/rc2.d/S22acct
```

Затем можно перезагрузить систему или сразу начать учет, запустив */etc/init.d/acct start*. Второй шаг – добавить в таблицу *crontab* записи о запуске пользователем adm команд для сбора отчетной информации. Пример 2.2 показывает, что нужно добавить.

Пример 2.2. Добавление строк в crontab

```
# min  hour  day  month  wkday  command
0      *     *    *      *      /usr/lib/acct/ckpacct
30     2     *    *      *      /usr/lib/acct/runacct 2> /var/adm/acct/nite/fd2log
30     9     *    *      5      /usr/lib/acct/monacct
```

Программа *ckpacct* контролирует размер учетного файла */usr/adm/pacct/*. Команда *runacct* на основе этих данных формирует учетную информацию. Наконец, *monacct* готовит «фискальный» отчет для каждого пользователя. Эти отчеты хранятся в */var/adm/acct*.

Просмотр учетных данных

Самым полезным инструментом для просмотра учетной информации является *acctcom*. Он показывает текущие учетные данные. Запускать его можно в нескольких режимах: опция *-a* даст усредненные данные для каждого запущенного процесса, *-t* предоставит разбивку «системное/пользовательское время» для каждого процесса, *-u user* покажет все процессы, исполняемые пользователем user, и *-C time* отобразит все процессы, потребившие более чем *time* секунд процессорного времени. У *acctcom* есть и другие полезные опции. Они могут отличаться от системы к системе, поэтому для более полной информации следует обратиться к своей документации (manual pages).

Системные учетные программы также порождают несколько файлов, которые можно посмотреть на досуге. Они создаются ежедневно в конце каждого учетного периода (указанного при запуске *monacct*).

Автоматизация работы sar

Время от времени в книге будет обсуждаться *sar* – средство для накопления данных о производительности. Кроме того, *sar* можно применять для автоматического сбора данных и их хранения с целью последующего просмотра.

Включение sar

Начать автоматический сбор данных с помощью *sar* достаточно просто. Необходимо раскомментировать соответствующие строки в */etc/init.d/perf* (а именно последние 13) и добавить в системный файл *crontab* поддержку для автоматической записи данных.

Изменения в системных записях *cron*, расположенных в */var/spool/crontabs/sys*, подразумевают принятие некоторых решений: когда необходимо записывать данные (с помощью команды *sa1*) и когда данные следует выдавать (с помощью команды *sa2*). Вот как эти записи выглядят по умолчанию:

```
# min   hour    day     month   wkday   command
# 0 * * * 0-6 /usr/lib/sa/sa1
# 20,40 8-17 * * 1-5 /usr/lib/sa/sa1
# 5 18 * * 1-5 /usr/lib/sa/sa2 -s 8:00 -e 18:01 -i 1200 -A
```

Первая запись означает, что *sar* будет сохранять данные каждый час семь дней в неделю. Вторая инициирует запись два раза в час, на 20-й и 40-й минутах с начала часа в период повышенной нагрузки (с 8 утра до 5 вечера, с понедельника по пятницу).

Третья запись более сложна. В 18.05 с понедельника по пятницу она инициирует выдачу данных, собранных между 8 часами утра (*-S time*) и одной минутой седьмого (*-e time*) с интервалом 1200 секунд, или 20 минут (*-I seconds*), а также выдает все данные (*-A* для *sar*).

Извлечение данных

Извлечь данные из записей *sar* необыкновенно просто. Нужно лишь запустить *sar*, указав с помощью соответствующих ключей необходимые данные, а также требуемый период времени. Ключи *-S starting-time* и *-e ending-time* применяются для временного интервала, а *-f /var/adm/sa/sadd* (*dd* – день месяца) для указания выбранного дня.

Виртуальный Адриан

Адриан Кокрофт (Adrian Cockcroft) – знаменитый инженер компании Sun, занимающийся вопросами анализа производительности. Вместе с Риком Петитом (Rich Pettit) он разработал набор программ для сбора и анализа данных в системах Solaris. Эти программы очень подробно описаны в их книге «Производительность и настройка Sun» («Sun Performance and Tuning»), изданной Prentice Hall. Хотя книга слегка устарела, она, безусловно, заслуживает прочтения. Сами программы представляют собой замечательный инструментарий. Этот пакет, называемый SE Toolkit, размещен по адресу *http://www.setoolkit.com/*.

Анализ конфигурации сети

Исторически[1] сети практически не ограничивали производительность. Поэтому значительная часть усилий разработчиков была направлена на изучение и улучшение других аспектов работы системы. Однако вследствие бурного роста приложений Интернета сетевой уровень все чаще и чаще становится ограничивающим фактором. В результате инструменты для анализа производительности сети не настолько совершенны, как средства для анализа других показателей работы. Как же в таком случае анализировать сетевой уровень операционной среды?

Вначале небольшое предупреждение: этот раздел предполагает некоторую осведомленность читателя в концепциях сети.

Начнем с рассмотрения трех основных типов трафика, который присутствует в сетях TCP/IP. Затем кратко обсудим некоторые параметры сетевого трафика, например какова средняя величина пакета. В заключение разберем средства, позволяющие определять типичный трафик в конкретной сети.

Краткие замечания по терминологии

Вот понятия сетевого трафика, которые будут применяться в дальнейшем обсуждении:

- В соединении TCP/IP *клиент* – это сторона, которая *открыла* соединение (инициирующая сторона). *Сервер* – это сторона, которая *приняла* соединение (принимающая сторона). Заметим, что это ровным счетом ничего не говорит о ролях машин вне самого факта соединения!
- *Входящий трафик* – это трафик от клиента к серверу, *исходящий трафик* – трафик в обратном направлении.
- *Пакет без полезной нагрузки* – обычно около 60 байт в пакете (напомним, что существуют различные заголовки – обычно Ethernet, IP и TCP). Эти пакеты имеют прямое отношение к выполнению некоторых действий, например, связанных с протоколом TCP.
- *Маленький пакет* – менее 400 байт в пакете (включая все заголовки).
- *Средний пакет* – пакет размером от 400 до 900 байт.
- *Большой пакет* – пакет размером от 900 до 1500 байт (1500 – для Ethernet).

[1] То есть до эпохи Сети.

Тип 1: запрос–ответ

Первый тип трафика – запрос-ответ. Классические примеры из практики: HTTP, протоколы доставки электронной почты (например, POP3) и исходящие транзакции SMTP (по доставке на удаленный хост). Рисунок 2.1 показывает трафик типа 1.

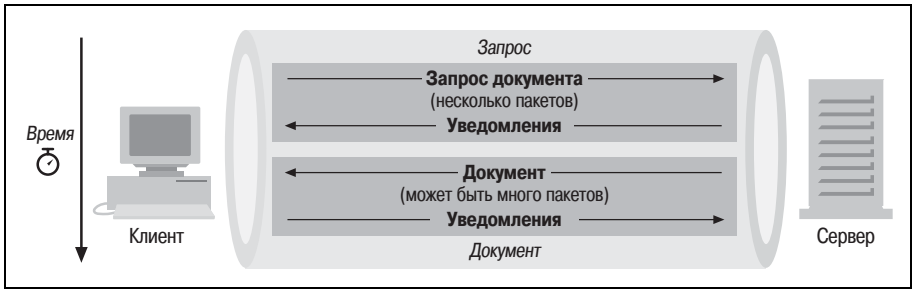


***Рис. 2.1.** Трафик типа 1*

Трафик запрос-ответ характеризуется высокими скоростями соединения. Однако иногда можно наблюдать и довольно медленные соединения, особенно в современной веб-среде, где широко применяются пакеты сохранения соединения HTTP (keepalives). Входящие пакеты обычно маленькие; исходящие – средней величины, но обычно их немного (меньше двадцати).

Тип 1B: обратный запрос–ответ

Вариация трафика типа 1 – обратный запрос-ответ. Обычно его можно наблюдать во входящих соединениях SMTP. Интересно, что в противоположность трафику типа 1 входящая и исходящая роли здесь поменялись местами: клиент инициирует большую часть передачи данных. Рисунок 2.2 иллюстрирует трафик типа 1B.

Скорость соединения часто умеренная, но иногда носит импульсный характер. На начальном этапе соединения входящие пакеты обычно малы, но затем становятся большими. Исходящие пакеты – это маленькие пакеты или пакеты без полезной нагрузки.

***Рис. 2.2.** Трафик типа 1B*

Тип 2: передача данных

Трафик типа 2 обычно представляет собой передачу больших объемов данных. Наиболее часто это можно встретить в трафике *ftp* и при передаче файлов во время резервного копирования по сети (рис. 2.3).

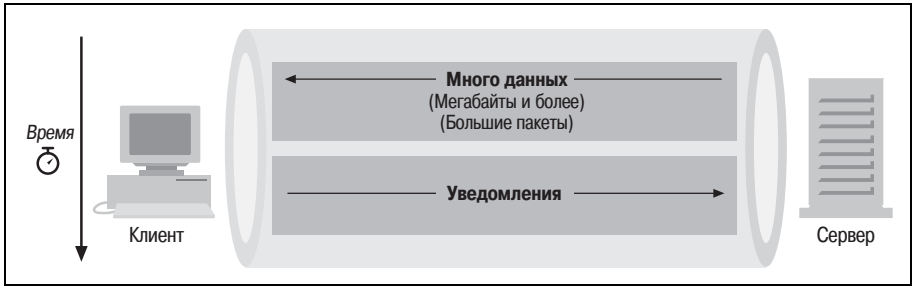


Рис. 2.3. *Трафик типа 2*

В основном скорость соединения очень низкая. По существу, поток входящих пакетов – это пакеты без полезной нагрузки, однако поток исходящих пакетов почти полностью состоит из полных пакетов.

Тип 3: передача сообщения

Третий и последний тип трафика – передача сообщения. Это наиболее типичный трафик для символьных приложений, таких как *telnet*, *rlogin* и *ssh*. Также его можно часто встретить в таких схемах транзакций баз данных, как SQLnet. Некоторые высокопроизводительные программы с параллельной обработкой данных также применяют такие передачи данных. Рисунок 2.4 иллюстрирует трафик типа 3.

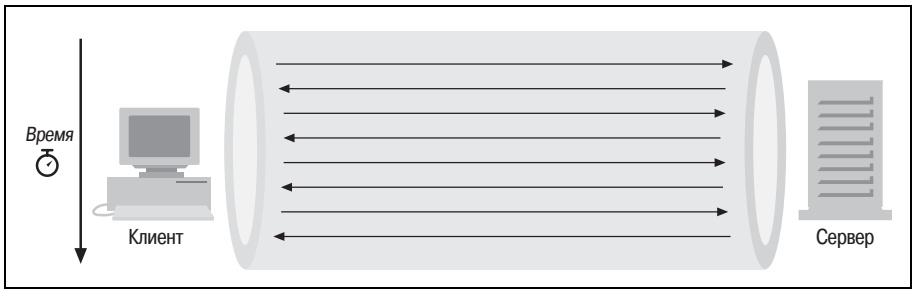


Рис. 2.4. *Трафик типа 3*

Скорость соединения различна. Обычно она мала, но может быть и достаточно высокой – это в значительной степени зависит от приложений. Для этого типа трафика характерно большое количество разносимых маленьких пакетов. Для системы подобный трафик представляет большую нагрузку, так как усилия по обработке маленького пакета,

по существу, те же, что и по обработке большого: тот же объем работы, но полезных данных меньше.

Настройка систем с нагрузкой по типу 3 очень трудна. Если при наличии такой нагрузки есть уверенность, что сеть – слабое место в производительности, и производительность нужно повысить, то существуют два пути: либо изменить алгоритм работы приложения, чтобы с трафика типа 3 перейти на трафик типа 1, либо вложить деньги в сетевые аппаратные средства с очень коротким периодом ожидания.[1]

Распределение размеров пакетов

Сбор данных о работе сети (например, с помощью *snoop* или *tcpdump*) может дать очень интересную информацию о том, какой трафик есть в сети. Один из возникающих простых вопросов таков: «Если построить график с осями «размер пакета» и «количество пакетов», то что можно будет увидеть?» Проанализировав системы в Интернете, можно заметить, что такой график – трехвершинный (имеет три отчетливых пика). Обычно это веб-системы или почтовые серверы.

- Первый пик в большей степени относится к входящему трафику и почти полностью состоит из пакетов размером около 60 байт. Это уведомления TCP, идущие от броузеров к веб-серверу и подтверждающие прием порции данных.
- Второй пик почти полностью состоит из исходящего трафика (пакеты размером 1540 байт). Это полные пакеты данных, идущие от сервера к броузерам.
- Третий пик замечаешь не сразу: он соответствует пакетам размером около 540 байт и относится к исходящему трафику. Такой трафик встречается в определенных Windows-реализациях TCP/IP, которые для соединения TCP устанавливают «размер максимального сегмента» 536 байт. В результате другая сторона не может передать клиенту любой пакет размером более 536 байт.

Если существует возможность влиять на трафик в сети, то такие данные могут послужить поводом к серьезным изменениям. Правильно настроенные алгоритмы приложений и стеки TCP – путь к устранению «средних пиков» на отметке 540 байт и соответствующему повышению эффективности.

Регулирование рабочей нагрузки

Следующий этап настройки системы – это внедрение регулирования рабочей нагрузки. В техническом плане этот этап часто прямолинеен, но в

[1] Myrinet имеет репутацию быстрого сетевого оборудования с коротким периодом ожидания. Такое оборудование не нашло широкого распространения вне специализированных рынков сбыта.

нем могут быть и политические аспекты. Жесткий контроль пользователей системы – не лучший шаг, и решаться на него надо лишь в случаях, когда обучение и попытки взаимодействия с пользователями провалились. Если руководство принуждает администратора к такой деятельности, то администратору стоит запросить письменное указание.

К чему все эти предостережения? Таких трудностей значительно меньше в академической среде или в маленьких компаниях, где каждый знает системного администратора и где администратора всегда можно найти и спросить, что же случилось. Однако в большой профессиональной среде или там, где управление пользователями отделено от управления администраторами, все намного сложнее. Очень часто ограничение прав неприятно задевает пользователей (возможно, они разрабатывали программы и оборудование, которые обслуживает администратор). Такие ограничения снижают их гибкость в решении своих задач. Если пользователи взывают к руководству, то администратору приходится доказывать разумность конкретного технического решения взбешенным старшим вице-президентам. Нужно быть осторожным, если хочешь быть сильным.

С другой стороны, обучение пользователей – это одно из самых действенных средств администратора, контролирующего нагрузку компьютерной системы. В этом разделе будет обсуждаться обучение пользователей, а также близкая тема письменных соглашений о производительности. Кроме того, будут рассмотрены несколько непосредственных методов, призванных ограничить потребление системных ресурсов.

Обучение

Обучение пользователей – самый мощный инструмент в управлении рабочей нагрузкой. Ограничение выделяемого процессорного времени и дисковые квоты эффективны, однако они часто усиливают «негодование от незнания». Пользователи чувствуют себя средневековыми невольниками. Много чего сделать они не в состоянии, скажем, вызвать дождь в сухой период. А все, что пользователи могут, – это идти и умолять о помощи весьма таинственных людей, обычно живущих в пещерах. В конечном итоге пользователи раздражены.

Значительно лучше объяснить пользователям, какие их действия приводят к трудностям и каким может быть решение. Такие откровенные обсуждения неоднократно давали отличные результаты без неприятных осложнений.

Соглашения по применению и производительности

Один из первостепенных подходов к обучению пользователей состоит в разработке *соглашений по применению и производительности*. Это очень напоминает «соглашения об уровнях сервиса», обычные для мира сетей. Такой документ точно устанавливает, что обе договариваю-

щиеся стороны ожидают друг от друга. По желанию он может быть в официальной или свободной форме. Авторам встречались как довольно длинные описания, будто изданные для судебного рассмотрения, так и документы на полстраницы, подводящие итог под соглашениями об оценках производительности. Все администраторы были довольны такими документами, какой бы длины они ни были. Авторы предпочитают лаконичные документы (пользователям проще их читать), но это вопрос вкуса.

В соглашениях по применению и производительности можно отразить многое:

- Каков минимальный приемлемый уровень производительности? Это весьма насыщенный вопрос. Попросите пользователей ответить на него. Существуют ли «конкурирующие» задания или пользователи, поддержку которых нужно обеспечить?
- Какими средствами оценивать производительность? Из чего состоит нагрузка на систему и каковы ее характеристики перед появлением неполадок? Есть ли у пользователей способ сообщать о том, что машина «забита»?
- Каково расписание работы машины? Когда может проводиться техническое обслуживание?
- Каким образом системный администратор может быть оповещен о неполадках? Проводится ли резервное копирование, если администратор за городом или недосягаем по другим причинам?
- Как можно оповестить пользователей о проблемах? Достаточно ли для этого пункта в «новостях дня»?[1] Нужно ли создавать почтовый псевдоним для уведомлений?

Жизненно важно периодически пересматривать эти соглашения (скажем, раз в квартал). Этот пересмотр может быть очень кратким, если документ по-прежнему верен. А может быть и не очень – обычно в случаях, когда работа стала более сложной или возникли различные трудности.

Опыт авторов показывает: мало кто выделяет время на то, чтобы создать такие соглашения. Это абсурдно. Такие соглашения дают солидные данные о том, что пользователи ожидают от своей операционной среды. Кроме того, это проявление доброй воли и желания взаимодействовать. Иметь дело с неконтактными пользователями – одна из наименее приятных сторон работы системного администратора. К со-

[1] Во многих корпоративных системах принято при входе пользователя в систему выводить на экран «новость дня» – короткое сообщение, в котором сообщается о каких-то важных для пользователя событиях в системе – грядущей профилактике, отключении части сети, переносе каталогов на другие диски и т. п. Текст сообщения придумывает администратор, а выводится оно системными средствами. – *Примеч. науч. ред.*

жалению, возможно, это часть его работы. Следует выделить несколько минут, чтобы вместе с пользователями прийти к такому соглашению. Необходимо получить подтверждение от руководства, что это соглашение понимает каждый. Тогда от соглашения будет польза.

Параметры maxusers и pt_cnt

В ходе разработки типового дистрибутива Беркли (Berkeley Standard Distribution, BSD) системы UNIX был введен параметр, позволяющий масштабировать размер нескольких внутренних таблиц ядра и буферов. Коэффициент масштабирования напрямую зависел от того, какое количество пользователей, работающих в режиме разделения времени, поддерживала система. Поэтому этот параметр был назван `maxusers`. Сегодня нет прямой взаимосвязи между количеством пользователей, которое поддерживает система, и значением `maxusers`. Однако набор параметров в Solaris[1] по-прежнему опирается на значение `maxusers`. Часть из них относятся к контролю рабочей нагрузки, как описано в табл. 2.1.

Таблица 2.1. Параметры, зависящие от maxusers

Переменная	Ресурс ядра	Значение по умолчанию
`max_nprocs`	Максимальное количество процессов в масштабе системы	`10 + 16 x maxusers`
`reserved_procs`	Количество процессов в системе, зарезервированное для пользователя root	`5`
`maxuprc`	Максимальное количество процессов для не-root-пользователей	`max_nprocs - reserved_procs`
`ndquot`	Размер таблицы квот	`(10 x maxusers) + max_nprocs`
`ncsize`	Размер DNLC (см. раздел «Кэш поиска имени директории (DNLC)» главы 5)	`4 x (max_npocs + maxusers) + 320`
`ufs_ninode`	Размер кэша индексных дескрипторов (см. раздел «Кэш индексных дескрипторов» главы 5)	Как `ncsize`

Сам параметр `maxusers` устанавливается автоматически (если это не Solaris 2.3 или более ранняя версия). Его значение примерно равно количеству мегабайт RAM в системе. Минимальный предел – 8, а максимальный предел, равный 1 024 (без ручной настройки), применяется в любой системе с объемом памяти 1 Гбайт и более. Вручную параметру

[1] А также во всех системах от Беркли, включая FreeBSD. – *Примеч. науч. ред.*

maxusers можно присвоить и большее значение (*/etc/system*[1]), однако максимальный общий предел – 2048.

Вопреки распространенному мнению параметр maxusers не ограничивает количество одновременных подключений к системе. Он лишь определяет некоторые параметры, большей частью относящиеся к количеству пользователей, которые система может поддерживать. Многие компоненты системы учитывают количество поддерживаемых пользователей; maxusers является лишь общей отправной точкой. Параметр, который определяет количество поддерживаемых пользователей, называется pt_cnt. По умолчанию его значение равно 48, что может быть ощутимым пределом. Есть практическое ограничение на значение pt_cnt, а именно формат файла *utmp* накладывает предел 3884 для сессий *telnet* и *rlogin*. Вероятно, следует установить значение pt_cnt в пределах трех тысяч, чтобы избежать влияния на другие ненастраиваемые параметры системы. После установки pt_cnt необходимо создать записи в */dev/pts*. Лучше всего это сделать с помощью перезагрузки с флагом реконфигурации (*touch /reconfigure; reboot*; это идентично *boot -r*).

Ограничения, налагаемые на пользователей

К сожалению, время от времени на пользователей приходится налагать ограничения. Иногда эти ограничения устанавливаются вследствие того, что обучить их очень трудно, или потому, что ложка дегтя портит бочку меда. Подход авторов таков: ограничения, налагаемые на пользователей, – лучшее средство, если нужен экстренный тормоз. Когда какой-то важный параметр превышен, то безопаснее остановиться, к какому бы осложнению это ни привело, вместо того, чтобы разгребать завалы впоследствии.

Существует два основных подхода к ограничениям: квоты и контроль окружения.

Квоты

Дисковые квоты – это один из наиболее распространенных типов ограничений, налагаемых на пользователей. Предназначение дисковых квот – не допустить захвата пользователем большего дискового пространства, чем его справедливая доля. Такая часть определяется тем, на что пользователи имеют право и за что они ответственны.

Обычно дисковые квоты реализуются в два этапа. Когда пользователи превышают определенный порог потребления[2], обычно называемый

[1] В файле */etc/system* этот параметр устанавливается для Solaris. В других системах UNIX это делается иначе; например, во FreeBSD его можно установить при изменении файла конфигурации ядра перед компиляцией последнего. – *Примеч. науч. ред.*

[2] Обычно потребление измеряется либо занятым дисковым пространством, либо количеством файлов.

строгим лимитом, то они больше не могут записывать данные на диск, пока не удалят часть файлов и их потребление дискового пространства не упадет ниже этого лимита. Для меньших уровней потребления существует другой, так называемый *нестрогий лимит*. Если пользователи его превышают, то они могут продолжать записывать данные до момента достижения строгого лимита. Однако отсчет времени уже начался. Когда заданный период (обычно измеряемый в днях) истекает, пользователь больше не может записывать файлы, пока не уберет данные и не окажется ниже нестрогого лимита. Такой механизм дает пользователям возможность временно потреблять большой объем дискового пространства.

Включить квоты довольно просто как в Solaris, так и в Linux. В Solaris необходимо инсталлировать пакеты программ учета (`SUNWaccr` и `SUNWaccu`) и создать символические ссылки для двух стартовых файлов:

```
# ln /etc/init.d/acct /etc/rc0.d/K22acct
# ln /etc/init.d/acct /etc/rc2.d/S22acct
```

Затем в таблицу *crontab* пользователя *adm* следует добавить такие строки:

```
# min   hour    day     month   wkday   command
0     *   *     *   *   /usr/lib/acct/ckpacct
30    3   *     *   *   /usr/lib/acct/runacct 2> /var/adm/acct/nite/fd2log
30    9   *     *   5   /usr/lib/acct/monacct
```

Далее нужно включить учет процессов посредством перезагрузки или запуска */etc/init.d/acct start*.

В Linux порядок сходный. Первый шаг – редактирование системных стартовых сценариев, чтобы включать учет процессов при загрузке:

```
# включает учет процессов
if [ -x /sbin/accton ]
then
    /sbin/accton /var/log/pacct
    echo "activating process accounting"
fi
```

Затем необходимо создать файл учетных записей */var/log/pacct*. Владельцем этого файла должен быть root, и файл должен быть доступен для чтения «всем остальным»:

```
# touch /var/log/pacct
# chown root /var/log/pacct
# chmod 0644 /var/log/pacct
```

Далее, для включения учета данных о процессах нужно перезагрузиться либо запустить */sbin/accton /var/log/pacct*.

Предприимчивые пользователи всегда могут обойти дисковые квоты. Они посылают файлы сами себе. Чтобы замести следы, пользователи

находят каталоги на других файловых системах, где квоты не введены. И даже «перекидывают» файлы по сети между многими машинами с помощью доморощенных программ. Чем более разумны дисковые квоты, тем меньше у системного администратора забот с пользователями, творчески подходящими к проблеме. Вернемся к аналогии с экстренным тормозом – представьте электронные ограничители скорости на современных автомобилях. Если ограничитель установлен на отметке 140 миль в час, то большинство людей даже не побеспокоятся его отключить.[1] Они и не собирались ехать так быстро и потому могут даже и не знать о существовании ограничителя. Однако если ограничитель скорости установлен в районе 80 миль в час, огромная масса людей станет искать выход из положения. Они отключат ограничитель и примутся обгонять кого-либо на дороге. Вывод таков: чем разумнее дисковые квоты, тем меньше хлопот они доставляют.

Ограничения операционной среды

Более изящный способ контроля над процессами пользователей – наложить ограничения на системные ресурсы, которые потребляют приложения. Это реализуется на уровне командного интерпретатора с помощью команд *limit* в *csh* и *ulimit* в *sh*. Таблица 2.2 иллюстрирует основные параметры и их типичные значения по умолчанию.

Таблица 2.2. Ограничения ресурсов

Ресурс	Нестрогий предел	Строгий предел
время процессора	Не ограничено	Не ограничено
размер файла[a]	Не ограничено	Не ограничено
объем данных процесса	2 Гбайт	2 Гбайт
размер стека	8 Кбайт	2 Гбайт
размер дампа ядра	Не ограничено	Не ограничено
дескрипторы	64	1024
объем памяти	Не ограничено	Не ограничено

[a] Размер файла, конечно, ограничен, но это ограничение определяется либо типом файловой системы, либо физическим размером носителя. – *Примеч. науч. ред.*

Пользователи могут увеличить свои лимиты до уровня строгого лимита, который обычно модифицируется привилегированным пользователем. Более старые системы могут иметь другие лимиты, вызванные ограничениями оборудования на размер данных и стека (например, ма-

[1] Как и в случае с дисковыми квотами, предприимчивый водитель всегда может отключить ограничитель скорости.

шины архитектуры sun4c имеют ограничения в 512 Мбайт и 256 Мбайт соответственно).

Изменения лимитов можно поместить в файлы конфигурации пользовательских интерпретаторов команд. Одно из самых типичных изменений призвано не допускать записи дампа ядра на диск при аварии. Это достигается путем установки лимита coredumpsize в 0.

```
csh% limit coredumpsize 0
```

Следует помнить, что предприимчивые пользователи всегда могут найти обходные пути для этих ограничений.

Сложные операционные среды

В некоторых сложных операционных средах управление пользовательской рабочей нагрузкой становится очень серьезной задачей. Сотни, если не тысячи, процессоров размещены повсюду: от одной машины до крупного кластера однопроцессорных рабочих станций. Постановка задачи в таких обширных системах смещается от избежания перегрузки к достижению оптимального потребления ресурсов. Это очень сложная область из тех, к которым коммерческие поставщики начинают подступать. Ведущая идея ныне сводится к *расчетным сеткам*, которые обеспечивают распределение работы по географически удаленным участкам. Подробное обсуждение этой темы выходит за рамки книги. Дадим лишь несколько советов. Самое главное – не бояться соединять воедино разные инструменты, если это работает для конкретной системы. Какой-нибудь сценарий Perl, действующий вместе с механизмом учета процессов, может выдать замечательные итоговые данные по потреблению ресурсов приложениями или пользователями. Он может определить «издержки потребления» в зависимости от того, когда приложения были запущены и на каких машинах. Можно также с некоторой осторожностью написать простую программу, формирующую *пакетные очереди*. Пользователь передает свое задание контролирующему процессу, который находит систему с определенным установленным диапазоном параметров и запускает задания на соответствующих системах (например, на некоторых рабочих станциях, даже если они простаивают, можно работать между 8 часами утра и 6 вечера, или какие-то задания требуют большого объема физической памяти и могут быть запущены только на машинах с памятью более 2 Гбайт). Существуют коммерческие программы, решающие такие задачи.

Оценка производительности

Тестовая программа (*benchmark*) измеряет производительность при типовой рабочей нагрузке. Зафиксировав рабочую нагрузку, можно варьировать базовые системные параметры и сравнивать производительность системы в разных конфигурациях.

По существу, есть три вида тестовых программ: компоненто-зависимые инструменты (подобные SPECint и SPECfp для микропроцессоров), средства оценки всей системы, разработанные для эмуляции коммерческих сред (например, серия TPC и SPECweb99), и, наконец, программы, разработанные пользователем. В этом разделе будут рассмотрены соответствующие подходы к тестированию и приведены примеры программ из практики.

Эта книга не о настройке систем с целью достичь мировых рекордов в производительности. Вообще, покорение мировых рекордов требует огромного объема работы, серьезной поддержки инженеров, ответственных за базовые подсистемы, а также понимания, что конечная цель – это мировой рекорд (такой результат вовсе не обязательно найдет применение в среде заказчика).

MIPS и megaflops

Перед обсуждением измерений производительности нужно обратиться к терминологии и истории – единицам MIPS и megaflops.

MIPS

Один из первых подходов к замеру производительности приложения в компьютерной системе сводился к оценке того, насколько быстро система может выполнять команды процессора. Однако это темный показатель. Скорость работы процессора (обычно выражаемая в миллионах операций в секунду (millions of instructions per second), или MIPS) сильно привязана к его тактовой частоте. Возможно, будет разумно предположить, что 5-MIPS CPU превзойдет 100-MIPS CPU, но процессор 50-MIPS может в действительности превзойти 100-MIPS CPU. По существу, применение MIPS для сравнения производительности различных компьютеров некорректно: миллион операций на одном микропроцессоре могут не выполнить ту работу, что и такое же количество операций на другом. Допустим, есть процессор, выполняющий операции с плавающей точкой с помощью программного эмулятора. Пусть он выполняет 50 целочисленных операций для выполнения каждой операции с плавающей точкой. Значит, процессор 50-MIPS выполняет 1 миллион полезных операций в секунду. Добавим к этому факт, что различные операции требуют разного времени для своего выполнения, особенно на процессорах CISC. Являлись ли эти миллион операций операциями «ничего-не-делаю» («do-nothing») или же умножениями с плавающей точкой?

Хуже того, некоторые производители переопределили значение «MIP» как свой внутренний показатель. Особенно знаменита этим компания DEC. Длительное время VAX 11/780 считалась типовой машиной «1 MIP». DEC публично утверждала, что на разумной смеси задач система 11/780 способна выполнять около 470 000 операций в секунду – меньше половины MIP. По этой причине иногда можно услы-

шать, как люди ссылаются на «VAX MIPS» или «VUPs» (единицы обработки VAX), чтобы прояснить сравнение.

Ключевой вывод из этого: MIPS является полезным показателем лишь при сравнении процессоров одного производителя. Такие процессоры должны поддерживать одинаковую систему команд. Кроме того, следует применять одинаковые компиляторы. Существенная слабость MIPS как показателя производительности часто являлась поводом для шутки: MIPS – это сокращенная форма выражения «бессмысленный индикатор скорости процессора» («Meaningless Indicator of Processor Speed»).

Megaflops

Производительность работы с плавающей точкой часто измеряется в миллионах операций с плавающей точкой в секунду. Такую единицу измерения часто называют *megaflops*, или MFLOPS. Операции с плавающей точкой обычно выполняются специализированной частью микропроцессора, ответственной за такие действия. Это операции сложения, умножения, сравнения и преобразования формата. Реже – квадратный корень и деление. Однако обычно скорость в *megaflops* вычисляется для смеси сложений и умножений. Поскольку значения MIPS признаны вводящими в заблуждение, производители регулярно ссылаются на максимальные значения MFLOPS. Максимум MFLOPS процессора обычно кратен целым множителям (1, 2 или 4) тактовой частоты. Причина состоит в том, что оптимизация в микропроцессорах основана на быстрых операциях умножения/сложения (скажем, при накоплении текущих сумм векторных элементов – например, при скалярном произведении двух матриц). Поскольку микропроцессоры становятся все быстрее и быстрее, значение максимума MFLOPS перестает быть полезным в качестве разумной меры производительности операций с плавающей точкой: ограничивающим фактором становится пропускная способность памяти (насколько быстро данные можно перемещать из процессора и в процессор).

Компоненто-зависимые тестовые программы

Первые из обсуждаемых здесь тестовых программ разработаны для оценки производительности памяти и/или производительности микропроцессора. Два наиболее типичных инструмента – Linpack и SPECcpu.

Linpack

Тестовая программа Linpack – одна из первых тестовых программ, принятых на вооружение. Ее написал Джек Донгарра (Jack Dongarra), а затем она была усовершенствована сотрудниками аргонской национальной лаборатории. Интересно, что проект Linpack не сводился к разработке средств оценки. Скорее было намерение создать библиоте-

ку высокопроизводительных подпрограмм для решения задач линейной алгебры. Тестовая программа Linpack определяет величины megaflops так: измеряется время и количество операций, используемые этими подпрограммами для решения плотной системы линейных уравнений методом исключения Гаусса. Существует несколько версий Linpack, различающихся по размеру системы линейных уравнений, которые они решают, а также по числовой точности и основным нормам для тестирования. Наиболее часто применяли версию с матрицей 100×100 и результатами двойной точности, а также с компиляцией текста на Фортране с запретом оптимизации (ручные оптимизации тоже были недопустимы).

В дополнение к тесту 100×100 существует тест 1000×1000, в котором система уравнений может быть решена с помощью любого метода по выбору производителя. Более мощный Linpack – это важная тестовая программа, поскольку она дает корректный верхний предел для производительности операций с плавающей точкой. Это актуально при решении очень больших высокооптимизированных научных задач в конкретной системе.

Сердцевиной тестовой программы Linpack 100×100 является подпрограмма, называемая *daxpy*, которая умножает вектор на константу и добавляет его к другому вектору:[1]

```
for (i = 0; i <= N, i++) {
    dy[i] = dy[i] + da * dx[i];
}
```

На каждой итерации этого цикла выполняется две операции с плавающей точкой (умножение и сложение) и три операции с памятью (два чтения и запись). По существу, тестовая программа Linpack доминировала среди инструментов оценки производительности операций с плавающей точкой и обращений к памяти до начала 1990-х годов. Когда размеры кэша увеличились настолько, что смогли полностью вмещать все данные тестовой программы, Linpack 100×100 перестал быть полезным – структуры данных для всей программы составляют только 320 Кбайт. Первой системой, послужившей падению репутации Linpack 100×100, стала система IBM серии RS/6000. К сожалению, методы оптимизации не позволяют масштабировать Linpack 100×100 до уровня, на котором она снова смогла бы тестировать подсистему памяти.

Таким образом, Linpack 100×100 из разряда единственного общепринятого средства оценки перешел в разряд исторических экземпляров. Такой переход занял около двух лет, в течение которых производители настолько расширили кэши, что они стали вмещать тестовую программу полностью.

---

[1] На самом деле Daxpy написана на Фортране; здесь она представлена на языке C.

SPECint и SPECfp

В конце 1980-х была сформирована группа (впоследствии названная консорциумом по оценке производительности систем (Systems Performance Evaluation Cooperative), или SPEC) для установления, утверждения и подписания соглашений о стандартном комплекте средств оценки.[1] Первая тестовая программа SPEC для микропроцессора была названа SPEC89. В 1992, 1995 и 2000 годах она подверглась изменениям. За счет выпуска новых версий организация SPEC не отстает от новейших тенденций в аппаратных средствах и в разработке компиляторов, не давая производителям «подгонять» свои системы под хорошие результаты этих тестов. В то время как наиболее распространены средства SPEC по оценке производительности CPU, SPEC выпускает и другие типовые тестовые программы. Для получения большей информации о SPEC можно обратиться к ресурсу *http://www.spec.org*.

Первым тестовым инструментом SPEC был SPEC89, основанный на вычислении среднего арифметического времени работы 10 программ. Он не проводил различия между производительностью операций с плавающей точкой и целочисленных операций, что вело к сокрытию некоторых тонких моментов в производительности. Например, система с низкой производительностью целочисленных операций, но с высокоскоростными операциями с плавающей точкой имеет такую же оценку SPECmark, как и система с обратными характеристиками. Интересно, что когда был выпущен SPEC89, он не стал особо важным тестовым инструментом. Различные конкурирующие архитектуры того времени имели разные характеристики производительности, и не было инструментов для эффективной оценки всех вариантов. В области высокопроизводительных вычислений Linpack 100×100 полностью доминировал.

Желая улучшить качество оценки, SPEC переработал SPEC89 и выпустил инструмент SPEC92, ставший правопреемником SPEC89. Он состоит из 20 тестовых программ, разбитых на 6 программ с целочисленными вычислениями и 14 программ с операциями с плавающей точкой. Тестовые программы также выдают два итоговых значения, SPECint92 и SPECfp92, которые соответственно сообщают о производительности процессора при выполнении целочисленных операций и операций с плавающей точкой. После дискредитации Linpack 100×100 SPECfp92 стал боевым показателем для оценки производительности операций с плавающей точкой. На рубеже 1994 года большие кэши и быстрые компиляторы свели SPECfp92 на нет. К счастью, в 1995 году SPEC выпустил новую редакцию своего тестового комплекта. Размер тестовых программ увеличился, поэтому кэши не могли их вместить. Кроме того, повысилось время прогона. Пакет стал похож на большие пользо-

[1] Сейчас это корпорация по оценке типовой производительности (Standart Performance Evaluation Corporation).

вательские приложения. Подобный процесс произошел в 2000 году, когда SPEC снова доработал свои тестовые программы.

Довольно представительная выборка приложений, тестируемых с помощью SPECcpu2000, определяется типовыми кодами пользователей, которые обобщены в табл. 2.3.

Таблица 2.3. Коды приложений, тестируемых с помощью SPECcpu2000

Класс	Код	Язык	Описание
Целочисленный	164.gzip	C	Сжатие данных
	175.vpr	C	Расчет печатных плат методом программируемых матриц (FPGA, Field Programmable Gate Arrays)
	176.gcc	C	Компилятор C
	181.mcf	C	Определитель сетевого потока с минимальной стоимостью
	186.crafty	C	Шахматы
	197.parser	C	Анализатор естественного языка
	252.eon	C++	Трассировщик лучей
	253.perlbmk	C	Perl
	254.gap	C	Вычислительная теория групп
	255.vortex	C	Объектно-ориентированная база данных
	256.bzip2	C	Сжатие данных
	300.twolf	C	Имитатор кругового размещения/маршрутизации
С плавающей точкой	168.wupwise	F77	Физика (квантовая хромодинамика)
	171.swim	F77	Моделирование мелководья
	172.mgrid	F77	Многосеточный решатель: трехмерное потенциальное поле
	173.applu	F77	Решатель уравнений с частными производными
	177.mesa	C	Трехмерная графика
	178.galgel	F90	Вычислительная гидрогазодинамика
	179.art	C	Распознавание образов
	183.equake	C	Имитация распространения сейсмических волн
	187.facerec	F90	Процесс распознавания образов лица
	188.ammp	C	Вычислительная химия

Класс	Код	Язык	Описание
	189.lucas	C	Тестирование простых чисел
	191.fma3d	F90	Имитация разрушения конечных элементов
	200.sixtrack	F77	Физика (разработка высокоэнергетического ускорителя)
	301.apsi	F77	Метеорология (распространение загрязняющих веществ)

Тестовая программа SPEC выдает три значения как для целочисленных операций, так и для операций с плавающей точкой: *чистый* показатель, *базовый* показатель и *пропорциональный* показатель.

- *Базовый* показатель (например, SPECint_base95) оценивает производительность при наличии набора ограничений: приложения могут использовать только четыре ключа компилятора[1], ключи должны быть идентичны для всех приложений и т. д. Идея состоит в том, что базовый показатель более точно представляет типичную производительность пользовательских приложений.
- *Пропорциональный* показатель (например, SPECfp_rate95) оценивает производительность всей, возможно, многопроцессорной, системы. Производитель может одновременно запустить необходимое количество копий тестовой программы; пропорциональный показатель – общий для всей системы.

Оценка коммерческой рабочей нагрузки

Конечно, не все приложения привязаны к вычислениям. По сути, большинство бизнес-приложений вовлекают в работу почти каждую часть системы. С учетом признания этого факта были разработаны тестовые программы, призванные оценивать более типичные пользовательские среды. Обсудим два из наиболее распространенных инструментов оценки *коммерческой рабочей нагрузки*: серию TPC для оценки производительности баз данных и комплект SPECweb для веб-серверов.

TPC

Совет по обработке транзакций (Transaction Processing Council), или TPC (*http://www.tbc.org*), – это вышедшая из индустрии организация, представляющая производителей компьютерных систем и баз данных. TPC дает методику оценки относительной производительности таких систем. Производительность оценивается по количеству времени, необходимому для выполнения транзакций при различных нагрузках.

[1] Ключи компилятора, конечно, использует не приложение, а тот, кто его компилирует. – *Примеч. науч. ред.*

В области баз данных к показателям TPC относятся с большим вниманием. Подобно SPEC, TPC должен был время от времени обновлять свои тестовые программы – по мере изменения потребностей пользователей и технических возможностей. В результате тесты TPC-A и TPC-S вышли из моды, TPC-C и TPC-D наиболее распространены, а TPC-W становится популярным в области электронной коммерции.

В серии TPC есть семь тестовых инструментов:

- TPC-A, выпущенный в ноябре 1989 года. Предназначен для оценки производительности баз данных с интенсивными обновлениями и терминальным пользовательским интерфейсом. Такие операционные среды встречаются в сценариях обработки транзакций в режиме онлайн (online transaction processing, OLTP). Подобные среды обычно имеют множественные терминальные сессии, существенный дисковый ввод-вывод, небольшое время выполнения системных задач и приложений, а также требуют целостности транзакций. TPC-A устарел.
- TPC-B, выпущенный в августе 1990 года. В отличие от TPC-A, TPC-B не является средством оценки OLTP. Скорее TPC-B есть тестовый инструмент по оценке баз данных в критическом режиме. Особое значение придается дисковому вводу-выводу без удаленных терминальных сессий. TCP-B устарел.
- TPC-C, принятый в июле 1992 года. TPC-C оценивает OLTP подобно TPC-A, но является значительно более серьезным инструментом. Он рассчитан на множество видов транзакций и более сложные базы данных. TPC-C полностью имитирует вычислительную среду, подобную среде оптового поставщика, где совокупность пользователей осуществляет транзакции с базой данных. TPC-C ориентируется на основные транзакции системы учета поступивших заказов: введение и доставку заказов, запись платежей, проверку статуса заказа и мониторинг имеющегося в распоряжении ассортимента.
- TPC-D, представленный в декабре 1995 года. Разработан для оценки производительности поддержки решений. По существу, тест состоит из базы данных, которая медленно модифицируется в режиме онлайн, в то время как другие процессы выполняют большие запросы объединения таблиц и поиски в базе данных. TPC-D устарел.
- TPC-H – это тестовая программа поддержки решений. Состоит из запросов для коммерческих задач и одновременных модификаций данных. Довольно похожа на TPC-D. Этот инструмент был разработан с целью оценки производительности систем, поддерживающих решения. Такие системы анализируют большие объемы данных, выполняют очень сложные запросы и дают ответы на насущные вопросы бизнеса.
- TPC-R – другая тестовая программа поддержки решений. Очень похожа на TPC-H, за исключением того, что она позволяет дополнительные оптимизации, основанные на опережающем знании запросов.

- TPC-W – это инструмент для оценки скорости транзакций в электронной коммерции. Имитируется коммерческая операционная среда Интернета, управляемая веб-сервером с поддержкой транзакций. Это и множественные одновременные сессии броузеров, и генерация динамических страниц, требующая обращения к базе данных и модификации данных, и согласованные веб-объекты, и выполнение транзакций в режиме онлайн, а также целостность транзакций и конфликты при одновременном обращении к данным и их модификации.

SPECweb99

В середине 1990-х, когда популярность Сети резко возросла, SPEC выпустил инструмент для оценки веб-производительности, названный SPECweb96. SPECweb96 был полностью сфокусирован на скорости, с которой динамические страницы доставлялись клиентам. В конечном итоге SPECweb96 был повержен новыми программными средствами. Дорогу к этим средствам проложила Sun Microsystems. Эти программы известны как ускорители сетевого кэша (Network Cache Accelerator), или NCA. NCA использует модуль ядра, чтобы явно кэшировать статический веб-контент в буфере памяти ядра, и отвечает на HTTP-запросы страниц, беря их из кэша, даже не обращаясь к самому веб-серверу.

К концу 1990-х взоры все более и более обращались к динамическому контенту – точное содержимое страницы зависело от данных формы или иным образом было уникально для пользователя. Под воздействием этих изменений SPEC выпустил новую редакцию тестовой программы SPECweb, известной как SPECweb99.

SPECweb99 оценивает количество одновременных *согласованных* соединений в секунду. Чтобы поддерживать соединение, необходимо выдерживать скорость между 400 и 320 Кбит/с. Набор транзакций включает в себя запрос и передачу как статического, так и динамического контента; набор по умолчанию показан в табл. 2.4.

Таблица 2.4. Состав транзакций SPECweb99

Тип запроса	Процентное отношение
Статический GET	70,00%
Динамический GET (простой)	12,45%
Динамический GET с изменяемым рекламным объявлением	12,60%
Динамический POST	4,80%
Динамический POST, вызывающий CGI-скрипт	0,15%

Статическая порция рабочей нагрузки моделирует гипотетического провайдера веб-контента. Страницы контента находятся в файлах разного размера. Обращение к ним происходит с разной частотой. SPEC

получил точные значения этих параметров на основе анализа log-файлов нескольких загруженных веб-сайтов. Как показано в табл. 2.5, рабочий комплект файлов контента умещается в четыре класса.

Таблица 2.5. Классы статического контента

Класс	Диапазон размеров (Кбайт)	Шаг размера (Кбайт)	Процентный состав
0	< 1	0,1	35%
1	< 10	1	50%
2	< 100	10	14%
3	< 1000	100	1%

Комплект рабочих файлов состоит из некоторого количества каталогов, каждый из которых содержит 9 файлов каждого класса – всего 36 файлов в каталоге. Файлы класса 0 увеличиваются на 0,1 Кбайт, класса 1 – на 1 Кбайт и т. д. Например, файлы класса 2 будут размером в 10, 20, 30, 40, 50, 60, 70, 80 и 90 Кбайт. Так как ожидается, что более крупные веб-серверы обслуживают большее количество файлов, точный размер комплекта рабочих файлов (количество создаваемых каталогов) определяется количеством одновременных соединений.

В ходе оценочного теста класс файла, извлекаемого из случайной директории, определяется на основе жесткого (fixed) распределения (как показано в табл. 2.4). При отборе файла из выбранного класса используется распределение Zipf.[1] Относительные величины «попаданий» для каждого файла в классе показаны в табл. 2.6.

Таблица 2.6. Относительная частота выборки файлов для заданного класса

Номер файла	Процентный состав	Номер файла	Процентный состав
0	3,9%	5	11,8%
1	5,9%	6	7,1%
2	8,8%	7	5,0%
3	17,7%	8	4,4%
4	35,3%		

SPECweb99 – это важный тест, поэтому поставщики идут на уловки, чтобы увеличить производительность, определяемую по его показателям. Но, скорее всего, это не те приемы, которые можно применить на

[1] Вероятность выбора n-го элемента в распределении Zipf пропорциональна 1/n. Обычно распределение Zipf применимо в ситуациях, где имеется много равнозначных альтернатив, таких как отбор книг в публичной библиотеке.

практике. Если при приобретении системы для ее оценки применяется SPECweb99, то к результатам следует подходить с долей скептицизма. Необходимо внимательно прочитать описание конфигурации. Производители обязаны предоставить информацию об использованных аппаратных средствах и о том, какие параметры были настроены.[1]

Тестовые программы пользователя

Самые значительные средства оценки – это те, которые администраторы разрабатывают сами с учетом своей операционной среды. По существу, есть несколько принципов, которым стоит следовать при разработке таких программ. Инструменты должны быть нацелены на реальные задачи, обладать сравнительно малым периодом прогона, быть в высшей степени автоматизированными. Кроме того, должны быть установлены правила для запуска таких программ.

Выберите круг задач

Прежде всего, необходимо определить задачи из реальной практики. К примеру, если речь идет о покупке компьютера для анализа генетических последовательностей, то следует отобрать несколько характерных последовательностей и для каждой подготовить свою программу. Замечательно, если большая часть работы связана с запуском одного приложения – нужно лишь увеличить долю нагрузки, возлагаемую на него при тестировании. Необходимо разработать инструменты не только для тестирования текущих операций, но и для оценки той работы, которую планируется проводить в ходе всего срока службы машины (скажем, от трех до пяти лет). Следует оценить потребности в памяти, выявить задачи с высокой вычислительной нагрузкой – другими словами, проанализировать разумное подмножество из того, чем будет загружена система в ближайшие пять лет.

Выберите время прогона

Конечно, прогнозировать время прогона трудно. Одним из лучших помощников может стать хорошо себя зарекомендовавшая и повсеместно используемая тестовая программа SPECfp2000. Если производительность текущей системы оценивается в 100 SPECfp2000, то система, оцениваемая в 1000 SPECfp2000, должна быть в десять раз быстрее (примерно так). Тестовая программа, работающая 20 мин. на текущей системе, может работать всего 2 мин. на новой. Хорошее правило заключается в том, что на новом тестируемом оборудовании каждая «задача» должна быть запущена на срок от 10 до 20 мин. Запуск полного

[1] Заметим, что если необходимо побить мировой рекорд, то действия по настройке системы будут отличаться от действий, предпринимаемых каждый день на практике. Производители слишком серьезно относятся к мировым рекордам.

комплекта отнимет не больше часа или двух, если нет других причин сидеть и запускать его много-много раз.

Максимальная автоматизация

Администратор или производитель, желающий вытрясти из него деньги, будут часто запускать эту программу. Поэтому ее нужно максимально автоматизировать. Лучше всего написать программу-оболочку (зачастую – скрипт командного интерпретатора), который будет выполнять всю работу: от компиляции тестовой программы до ее запуска и печати итоговых результатов вместе с текущими наиболее важными элементами конфигурации системы. Эти элементы могут включать в себя количество задействованных процессоров, объем установленной памяти или использованные ключи компилятора.

Установите правила запуска тестовой программы

Такие правила важно установить, особенно если программу будет запускать кто-либо кроме разработчика. Есть много вопросов, которые следует учесть при написании правил. Разрешены ли модификации? Допустимы ли изменения в коде с целью переносимости или может быть изменен алгоритм? Какие способы оптимизации допустимы при компилировании? Насколько близки должны быть результаты тестирования к базовым результатам? По сути, для полного взаимопонимания с производителем необходимы правила о правилах.

В этом разделе были затронуты лишь основы разработки тестовых средств. Настоятельно советуем обратиться к книге «Высокопроизводительные вычисления» («High Performance Computing», издательство O'Reilly) Кевина Доуда (Kevin Dowd) и Чарльза Северанса (Charles Severance) для получения значительно более глубоких сведений по этой теме.

Заключение

Будем надеяться, что эта глава послужила теоретическим и практическим фундаментом для понимания рабочей нагрузки и управления ею. В ней были обсуждены средства, помогающие оценить рабочую нагрузку на систему, а также некоторые методы наложения ограничений на эту нагрузку. Также речь шла о тестовых программах для сравнения производительности систем с типовой нагрузкой.

Часть этой главы может показаться по-детски простой. По мере продвижения в анализе производительности динамических систем становится ясно, что простые на первый взгляд вещи на самом деле оказываются сложными. Одна из целей этой книги – проверить некоторые упрощения, чтобы выяснить, что скрывается за ними. Такое исследование поможет принимать наилучшие решения в отношении того, куда направить усилия по настройке.

- *Архитектура микропроцессора*
- *Кэширование*
- *Планирование процессов*
- *Многопроцессорная обработка*
- *Периферийные соединения*
- *Инструменты для контроля производительности процессора*
- *Заключение*

3

Процессоры

Оценивайте производительность в показателях загрузки процессора, на основе параллельных ускорений или в MFLOPS на доллар.

D. H. Bailey, «Двенадцать способов надуть публику, говоря о показателях производительности на параллельных суперкомпьютерах». Supercomputing Review, 1991

Повышение производительности микропроцессоров за последние 20 лет феноменально. Основатель Intel Гордон Мур (Gordon Moore) предсказывал, что количество транзисторов на микропроцессорах будет удваиваться каждые 18 месяцев (и пропорционально будет расти производительность). Безусловно, его предсказание сбылось, если не сказать больше. В начале 1980-х годов самый быстрый микропроцессор Intel (4004) работал с тактовой частотой менее 4 кГц. Сравните эту цифру с показателями 2001 года. Процессоры IA-32 последнего поколения, выпускаемые компаниями Intel и AMD, работают с тактовой частотой около 2,0 ГГц и выполняют больше чем одну операцию за такт (некоторые ключевые элементы конструкции микропроцессоров будут представлены ниже). Такой огромный скачок трудно постичь. Если бы мощность автомобиля за последние 20 лет повышалась такими же темпами, то быстроходность преемника DeLorean DMC-12 (1981–1983), обладавшего максимальной скоростью примерно 140 миль в час[1], удваивалась бы 12 раз: автомобиль перемещался бы со скоростью 287 *тысяч* миль в час. Это выше скорости звука в 410 раз или, другими словами,

[1] Хотя DeLorean DMC-12 позиционировался как американский спортивный автомобиль и конкурент Corvette, это была достаточно тяжелая машина из-за широкого применения нержавеющей стали «dairy-grade» (пищевой стали). – *Примеч. науч. ред.*

составляет около одной шестнадцатой скорости света. И даже это трудно представить. Если бы цена блока жевательной резинки, стоившего четверть доллара в 1983 году, увеличивалась так же быстро, как и производительность микропроцессора, то сегодня этот блок стоил бы больше тысячи долларов.

Неясно, можно ли удерживать такие темпы роста. Часто предсказывалось, что темпы должны замедлиться. Однако, поскольку инженеры-производственники нашли способы усовершенствовать технологии, применяемые в разработке и производстве процессоров, такого замедления пока не произошло.

Повышение производительности микропроцессоров значительно опережало рост производительности других компонентов компьютерной системы. Поэтому основополагающая тема этой книги такова: как добиться максимальной производительности других компонентов, чтобы оптимизировать коэффициент полезного действия процессора. Несомненно, производительность процессора – главный персонаж при обсуждении прозводительности компьютера. Это легко проверить, если зайти в ближайший компьютерный магазин и посмотреть на описания. Тактовая частота микропроцессора крупно выделена и подчеркнута. Намного притягательнее возможность обладать новейшим процессором 2,0 ГГц, чем несколькими очень быстрыми жесткими дисками. Безусловно, производительность микропроцессора – это важнейший компонент полной производительности системы. Но в то же время микропроцессор часто переоценивают.

Несмотря на важность микропроцессора для большинства системных администраторов, микропроцессор – это съемный черный керамический блок с радиатором. Большинство задач, стоящих перед системными администраторами, не требует глубоких знаний электроники и схемотехники. Однако понимание работы устройств – сердцевина настройки производительности. Очень трудно улучшить производительность блока, не представляя, как он работает. В отношении производительности процессора основой таких знаний являются сведения о его конструкции. В этой главе за обзором базовой архитектуры микропроцессора последует обсуждение других, близких тем. Это кэши, играющие важнейшую роль в производительности современных процессоров. Это планирование процессов – как операционная система принимает решения о приоритете процессов в использовании CPU. Это и многопроцессорная обработка, и средства связи процессоров друг с другом и с периферийными устройствами. Наконец, будут обсуждены инструменты, служащие для мониторинга производительности микропроцессора.

Архитектура микропроцессора

В основе микропроцессора лежит физическая реализация набора правил. Эти правила, называемые *командами*, точно устанавливают, какие действия может выполнять микропроцессор. Набор всех этих правил называется *системой команд* (*instruction set*). Вместе с другими характеристиками система команд определяет *структуру системы команд* (*instruction set architecture*), или ISA, которая содержит всю необходимую информацию для написания корректно работающих программ. Для получения более детальных сведений о теоретических основах можно обратиться к разделу «Уровни представления» главы 1.

Двадцать лет назад разработчики микропроцессоров разбились на два конкурирующих лагеря, каждый со своей философией. Один лагерь предпочел систему команд, в которой отдельные инструкции были очень мощными, почти на уровне примитивов C. Такие команды довольно сложны. Этот лагерь был известен как пристанище разработчиков *вычислительных устройств со сложной системой команд* (*complex instruction set computing*, CISC). Другой лагерь приступил к созданию системы команд, в которой отдельные команды были просты, но их простота позволяла оптимизировать многие участки. Такая конструкция процессора известна как схема *вычислений с сокращенной системой команд* (*reduced instruction set computing*, RISC). Обе конструкции выполняют один и тот же объем работы. Разница состоит в том, что при использовании RISC, вероятно, выполняется больше команд, однако эти команды, скорее всего, выполняются за меньшее время.

RISC выиграл сражение за производительность. Все больше и больше конструкций процессоров CISC становятся похожи на RISC. Например, ядро процессора P6 Intel преобразует сложные команды IA-32 в более простой внутренний формат и лишь затем их выполняет. Другой хороший пример – процессор Transmeta Crusoe, который использует специальные аппаратные средства (так называемые *преобразователи кода*, *code morphing*) для частичного преобразования команд IA-32 во внутренние команды Crusoe.

В последнее время разработчики стали добавлять дополнительные команды, обычно для поддержки графики. Расширения VIS SUN и MMX Intel – хорошие тому примеры.[1] Такие команды основаны вот на чем. Хотя самый маленький адресуемый участок памяти обычно составляет 32 бит, работать можно и с 8 из них. То есть можно выполнить 4 операции с памятью и 4 арифметические операции, данные которых вмещены в 1 операцию с памятью. Таким образом, эти расширения работают за счет объединения всех 32 бит данных, а значит, необходима лишь

[1] Архитектура IA-32 компании Intel создана, как было отмечено, по варианту CISC. Однако MMX – это хороший пример системы специализированных графических команд.

1 операция с памятью и 1 арифметическая операция. Однако для того, чтобы эти расширения давали максимальный результат, нужны большие усилия команды разработчиков компилятора. Рисунок 3.1 показывает параллельный принцип работы расширений мультимедиа.

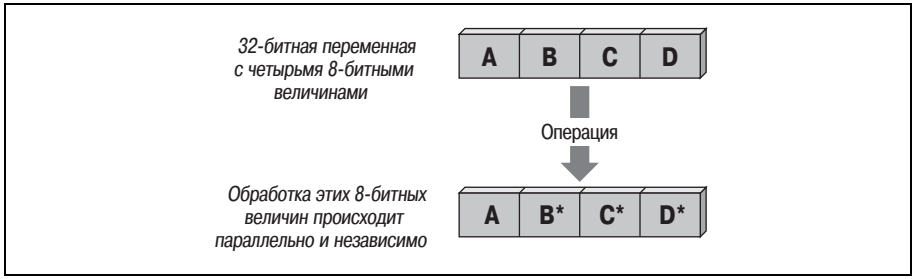


Рис. 3.1. *Параллельный принцип работы расширений мультимедиа*

Остаются ли процессоры RISC с добавленными расширениями попрежнему RISC (например, Ultra-SPARC Sun)? Важно помнить, что целью RISC было построение максимально быстродействующего микропроцессора с учетом специфичных технических ограничений, а не RISC сам по себе. Затем стали говорить, что процессоры RISC «второго поколения» на самом деле не являются RISC, а реализуют *вычисления с быстрой системой команд* (*fast instruction set computing*, FISC). Приверженцы этого названия считают, что оно более точно описывает устройство таких процессоров.

Тактовые частоты

Вообразим микропроцессор афинским гребным судном[1], плывущим по Средиземному морю. Чтобы корабль плыл со всей возможной скоростью, гребцы должны грести в одном темпе. Их действия синхронизированы ударами большого барабана. Точно так же все в микропроцессоре синхронизируется с помощью *внешнего сигнала синхронизации*. Сигнал циклично повторяется миллионы раз в секунду (один *герц* – это один цикл в секунду; сигналы синхронизации процессора обычно исчисляются в мегагерцах (МГц), или в миллионах герц). Такой сигнал обычно вырабатывается кварцем на плате с логическими схемами. В больших центрах обработки данных тактовый сигнал обычно генерируется на выделенной плате синхронизации. Эта тактовая частота увеличивается заданным *множителем*, в результате чего вырабатывается *внутренняя тактовая частота*. Такой множитель определяет, насколько быстр данный процессор по сравнению с другими подобными процессорами.

---

[1] Это мореходный корабль (обычно военный) с тремя рядами весел. Греческий город-государство Афины был известен своей военно-морской мощью, так же как Спарта своей армией.

Трудно выразить, насколько рискованно сравнивать производительность только на основе тактовой частоты. Таблица 3.1 иллюстрирует сравнение микропроцессора PowerPC 604e, применяемого IBM в серии RS/6000, и MIPS R10000, задействованного в компьютерах SGI.

Таблица 3.1. Сравнение производительности процессор/тактовая частота

Процессор	Тактовая частота (МГц)	SPECint95	SPECfp95
IBM PowerPC 604e	332	12,9	6,2
IBM PowerPC 604e	266	9,2	5,8
MIPS R10000	250	12,4	9,7

О тестовой программе SPEC речь пойдет далее (см. также раздел «SPECint и SPECfp» главы 2), однако ориентировочная прикидка здесь вполне уместна. При простом сравнении тактовых частот кажется, что процессор MIPS R1000 будет чуть-чуть медленнее, чем 266 МГц PowerPC и на 25% медленнее процессора 332 МГц. Однако тесты дают другой расклад: R10000 почти так же быстр, как 332 МГц PowerPC в целочисленных операциях и является лучшим (с большим отрывом) в вычислениях с плавающей точкой.

Проницательный читатель, проанализировав данные табл. 3.1, уже заметил, что повышение тактовой частоты не приводит к эквивалентному повышению производительности. Хотя тактовая частота 332 МГц IBM PowerPC на 25% выше, чем у его собрата с 266 МГц, производительность первого в операциях с плавающей точкой больше лишь на 6,8%. Причина состоит в том, что повышение тактовой частоты процессора увеличивает постоянную и существенную задержку при обращении к памяти. Особенно это характерно для операций с плавающей точкой, поскольку они большие.[1]

Предельная величина тактового сигнала зависит от неких эзотерических параметров архитектуры процессора – скажем, от параметров базовой полупроводниковой технологии. Регулированием таких параметров можно оптимизировать тактовую частоту. Однако на практике стоимость, теплоотдача и потребление энергии, характерные для конечного продукта, могут не дать ему выйти на рынок. Кроме того, тактовая частота зависисит от сложности выполняемой работы. Для выполнения конкретной задачи процессору необходимо определенное время. Самый медленный путь «внутри» процессора называется *критическим путем*. Уменьшение критического пути (например, с помощью упрощения конструкции) позволяет поднять тактовую частоту еще выше.

[1] Имеется в виду, что числа с плавающей точкой занимают в памяти больше места, чем, например, целочисленные операнды. – *Примеч. науч. ред.*

Другой подход к анализу производительности процессора сводится к выяснению среднего количества циклов, необходимого для выполнения команды. Название такого показателя – количество *циклов на команду* (*cycles per instruction*, CPI). Обычно чем меньше, тем лучше.

Конвейерная обработка

Так как не всегда возможно повысить производительность процессора за счет простого увеличения тактовой частоты, то к цели следует двигаться обходными путями. Один из методов подразумевает применение двух *функциональных единиц* в конструкции. Например, вместо одного устройства для целочисленной арифметики разместить два.[1] Однако такой подход недешев, если говорить о размере процессора.

Вместо этого можно внимательно рассмотреть, как же работает микропроцессор. Пока команды извлекаются из памяти, блок выполнения целочисленных арифметических операций простаивает. Конечно, это не лучший путь в создании быстродействующего процессора! Скажем, можно разработать процессор, который разделяет обработку команд на пять этапов:

Выборка команды (Instruction fetch, IF)

Выборка из памяти следующей команды для выполнения.

Декодирование команды (Instruction decode, ID)

Выяснение, что же делает команда.

Выполнение (Execute, EX)

Выполнение команды.

Память (MEM)

Запись и чтение данных из памяти.

Запись заново (Writeback, WB)

Запись полученных результатов в регистры.

Возможно, не каждая команда проходит через все эти этапы. Например, команде загрузки в память вовсе не нужно проходить через этап выполнения.[2] Итак, попробуем частично перекрывать выполнение команд.

Пусть выбрана первая команда. После того как она переходит на этап декодирования, выбирается следующая команда. По мере своей работы процессор таким же образом заполняет все свои «участки». То есть в текущий момент пять команд выполняются одновременно. Такой процесс называется *конвейерной обработкой* (*pipelining*).

---

[1] Процессор, использующий такой метод, называется суперскалярным. Этот процессор будет обсуждаться в разделе «Второе поколение конструкций процессоров RISC» далее в этой главе.

[2] Но такое может быть. Это зависит от особенностей конструкции процессора.

В описанном конвейере с пятью участками результаты замечательно выдаются каждый тактовый цикл. Однако это не означает, что каждая команда для своего выполнения требует только один цикл. Правильнее сказать, что для каждой команды необходимо пять циклов. Об этом важно помнить при измерениях CPI. В совершенном конвейерном процессоре значение CPI равно 1,0. Это означает, что одна команда выполняется в среднем за один тактовый цикл. Таким образом, конвейерная обработка не повышает скорость выполнения одной команды. Скорее она увеличивает производительность микропроцессора. Добавим, что разбивка команд на пять стадий привела к упрощению каждого этапа. Чем больше этапов, тем проще пройти каждый из них. Значит, таким путем можно повысить тактовую частоту процессора. Рисунок 3.2 иллюстрирует команды «в полете по конвейеру».

Рис. 3.2. *Команды «в полете по конвейеру»*

Конвейерная обработка – это один из основных приемов в истории повышения производительности процессора. Однако на деле не все так просто. Иногда, выполнение команды может длиться очень долго. Например, чтение данных из памяти может быть существенно дольше, если они не расположены в кэше. Это может привести к «залипанию» конвейера, его *останову* (*stall*). Разработчики микропроцессоров прилагают огромные усилия, чтобы избежать остановов конвейера. Такие «залипания» губительно сказываются на производительности.

Команды переменной длины

Плавная работа конвейера зависит от всех команд. Их исполнение занимает приблизительно одно время. В конструкции RISC все команды, как правило, работают с одним и тем же объемом данных. Однако команды CISC, в силу своей сложности, различаются по длине. Например, команда «возврат из подпрограммы» довольно компактна, а сложная команда «умножить-добавить» может быть длиннее. Рисунок 3.3 иллюстрирует различие между командами фиксированной и переменной длины.

Процессор не знает длину команды, пока он не декодирует ее и не определит, что команда собой представляет. Процессор должен извлечь ко-

манду до ее декодирования. Поэтому если команда длиннее извлеченной части, то процессор должен вернуться назад к этапу выборки и извлечь оставшуюся часть. Это вызывает дорогостоящее обращение к памяти и блокирует конвейер. Если, как в большинстве систем команд RISC, длина каждой команды фиксирована по определению, то таких трудностей не возникает, и конвейер может двигаться более гладко.

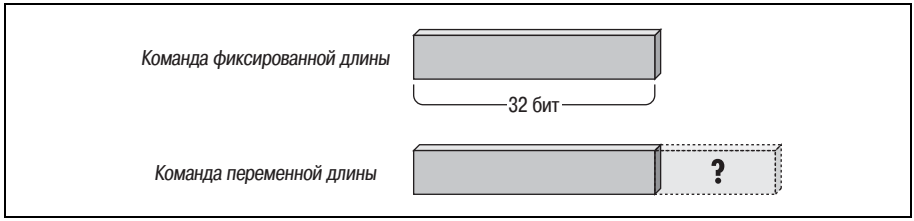


Рис. 3.3. *Команды фиксированной и переменной длины*

Условные переходы

Другая сложность в конвейерной обработке – это команды *условных переходов*. В зависимости от результатов вычислений условный переход определяет следующую команду, которую предстоит выполнить. Рассмотрим следующий фрагмент кода:

```
if (i == j) {
   delta++;
}
```

Оператор *if* – пример условного перехода. Такие операторы в большинстве систем команд RISC реализованы как команды условного *перехода по равенству* (*branch if equal*, BEQ) и условного *перехода по неравенству* (*branch if not equal*, BNE).

Когда процессор извлекает команду из памяти, он не знает, что собой представляет команда. Поэтому он извлекает следующую команду. О том, что команда является переходом, процессор узнает, когда команда уже декодирована. Если это «сквозной» переход (то есть следующая команда идет по направлению перехода), то все работает хорошо. Однако если выясняется, что переход «*изъят*» (то есть извлеченная команда не является следующей для выполнения), то процессору приходится возвращаться назад, удалять все незавершенные команды из конвейера и извлекать следующую команду, которую надо выполнить.

К сожалению, анализ показал, что почти каждая седьмая команда – это команда условного перехода. Если даже половина переходов – «сквозные», теряется около 20% производительности. Чтобы избежать очистки конвейера, многие процессоры RISC реализуют *отсрочку перехода*. Это означает, что команда, которая попадает на конвейер сразу после перехода, является либо «безобидной», либо полезной – какой бы переход ни был. Например, рассмотрим псевдокод:

```
loop:
   c--;
   i = j++;
   j = k + 4;
   l = m + n;
if (c == 0) goto loop;
```

Так как сложение *m* и *n* и сохранение результата в *l* не зависят от других вышестоящих команд в цикле, то для применения отсрочки перехода команды можно переставить в таком порядке:

```
loop:
   c--;
   i = j++;
   j = k + 4;
if (c == 0) goto loop;
l = m + n;
```

К тому времени, когда процессор решит, какой переход должен быть осуществлен, эта команда почти извлечена. Если нельзя найти хорошую команду для отсрочки перехода, можно просто вставить *пустой оператор* (*no-operation*, NOP). Обычно это не возлагается на программиста. Такую работу берет на себя ассемблер.[1] Это изящное решение уменьшает воздействие остановов процессора, происходящих из-за условных переходов. Однако когда процессор стал способен параллельно выполнять больше команд, потребовались более мощные подходы.

Если бы было можно и дальше снижать коэффициент неудач условных переходов, то конвейер работал бы более эффективно. Помня об этом, разработчики процессоров придумали схему, которая позволяет процессору предсказывать направление перехода. Когда команда декодирована и процессор видит, что это переход, он обращается к таблице последних операций перехода. Основываясь на характере последних переходов, он делает предположение и немедленно начинает извлечение команд из предсказанного участка. Если прогноз неверный, то следует отменить все текущие команды. Один заведомо простой алгоритм предсказания переходов – всегда предполагать, что условный переход, ссылающийся на более ранний адрес, «изъят». Такой подход удивительно верен, поскольку многие программы выполняют многократные итерации по циклу.

```
for (i = 0; i <= 100000; i++) {
   j[i] = k[i] + j[i];
}
```

Если переход предсказан верно, то затраты на условные переходы не больше затрат на любые другие команды. Если нет, то следует остано-

[1] В данном случае под «ассемблером», очевидно, понимается часть компилятора, которая генерирует исполняемый код. – *Примеч. науч. ред.*

вить процессор. Однако будем надеяться, что это не будет случаться часто. Простая схема предсказания условных переходов точна на 95%. Это существенно снижает издержки из-за остановов вследствие переходов.

Второе поколение конструкций процессоров RISC

Совершенный процессор RISC первого поколения смог бы выполнить одну команду за тактовый цикл. Даже если бы пятиэтапный конвейер был полностью загружен пятью командами, полное значение CPI лишь приближалось бы к 1,0. Достигнув пределов в усовершенствовании конструкций первого поколения, компании-разработчики микропроцессоров начали гонку по созданию самого быстродействующего RISC-процессора. Стратегии разделились на три основные части: игры со скоростью, суперскалярные конструкции и суперконвейерные разработки.

Игры со скоростью основаны на оптимизации простого, маленького, эффективного ядра для очень больших тактовых частот. Этот подход представлен в процессоре Alpha компании Digital Equipment Corporation (DEC), тактовая частота которого исторически примерно в два раза превышала тактовую частоту ближайшего конкурента.

Повышения производительности без увеличения тактовой частоты можно достичь за счет добавления дополнительных вычислительных элементов внутрь процессора. На таком подходе основана конструкция так называемого *суперскалярного* процессора. Количество и тип операций, которые могут выполняться одновременно, зависят как от приложения, которое должно быть создано в расчете на параллелизм, так и от процессора, который должен иметь достаточно функциональных блоков и плотно загружать их работой. Такая идея кажется простой, но на самом деле это довольно сложно как для разработчика компилятора, так и для проектировщика аппаратных средств. Современные суперскалярные процессоры могут выполнять шесть команд за тактовый цикл. На пике производительности значение CPI в них составляет 0,17!

Одна из реализаций суперскалярных вычислений называется архитектурой *с командными словами сверхбольшой длины* (*very long instruction word*, VLIW). В такой схеме процессор сам по себе очень простой. В нем почти нет передовых аппаратных средств планирования, обычных для скалярных процессоров. В процессоре VLIW, таком как Transmeta Crusoe, команды сгруппированы в пачки (командные слова) – от трех до четырех в одной пачке.[1] Процессор последовательно загружает каждое командное слово и параллельно выполняет все команды на нескольких функциональных блоках. Разработчикам компиляторов приходится нелегко: написать оптимизированный компилятор для процессора VLIW исключительно трудно. В этой области по-прежнему

[1] В ранних процессорах VLIW, созданных Multiflow Computer, ни много ни мало 28 операций могли быть объединены в одно командное слово.

ведутся активные исследования. Конструкция Crusoe облегчает эту задачу за счет аппаратной реализации рекомпилирования «на лету».

В процессорах RISC «проще» часто означает «быстрее». Возможно, некоторые этапы прохождения команд в конвейере излишне сложны. Особенность конструкции *суперконвейера* – это снижение сложности этапов за счет увеличения их количества. Если уменьшенная сложность дает процессору возможность работать быстрее, то суперконвейерные процессоры могут достичь такого же увеличения производительности, что и суперскалярные процессоры. Однако в длинных конвейерах условные переходы могут быть обработаны достаточно поздно, что делает остановы из-за ошибочно предсказанных переходов очень накладными. В то же время суперконвейерная обработка может быть объединена с другими подходами. Примером может служить MIPS R8000, который сочетал суперскалярную конструкцию с двумя длинными конвейерами.

Однако разработать быстрый компьютер – это больше, чем просто создать быстрый микропроцессор. Устранение узких мест в вычислениях выявляет еще больше узких мест, особенно в том, как информация извлекается из процессора и помещается в него.

Кэширование

Главная трудность при оптимизации микропроцессора заключается в том, что остановы конвейера приводят к серьезному снижению производительности. Если каждый запрос к памяти связан с обращением к оперативной памяти, то конвейер будет останавливаться очень часто. Для того чтобы обойти это затруднение, в современных процессорах повсюду применяют кэши. В некоторых процессорах кэш реализован как единый блок – так называемая *объединенная* структура кэша. Однако в большинстве случаев кэш разделен на две части: для данных и для команд. Такая реализация называется *гарвардской* (*Harvard*) структурой кэша. На практике гарвардский кэш значительно эффективнее, особенно в случае маленького кэша.

Иерархия кэша

Быстродействие памяти увеличивалось значительно медленнее, чем скорость процессора. Производительность процессора удваивалась приблизительно каждые восемнадцать месяцев, а производительность памяти удваивалась примерно каждые семь лет. Важно понимать: так как скорость процессора опередила быстродействие памяти, то даже кэши статической RAM, расположенные вне корпуса процессора, недостаточно быстры.[1] Полное время, необходимое для передачи данных

[1] Более подробно о различных типах памяти будет рассказано в главе 4.

в память или из памяти, больше времени физического доступа к памяти, поскольку существуют накладные расходы при передаче данных между чипами и при синхронизации кэшей и оперативной памяти. Самый типичный алгоритм для уменьшения времени доступа – это создание иерархии кэшей с монотонно увеличивающимся размером и монотонно уменьшающейся производительностью:

- Процессор имеет отдельные кэши для команд и данных: *i-кэш* и *d-кэш* соответственно. Зачастую емкость каждого около 16 Кбайт. Это самые быстрые кэши в системе. Они называются кэшами *уровня 1*, или кэшами *L1*.
- Кэши второго уровня, хранящие как данные, так и команды. Они большей емкости – обычно между 256 Кбайт и 4 Мбайт. «Промах» кэша на уровне L1 может, тем не менее, привести к удачному обращению к кэшу второго уровня. Хотя такая операция займет немного больше циклов, она по-прежнему будет очень эффективной. Это кэши *уровня 2*, или *L2*.
- Иногда в схеме присутствует кэш третьего уровня. Его емкость обычно от 4 до 32 Мбайт. И хотя он еще медленнее, чем кэш L2, такой кэш на порядок быстрее оперативной памяти.
- Если промахи кэша произошли на всех этих уровнях, необходимо обратиться к оперативной памяти. Если система памяти сильно загружена, то возможна долгая задержка. Этого хочется избежать.

В большинстве микропроцессоров, обладающих самой высокой тактовой частотой, реализована трехуровневая структура кэша. Уровни кэша и время доступа к ним на примере типичного процессора 1,0 ГГц представлены в табл. 3.2.

Таблица 3.2. Типичные скорости доступа к памяти

Тип и размещение кэша	Время доступа (в нс)	Размер	Процент «попаданий»[a]
L1 на чипе	1–3	8–64 Кбайт	>90%
L2 на чипе	6–18	1–8 Мбайт	>50%
L3 вне чипа	30–40	8–32 Мбайт	>30%
Оперативная память	220	Очень большой	–

[a] Лучше всего это описать как «процент запросов к кэшу, которые могут быть обеспечены данными, уже находящимися в кэше».

Каждый раз, когда данные загружаются из кэша более низкого уровня, они также загружаются и в кэши над ним. Когда обращение к памяти сводится к обращению в область кэша, такое попадание называется *удачным обращением в кэш* (*cache hit*). Когда необходимо «идти сквозь» кэш и обратить взор на следующий уровень иерархии, то это означает, что произошел *промах кэша* (*cache miss*). Каждый промах кэша привносит издержки. Предсказание издержек часто затрудни-

тельно. Новейшие реализации процессоров, подобные UltraSPARC компании SUN, имеют переменные издержки кэша. В основном издержки пропорциональны разнице между тактовой частотой процессора, тактовой частотой кэша и скоростью оперативной памяти. С точки зрения производительности очень важно удачно «попадать» в кэш как можно чаще. Рассмотрим два сценария обращения к памяти исследуемой системы и определим среднее время доступа для каждого из них:

- Попадание в кэш L1 в 65% случаев, в кэш L2 в 25% и в оперативную память в 10% случаев.

  $0{,}65(4\ нс) + 0{,}25(5\ нс) + 0{,}10(220\ нс) = 25{,}85\ нс$

- Попадание в кэш L1 в 90% случаев, в кэш L2 в 8% и в оперативную память в 2% случаев.

  $0{,}90(4\ нс) + 0{,}08(5\ нс) + 0{,}02(220\ нс) = 8{,}4\ нс$

Хотя 8,4 наносекунды кажутся чрезвычайно хорошим результатом, следует иметь в виду, что период тактовых импульсов этой системы – порядка наносекунды. Тогда становится ясно, почему так важно, чтобы удачные обращения к кэшу случались как можно чаще и как можно выше в его иерархии (рис. 3.4).

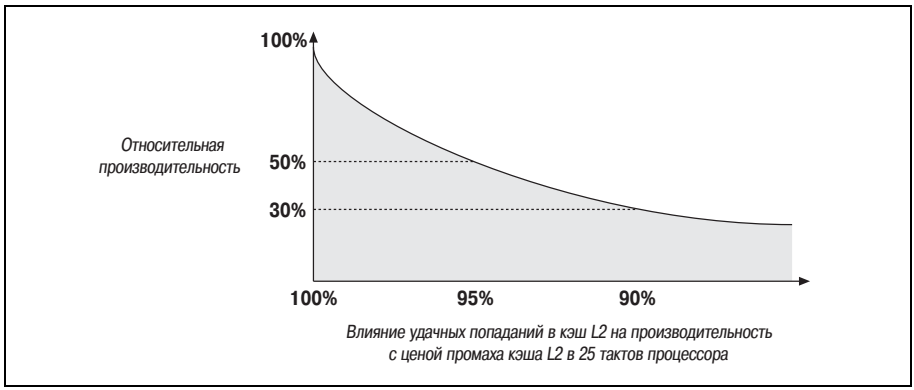


Рис. 3.4. *Производительность как функция частоты удачных обращений к кэшу L1*

Организация и работа кэша

Кэши разбиты на участки равного размера, называемые *строками*. Строка обычно содержит от четырех до шестнадцати последовательных участков памяти. Хотя данные в строке физически прилегают друг к другу в оперативной памяти, две отдельные строки кэша могут содержать данные, которые хранятся в несмежных ячейках оперативной памяти.

Когда данные запрашиваются из кэша, система проверяет, есть ли эти данные в одной из строк кэша. Если есть, то они быстро извлекаются;

если нет, то запрос переходит на следующий уровень кэша (или к оперативной памяти). Когда данные извлекаются из кэша более низкого уровня, они помещаются в некую строку в кэше, перезаписывая в ней данные, если это необходимо.

Важно отметить, что данные в кэше – это не «оригинал» данных. Поэтому когда данные в кэше изменяются, у системных администраторов есть три возможности:

- Обновить только копию в кэше, что можно сделать очень быстро. Однако сейчас «оригинал» и кэшированный эквивалент различаются. Это модель *с обратной записью*.[1]
- Непосредственно обновить оба экземпляра. Такая операция может быть медленной, но эффективной, если затраты на запись в кэш невелики. Это *сквозная запись*.
- Отбросить кэшированный экземпляр и обновить только оригинал. Это имеет смысл, если затраты на запись в кэш высоки.

Не все в оперативной памяти можно кэшировать: обычно участки, связанные с устройствами ввода-вывода, не кэшируются. Это определяется диапазоном адресов, а не данными или командами, хранимыми в участках памяти.

Ассоциативность

Каждая строка в кэше образует пару с каноническими данными[2], хранимыми в оперативной памяти. Это называется *отображением* (*mapping*). Кэш содержит специальную запись для каждой строки. Такая запись называется *тегом* (*tag*). Здесь хранится информация о том, с какими адресами в памяти ассоциирована эта строка. В некоторых реализациях хранится также время последнего обращения. Такие отображения могут быть выполнены тремя разными способами.

Прямое отображение (direct mapping)

Это непосредственный способ отображения оперативной памяти в строках кэша. В этой схеме нулевая строка кэша хранит целые кратные размера кэша.[3] Для кэша в 32 Кбайт адреса памяти от 0 и

---

[1] Также эту модель называют «записью с обратным копированием», так как запись данных из кэша в оперативную память происходит только тогда, когда соответствующие данные в кэше замещаются другими. – *Примеч. науч. ред.*

[2] «Канонические данные» – это просто «оригинал» данных; каноны здесь ни при чем. – *Примеч. науч. ред.*

[3] Ячейки оперативной памяти, имеющие одинаковые младшие разряды в своем адресе, попадают в одну ячейку кэш-памяти, т. е. <адрес ячейки кэш-памяти> = <адрес блока оперативной памяти> mod <число ячеек в кэш-памяти>; в зависимости от организации кэш-памяти это могут быть не ячейки, а строки, совокупности ячеек». – *Примеч. науч. ред.*

далее с шагом 32 Кбайт (0, 32 Кбайт, 64 Кбайт и т. д.) будут отображены в первой строке кэша. Реализовать это просто и недорого. Однако при работе с двумя областями памяти, каждая из которых кратна размеру кэша, такой подход может привести к снижению производительности (*пробуксовке*). Представим копирование содержимого массива A в массив B, причем и A, и B составляют ровно 32 Кбайт. Содержимое A будет считано, но содержимое B должно быть помещено в ту же самую строку кэша. Это быстро вызовет большое количество промахов кэша.

Полностью ассоциативное отображение (fully associative mapping)

Полностью ассоциативное отображение дает возможность любой строке кэша отображать любой адрес памяти. В отличие от прямого отображения, каждая строка кэша в полностью ассоциативном кэше кое-что знает о данных, которые она содержит. Когда процессор запрашивает данные, опрашивается каждая строка кэша – не содержит ли она данные. Если данных нет ни в одной из них, значит, произошел промах кэша. Строки кэша отслеживают последнее время обращения к данным. Поэтому когда необходимо перезаписать строку, старые данные можно отбросить.[1] Полностью ассоциированные кэши имеют хорошие показатели эффективности, но такие кэши очень дороги.

Кэширование с заданной ассоциативностью (set associative caching)

Кэш с заданной ассоциативностью – это компромисс между методами прямого и полностью ассоциированного отображения. Такие кэши состоят из групп по два или четыре кэша с прямым отображением. Всякий раз, когда строка кэша должна быть замещена, можно выбрать один иэ этих кэшей. Такой подход избавляет от трудностей с пробуксовкой, которые наблюдались в кэшировании с прямым отображением.

Локальность и «кэш-нарушители»

Кэши так эффективны потому, что при работе программ проявляются особые свойства обращения к памяти, называемые *временно́й локальностью* и *пространственной локальностью*. После обращения к команде или данным следующая выполняемая команда или следующие извлекаемые данные оказываются рядом с предыдущей порцией – либо с точки зрения пространства между ними, либо по времени. Временнбя локальность зависит от повторного использования чего-либо. Пространственная локальность зависит от одновременного использования команд или данных, которые расположены близко друг к другу.

[1] «Старость» данных в строке кэша определяется методом LRU (Least-Recently Used): заменяется тот блок, который не использовался дольше всех. – *Примеч. науч. ред.*

Фрагменты кода, приводящие к нарушению функциональности кэша, называются «кэш-нарушителями». Обычно «кэш-нарушители» разбивают временную/пространственную локальность, на которую опирается кэш для эффективности работы. Три наиболее типичных вида «кэш-нарушителей» – это неединичный шаг, связные списки и копирование блоков, выровненных по размеру кэша.

Единичный шаг

Лучше всего кэши действуют, когда программа работает с памятью последовательно. Для примера рассмотрим короткие целые числа (16 бит) и строки кэша длиной 128 бит. При обращении к первому целому числу следующие семь будут также загружены в строку кэша, поэтому следующие семь обращений будут произведены очень быстро. Это называется *единичным шагом* (*unit stride*), потому что адрес каждого последующего обращения к данным на единицу больше предыдущего. Следующий код демонстрирует единичный шаг:

```
for (i = 0; i <= 100000; i++) {
   j = j + Array[i];
}
```

Если программа выполняется с неединичным шагом, то страдает ее производительность, потому что каждое обращение к памяти требует обращения к оперативной памяти. Вот пример цикла с неединичным шагом:

```
for (i = 0; i <= 100000; i += 16) {
   j = j + Array[i];
}
```

Хотя этот код выполняет одну шестнадцатую работы цикла с единичным шагом, оба цикла выполняются одинаково! Обращение к некэшированной памяти оказывает сильнейшее влияние на производительность цикла с неединичным шагом. Для дополнительных примеров оптимизации цикла следует обратиться к разделу «Примеры оптимизации» главы 8.

Связные списки

Представим связный список с несколькими тысячами записей. Каждая запись состоит из данных и ссылки на следующий элемент. Поиск по списку осуществляется путем его обхода. Так как код для поиска по списку может быть очень компактным, такой подход хорошо работает в случае кэша команд. Однако при каждом запросе к данным будет происходить обращение к различным строкам кэша. Поэтому каждое обращение к данным влечет за собой промах кэша. Если размер связного списка превышает размер кэша и необходимо очищать строки кэша для записи следующего элемента из списка, то следующий шаг по-

иска не найдет начала списка в кэше! Важно отметить, что элементы списка редко расположены рядом в памяти. Поэтому итерация по списку разрушает принцип пространственной локальности, на который опирается кэш.

Единственный способ преодолеть эту трудность – придумать новый алгоритм. Когда растет быстродействие процессора, дело становится еще хуже, потому что несоответствие тактовой частоты процессора/кэша и времени обращения к памяти увеличивается еще больше.

Копирование блоков, выровненных по размеру кэша

Немалая часть времени ядра уходит на копирование и обнуление блоков данных, особенно страничными участками (обычно размером 4 или 8 Кбайт).[1] Эти данные не нужно кэшировать, потому что наверняка они не будут использоваться сразу. По существу, такие операции служат лишь для перемещения полезных данных из кэша. Наиболее эффективный способ копировать страничные фрагменты данных при кэшировании с обратной записью – это читать строку кэша, а затем записывать ее как блок. Выглядит это так:

1. Читаем первый фрагмент источника, вызывая промах кэша.
2. Читаем всю строку кэша источника.
3. Записываем первый фрагмент в целевой приемник, вызывая промах кэша.
4. Записываем всю строку кэша в целевой приемник.
5. Повторяем до завершения. На более позднем этапе целевая строка кэша будет очищена для более полезных данных.

Если размер участка между приемником и источником кратен размеру кэша, а система применяет кэш с прямым отображением, то источник и приемник будут использовать одну и ту же строку кэша. Тогда произойдут промахи чтения и записи для каждого элемента строки. Это крайне неэффективно. Вот пример:

```
#define CACHESIZE 0x40000
char source[CACHESIZE], target[CACHESIZE];
for (i = 0; i < CACHESIZE; i++) {
   source[i] = target[i];
}
```

В процессорах реализовано ускорение копирования блоков. В системе команд VIS SUN, реализованной в серии процессоров UltraSPARC, есть команда копирования блока. При ее выполнении удерживается соответствие между кэшами и оперативной памятью, но не задействуются кэши. На VIS-процессорах вызовы функций `bcopy(3C)` и `memcpy(3C)` такой механизм применяют автоматически.

[1] Страницы будут обсуждены в главе 4.

Неправильность размера кэша

Как явствует из вышесказанного, «увеличение кэша означает увеличение производительности». Если код работает в режиме «нарушения кэша», то может оказаться, что команды выполняются гораздо быстрее вообще без доступа к кэшу процессора! Иногда можно наблюдать очень странные результаты. Например, когда SPARCstation 5 Model 110 превосходит SPARCstation 20 Model 85 при большом моделировании. В чем же дело?

SPARCstation 5 применяет процессор microSPARC, разработанный для недорогих, эффективных операций. В нем маленький кэш, высокий процент промахов кэша, но очень низкая стоимость таких промахов. Процессор SuperSPARC-II, применяемый в SPARCstation 20, обладает намного большим кэшем и значительно меньшим процентом промахов. Издержки при промахах кэша в нем существенно выше. В некоторых приложениях (таких как обработка баз данных) большой кэш работает эффективно, поэтому SPARCstation 20 значительно быстрее. Но в моделировании, где размер кэша не настолько важен для производительности, система SPARCstation 5 чрезвычайно эффективна вследствие меньшей стоимости промахов кэша. Конструкция UltraSPARC объединила в себе элементы, способствующие очень маленькому периоду ожидания при доступе к оперативной памяти.

Хотя рассмотренный пример относился к несовременному оборудованию, сложности такого рода стали появляться все чаще и чаще. Например, процесоры Celeron порой превосходят свои «профессиональные» аналоги (Pentium-II) за счет сниженной стоимости промахов кэша. Для точного выяснения того, как действует кэш в конкретных приложениях, может потребоваться всестороннее тестирование.

Еще один источник влияния на эффективность кэша – это механизм окрашивания страниц. Он имеет отношение к алгоритму, применяемому ядром операционной системы при решении о том, какие страницы следует поместить в кэш процессора. Более подробную информацию по этой теме можно найти в разделе «Окрашивание страниц» главы 4.

Планирование процессов

Другим источником воздействия на производительность процессора является планирование процессов. Планирование процессов – это механизм, согласно которому операционная система определяет, какой процесс отдать на выполнение процессору. Следующие четыре раздела посвящены различным моделям планирования процессов, применяемым в мире UNIX.

Модель System V: модель Linux

Linux воплощает относительно простую, стандартную для UNIX модель планирования. Она поддерживает приоритетное прерывание обслуживания процессов (прерывание запущенного процесса процессом с более высоким приоритетом). В то же время в ядре Linux такая возможность не реализована.[1]

Алгоритм планирования ядра Linux основан на разбиении времени CPU на единицы, называемые *периодами* (*epochs*). В каждом периоде любой процесс имеет определенный *квант*, продолжительность которого вычисляется в начале периода. Следовательно, каждый процесс может иметь различный квант. Этот квант представляет собой наибольшее время CPU, которое может быть выделено процессу в этом периоде. Когда процесс израсходовал свой квант, он извлекается из процессора и замещается другим запущенным процессом. Конечно, планировщик может выбирать процесс несколько раз, пока квант процесса не истечет (например, процессор может быть освобожден от ожидания завершения дискового ввода-вывода процесса).

Когда планировщик решает, какой процесс запустить следующим, он рассматривает приоритет каждого процесса. Существует два типа *приоритета*: *статический* и *динамический*. Пользователи назначают статический приоритет процессам реального времени. Такой приоритет никогда не регулируется. Динамический приоритет применяется только к фоновым процессам. Это сумма *кванта базового времени* (так называемый *базовый приоритет* процесса) и количества тиков времени CPU, остающихся у процесса до того, как его квант в текущем периоде истечет. Статический приоритет процесса реального времени всегда выше любого динамического приоритета. Планировщик запускает условный процесс лишь тогда, когда нет запущенных процессов реального времени.

Период заканчивается, когда все запущенные процессы израсходовали свои кванты. Далее планировщик пересчитывает продолжительность квантов для всех процессов, и начинается новый период. Если процесс израсходовал свой квант в предыдущем периоде, то по умолчанию ему назначается квант, равный кванту базового времени процесса. Новый процесс наследует квант базового времени своего родителя. Так как процесс 0 (планировщик свопинга) по умолчанию имеет квант базового времени 20 тиков (около 200 мс), то каждый процесс по умолчанию имеет такой же квант базового времени. Значение кванта базового времени для конкретного процесса настраивается с помощью системных вызовов `nice(3C)` и `setpriority(3C)`, а также командой *nice*.

[1] Имеется в виду, что выполнение кода ядра не может быть прервано. – *Примеч. науч. ред.*

Определение приоритета процесса

Определить приоритет запущенного процесса можно по колонке PRI вывода команды *ps -el* (пример 3.1).[1]

Пример 3.1. Определение приоритета процесса в Linux

```
% ps -el
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
100 S     0     1     0  0  60   0    -   326 do_sel ?        00:00:48 init
040 S     0     2     1  0  69   0    -     0 contex ?        00:00:00 keventd
040 S     0     3     1  0  69   0    -     0 kswapd ?        00:05:32 kswapd
040 S     0     4     1  0  69   0    -     0 krecla ?        00:00:00 kreclaimd
040 S     0     5     1  0  69   0    -     0 bdflus ?        00:24:43 bdflush
040 S     0     6     1  0  69   0    -     0 kupdat ?        00:23:01 kupdate
...
140 S     0 11602   488  0  68   0    -   745 tcp_da ?        00:00:00 sendmail
140 S     0 11727   488  0  68   0    -   745 tcp_da ?        00:00:00 sendmail
140 S     0 11793   488  0  69   0    -   745 tcp_da ?        00:00:00 sendmail
000 R   563 11877  5019  0  75   0    -   759 -      pts/3    00:00:00 ps
```

Настройка эффективного приоритета процесса

Настройка эффективного приоритета процесса осуществляется напрямую с помощью команды *renice*. Команда *renice* изменяет абсолютное значение nice процесса.[2] Это значение может варьироваться от +20 (такие процессы можно запустить лишь при простаивании системы) до -20 (процессы планируются в зависимости от их приоритетов). Значение по умолчанию – ноль. Например, рассмотрим такой процесс (*rhof*):

```
% ps -el |grep rhof
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   563  5620  5601  0  69   0    -   514 do_sel pts/6    00:00:00 rhof
```

Снизим приоритет этого процесса до -20:[3]

```
% renice -20 5620
5620: old priority 0, new priority 20
% ps -el |grep rhof
```

---

[1] В разных системах UNIX ключи команды ps отличаются. Например, в современных версиях Linux и FreeBSD для получения той же информации надо дать команду `ps auxw`. – *Примеч. науч. ред.*

[2] В оригинале – «niceness». От nice (англ.) – здесь: любезный. Это значение говорит о том, насколько данный процесс «любезен» к остальным. – *Примеч. перев.* (Другое название – «фактор уступчивости». – *Примеч. науч. ред.*)

[3] Помните, это лишь пример! Наделение пользовательского процесса слишком высоким приоритетом в системе чревато узурпацией процессора этим процессом, а это затруднит выполнение других, намного более важных задач в системе, например подгрузку страниц с диска (swaping). – *Примеч. науч. ред.*

```
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   563  5620  5601  0  69  19    -   514 do_sel pts/6    00:00:00 rhof
```

Отметим, что значение в колонке NI стало равным 19 – приоритет процесса уменьшился. Непривилегированные пользователи не могут повысить приоритет своих процессов, даже если они сначала его уменьшили.[1]

```
% ps -el |grep rhof
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   563  5865  5601  0  70   0    -   514 do_sel pts/6    00:00:00 rhof
% renice +5 5865
5865: old priority 0, new priority 5
% ps -el |grep rhof
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   563  5865  5601  0  71   5    -   514 do_sel pts/6    00:00:00 rhof
% renice +3 5865
renice: 5865: setpriority: Permission denied
```

Чтобы уменьшить приоритет процесса, необходимо иметь права root:

```
% sudo renice -20 5865
5865: old priority 0, new priority -20
% ps -el |grep rhof
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S  1000  3274  3240  0  59 -20    -  1619 select pts/1    00:00:00 rhof
```

Это простой и легкий способ регулирования приоритета запущенного процесса.

Модификации для систем SMP

В многопроцессорных системах (SMP) возможны интересные головоломки. Представим два процессора: один из них бездействует, а другой обрабатывает процесс с относительно низким приоритетом. На занятом процессоре появляется процесс с высоким приоритетом. Следует ли немедленно запустить процесс с высоким приоритетом на бездействующем CPU и при этом перемещать между процессорами строки кэша, задействованные процессом? Или отложить его и запустить позднее на текущем процессоре? В Linux принято эмпирическое правило, связанное с размером кэша процессора: чем больше кэш процессора, тем больше процесс будет ждать выделения ему времени этого процессора.

---

[1] В книгах по UNIX часто бывает характерная путаница с увеличением и уменьшением приоритета процесса. Дело в том, что чем меньше числовое значение NICE, тем выше (больше) приоритет процесса. Поэтому если числовое значение уменьшается, реальный приоритет процесса растет. Все, кроме пользователя root, могут только понижать приоритет своих процессов, то есть увеличивать числовое значение NICE. – *Примеч. науч. ред.*

Многоуровневые классы планирования: модель Solaris

Одна из замечательных особенностей операционной среды Solaris – возможность контроля администратора над планированием обработки процессов в операционной системе. Такая возможность обеспечивает многоуровневое управление приоритетами запускаемых процессов. Важно помнить, что Solaris непосредственно не планирует процессы; скорее она планирует базовые потоки ядра (threads).

Модель поточной работы Solaris

Перед тем как продолжить разговор о планировании процессов, необходимо детально обсудить, что собой представляет поток[1] ядра. Solaris реализует многоуровневую поточную модель. Ее назначение – разделить управление потоками пользовательского уровня и работу ядра. Потоки уровня пользователей имеют свою схему приоритетов. Их планирование проводится с помощью потока-планировщика. Такой поток создается библиотекой потоков, когда многопоточное приложение компилируется и выполняется. Благодаря этому многопоточные приложения могут порождать тысячи потоков без существенной загрузки ядра. По сути, ядро не видит пользовательские потоки, пока они не присоединяются к *легковесному процессу* (*lightweight process*, LWP[2]), имеющему определенное положение в операционной системе. Поток-планировщик отвечает за отображение пользовательских потоков в LWP, который связан с потоком ядра для выполнения процессором.[3] Каждый LWP имеет поток ядра, но необязательно каждый поток ядра имеет LWP. Часть потоков ядра задействованы исключительно операционной системой, поэтому LWP здесь не требуется.

---

[1] Thread (поток) в переводной литературе часто называют «нитью». Несмотря на то что в UNIX есть еще одна разновидность потоков (потоки ввода-вывода), в этом разделе мы будем придерживаться термина «поток» и для последовательностей команд, грубо говоря, для подпроцессов. – *Примеч. науч. ред.*

[2] Иерархия процессов, легковесных процессов и потоков ядра в Solaris (SunOS) такова: любой процесс может состоять из любого количества пользовательских, т. е. запрограммированных программистом-пользователем потоков. Пользовательские потоки одного пользовательского процесса объединяются в один или несколько легковесных потоков. Потоки одного LWP выполняются в общей виртуальной памяти и имеют общие ресурсы (в частности, дескрипторы открытых файлов). Каждому LWP при запуске этого LWP ядро ставит в соответствие один поток ядра. Подробнее о планировании процессов в Solaris можно прочесть по адресам *http://sakima.ivy.net/~carton/academia/solaris-lwp.pdf* и *http://www.sdteam.com/articles/glava_21.html*. – *Примеч. науч. ред.*

[3] Приложение без поддержки потоков имеет один LWP и поток ядра, ассоциированный с ним.

Время от времени встречаются высказывания, что потоки ядра и потоки LWP – одно и то же. На самом деле они четко различаются, но в рассуждениях о процессах пользователей вполне приемлемо представлять их одним целым. Между потоками LWP и потоками ядра отношение всегда «один к одному», без исключений.

Далее при обсуждении планирования под словом «поток» будем подразумевать именно *поток ядра*.

Классы планирования

Обычно при планировании потоки делятся на пять категорий: с разделением времени (`ts`), интерактивные (`ia`), ядро, реального времени (`rt`) и прерывание. Относительные приоритеты в каждом из этих классов могут быть отображены в абсолютные приоритеты. Такое отображение обобщено в табл. 3.3.

Таблица 3.3. Табличное представление приоритетов процессов в Solaris

Диапазон абсолютных приоритетов	Диапазон относительных приоритетов	Очередь планирования
169–160 (или 109– 100)	9–0	Потоки прерывания (если класс реального времени не загружен)
159–100	59–0	Приоритеты реального времени
99–60	39–0	Приоритеты ядра
59–0	59–0	Приоритеты с разделением времени и интерактивные приоритеты

По умолчанию процессы попадают в класс с разделением времени. Кроме того, есть интерактивный класс, предназначенный для повышения интерактивной производительности в оконных системах. Такое повышение достигается за счет назначения главным процессам активного окна наивысшего приоритета. Процессы, отбираемые для помещения в интерактивный класс, – это процессы, возникшие в оконной среде. Удаленные[1] процессы по-прежнему попадают в класс с разделением времени. Однако приоритеты интерактивного класса и класса с разделением времени перекрываются, поэтому интерактивный процесс с приоритетом 59 запускается с тем же приоритетом, что и процесс с разделением времени с приоритетом 59.

Рассмотрим, как операционная система решает, какие потоки следует запустить.

[1] В смысле «запущенные с удаленных компьютеров». Разумно предполагается, что процессы реального времени не будут запускать через сеть. – *Примеч. науч. ред.*

Заметки о планировании в реальном времени

Планирование в реальном времени относится к средствам, с помощью которых процессу по мере необходимости обеспечивается квант процессора. Такая схема характерна для некоторых типов приложений, например в робототехнике или управлении полетами. Вообще, поддержка такого планирования в операционной системе делится на три категории: поддержки нет совсем, *строгая* поддержка реального времени (для приложений соблюдаются все предельные сроки) и *нестрогая* поддержка реального времени (предельные сроки могут быть пропущены, но поддерживается статистический минимум). Solaris – это среда с нестрогой поддержкой реального времени.

В нестрогой среде реального времени период отклика должен иметь конечную продолжительность (например, верхний предел времени, в течение которого приложение может освободить процессор; в Solaris такой период может составлять 5 мс). Обычно в UNIX поддержка реального времени не реализована. Причин две. Во-первых, ошибки из-за отсутствия страниц в оперативной памяти могут вызвать непредсказуемую задержку (см. раздел «Пейджинг и свопинг» в главе 4). В Solaris решение сводится к блокированию в памяти всех страниц процесса реального времени. Другая сложность состоит в том, что в большинстве систем UNIX ядро не является *прерывающим* (*preemptible*). Это означает, что оно может быть смещено из процессора при запуске другого процесса. Ядро может не быть прерывающим при наличии некоторых обстоятельств. Они называются *точками непрерываемости* (*nonpreemption points*). Непрерываемость проявляется лишь в отдельных случаях. Вот наиболее типичный пример: ядро исполняется в контексте прерывания. Однако прерывание с высоким приоритетом может замещать прерывание текущего обслуживаемого процесса.

Диспетчер

Когда поток готов к выполнению, он помещается в *очередь обработки*. Каждый процессор имеет отдельный набор таких очередей. Состояние ожидания в очереди обработки называется состоянием запуска (RUN). Когда процессор освобождается, диспетчер берет поток с наивысшим приоритетом из очереди обработки этого процессора и помещает его в процессор. Теперь поток в состоянии «внутри процессора» (ONPROC) и может выполнять вычисления.

Поток может покинуть процессор в силу разных обстоятельств. Первый вариант: поток выполняет вычисления, пока не достигнет предела выделенного ему времени (определяемого приоритетом потока). Та-

кой период называется *квантом*. Так как процессы с бульшим приоритетом выбираются более часто, то квант обычно обратно пропорционален приоритету (процессы с бульшим приоритетом имеют меньшие кванты). Когда квант исчерпан, потоку присваивается приоритет нового уровня (значение `tqexp`). Такой уровень почти всегда ниже приоритета потока при старте. Это сделано для обеспечения «справедливой разбивки» при планировании.

Другой случай выхода потока из процессора: поток блокируется. Например, при ожидании завершения ввода-вывода или выдачи блокировки. В этом случае поток переходит в состояние «сна» (`SLEEP`). Потокам, переходящим «ко сну» и при этом удерживающим «критические» ресурсы (такие как блокировки чтения/записи), назначается приоритет в классе ядра. Таким образом, эти потоки имеют шанс быстро снять блокировку. Когда процессы из «спящей» очереди возвращаются в процессор, им присваивается новый приоритет (определяется параметром `slpret`). Такое значение обычно выше, чем предыдущий приоритет потока. Поэтому после «сна» поток быстро получает время для своего выполнения.

Третий, и последний, случай выхода потока из процессора – принудительный, обычно по инициативе ядра, *прерывающего* поток. Если прерванный поток достаточно долго ожидает «внимания» процессора, то его приоритет повышается до значения, задаваемого параметром `lwait`. Таким образом, потоки получают компенсацию за долгое ожидание в очереди обработки после прерывания.

Проверка приоритета процесса

Определить, с каким легковесным процессом и классом планирования ассоциирован конкретный процесс, можно с помощью */usr/bin/ps* и ключей *-L* и *-c* (пример 3.2).

Пример 3.2. Просмотр приоритета процессов в Solaris

```
% /usr/bin/ps -efcL
     UID   PID  PPID   LWP  NLWP  CLS PRI    STIME TTY      LTIME CMD
    root     0     0     1     1  SYS  96   Mar 20 ?        0:00 sched
    root     1     0     1     1   TS  58   Mar 20 ?        0:06 /etc/init -
    root     2     0     1     1  SYS  98   Mar 20 ?        0:00 pageout
    root     3     0     1     1  SYS  60   Mar 20 ?       22:37 fsflush
...
    root   200     1     1     7   TS  51   Mar 20 ?        0:00 /usr/sbin/nscd
    root   200     1     2     7   TS  58   Mar 20 ?        0:00 /usr/sbin/nscd
    root   200     1     3     7   TS  59   Mar 20 ?        0:00 /usr/sbin/nscd
    root   200     1     4     7   TS  58   Mar 20 ?        0:00 /usr/sbin/nscd
    root   200     1     5     7   TS  59   Mar 20 ?        0:00 /usr/sbin/nscd
    root   200     1     6     7   TS  59   Mar 20 ?        0:00 /usr/sbin/nscd
    root   200     1     7     7   TS  58   Mar 20 ?        0:01 /usr/sbin/nscd
    root   185     1     1     1   TS  48   Mar 20 ?        0:01 /usr/sbin/cron
...
```

```
 root  4897   164     1     1   TS  38 20:28:02 ?        0:00 in.telnetd
 root  5235  4899     1     1   TS  48 21:10:19 pts/1    0:00 /usr/bin/ps -cefL
  gdm  4899  4897     1     1   TS  58 20:28:02 pts/1    0:00 -csh
```

Вывод этой команды значительно сокращен. Особенно интересны колонки LWP, CLS и PRI, в которых соответственно представлены номер легковесного процесса, класс планирования и приоритет. Отметим, что многопоточные процессы (например, кэширующий демон службы имен, */usr/sbin/nscd*) могут одновременно иметь несколько LWP (количество LWP, с которыми ассоциирован процесс, представлено в колонке NLWP).

Настройка таблиц диспетчера

Таблицы диспетчера настраиваются с помощью команды *dispadmin*. Настройка сводится к выводу диспетчерской таблицы для конкретного класса в файл, редактированию этого файла и обратной загрузке файла в таблицу диспетчера.

Чтобы узнать, что же доступно для редактирования, следует запустить *dispadmin -l* и определить активные классы планирования:

```
% dispadmin -l
CONFIGURED CLASSES
==================

SYS     (System Class)
TS      (Time Sharing)
RT      (Real Time)
IA      (Interactive)
```

Далее с помощью *dispadmin -c class -g* можно сделать дамп таблицы планирования:

```
% dispadmin -c TS -g
# Time Sharing Dispatcher Configuration
RES=1000

# ts_quantum  ts_tqexp  ts_slpret  ts_maxwait ts_lwait  PRIORITY LEVEL
       200         0        50           0        50        #     0
       200         0        50           0        50        #     1
       200         0        50           0        50        #     2
       200         0        50           0        50        #     3
...
       200         0        50           0        50        #     9
       160         0        51           0        51        #    10
       160         1        51           0        51        #    11
       160         2        51           0        51        #    12
       160         3        51           0        51        #    13
       160         4        51           0        51        #    14
       160         5        51           0        51        #    15
...
        40        42        58           0        59        #    52
```

```
40      43      58      0       59      #       53
40      44      58      0       59      #       54
40      45      58      0       59      #       55
40      46      58      0       59      #       56
40      47      58      0       59      #       57
40      48      58      0       59      #       58
20      49      59      32000   59      #       59
```

Вывод этой таблицы значительно урезан. Рассмотрим его по частям. Первая строка определяет временнэю дискретность для таблицы ниже этой строки. Обратная величина временнуй дискретности используется для толкования колонки `ts_quantum` таблицы диспетчера: по мере возрастания значения `RES` значения `ts_quantum` соответственно растут. По умолчанию `RES` равен 1000. Обратная величина `RES` равна 0,001, что в долях секунды составляет одну миллисекунду. Значит, по умолчанию единицами для поля `ts_quantum` являются миллисекунды. Хотя значение дискретности можно менять, лучше оставить значение по умолчанию.

Назначение большинства из этих колонок уже обсуждалось выше в этой главе в разделе о работе диспетчера (см. раздел «Диспетчер»). Подведем краткий итог:

- Поле `ts_quantum` представляет максимальный период времени, в течение которого поток может оставаться в процессоре.
- Поле `ts_tqexp` определяет приоритет, который будет назначен потоку после перемещения из процессора, когда истечет его полный квант времени.
- Поле `ts_slpret` сообщает приоритет, который будет иметь процесс, когда он будет обрабатываться после «сна» (например, после ожидания завершения ввода-вывода).
- Поля `ts_maxwait` и `ts_lswait` контролируют компенсацию процессу (присваиваемый приоритет задается `ts_lswait`), когда процесс на долгое время был вытеснен потоком с приоритетом ядра (больше чем `ts_maxwait`).

Некоторые детали представляют особый интерес. Во-первых, значения кванта уменьшаются в соответствии с уровнем приоритета. Во-вторых, значение `maxwait` для уровня приоритета 59 очень высоко. Поэтому справедливый шанс быть выбранным есть у каждого потока.

Каждый класс приоритетов имеет нулевое значение `ts_maxwait` для каждого приоритета. Исключение составляют приоритеты наивысшего уровня. Поэтому приоритеты всех потоков, за исключением потоков разделения времени с наивысшим приоритетом, будут переназначаться в пределах 50–59 примерно один раз в секунду. Таким образом, поток, обрабатываемый процессором в течение длительного периода, не будет «наказываться». Со временем потоки с излишним потреблением времени CPU прекратят работать основную часть своего времени с приоритетом от 0 до 9, так как они будут продолжать расходовать

квант времени и претерпевать переназначение приоритетов из-за ts_tqexp. Раз в секунду их приоритет будет повышаться. Он может быть снова снижен лишь при условии, что потоки продолжат интенсивно загружать процессор.

Класс реального времени (RT) представлен гораздо более простой таблицей:

```
# Real Time Dispatcher Configuration
RES=1000

# TIME QUANTUM                    PRIORITY
# (rt_quantum)                      LEVEL
      1000                    #        0
      1000                    #        1
      1000                    #        2
      1000                    #        3
      1000                    #        4
...
       100                    #       55
       100                    #       56
       100                    #       57
       100                    #       58
       100                    #       59
```

Таблица проще, потому что планирование потока реального времени не вовлекает ни одну из сложных настроек, описанных в предыдущем случае. Потоки реального времени выполняются до их останова или до истечения их кванта (когда многочисленные потоки реального времени находятся в состоянии ожидания). Также отметим, что, хотя уровень приоритета и варьируется от нуля до 59, потокам реального времени присваивается более высокий приоритет, нежели потокам ядра, так как эти относительные приоритеты отображаются в абсолютные с помощью отображения, обсужденного выше).

Таблицы диспетчера можно «прилизать» с помощью записи вывода *dispadmin* в файл (вывод *dispadmin* был показан выше) и редактирования файла. Затем этот файл следует загрузить, запустив *dispadmin -C class -S file*.

Наиболее типичная методика – *сдвинуть* класс разделения времени, чтобы высокоприоритетные задания с высокой загрузкой процессора выполнялись без принудительного снижения их приоритета. Выполнить это можно путем редактирования диспетчерской таблицы, относящейся к классу разделения времени (пример 3.3).

Пример 3.3. Создание предела приоритета в классе разделения времени

```
# Time Sharing Dispatcher Configuration
RES=1000

# ts_quantum  ts_tqexp  ts_slpret  ts_maxwait ts_lwait  PRIORITY LEVEL
...
        40         36         57           0         57        #     46
```

```
40      37      58      0       58      #       47
40      38      58      0       58      #       48
40      39      58      0       58      #       49
40      40      58      0       58      #       50
40      41      58      0       58      #       51
40      42      58      0       58      #       52
40      43      58      0       58      #       53
40      44      58      0       58      #       54
40      45      58      0       58      #       55
40      46      58      0       58      #       56
40      47      58      0       58      #       57
20      48      58      32000   58      #       58
200     49      59      0       59      #       59
```

Изменившиеся значения выделены. В результате ни один процесс не попадет на уровень приоритета 59 (вот для чего был выверен параметр `ts_lwait`). Уровень приоритета 58 действует так же, как и уровень приоритета 59. Он имеет необычно низкий квант и высокое значение `ts_maxwait`. Увеличение кванта уровня 59 приведет к тому, что потоки этого класса будут выполняться дольше – до тех пор, пока они не будут вынуждены покинуть процессор, освободив место другим потокам. Как будет описано в следующем разделе, для вывода процесса на этот уровень приоритета необходимо запустить *priocntl*.

В этой схеме есть одна маленькая загвоздка: все процессы на уровне приоритета 59 необходимо убрать. Для этого нужно создать копию примера 3.3 и изменить последнюю строку следующим образом (для уровня приоритета 59):

```
# Time Sharing Dispatcher Configuration
RES=1000

# ts_quantum  ts_tqexp  ts_slpret  ts_maxwait ts_lwait  PRIORITY LEVEL
...
      200         0         0           0         0       #     59
```

Теперь любой процесс на уровне приоритета 59 быстро его покинет. Далее для нахождения процессов, по-прежнему имеющих приоритет 59, можно запустить *ps –ecL | grep 59*. Такие процессы необходимо остановить и перезапустить.

Настройку таблиц диспетчера следует проводить с осторожностью. Неверные действия могут привести к потере устойчивости и неполадкам в работе производственной системы. Тестируйте внимательно!

Регулирование приоритетов

Регулировать приоритет процесса можно напрямую с помощью команды *priocntl*. Ключ *-s* служит признаком установки приоритета. Также необходимо применить либо *-p priority*, чтобы задать нужный уровень приоритета, либо *-i pid process-id*, чтобы указать ID процесса, чей приоритет будет изменен. Рассмотрим пример. Прежде всего, с помощью

настройки таблиц диспетчера вытесним процессы с разделением времени с уровня приоритета 59 (пример 3.3). Затем переместим процесс *nscd* на уровень приоритета 59:

```
# /usr/bin/ps -ecL | grep 59
# /usr/bin/ps -ecL |/usr/xpg4/bin/grep -E 'nscd|PID'
   PID   LWP  CLS PRI TTY     LTIME CMD
   200     1   TS  51 ?        0:00 nscd
   200     2   TS  58 ?        0:00 nscd
   200     3   TS  58 ?        0:00 nscd
   200     4   TS  58 ?        0:00 nscd
   200     5   TS  58 ?        0:00 nscd
   200     6   TS  58 ?        0:00 nscd
   200     7   TS  58 ?        0:01 nscd
# priocntl -s -p 59 -i pid 200
# /usr/bin/ps -ecL |/usr/xpg4/bin/grep -E 'nscd|PID'
   PID   LWP  CLS PRI TTY     LTIME CMD
   200     1   TS  59 ?        0:00 nscd
   200     2   TS  59 ?        0:00 nscd
   200     3   TS  59 ?        0:00 nscd
   200     4   TS  59 ?        0:00 nscd
   200     5   TS  59 ?        0:00 nscd
   200     6   TS  59 ?        0:00 nscd
   200     7   TS  59 ?        0:01 nscd
```

Преимущество подобного приема заключается в том, что высокоприоритетные задания получают максимально много процессорного времени вне зависимости от окружения, в котором они запущены.

Команда *renice* уже обсуждалась в разделе «Настройка эффективного приоритета процесса» этой главы. Она предоставляет другой способ установки приоритета процесса, а именно за счет статического смещения в таблице приоритетов. Это смещение равно значению аргумента *renice*, но с обратным знаком. Например, аргумент –20 команды *renice* приведет к смещению процесса в таблице приоритетов на +20 (то есть его приоритет никогда не упадет ниже 20).

Многопроцессорная обработка

В середине 1980-х многопроцессорные системы с разделением памяти казались скорее экзотическими. Они были дорогостоящими. С тех пор стоимость аппаратных средств упала, в операционных системах появилась серьезная поддержка мультипроцессинга, а потребность в приемлемых по стоимости, мощных комплексах уровня рабочих групп и даже уровня настольных систем увеличилась. Такие многопроцессорные системы основаны на архитектуре *унифицированного обращения к памяти* (*uniform memory access*, UMA), где вся физическая память в равной мере доступна всем процессорам. Некоторые крупные системы применяют архитектуру *неунифицированного обращения к памяти* (*nonuniform memory access*, NUMA), где отдельный процессор

может иметь более быстрый доступ к физической памяти. Рисунок 3.5 иллюстрирует архитектуры UMA и NUMA.

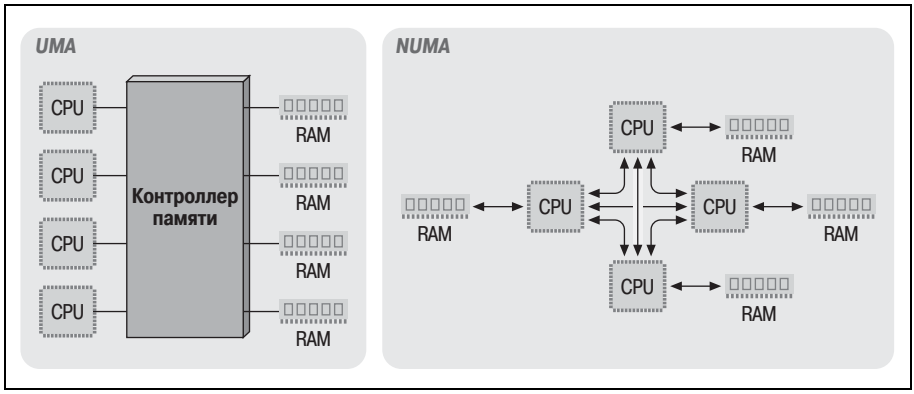


Рис. 3.5. *Архитектуры UMA и NUMA*

Здесь будут обсуждаться только архитектуры UMA.

Взаимодействие процессоров

Для деятельности человеческого организма очень важно, чтобы информация могла поступать в мозг и разноситься из него. Точно так же для успешной работы многопроцессорных систем необходимо взаимодействие участков системы. Если провести аналогию между компьютером и человеческим организмом и сравнить процессор с головным мозгом, то «спинной мозг» содействует обмену данными между процессором с одной стороны и оперативной памятью и периферийными устройствами с другой. В архитектуре компьютера для этого служат шины и коммутационные соединители. Кроме того, они делают возможным арбитраж взаимодействия. Это означит, что компоненты способны определить, кто может общаться в каждый момент времени. Важно заметить, что речь здесь идет только о взаимодействии между процессорами и оперативной памятью. Для более подробной информации о взаимодействии между периферийными устройствами следует обратиться к разделу «Периферийные соединения» далее в этой главе.

Шины

Шина состоит из трех частей: протокола коммуникации, совокупности параллельных проводов, которые связывают все компоненты системы, и вспомогательной аппаратуры. Шины дешевы и их легко создавать. Но по мере увеличения количества подключенных к шинам устройств и роста нагрузки на шины последние становятся узким местом производительности. Поэтому размер и производительность кэша – это важные характеристики многопроцессорных конфигураций, основанных на шинах. Обычно такая архитектура включает в себя не более

четырех процессоров. В то же время более крупные системы поддерживают несколько десятков процессоров. Так что вполне возможно, что подсистема памяти или шинная подсистема будут переполнены еще до установки максимального количества процессоров!

Шины реализуют протокол с *коммутацией каналов* или с *коммутацией пакетов*. В протоколе с коммутацией каналов два устройства организуют на шине коммуникационный канал и поддерживают его до окончания транзакции между ними. Такой протокол реализовать просто. Однако если взаимодействие происходит с медленным устройством (например, с оперативной памятью), то процессор вынужден ждать. Эти состояния ожидания уменьшают коэффициент полезного действия шины и растрачивают тактовые циклы: чем больше тактовая частота процессора, тем больше уходит тактовых циклов.

Старшие системы Sun применяли шинную архитектуру MBus, основанную на коммутации каналов.[1] В SPARCserver 20 можно было установить четыре модуля 50 МГц SuperSPARC с SuperCache. Для того времени это была в высшей степени быстродействующая система. К сожалению, когда процессоры 50 МГц были инсталлированы на SPARCstation 20, тактовая частота MBus автоматически снизилась с 50 МГц до 40 МГц – для обеспечения синхронизации данных между процессорами. Печально, что система, которой была необходима быстрейшая MBus, не получила ее. Вследствие ограничений в блоках SPARCstation 20, связанных с поглощением тепла, в таких системах не было возможности установить процессоры быстрее 50 МГц.

Так как при взаимодействии быстрых и медленных устройств по протоколу с коммутацией каналов (например, MBus) происходят потери циклов, Sun реализовала протокол с коммутацией пакетов для шины XBus. Этот протокол был применен на каждой плате в сериях SPARCserver 1000 и 2000, а также в CRAY SuperServer CS6400.[2] Платы были связаны через XDBus, состоявшую из множественных перемежающихся шин XBus. Протокол с коммутацией пакетов основан на разбиении каждой шинной транзакции на пакет-запрос и пакет-ответ. Реализовать это трудно, так как каждый пакет должен содержать идентификатор устройства, выпустившего его. Также на шине необходимо специальное устройство (такое как подсистема памяти), чтобы ставить в очередь запросы, приходящие слишком быстро.

В связи с выпуском быстрых процессоров UltraSPARC компания Sun разработала новую архитектуру взаимосвязей, названную *UltraSPARC*

---

[1] SPARCstation 10, 20 и SPARCserver 6x0MP. Другие системы, такие как SPARCstation LX, Classic, 4 и 5, также основаны на MBus, но в них нет поддержки мультипроцессинга. Если *uname -a* возвращает sun4m, то в машине применяется архитектура MBus.

[2] Когда Cray был куплен Silicon Graphics в 1996 году, подразделение, отвечающее за SuperServer CS6400, было продано Sun. Впоследствии эта группа создала систему Ultra Enterprise 1000 («Starfire»).

Port Architecture switched interconnect (UPA). Это архитектура с коммутацией пакетов, характеризующаяся наличием в структуре «коммутатора». В этой архитектуре все шины проходят через централизованный коммутатор, который надлежащим образом их переназначает, а также масштабирует размер шин исходя из особенностей устройств. Например, устройствам памяти нужна очень широкая шина, а каналам ввода-вывода не нужно быть столь широкими. Такая подгонка размеров позволяет сохранять низкую стоимость. Все порты в коммутаторе работают с одной тактовой частотой. Однако часть из них – это *ведущие порты* (процессоры, контроллеры ввода-вывода), которые могут как инициировать, так и отвечать на запросы. Другая часть – это *ведомые порты* (видеобуферы, память), которые могут только отвечать на запросы. Архитектура UPA устойчива к конфликтным ситуациям на шинах при большой нагрузке, поскольку каждое устройство обладает своим соединением с шинным коммутатором.

В системах с несколькими системными платами серверной линии UltraEnterprise шина UPA была модифицирована для обеспечения работы многоплатных систем.[1] Это вылилось в создание пассивной внутренней магистрали Gigaplane, являющейся распределенным расширением средств шины UPA. В ней возможна оптимизация, которая повышает эффективность спин-блокировок и разделения памяти.

Производительность этих шин подробно представлена в табл. 3.4.

Таблица 3.4. Характеристики многопроцессорной шины/коммутатора Sun

Шина	Тактовая частота (в МГц)	Максимальная пропускная способность (в Мбайт/с)	Максимальная производительность чтения (в Мбайт/с)	Максимальная производительность записи (в Мбайт/с)
MBus	40	320	90	200
XBus (per board)	40	320	250	250
Single XDBus (SPARCserver 1000)	40	320	250	250
Dual XDBus (SPARCcenter 2000)	40	640	500	500
Quad XDBus (SuperServer CS6400)	55	1 760	1 375	1 375
UPA (системы UltraSPARC)	82	1 600	1 300	1 300
Gigaplane (системы Ultra Enterprise)	82	2 624	1 600	1 600
Gigaplane XB (Ultra Enterprise 10000)	100	До 10 622	До 12 800	До 12 800

[1] Системы Ultra Enterprise 250 и Ultra Enterprise 450 – системы с одной платой, реализующие UPA.

Коммутационные соединители

Коммутационные соединители были разработаны в связи с проблемой производительности в архитектуре с одной шиной. По существу, соединители – это совокупность нескольких параллельных шин, за счет которых каждый компонент может получить доступ к другому компоненту по любому тракту, проходящему через соединитель. Помимо связывания участвующих сторон, соединитель, подобно телефонисту, должен разрешать конфликты, например, когда два процессора пытаются обратиться к одному и тому же биту памяти. Поскольку в такой схеме нет одной разделяемой шины, производительность в целом хорошая. Однако вследствие необходимости дополнительных аппаратных средств коммутационные соединители обычно более дороги и присутствуют только в мощных многопроцессорных системах.

Пример такой реализации – коммутационный соединитель Gigaplane XB, применяемый в системе Sun Ultra Enterprise 10000. Это расширение объединительной платы Gigaplane, задействованной в других системах Ultra Enterprise. Он улучшает пропускную способность архитектуры, но при этом повышается стоимость и время обращения к памяти. Такое увеличение времени обращения к памяти вызвано дополнительным уровнем схемы. В Gigaplane XB нет оптимизации взаимных блокировок, включенной в Gigaplane, так как Gigaplane XB был разработан командой Cray SuperServer до того, как ее приобрела Sun. На самом деле при запуске с ограничением полосы пропускания соединитель Gigaplane быстрее, чем Gigaplane XB. Полная пропускная способность соединителя XB достижима лишь в полностью скомпонованной системе. Это проиллюстрировано в табл. 3.5.

Таблица 3.5. Изменение пропускной способности в зависимости от количества скомпонованных процессоров

Архитектура	Количество процессоров	Максимальная пропускная способность (в Мбайт/с)	Время задержки (в нс)
UPA	1–4	1 600	170
Gigaplane	1–30	2 624	240
Gigaplane XB	1–12	1 333	400
	13–20	2 666	
	21–28	4 000	
	29–36	5 331	
	37–44	6 664	
	45–52	7 997	
	53–60	9 330	
	61–64	10 622	

В конфигурациях с количеством процессоров меньше двадцати архитектура Gigaplane имеет преимущество в производительности. При

большем их количестве начинает себя проявлять Gigaplane XB: пропускная способность увеличивается в четыре раза при удвоении количества процессоров. В небольших системах с Gigaplane (четыре процессора или менее) повышение времени задержки неизбежно. Вследствие увеличенной сложности коммутации задержка примерно в два раза больше, чем в системах с UPA. Это означает, что четырехпроцессорный одноплатный сервер Ultra Enterprise часто превосходит четырехпроцессорный многоплатный сервер Ultra Enterprise при значительно меньшей стоимости!

Системы UltraSPARC–III: Fireplane

По мере перехода на архитектуру UltraSPARC-III Sun разработала новую технологию соединений, названную *Fireplane*. В архитектуре UPA предыдущего поколения Fileplane улучшает как производительность, так и набор характеристик соединений. В отличие от обсужденных ранее соединений, Fireplane не задает топологию устройства. Вместо этого он описывает протокол, который устройства, подключенные к соединителю, используют для коммуникаций. Это дает хорошую степень гибкости на будущее. Другое интересное новое свойство Fireplane – поддержка нескольких одновременных незавершенных транзакций (например, в Sun Blade 1000 в состоянии ожидания могут одновременно находиться до пятнадцати запросов передачи 64-байтных данных).

Sun Blade 1000 реализует шинную топологию с коммутацией пакетов как для адресных трактов, так и для трактов данных. Тракт данных – это 288-битная шина (256 бит данных плюс 32 бит информации об исправлении ошибок). Она основана на группе из шести чипов под общим названием *комбинированный коммутатор процессор/память* (*combined processor memory switch*, CPMS). Соединения «точка-точка» между трактом данных и устройствами, подсоединенными к нему, упрощают арбитражную логику.

Контроллер памяти, интегрированный в процессор UltraSPARC-III, снижает задержку и повышает пропускную способность. Адресоваться к памяти можно через 144-разрядный тракт данных, связанный с чипом коммутации данных, имеющим 576-разрядный тракт к памяти (512 бит данных плюс 64 бит информации о коррекции ошибок). Интересно, что в двухпроцессорных системах активен только один из процессорных контроллеров памяти. Этот процессор может напрямую обращаться к памяти, тогда как другой должен обращаться к памяти через соединение Fireplane.[1]

Само по себе соединение допускает пиковую пропускную способность 67,2 Гбайт/с с поддерживаемой пропускной способностью до 9,6 Гбайт/с (2,4 Гбайт/с на процессор до предела 9,6 Гбайт/с).

[1] Для облегчения управления связностью кэша все адресные запросы памяти идут через магистраль.

Архитектура «без соединений»

Некоторые процессоры UltraSPARC, обычно используемые в простых системах, не требуют формальных соединений. К числу таких процессоров относятся так называемые разработки «серии I» и «серии E», такие как UltraSPARC-IIi и UltraSPARC-IIe. Для процессоров серии I характерна высокая интегрированность и оптимизация цена/производительность, тогда как процессоры серии E разработаны для встроенных приложений.

Параллельная обработка в операционных системах

Большинство современных операционных систем поддерживают *многозадачность*. По существу, многозадачность не требует более одного процессора. Это просто возможность выполнять более одного процесса одновременно. Система делит свое время между процессами на основе контекстного переключения через заданные интервалы времени, а также в зависимости от активности ввода-вывода и прерываний. В многопроцессорной системе отдельные процессы могут быть распределены по процессорам для общего увеличения быстродействия (но без увеличения производительности отдельно взятого приложения). Большинство процессов – самодостаточные.[1] В то же время процессы могут прекрасно взаимодействовать друг с другом и совместно использовать данные с помощью средств *межпроцессной коммуникации* (IPC), таких как каналы и сокеты, или с помощью *тесно разделяемой памяти* (*intimately shared memory*, ISM). Вообще говоря, реализовывать IPC с целью увеличения производительности отдельного приложения на многопроцессорной системе UMA – это не лучшая идея. Однако на очень крупных системах с параллельными вычислениями такое вполне допустимо.

Потоки

Поток – это не процесс! Жизнь процесса начинается как единичный поток, но в ходе своего выполнения процесс может обрастать потоками или сбрасывать их. Все эти потоки разделяют одно и то же пространство памяти процесса, но каждый поток также обладает *собственным участком потока* (*thread private area*) для своих локальных переменных. Потоки могут увеличить производительность отдельного приложения двумя способами:

- В многопроцессорной системе каждый поток многопоточного процесса может быть передан разным процессорам. Такая совместная работа в многопроцессорной среде улучшает производительность отдельно взятого приложения.
- Потоки можно использовать, чтобы повысить пропускную способность приложения. Всякий раз при необходимости получить файл с

[1] То есть данные между процессами не разделяются. В рамках этого обсуждения авторы не будут затрагивать тему разделяемых библиотек.

диска однопоточный веб-сервер застывал бы в состоянии ожидания. Если же сервер многопоточный, то один поток может обрабатывать новый запрос, пока другой поток ждет приема данных с диска.

Согласно последнему утверждению, вполне реально запускать многопоточные процессы на единственном процессоре. На самом деле многопоточность существовала задолго до появления многопроцессорных систем. Такая методика очень полезна для повышения производительности. Однако количество идентичных активных потоков в программе с интенсивными вычислениями не должно быть больше количества процессоров.

Блокирование

Для ровной работы многопроцессорных систем очень важно иметь механизм синхронизации. Рассмотрим пример. Пусть существуют два потока, каждый из которых инкрементирует одну и ту же переменную путем выполнения следующего ассемблерного кода:

```
LOAD      r1,Counter
ADD       r1,1
STORE     r1,Counter
```

Пусть первоначальное значение переменной `Counter` равно `021400`. Что случится, если в то мгновение, когда первый поток завершил `LOAD`, но перед тем как он выполнил `ADD` и `STORE`, операционная система переключилась на выполнение второго потока? Второй поток может выполнить все три команды, установив значение `Counter` равным `021401`. Когда операционная система возобновит выполнение первого потока, она начнет с того места, где остановилась, и выполнит команды `ADD` и `STORE`, сохраняя `021401` в `Counter` и устанавливая окончательное значение `021401`, тогда как оно должно быть `021402`. Если в системе два процессора, то ситуация именно такая, если не хуже! Процессоры не заботятся о вмешательстве операционной системы, когда два потока выполняют одни и те же команды одновременно. Вместо этого процессоры чинно выполнят последовательности load-add-store. В результате получим `021401`, тогда как значение в `Counter` должно быть `021402`.

Для того чтобы только один поток мог за раз выполнять эти команды, необходимы простые *атомарные* (неделимые) операции. Фрагменты кода, подобные приведенному, назовем *критическими участками*.

В однопроцессорных системах на помощь приходило блокирование обслуживания прерываний при прохождении критических участков. За счет этого обслуживание потока не прерывалось. Такая схема не будет работать на многопроцессорных системах, так как каждый процессор обладает своей маской прерываний. Нужно найти обходной путь.

Все процессоры SPARC поддерживают команду `LDSTUB` (load-store-unsigned-byte), которая атомарно читает из памяти байт в регистр и за-

писывает 0xFF в память. Эта команда дает возможность установки *взаимоисключающих блокировок* (*mutual exclusion locks*), или *взаимных исключений* (*mutexes*). Благодаря им только один процессор за раз может удерживать блокировку. Чтобы убрать блокировку, поток просто записывает 0x00 обратно в память. Если процессор не может выполнить блокировку, он «крутится в цикле» (*spin*), пока такая возможность не предоставится. *Спин-блокировка* полностью выполняется в кэше процессора, поэтому это не приводит к чрезмерному шинному трафику. Спин-блокировки[1] полезны, когда предполагается, что ожидание блокировки будет коротким. Если предполагается более длинное ожидание, то поток должен «заснуть», позволив процессору выполнять другое задание. В системе команд SPARC V9, применяемой в процессорах UltraSPARC, есть атомарная операция compare-and-swap, а также атомарная операция load-store.

Когда для систем класса SPARCserver 6x0 была выпущена версия SunOS 4 с поддержкой мультипроцессинга, то в ней однопроцессорный метод маски прерывания был заменен на метод, основанный на взаимных исключениях. Запрос на блокирование прерываний был заменен на взаимные исключения, а запрос об отмене запрета на прерывания – на снятие взаимных исключений. На загруженных многопроцессорных системах блокировка с помощью взаимных исключений сама по себе может стать узким местом.[2] В таком случае введение дополнительных процессоров скорее будет помехой, нежели послужит повышению производительности.

Интересно, что феномен снижения производительности при добавлении процессоров не сводится лишь к такому случаю. Серьезные конфликты взаимных исключений возможны при запуске плохо написанного приложения. Одна большая компания, ведущая операции с ценными бумагами, работала с программой, написанной на C, которая запускалась на системе Solaris. На однопроцессорной системе эта программа работала удовлетворительно. Но с добавлением второго CPU производительность приложения упала почти на 30%. Ничего удивительного: плохой алгоритм и реализация приложения могут стать (и, вероятно, станут) основной причиной низкой производительности на многопроцессорных системах.[3]

---

[1] Спин-блокировки также называют взаимными блокировками. – *Примеч. науч. ред.*

[2] Авторы никогда не сталкивались с этим на системах с менее чем четырьмя CPU (подразумевается приложение с «хорошим поведением»).

[3] Это не единственный случай, когда плохой алгоритм может быть причиной снижения производительности. Это может случиться, например, при росте нагрузки на плохо спроектированную базу данных. Что касается многопроцессорных систем, то сейчас в мире принято применять специальные приемы программирования для разработки многопроцессорных приложений, а в Интернете можно отыскать много объявлений о курсах для желающих научиться это делать (за рубежом). – *Примеч. науч. ред.*

Иногда всем потокам процесса в некой точке нужно дождаться, пока каждый из потоков не «сделает свое дело». Например, пусть речь идет о временнуй имитации смятия передней панели автомобиля во время удара о бетонную стену.[1] Для потоков нет смысла вычислять состояние автомобиля в разные моменты времени, так как потокам, выполняемым позднее, нужно знать, что же случилось раньше. Такую синхронизацию можно реализовать с помощью конструкции, называемой *барьером*. Все потоки должны достигнуть барьера (скажем, закончить обработку данных за какой-то интервал времени), прежде чем всем потокам будет позволено продолжить работу.

Влияние кэша на производительность многопроцессорной обработки

Рассмотрим еще одну особенность архитектуры многопроцессорной системы. Пусть есть два процессора, CPU 0 и CPU 1. CPU 0 выбирает некую строку кэша из основной памяти в свой кэш. Далее CPU 1 помещает ту же самую строку в свой кэш. Когда CPU 0 осуществляет запись в эту строку кэша, он обновляет свой локальный кэш и основную память. Однако должен существовать какой-то способ уведомить CPU 1, что его копия строки кэша неверна и ее необходимо отбросить. Это можно сделать с помощью взаимодействия по многопроцессорной шине. Так как подобное взаимодействие должно происходить при большинстве операций с кэшем, то возникает большой трафик.

Таким образом, увеличение кэша процессора часто влечет за собой значительное увеличение производительности многопроцессорных систем. Обращения к оперативной памяти должны идти через шину, которая в многопроцессорной системе обычно очень загружена.

Периферийные соединения

Обсудим еще одно, последнее взаимодействие. Это взаимодействие между шиной или соединителем, которые способствуют коммуникациям между процессорами и оперативной памятью (см. раздел «Взаимодействие процессоров» ранее в этой главе), и шиной, которая связана с периферийными устройствами.

Широко распространены две архитектуры шины: SBus компании Sun и *локальная шина соединения периферийных устройств* (*Peripheral Component Interconnect*, PCI). Вторая шина является стандартом в индустрии. SBus сдает свои позиции в пользу стандарта PCI, однако в силу большого количества установок SBus будет здесь рассмотрена.

---

[1] К сожалению, автор сталкивался с этим на практике. Его персональный опыт показывает, что имитация – это гораздо менее дорогостоящий способ убедиться в том, что автомобиль защитит пассажиров от телесных повреждений.

SBus

Архитектуры SBus и PCI довольно схожи: обе разработаны как шины ввода-вывода (а не для более широкого применения) и имеют маленький форм-фактор, требующий высокой степени интеграции. Кроме того, у них примерно сходные показатели производительности. Если бы PCI появилась в 1988 году, когда Sun проектировала SBus для SPARCstation 1, то SBUS, вероятно, не была бы разработана. В каждой системе Sun шина SBus реализована немного по-разному. Выбор конструкции осуществлялся исходя из критериев производительности, стоимости, легкости реализации и т. д. Однако на практике все современные шины SBus отвечают почти всем этим требованиям.

SBus – это архитектура шины с параллельной передачей (много бит передается одновременно). В оригинальной спецификации было 32 адресных линии и 32 линии данных. Спецификация Rev B.0 SBus допускает задействование адресных линий во время цикла данных, чтобы можно было передавать 64 бит. Этот алгоритм реализован в системах SPARCstation 10SX, SPARCstation 20 и во всех системах, основанных на UltraSPARC. Такая схема означает, что 64-разрядные и 32-разрядные реализации SBus обладают одним форм-фактором. Однако влияние разрядности шины на производительность часто переоценивается (хотя это и важный фактор). Очень мало устройств могут выйти за пределы пропускной способности 32-битной шины. В основном это графические видеобуферы с очень высокой производительностью и очень быстрые сетевые интерфейсы (со скоростью передачи данных более 250 Мбит/с).

Тактовая частота

В большинстве старших систем Sun частота SBus производилась из системной частоты либо напрямую, либо через делитель. В большей части систем SPARCstation 10 системная частота равна 40 МГц, а частота SBus – 20 МГц.[1] Системы SPARCstation 20 работают с системной частотой 50 МГц и частотой SBus 25 МГц. SPARCserver 1000 и SPARCcenter 2000 запускают SBus на 20 МГц вне зависимости от других факторов. В системах microSPARC и microSPARC-II частота SBus задается делением частоты процессора. Данные о частоте шины SBus приведены в табл. 3.6.

В большинстве новых систем Sun (SPARCserver 1000E, SPARCcenter 2000E и системы на базе UltraSPARC) частота шины SBus равна 25 МГц вне зависимости от любых факторов.

---

[1] Исключение – ранние модели (SPARCstation 10 модель 20 и модель 30), которые имели системную частоту 33 МГц и 36 МГц соответсвенно. При этом частота SBus была равна 16,5 МГц и 18 МГц соответственно. Большинство этих систем были обновлены путем установки более новых версий плат SPARCstation 10.

Таблица 3.6. Тактовые частоты SBus для систем microSPARC

CPU clock (МГц)	Divider	SBus clock (МГц)
50	2:1	25
70	3:1	23,3
85	4:1	21,25
110	5:1	22

Размер пакета передачи

Архитектура SBus позволяет передавать данные в пакетах. Пакет – это объем данных, которые шина может принять каждый раз, когда устройство захватывает шину. Размер пакета примерно равен максимальному размеру единицы передачи в сети. Каждый раз при инициации передачи аппаратура интерфейса шины принимает один пакет данных. Аппаратура проводит контроль шины, передает адрес назначения и пересылает за тактовый цикл такой объем данных, который позволяет разрядность шины. Ширина шины оказывает влияние на эффективность шинных транзакций. На платформе с размером пакета 16 байт передача по шине 2 Кбайт требует 128 шинных циклов арбитраж/адрес/данные по сравнению с 32 циклами при размере пакета 64 байт.

Пакеты SBus могут иметь размер 1, 2, 4, 8, 16, 32 и 64 байт. Системы UltraSPARC и более поздние, основанные на MBus настольные системы поддерживают также 128-байтный размер пакета. Однако максимальный поддерживаемый размер пакета зависит от платформы. SPARCstation 2, SPARCstation 10, SPARCstation 20 и SPARCsystem 600MP поддерживают 32-байтные пакеты. Все другие системы не-UltraSPARC поддерживают 16-байтные пакеты. Системы UltraSPARC поддерживают все размеры пакетов. Размер пакета устанавливается во время загрузки на уровне слотов. Размер выбирается меньше того максимального размера, который поддерживается ведущим узлом SBus и контроллером.

Режим передачи

Каждая реализация SBus предлагает по меньшей мере *программируемый ввод-вывод* (*programmed I/O*, PIO) и *прямой доступ к виртуальной памяти в последовательном режиме* (*consistent mode direct virtual memory access*, DVMA).[1] В PIO для передачи данных по шине процессор применяет команды load и store. Главным образом, режим PIO используется для управления регистрами контроля и статуса на платах SBus, однако некоторые платы реализуют PIO для передачи данных. Так как такой процесс требует вмешательства процессора, а раз-

[1] DVMA очень похож на более распространенный метод прямого доступа к памяти (DMA). Их отличие лишь в том, что DVMA определяет адрес транзакции памяти в виртуальном пространстве, а не физический адрес.

меры пакетов непременно малы (процессоры не-UltraSPARC допускают команды load и store не более 8 байт; процессоры UltraSPARC имеют 64-байтные блоки команд load/store), то PIO почти всегда более дорог, чем DVMA. Однако режим DVMA обладает довольно сложным процессом установки и разрыва для каждой операции ввода-вывода. Поэтому PIO более эффективно обслуживает маленькие передачи.

Некоторые реализации SBus предлагают дополнительный режим передачи, называемый *потоковым режимом DVMA* (*streaming mode*). Потоковый режим DVMA обеспечивает существенно более быстрые передачи, особенно для считываний. Однако он требует значительных модификаций драйвера для согласования работы оборудования передачи и разделяемой памяти. Большинство драйверов устройств реализуют поддержку потокового режима.

Итог по реализациям SBus

Таблица 3.7 обобщает реализации SBus в системах на базе SPARC.

Таблица 3.7. Подробности реализаций SBus

Платформа	Размер пакета (байт)	Ширина шины (бит)	Частота (МГц)	Потоковый режим	Скорость (чтения/записи) (Мбайт/с)
SPARCstation 1	16	32	20	Нет	12/20
SPARCstation 1+, IPC	16	32	25	Нет	15/25
SPARCstation 2, IPX	32	32	20	Нет	15/32
SPARCstation 10, SPARCstation 600	32	32	20	Нет	32/52
LX, Classic	16	32	25	Нет	17/27
SPARCstation 4, SPARCstation 5	16	32	21,25–25	Нет	36/55
SPARCstation 10 SX	128	32/64	20	Нет	40/95
SPARCstation 20	128	32/64	25	Нет	62/100
SPARCserver 1000, SPARCcenter 2000	64	32	20	Да	45/50
SPARCserver 1000E, SPARCcenter 2000E	64	32	25	Да	56/62
Все системы UltraSPARC	128	32/64	25	Да	90/120

Платы ввода-вывода SBus, применяемые для систем серии Ultra Enterprise, отделены от плат процессор/память. Существует два типа плат ввода-вывода SBus. Одна разработана для графических карт, тогда как другая – более общего назначения.

Универсальные платы ввода-вывода SBus имеют двухпортовый интерфейс Fibre Channel, порт Fast Ethernet и несимметричный порт Fast

Wide SCSI-2, а также три 64-разрядных слота SBus. На плате реализованы две независимые шины SBus. Одна отвечает за интерфейсы Fast Ethernet и SCSI и один из слотов SBus. Другая SBus обслуживает интерфейсы Fibre Channel и два других слота SBus.[1] В графической плате ввода/вывода один из слотов SBus заменен на слот UPA. Это означает, что в ней есть только один контроллер SBus. Оба слота SBus присоединены к такой шине SBus наряду с другими устройствами, интегрированными на плате ввода-вывода. Порт UPA полностью независим.

Применение карт SBus

Таблица 3.8 дает общее представление о картах SBus и их типичной пропускной способности. Особый интерес вызывает тот факт, что видеобуферы SBus занимают немалую часть полосы пропускания шины.

Таблица 3.8. Пропускная способность карт SBus

Описание	Типичная пропускная способность (Мбайт/с)	Тип передачи	Ширина
Fast/Wide SCSI-2	8–10	DVMA	32
25 Мбит/с Fibre Channel	16	DVMA	64
Fast/Wide SCSI-2 + Fast Ethernet (*hme*)	12	DVMA	64
155 Мбит/с ATM (Ревизия 1)	10	DVMA	32
155 Мбит/с ATM (Ревизия 2)	10	DVMA	64
655 Мбит/с ATM	50–100	DVMA	64
Fast Ethernet (Ревизия 2)[a]	4	DVMA	64
Quad Fast Ethernet	20	DVMA	64
Gigabit Ethernet	40	DVMA	64
100 Мбит/с FDDI	6–8	DVMA	32
16 Мбит/с Token Ring	1	PIO	32
Видеобуферы GX, GX+	4–12	PIO	32
Видеобуферы TurboGX, TurboGX+	8–20	PIO	32
Видеобуферы ZX	20–30	DVMA	32

[a] Карты Fast Ethernet Ревизии 1, использующие драйвер *be*, не поддерживаются в Solaris после версии 2.6. Если это возможно, их следует заменить на карты Ревизии 2, которые используют драйвер *hme*.

[1] На самом деле существует три типа универсальных карт ввода-вывода SBus. Карты первого типа (501–4287) имеют пару интерфейсов Fibre Channel 25 Мбайт/с и поддерживают частоту на панели до 83 МГц. Карты двух других типов имеют интерфейсы Fibre Channel 100 Мбайт/с, реализованные через слоты GBIC. Единственное различие между ними в том, что один (501–4266) работает на частоте 83 МГц, а частота другого (501–4883) может подняться до 100 МГц. Однако все они имеют две независимые шины SBus, которые разделены, как уже было описано.

Как правило, не стоит нагружать SBus более чем на 80 процентов. Это позволит избежать шинных конфликтов и сложностей при арбитраже, которые могут снизить производительность.

PCI

PCI, или *локальная шина соединения периферийных устройств*, является, пожалуй, самой распространенной конструкцией шины в современных компьютерных системах. Первоначально она была представлена в 1994 году в персональных компьютерах на платформе Intel и в значительной степени заменила все более старшие архитектуры периферийных шин. Сейчас она широко поддерживается и производителями рабочих станций. PCI является стандартом под патронажем PCI Special Interest Groop, или PCI SIG (*http://www.pcisig.com*).

PCI – это архитектура синхронной шины. Это означает, что все передачи данных выполняются в соответствии с системной частотой. Первоначальная спецификация PCI допускала максимальную тактовую частоту 33 МГц, однако более поздняя спецификация Ревизия 2.1 увеличила ее до 66 МГц. Большинство персональных компьютеров поддерживают только стандарт 33 МГц. Слоты 66 МГц в основном применяются в сетях с очень большими скоростями (например, дуплексный Gigabit-Ethernet сильно выигрывает от скорости слота 66 МГц). PCI поддерживает 32-разрядные уплотненные шины адреса и данных.[1] Также предусмотрена архитектурная поддержка для 64-битной шины данных через больший разъем. На рынке персональных компьютеров 64-разрядные карты с поддержкой 66 МГц не нашли широкого распространения (рис. 3.6).

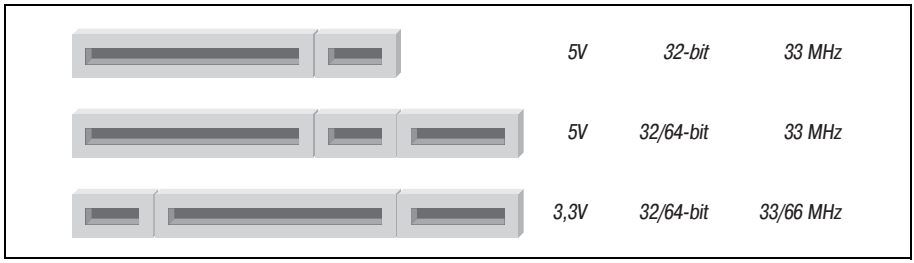


Рис. 3.6. *Типы разъемов PCI*

Вследствие электротехнических факторов высокая скорость шины PCI (до 528 Мбайт/с на 64-битном тракте данных с тактовой частотой 66 МГц) приводит к ограничению количества слотов расширения на одной шине до трех-четырех. Для обеспечения большего количества сло-

[1] Такое уплотнение позволяет снизить число выводов на разъеме PCI, что ведет к уменьшению физического размера и снижению стоимости. Типичные 32-разрядные платы PCI используют около 50 выводов.

тов группа SIG в Ревизии 2.1 PCI определила механизм *моста PCI-PCI*. Такие мосты представляют собой чипы, которые реализуют электротехническую изоляцию двух шин PCI. При этом они позволяют переадресацию шинных передач от одной шины к другой. В системах с несколькими шинами PCI многочисленные мосты могут каскадироваться.

Многоплатные системы Sun Enterprise поддерживают платы ввода-вывода PCI: две карты PCI на одну плату ввода-вывода. Каждый слот PCI расположен на отдельной шине PCI. Интерфейсы Fast Ethernet и Fast/Wide SCSI также расположены на отдельных шинах PCI.

Транзакции шины PCI

В терминологии PCI данные передаются между *инициатором* (*initiator*), или *ведущим шины* (*bus master*), и *целевым объектом* (*target*), или *ведомым шины* (*bus slave*). После того как инициатор захватывает шину, сама передача состоит из одной адресной фазы и любого количества фаз данных. Во время адресной фазы инициатор уведомляет о типе передачи (чтение из памяти, запись в память, чтение ввода-вывода, запись ввода-вывода и т. д). При передаче как инициатор, так и целевой объект могут переходить в состояние ожидания. Кроме того, инициатор или целевой объект могут в любое время завершить передачу.

Кроме того, PCI поддерживает четкий механизм автоматической конфигурации. Каждое устройство PCI включает в себя набор регистров, содержащих конфигурационные данные. Эти регистры определяют тип карты (SCSI, Ethernet, видеобуфер и т. п.), а также производителя карты, уровень прерывания и т. д.

Наконец, PCI поддерживает как 5-вольтовый, так и 3,3-вольтовый уровень передачи сигналов. Однако большинство более ранних реализаций PCI были только 5-вольтовыми и не подавали питание на 3,3-вольтовые выводы. Со временем 3,3-вольтовый интерфейс будет применяться более интенсивно. Чтобы предотвратить установку платы в слот, который не поддерживает соответствующий вольтаж, в стандарт разъема PCI введена схема «ключа».

CompactPCI

Шина CompactPCI – это одна из разновидностей PCI, разработанная для индустрии. Она предназначена для таких приложений, как сбор данных и управление в реальном масштабе времени, а также для измерительных приборов и телекоммуникаций. По сравнению со стандартным, настольным интерфейсом PCI CompactPCI поддерживает в два раза больше слотов (восемь против четырех), а также обладает улучшенным форм-фактором. В стойку плат карты CompactPCI вставляются с лицевой стороны. Такой способ в сочетании с более надежными разъемами дает CompactPCI значительное преимущество в отношении «горячей замены». Устройства CompactPCI работают так же, как и сопоставимые устройства PCI. Однако поскольку карты PCI только на-

чали выходить на рынок рабочих станций/серверов, их долгосрочные перспективы неясны.

Итог по периферийным соединениям

Периферийные соединения обобщены в табл. 3.9. Акцент сделан на оборудовании для персональных компьютеров.

Таблица 3.9. Сравнение периферийных соединений

Тип шины	Год	Ширина (бит)	Тактовая частота (МГц)	Максимальная пропускная способность (Мбайт/с)	Пакетные передачи
ISA	1984	16	8	8	Нет
MCA	1987	32	10	40	Да
EISA	1988	32	8	32	Да
Sbus	1988	32 или 64	До 25	До 200	Да
VESA	1992	32	33	132	Да
PCI	1992	32 или 64	33 или 66	От 132 до 534	Да

Прерывания в Linux

В системе Linux список всех прерываний можно получить из файла */proc/interrupts*:

```
% cat /proc/interrupts
           CPU0
  0:  216651906          XT-PIC  timer
  1:       1965          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  8:          1          XT-PIC  rtc
  9:   96370488          XT-PIC  eth0
 14:  381136978          XT-PIC  ide0
 15:   60751177          XT-PIC  ide1
NMI:          0
ERR:          0
```

Настраивать распределение прерываний в системе Linux лучше всего с помощью приложения *irqtune*. На время написания книги эта программа была доступна по адресу *http://www.best.com/~cae/irqtune/*. Она позволяет задавать прерывания, которые следует обслуживать с более высоким приоритетом. Приоритет прерываний определяет, какие прерывания обрабатывать первыми, когда одновременно происходит несколько прерываний. Вообще, приоритеты прерываний назначаются в порядке убывания IRQ, то есть системный таймер (IRQ 0) имеет приоритет над всеми другими IRQ. Утилита *irqtune* может устанавливать приоритет 0 для определенных IRQ. Для этого нужно прос-

то запустить *irqtune* и указать прерывания, которым следует отдавать предпочтение. Например, чтобы задать приоритет 0 для IRQ 9 и 14 (см. интерфейсы *eth0* и *ide0* в листинге выше), нужно выполнить:

```
# irqtune 9 14
```

Прерывания в Solaris

В системах Sun класса предприятия (Sparcserver 1000, SPARCcenter 2000 и серия Ultra Enterprise) есть много вариантов управления устройствами SBus. Начиная с Solaris 2.3 операционная система использует механизм *статического распределения прерываний*. Это означает, что каждое устройство ввода-вывода закрепляется за конкретным процессором. Каждый раз, когда устройство запрашивает прерывание, процессор, назначенный для этого устройства, обслуживает это прерывание. Таблица распределения прерываний устанавливается при загрузке, а также каждый раз, когда процессор переходит в состояние непрерывания или выходит из него (см. раздел «Управление процессорами в Solaris» далее в этой главе). В настоящее время нет способа вручную закреплять прерывания от конкретного устройства за конкретным процессором. При альтернативном подходе, известном как *распределение по круговой системе* (*round robin distribution*), каждое прерывание обслуживается «следующим» процессором. Такой режим включается с помощью установки параметра ядра `do_robin` в 1.

При распределении по круговой системе кэши процессоров используются значительно менее эффективно, так как копия кода обработки, специфичного для устройства, должна присутствовать в кэше каждого процессора. В то же время прерывания обрабатываются быстрее.

Как правило, статическая обработка прерываний означает, что высокопроизводительные платы следует распределять между максимально возможным количеством процессоров и интерфейсов SBus. Платы Quad Fast Ethernet (QFE) известны тем, что генерируют множество прерываний (до 20 000 в секунду при большой нагрузке). В SBus все слоты равнозначны. Поэтому размещение нескольких карт SBus на одной шине SBus иногда может повысить производительность, так как идентичные платы разделяют один и тот же код обработки прерываний. С другой стороны, полосу пропускания SBus можно легко перегрузить. Такое расположение повышает шансы, что код обслуживания прерывания постоянно находится в кэше процессоров. Также заметим, что поскольку прерывания назначаются статически, большая нагрузка на процессор может снизить производительность ввода-вывода устройств, закрепленных за этим процессором. Если один процессор обслуживает большое количество прерываний и регулярно загружен на 100% или около того, то перемещение плат SBus на другую шину SBus может принести существенное увеличение производительности. Более простое решение – включить планирование по круговой системе.

Один типичный подход к управлению распределением прерываний в многопроцессорных системах таков: для обслуживания всех прерываний применять малое количество процессоров. Другим процессорам следует запретить обрабатывать прерывания. Это можно сделать с помощью команды *psradm*, позволяющей перевести процессоры в режим `no-intr` (см. раздел «Управление процессорами в Solaris» далее в этой главе). Эта команда позволяет тонко сегментировать рабочую нагрузку.

Инструменты для контроля производительности процессора

Точно определить, является ли время процессора узким местом для системы, зачастую намного сложнее, чем это кажется на первый взгляд. Симптомы постоянной нехватки CPU могут свидетельствовать о многом: ошибки в дисковых конфигурациях, привязка периферийных устройств к одному процессору, управляющему прерываниями, зависание процессора при нехватке памяти и т. д. Поэтому говорить о том, что дело в процессоре, следует тогда, когда исключены все другие варианты.

Существует также философский аспект. Некоторым администраторам нравится видеть низкий средний уровень нагрузки на своих системах, когда процессор простаивает более чем на 50%. Такая система легко справится со всплесками нагрузки. Точно так же другим администраторам нравится видеть среднюю нагрузку, которая отвечает количеству процессоров. При такой нагрузке система работает на всех парах. Ради производительности процессора в ней задействованы все имеющиеся возможности. Персональный подход авторов таков: для сервера вычислений более подходит полная загрузка всех процессоров, а на исследовательской системе или сервере баз данных лучше иметь запас прочности, чтобы быть готовым к всплескам нагрузки.

Для мониторинга производительности процессора есть немало инструментов: средняя нагрузка, *vmstat*, *mpstat*, *prstat* и/или *top* и *lockstat*. Кроме того, в Solaris существует два инструмента для управления процессорами на многопроцессорных системах: *psrinfo* и *psrset*. Далее все они будут подробно рассмотрены.

Средняя нагрузка

Средняя нагрузка на систему определяется с помощью команды *uptime*:

```
% uptime
     4:02pm  up 19 day(s),  9:28,  119 users,  load average: 1.22, 1.21, 1.25
```

Средняя нагрузка представлена как среднее количество запущенных заданий (длина очереди на выполнение плюс количество текущих выполняемых заданий) за последние 1, 5 и 15 минут. Такие значения

позволяют навскидку оценить, насколько сильно нагружена система. Однако значения *uptime* не сообщают о нагрузке исключительно на процессор. В системе с нехваткой памяти показатель средней нагрузки обычно высокий.

Статистика средней нагрузки значима тогда, когда рассматривается в контексте количества инсталлированных процессоров. Общее правило таково: если средняя нагрузка меньше количества инсталлированных процессоров, значит, системе хватает мощности процессоров. Например, если средняя нагрузка на 4-процессорной системе равна 2.5, это не является признаком нехватки процессоров.

Очереди процессов

Vmstat – это очень мощный инструмент в арсенале средств настройки производительности. Постепенно он будет рассмотрен во всех подробностях. Вот простой вывод *vmstat* в системе Solaris (напомним, что первый аргумент *vmstat* – это интервал времени в секундах между срезами данных; кроме того, всегда следует отбрасывать первую строку вывода):

```
% vmstat 30
 procs     memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr s0 s1 -- --   in   sy   cs us sy id
 0 0 0   5928  1920   0 328 45 25 37 1360 2 4  8  0  0  306 1682  299 10  5 85
 0 0 0 1202648 17152  0 904  8  0  0 1200 0 7 36  0  0  547 5632  683 18 15 67
 0 0 0 1202744 19600  0 1136 0  0  0 1040 0 2 26  0  0  575 4542  752 33 12 55
```

Первая колонка выделена. Она сообщает о состоянии трех системных очередей процессов. По порядку: *очередь запуска* (*run queue*, `r`), *очередь запущенных-но-блокированных процессов* (*runnable-but-blocked queue*, `b`) и очередь *запущенных-но-свопированных процессов* (*runnable-but-swapped queue*, `w`). Очередь свопинга и все, что с ней связано, обсуждается в разделе «vmstat» главы 4.

Очередь запуска включает процессы, которые запущены, но в текущее время не выполняются. В этом примере процессор системы не перегружен работой. Общее правило таково: если очередь запуска более чем в четыре раза превышает количество процессоров в системе, то количество процессоров следует увеличить.

Очередь запущенных-но-блокированных процессов включает процессы, которые, будучи запущенными, тем не менее не выполняются вследствие ожидания передачи от блочных устройств (обычно дисковая операция ввода-вывода).

В Linux вывод *vmstat* слегка иной, однако содержит ту же информацию:

```
% vmstat 30
 procs                  memory    swap        io    system         cpu
 r b w  swpd  free  buff cache  si  so   bi   bo   in   cs  us  sy  id
```

```
 0 0 0 20068   9568 26524 50828    0    0     2     7     8     3     3     1     8
 0 0 0 20068   9568 26524 50828    0    0     0     0   115    22     8     4    89
 0 0 0 20068   9568 26524 50828    0    0     1     0   115    22     7     4    89
 0 0 0 20068   9568 26524 50828    0    0     0     0   114    20     7     4    89
```

Разбивка времени процессора

Время процессора разбито на четыре временных режима: *системное время* (*system time*), *пользовательское время* (*user time*), *время ожидания ввода-вывода* (*I/O wait time*) и *время простоя* (*idle time*).[1] Время простоя – это оставшееся время, когда процессор не пребывает в трех других режимах. Нормальное рабочее состояние программы – это пользовательский режим. Однако запущенная программа может генерировать запросы на обслуживание, которое предоставляет ядро, например ввод-вывод. Такие запросы требуют внимания операционной системы, поэтому программа переключается в системный режим, а после обработки запроса возвращается в пользовательский. Время нахождения в этих двух режимах подсчитывается независимо, чтобы получить значения пользовательского и системного времени. Большая часть времени выполнения процессов – это пользовательское и системное время.

Когда процесс ожидает завершения обработки запроса данных от блочного устройства, процессор находится в режиме ожидания ввода-вывода. Это означает следующее: когда процесс блокирован подобным образом, все время простоя становится временем ожидания. Если *vmstat* сообщает, что время простоя равно нулю, то в первую очередь следует проверить, нет ли у системы сложностей с пропускной способностью ввода-вывода.

Другое важное диагностическое средство – это сравнение системного и пользовательского времени. В серверах NFS, которые будут обсуждаться в разделе «NFS» главы 7, системное время обычно намного больше пользовательского времени, так как служба NFS полностью содержится внутри ядра.

В Solaris такие данные по процессам можно получить с помощью команды *ptime*. Вот пример:

```
% /usr/proc/bin/ptime tar -cf miscellany.tar miscellany

real        7.402
user        0.044
sys         0.287
```

Существуют другие команды, */bin/time* и *timex*, которые имеют другой формат вывода. Более подробно эти команды будут обсуждены в

[1] Заметим, что *vmstat* сообщает только пользовательское время, системное время и время ожидания (время ожидания складывается со временем простоя). Для получения отдельных значений времени ожидания и времени простоя нужно использовать *mpstat*.

разделе «Измерение времени работы приложения: time, timex и ptime» главы 8.

Большое значение системного времени в выводе *ptime* объясняется объемом запросов ввода-вывода, которые должны быть обработаны с помощью средств ядра. Получить такие сведения на уровне системы можно многими способами. *vmstat* и *mpstat* (они будут обсуждаться более подробно) предоставляют такие данные. Рассмотрим выделенные колонки в выводе *vmstat*:

```
% vmstat 30
 procs     memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr s0 s1 -- --   in   sy   cs us sy id
 0 0 0   3296  2344   0 365 44 22 33  0  1  4  8  0  0  304  700  318 11  6 84
 0 0 0 1221816 99992  0 450  0  0  0  0  0  0  2  0  0  269 1791  465 19  6 76
 0 0 0 1222568 100688 0  83  0  0  0  0  0  0  8  0  0  254  591  389  2  1 98
```

Эта система служит примером системы под большой нагрузкой. Процент времени простоя довольно высок, а пользовательское время больше системного времени.

Многопроцессорные системы

Обсужденные инструменты представляют достоверные данные и для многопроцессорных систем. Однако для получения статистики по каждому процессору в Solaris следует обратиться к более мощному инструменту *mpstat*.

Вот простой вывод *mpstat* на 2-процессорной системе:

```
% mpstat 30
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0  182   0  353   269   54  161   24   13    5    0   940   11   5   5  78
  1  182   1  531   134  116  157   23   13    6    0   931   10   6   5  79
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0 2856   228   28   72    0    6    0    0   454    2   2   4  92
  1    4   0    0    39   39   77    0    7    0    0    50    0   0   4  96
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0   80   0  515   489  288  108    2   11    1    0   275    1   6  37  56
  1   61   0   55    46   46  113    2    9    2    0   270    0   1  37  62
```

Наиболее интересные показатели:

- `intr` – это количество прерываний, обрабатываемых процессором; `ithr` – количество прерываний как потоков.
- `migr` – это количество миграций потоков. Миграция потока происходит, когда поток переключается с выполнения на одном процессоре на выполнение на другом процессоре.
- `smtx` – это количество спинов или *взаимных исключений* (в практическом исчислении эта колонка представляет количество блокировок, не полученных с первой попытки). Блокировки будут обсуждены в разделе «Статистика блокировок» далее в этой главе.

- `syscl` – это количество системных вызовов.
- `usr`, `sys`, `wt` и `idl` – процентное соотношение пользовательского, системного периодов, периодов ожидания и простоя с разбивкой по процессорам. Заметим, что *vmstat* представляет только системное среднее значение.

top и prstat

top – это общепринятый инструмент для мониторинга выполнения процессов в масштабе всей системы. Он предоставляет список процессов с интенсивным использованием CPU, которые запущены в системе. Кроме того, *top* выдает интересную статистику, такую как средняя нагрузка, количество процессов и емкость памяти и области свопинга. *top* широко распространен и очень прост в освоении, поэтому он больше не будет обсуждаться. В интересах полноты описания рассмотрим вывод *top*:

```
% top
 10:27pm  up 24 days, 21:24,  9 users,  load average: 0.05, 0.06, 0.01
125 processes: 124 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  8.0% user,  9.1% system,  0.0% nice, 82.7% idle
Mem:  255596K av, 251056K used,   4540K free,      0K shrd,   3200K buff
Swap: 112444K av,      8K used, 112436K free                  99112K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
 5752 gdm       17   0  1296 1296  1052 R       0 13.8  0.5   0:01 top
 3829 root       9   0  1252 1252  1072 S       0  1.1  0.4   0:06 sshd
 5753 root      10   0  1492 1492  1292 S       0  1.1  0.5   0:00 sendmail
    1 root       8   0   520  520   452 S       0  0.0  0.2   0:48 init
```

К сожалению, обратная сторона обновляющегося списка процессов и других славных особенностей *top* заключается в том, что он потребляет немало процессорного времени.[1] Поскольку это существенно искажает поведение системы, в состав Solaris включен сходный инструмент, называемый *prstat*. Он не предоставляет всю ту полезную информацию о потреблении памяти, которую выдает *top*. Однако *prstat* потребляет очень мало процессорного времени и не приводит к такому сильному искажению поведения системы:

```
% prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
  4585 gdm      1280K 1096K cpu0    48    0   0:00.00 0.4% prstat/1
  4554 gdm      1424K 1024K sleep   58    0   0:00.00 0.0% csh/1
   337 nobody     23M 6856K sleep   59    0   0:00.07 0.0% ns-slapd/15
  4552 root     1784K 1256K sleep   38    0   0:00.00 0.0% in.telnetd/1
...
Total: 33 processes, 143 lwps, load averages: 0.00, 0.00, 0.01
```

[1] Ходят слухи, что отсюда происходит название *top* – эта программа часто появляется вверху списка потребителей CPU.

Подведем итог. *top* очень хорош, но важно помнить, что он искажает состояние системы из-за того, что сам потребляет много ее ресурсов. В системах Solaris *prstat* предоставляет подобный вывод при значительно меньшей стоимости в показателях процессорного времени.

Статистика блокировок

Кроме сбора данных о состоянии процессоров *mpstat* можно использовать для получения важных сведений о прерываниях и взаимных исключениях (mutex). Основной показатель – это колонка `smtx`, которая представляет количество случаев за единицу времени, когда ядро не смогло немедленно получить запрашиваемое взаимное исключение. Если это количество будет в 250 и более раз превышать количество скомпонованных процессоров, то системное время станет заметно увеличиваться. Процессорам с большей производительностью, например UltraSPARC III/Pentium 3 Xeon, по силам более высокие значения `smtx`.

```
% mpstat 30
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0  182   0  353   269   54  161   24   13    5    0   940   11   5   5  78
  1  182   1  531   134  116  157   23   13    6    0   930   10   6   5  79
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0  112   0  365   259   55  141    6   12    2    0   486    6   3  10  81
  1  157   0  337    73   67  147    7   11    3    0   594    6   4   8  82
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0   97   0  598   229   26  135    6   11    2    0   389    5   3   5  87
  1  107   0   81    67   62  130    5   11    2    0   423    4   3   4  89
```

Избавиться от конфликтов с взаимными исключениями лучше всего с помощью производителя. Необходимо проверить все обновления операционной системы, а затем применить команду *lockstat*, чтобы выяснить, какие взаимные исключения не обрабатываются. Если потребление процессора постоянно очень высокое, а значения `smtx` в системе SMP велики, то вместо добавления новых процессоров следует заменить текущие на более быстрые.

Для нахождения необрабатываемых взаимных исключений *lockstat* реализует мониторинг блокировок ядра. Его может активизировать только пользователь root. При сборе данных ядро может динамически изменять свои блокировки. После завершения сбора данных ядро возвращается к нормальному, оптимизированному коду блокировки. *lockstat* может быть вызван для выдачи данных с разбивкой по процессорам. Кроме того, с помощью команды *sleep* можно получить данные для всей системы в целом. Чтобы привести хорошую иллюстрацию, авторы довольно долго искали простой пример для генерации большого количество спинов взаимных исключений и наконец нашли:[1]

[1] Данные были получены с помощью скриптов C shell с бесконечными циклами. Количество скриптов равно количеству скомпонованных в системе процессоров. Авторы надеятся, что это не то, чем занимается система читателя.

```
% mpstat 10
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    1   0    2   104    0   19    0    0    0    0    26    0   0   0 100
  1    1   0    1     8    6   19    0    0    0    0    37    0   0   0 100
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0    0   109    2   40   22    2  956    0 106954   47  53   0   0
  1    0   0    0    23    7   43   15    2 1117    0 107760   48  52   0   0
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0    0   116    0   57   26    2 1289    0 107204   47  53   0   0
  1    0   0    0    24    3   45   21    2 1099    0 107199   47  53   0   0
```

Очевидно, что эта система имеет большие сложности с взаимными исключениями. Применим *lockstat* для исследования операций в системе:

```
# lockstat sleep 10

Adaptive mutex spin: 19132 events

Count indv cuml rcnt     spin Lock             Caller
-------------------------------------------------------------------------------
 2716  14%  14% 1.00        1 flock_lock        cleanlocks+0x1c
 2388  12%  27% 1.00        3 0xe0452ab0        dev_get_dev_info+0x3d
 1905  10%  37% 1.00        3 0xe0452ab0        cdev_size+0x48
 1554   8%  45% 1.00        1 0xe07c9f30        vn_rele+0x22
 1307   7%  52% 1.00        5 stable_lock       specvp+0x8f
 1050   5%  57% 1.00        2 0xe0452ab0        mod_rele_dev_by_major+0x21
  995   5%  62% 1.00        2 0xe0452ab0        ddi_hold_installed_driver+0x36
...
-------------------------------------------------------------------------------
...
```

К сожалению, по умолчанию вывод *lockcstat* весьма подробный. Однако с первого взгляда ясно, что большую часть своего времени система работает с блокировкой *flock_lock*, относящейся к файловой системе. Исходя из этого есть смысл применить *truss*, чтобы выяснить, какие системные вызовы в приложении порождают такие конфликты.

Если есть сомнения в том, какая блокировка могла вызвать затруднения, то можно использовать способность *lockstat* искать конкретную блокировку. Воспользуемся ключом *-l lockname*:

```
# lockstat -l flock_lock sleep 10

Adaptive mutex spin: 2594 events

Count indv cuml rcnt     spin Lock                   Caller
-------------------------------------------------------------------------------
 2594 100% 100% 1.00        1 flock_lock             cleanlocks+0x1c
-------------------------------------------------------------------------------

Adaptive mutex block: 9 events

Count indv cuml rcnt     nsec Lock                   Caller
```

```
-------------------------------------------------------------------------------
    9 100% 100% 1.00 64490487 flock_lock            cleanlocks+0x1c
-------------------------------------------------------------------------------
```

Управление процессорами в Solaris

В состав Solaris 2.6 и более поздних версий входят команды, помогающие системному администратору в управлении процессорами. В большинстве приложений по настройке производительности роль этих команд невелика, однако знать о них полезно.

psrinfo

Команда *psrinfo* показывает данные о скомпонованных процессорах. Особенно полезная информация может быть получена с помощью ключа *-v*.

```
% /usr/sbin/psrinfo -v
Status of processor 0 as of: 09/20/99 16:24:34
  Processor has been on-line since 09/01/99 06:33:49.
  The sparc processor operates at 200 MHz,
        and has a sparc floating point processor.
Status of processor 1 as of: 09/20/99 16:24:34
  Processor has been on-line since 09/01/99 06:33:52.
  The sparc processor operates at 200 MHz,
        and has a sparc floating point processor.
```

Эта утилита оказывалась полезной при работе с пользователями, которые не были уверены в том, какое оборудование у них установлено, однако хотели его настроить. В Linux подобные данные можно получить, исследовав содержимое */proc/cpuinfo*.

psradm

Команда *psradm* позволяет изменить рабочее состояние процессоров. Процессор может быть в следующих состояниях:

- Процессор в состоянии онлайн (on-line) выполняет вычисления[1]. Его работа может быть прервана системными устройствами ввода-вывода.
- Процессор в состоянии оффлайн (off-line) выполняет вычисления и обычно не обслуживает прерывания. В некоторых случаях запретить прерывания для процессора в состоянии оффлайн не представляется возможным.
- Процессор в состоянии без прерываний (no-intr) выполняет вычисления, но прервать его работу нельзя.

[1] То есть on-line-процессор – в процессе обработки процессов (processor processes processes).

По крайней мере один процессор должен быть способен проводить вычисления. Кроме того, должна быть возможность прерывать работу по крайней мере одного процессора.

psrset

Порой бывает полезным создать объединение процессоров с помощью *psrset*. Объединение процессоров позволяет привязывать процесс к определенной группе процессоров, а не к одному. Иногда объединение процессоров создается системой. В этом случае такие процессоры выполняют как системные, так и пользовательские процессы. Кроме того, объединение может быть создано с помощью *psrset*. Тогда объединение может выполнять только процессы, привязанные к этим процессорам. Рассмотрим пример вывода *psrset* на системе Sun Ultra Enterprise 3500. Затем привяжем процесс к этому объединению процессоров.

В системе, использованной для примера, установлено восемь процессоров 250 МГц UltraSPARC-II:

```
# psrinfo
6       on-line   since 11/15/01 16:47:33
7       on-line   since 11/15/01 16:47:40
10      on-line   since 11/15/01 16:47:40
11      on-line   since 11/15/01 16:47:40
14      on-line   since 11/15/01 16:47:40
15      on-line   since 11/15/01 16:47:40
18      on-line   since 11/15/01 16:47:40
19      on-line   since 11/15/01 16:47:40
```

Чтобы увидеть объединения процессоров в системе, можно запустить *psrset -i* (для вывода в разрезе процессоров, а не объединений можно применить *psrset -p*). По умолчанию объединений нет – все процессоры находятся в общем пуле выполнения. С помощью *psrset -c* создадим объединение процессоров, состоящее из четырех процессоров (с номерами 14, 15, 18 и 19):

```
# psrset -i
# psrset -c 14 15 18 19
created processor set 1
processor 14: was not assigned, now 1
processor 15: was not assigned, now 1
processor 18: was not assigned, now 1
processor 19: was not assigned, now 1
# psrset -i
user processor set 1: processors 14 15 18 19
# psrset -p
processor 14: 1
processor 15: 1
processor 18: 1
processor 19: 1
```

Добавить процессоры к этому объединению можно с помощью *psrset -a set-id processor-list*. Подобным образом с помощью *psrset -r processor-list* можно удалить процессоры из объединения.

```
# psrset -i
user processor set 1: processors 14 15 18 19
# psrset -a 1 10 11
processor 10: was not assigned, now 1
processor 11: was not assigned, now 1
# psrset -i
user processor set 1: processors 14 15 18 19 10 11
# psrset -r 18 19
processor 18: was 1, now not assigned
processor 19: was 1, now not assigned
# psrset -i
user processor set 1: processors 14 15 10 11
```

Полностью удалить объединение процессоров можно с помощью *psrset -d set-id*:

```
# psrset -i
user processor set 1: processors 14 15 10 11
# psrset -d 1
removed processor set 1
# psrset -i
#
```

При конфигурировании системы смешанного применения, скажем, для разработки программного обеспечения и выполнения вычислений, объединения процессоров можно применять для сегментирования рабочей нагрузки. Например, на восьмипроцессорной системе можно создать одно объединение из двух процессоров для компилирования, одно четырехпроцессорное объединение для вычислений, а оставшиеся два процессора никуда не назначать (тем самым оставив их для интерактивной работы). Процессорам, вовлеченным в компилирование и вычисления, можно запретить обрабатывать прерывания (например, для дискового или сетевого ввода-вывода). Для этого следует запустить *psrset -f set-id*. Или же для нескольких процессоров можно разрешить обработку прерываний. В этом случае обработку прерываний для объединения можно включить с помощью *psrset -n set-id*. По умолчанию все процессоры могут обрабатывать прерывания. Например, создадим объединение из четырех процессоров и запретим ему обработку прерываний:

```
# psrinfo
6       on-line   since 11/15/01 16:47:33
7       on-line   since 11/15/01 16:47:40
10      on-line   since 11/15/01 16:47:40
11      on-line   since 11/15/01 16:47:40
14      on-line   since 11/15/01 18:26:23
15      on-line   since 11/15/01 18:26:23
```

```
18      on-line   since 11/15/01 18:26:23
19      on-line   since 11/15/01 18:26:23
# psrset -i
# psrset -c 14 15 18 19
created processor set 1
processor 14: was not assigned, now 1
processor 15: was not assigned, now 1
processor 18: was not assigned, now 1
processor 19: was not assigned, now 1
# psrset -f 1
# psrinfo
6       on-line   since 11/15/01 16:47:33
7       on-line   since 11/15/01 16:47:40
10      on-line   since 11/15/01 16:47:40
11      on-line   since 11/15/01 16:47:40
14      no-intr   since 11/15/01 18:27:09
15      no-intr   since 11/15/01 18:27:09
18      no-intr   since 11/15/01 18:27:09
19      no-intr   since 11/15/01 18:27:09
```

Заметим, что удаление объединения процессоров, для которого были запрещены прерывания, не имеет побочного эффекта возобновления обработки прерываний для членов этого объединения. Возобновить прерывания для конкретного процессора всегда можно с помощью ***psradm*** *-**n** processor-list*.

Теперь, когда создано объединение процессоров, как запускать на нем процессы? По существу, есть два пути: или применить ***psrset*** *-**e** set-id command*, чтобы напрямую запустить команду на объединении процессоров, или воспользоваться ***psrset*** *-**b** pid-list* для привязки процессов к объединению.[1] Например, запустим многопоточное приложение на объединении процессоров 1:

```
# psrset -i
user processor set 1: processors 14 15 18 19
# psrset -e 1 mtsieve &
[1] 3701
# mpstat 3
...
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  6    0   0  747   401  301    0    0    0    0    0     0    0   0   0 100
  7    0   0    0   104  104   45    0    0    0    0    27    0   0   0 100
 10    0   0    0   100  100   13    0    0    0    0     9    0   0   0 100
 11    0   0    0   100  100   11    0    0    0    0     3    0   0   0 100
 14    0   0    0   111  100   11   11    0    0    0     0  100   0   0   0
 15    0   0    0   111  100   11   11    0    0    0     0  100   0   0   0
 18    0   0    0   111  100   11   11    0    0    0     0  100   0   0   0
```

[1] На самом деле происходит привязка базовых легковесных процессов к объединению (подробнее о легковесных процессах см. раздел «Модель поточной работы Solaris» ранее в этой главе).

```
 19    0   0    0   111  100   11   11    0    0    0    0  100   0   0   0
...
```

В выводе *mpstat* легко увидеть, что процессоры в объединении процессоров всецело загружены, тогда как другие процессоры полностью простаивают.

Инструменты для контроля производительности периферийных соединений

Для контроля производительности периферийных соединений устройств существует очень мало средств. Безусловно, лучшее из них – это утилита Solaris *busstat*, которая позволяет контролировать состояние счетчиков производительности, встроенных в оборудование Sun. Так как для этого необходима высокая степень аппаратной поддержки, возможности *busstat* могут меняться со временем и от платформы к платформе. Рассматриваемые примеры собраны на Sun Ultra Enterprse 3500, запускающей Solaris 8 7/01.

Прежде всего, с помощью *busstat* можно посмотреть, какие устройства доступны для мониторинга. Для этого запустим *bussat -l*:

```
# busstat -l
Busstat Device(s):
sbus0 sbus1 ac0 ac1 ac2 ac3 ac4
```

Итак, есть два устройства `sbus`, которые соответствуют двум SBus на единственной карте ввода-вывода SBus этой системы (см. раздел «Итог по реализациям SBus» ранее в этой главе). Устройства `ac` – это адресные контроллеры, которые есть на каждой установленной в системе плате. Каждое устройство имеет два *счетчика оценки производительности* (*performance instrumentation counter*, PIC), которые могут подсчитывать определенные события. Чтобы получить список учитываемых событий для каждого устройства, можно воспользоваться *busstat -e device*:

```
# busstat -e sbus0
pic0
dvma_stream_rd
dvma_stream_wr
dvma_const_rd
dvma_const_wr
dvma_tlb_misses
dvma_stream_buf_mis
dvma_cycles
dvma_bytes_xfr
interrupts
upa_inter_nack
pio_reads
pio_writes
sbus_reruns
```

```
pio_cycles
pic1
...
```

Таблица 3.10 содержит описание наиболее важных параметров из этого списка.

Таблица 3.10. События, учитываемые счетчиком производительности SBus

Событие	Описание
dvma_stream_rd	Количество потоковых чтений DVMA
dvma_stream_wr	Количество потоковых записей DVMA
dvma_const_rd	Количество последовательных чтений DVMA
dvma_const_wr	Количество последовательных записей DVMA
dvma_tlb_misses	Количество неудачных быстрых переадресаций во время операции DVMA
dvma_stream_buf_mis	Количество неудачных чтений потоковых буферов DVMA
dvma_cycles	Количество циклов Sbus, предоставленных для операций DVMA
dvma_bytes_xfr	Количество байт, переданных через SBus
interrupts	Количество прерываний, порожденных SBus
upa_inter_nack	Количество безответных прерываний (на UPA), порожденных SBus
pio_reads	Количество программируемых чтений ввода-вывода
pio_writes	Количество программируемых записей ввода-вывода
sbus_reruns	Количество повторных запусков SBus
pio_cycles	Количество циклов Sbus, предоставленных для операций PIO

С помощью *busstat -w device,pic0=event,pic1=event* можно указать счетчики, показания которых нужно исследовать. Эти счетчики остаются программируемыми до их сброса. Программировать счетчики может только привилегированный пользователь.

Обычно это очень подробные события, которые могут не представлять большого интереса ни для кого, кроме разработчиков драйверов и аппаратных средств. Однако бывает интересно взглянуть на количество байт, переданных через SBus. Например, с помощью *busstat* можно наблюдать работу SBus с сетевым интерфейсом, присоединенным к сетевому устройству SunRay:[1]

[1] SunRay всецело отвечает за передачу команд обновления изображения на головную систему. Его действие подобно работе X terminal.

```
# busstat -w sbus1,pic0=dvma_cycles,pic1=dvma_bytes_xfr 1
time dev    event0              pic0        event1              pic1
1    sbus1  dvma_cycles         1449        dvma_bytes_xfr      1940
2    sbus1  dvma_cycles         1361        dvma_bytes_xfr      1800
...
12   sbus1  dvma_cycles         1328        dvma_bytes_xfr      1800
13   sbus1  dvma_cycles         3469        dvma_bytes_xfr      6424
```

До этой точки система просто обслуживает экран *dtlogin*. Вероятно, небольшой объем передачи данных происходит во время «мерцания» курсора. Подключимся, набрав достоверное имя пользователя и пароль:

```
14   sbus1  dvma_cycles         7282        dvma_bytes_xfr      19812
15   sbus1  dvma_cycles         7620        dvma_bytes_xfr      21172
16   sbus1  dvma_cycles         1399        dvma_bytes_xfr      2352
```

В этой точке произошло успешное подключение к системе и появляется экран CDE «Welcome to Solaris», а за ним запускаются процессы настольной среды пользователя (*twm* и несколько процессов *xterm*):

```
17   sbus1  dvma_cycles         182874      dvma_bytes_xfr      590212
18   sbus1  dvma_cycles         1361        dvma_bytes_xfr      2004
19   sbus1  dvma_cycles         247957      dvma_bytes_xfr      814420
20   sbus1  dvma_cycles         2691        dvma_bytes_xfr      3688
21   sbus1  dvma_cycles         13542       dvma_bytes_xfr      25392
22   sbus1  dvma_cycles         7931        dvma_bytes_xfr      14248
```

Через некоторое время активность SBus возвращается на близкий к нулю уровень, когда происходят лишь минимальные графические обновления вследствие работы в сети:

```
31   sbus1  dvma_cycles         225         dvma_bytes_xfr      536
32   sbus1  dvma_cycles         597         dvma_bytes_xfr      760
```

Хотя этот пример очень ограничен, он показывает, какой подробный анализ можно проводить с помощью *busstat*.

Расширенная статистика производительности процессора

В Solaris для получения подробнейшей статистики о производительности процессора можно применять инструменты *cpustat* и *cputrack*. *cpustatt* генерирует статистику в масштабе всей системы, тогда как *cputrack* позволяет собирать данные о конкретном процессе.

Список учитываемых событий можно получить с помощью *cpustat -h*. Некоторые события доступны только на конкретных счетчиках PIC. Таблица 3.11 обобщает различные события вместе и счетчики PIC, на которых эти события можно учитывать.

Таблица 3.11. События, учитываемые при оценке производительности CPU

Событие	Доступные PIC	Описание
Cycle_cnt	0, 1	Количество накопленных тактовых циклов
Instr_cnt	0, 1	Количество завершенных команд
Dispatch0_IC_Miss	0	Количество командных «промахов» кэша
Dispatch0_mispred	1	Количество непредсказанных переходов (заметим, что такая ошибка обычно приводит к аннулированию команд, «забежавших вперед», поэтому количество откатов конвейера почти в два раза больше количества переходов)
Dispatch0_storeBuf	0	Количество случаев, когда была выдана команда хранения, но буфер хранения был полон и не мог обработать дополнительные записи в память
Dispatch0_FP_use	1	Количество случаев, когда команда, зависящая от результата операции с плавающей точкой, была еще не доступна по причинам, не связанным с загрузкой (т. е. Dispatch0_FP_use и Load_use взаимоисключающие)
Load_use	0	Количество случаев, когда команда, ожидающая выполнения, требовала загрузки из памяти, которая еще не была завершена; залипание всех команд в конвейере
Load_use_RAW	1	Количество случаев, когда грядет load-use, но грядет также и read-после-write для более ранней незавершенной load. Это свидетельствует о том, что данные load ожидают завершения более ранней store
IC_ref	0	Количество обращений к кэшу команд
IC_hit	1	Количество попаданий в кэш команд
DC_rd	0	Количество обращений к кэшу данных для чтения
DC_rd_hit	1	Количество попаданий в кэш данных при чтении
DC_wr	0	Количество обращений к кэшу данных для записи
DC_wr_hit	1	Количество попаданий в кэш данных при записи

Событие	Доступные PIC	Описание
EC_ref	0	Количество обращений к E-Cache (L2 cache)
EC_hit	1	Количество попаданий в E-Cache
EC_write_hit_RDO	0	Количество попаданий при записи в E-Cache, приведших к переносу принадлежности (ownership)
EC_wb	1	Количество «промахов» E-Cache, приведших к обратной записи в оперативную память
EC_snoop_inv	0	Количество несовпадений E-Cache, обнаруженных при слежении за шиной
EC_snoop_cb	1	Количество обратных копирований E-Cache при слежении за шиной
EC_rd_hit	0	Количество попаданий при чтении из E-Cache после промаха в кэше даннах
EC_ic_hit	1	Количество попаданий при чтении из E-Cache после промаха в кэше команд

Снова отметим, что часть этих счетчиков относится к анализу производительности на очень низком уровне. Главным образом, их применяют люди, очень хорошо знакомые с архитектурой процессора. Более подробную информацию об архитектуре семейства процессоров UltraSPARC можно найти в руководстве пользователя, опубликованном Sun Microelectronics по адресу *http://www.sun.com/microelectronics/manuals/index.html*.

Часто бывает интересно определить количество циклов на одну команду. Для этой цели авторы позаимствовали исходный код (см. раздел «Компиляция с поддержкой профилирования»[1] главы 8) приложения «решето Эратосфена» (Sieve of Eratosthenes) и незначительно его изменили, чтобы не отображать выходные данные (а просто их вычислять). Цель эксперимента – сравнить требуемое количество циклов на команду в зависимости от уровня оптимизации компилятора (Forte 6 Update 1). С помощью *cputrack* соберем данные для каждого процесса:

```
% cputrack -c pic0=Cycle_cnt,pic1=Instr_cnt sieve.o0
   time lwp      event      pic0      pic1
  1.046   1       tick 170075841 128955109
  2.056   1       tick 221955530 162910824
```

[1] Профилирование (profiling) – процедура, позволяющая выяснить, сколько раз в процессе выполнения программы была выполнена каждая инструкция программы. Теоретически, можно подсчитать это значение как для строки исходного текста, так и для каждой инструкции процессора. – *Примеч. науч. ред.*

```
  3.056   1       tick 240994274  93274214
  4.056   1       tick 231824966  52868531
  5.056   1       tick 198733670  53220280
  6.056   1       tick  92304003  51971543
  6.522   1       exit 1192528514 564103611
```

Далее, используя последнюю строку (итоговую статистику), вычислим среднее количество циклов на команду. Для этого разделим значение `pic0` (количество циклов) на значение `pic1` (количество команд). Получим CPI, равное 2,114. Повышая оптимизацию, можно получить следующие результаты:

Ключи компилятора	Всего команд	Количество циклов на команду	Полное время прогона (с)
`-x00`	564 103 381	2,114	6,507
`-x02`	168 642 699	5,022	5,480
`-x03`	137 109 359	5,929	5,275
`-fast -x05`	138 521 627	5,684	5,213

На такие результаты производительности опасно опираться. В то же время интересно наблюдать огромный скачок CPI при переходе от одной оптимизации к другой. Очевидно, что дальнейшая оптимизация приведет к снижению результатов.

При исследовании производительности можно применить *cputrack* для получения данных о времени прогона приложений. Например, можно вычислить показатель попаданий в кэш команд при работе неоптимизированного приложения:

```
%  ptime cputrack -c pic0=IC_ref,pic1=IC_hit sieve.o0
   time lwp      event      pic0      pic1
...
  6.475   1       exit 179350516 179325008
```

Исходя из полученных данных можно вычислить, что показатель попаданий составляет 99,986%. Это невероятно высокий результат. Можно также определить показатели попадания в кэш данных при чтении и при записи:

```
% cputrack -c pic0=DC_rd,pic1=DC_rd_hit sieve.o0
   time lwp      event      pic0      pic1
...
  6.469   1       exit 128490011 128460760

real        6.509
user        0.038
sys         0.063
% cputrack -c pic0=DC_wr,pic1=DC_wr_hit sieve.o0
   time lwp      event      pic0      pic1
```

```
...
  6.471   1       exit  69216839  33654774
```

Показатель попаданий в кэш данных при чтении составляет 99,977%, а при записи 48,622%. Становится интересно. Анализируется неоптимизированное приложение, и показатели попаданий в кэш данных при записи выглядят немного низкими. Было бы интересно выяснить, повысятся ли они с повышением уровня оптимизации. Проведенный эксперимент свидетельствует, что в высокооптимизированной версии (с наименьшим временем прогона) показатель попаданий в кэш данных при записи составляет только 0,031%. Сначала это сбивает с толку, но после размышлений становится ясно: на пике своей производительности приложение *не* делает частых записей в один и тот же участок памяти. Это хорошая иллюстрация того, как счетчики низкого уровня могут сбить с толку, если мало знаешь об анализируемом приложении.

Заключение

Микропроцессор – это одно из наиболее сложных устройств в расчете на квадратный сантиметр, которое когда-либо разработало человечество. Понять его работу очень притягательно и непросто. Авторы надеются, что из этой главы читатель вынес глубокие сведения о процессорах во всей их сложности, а также подготовился к оценке производительности процессоров в динамической системе.

Список обсужденных тем большой: от основ конструкции микропроцессора и архитектур шин и коммутационных соединителей до кэширования и периферийных соединений. Есть много чего, что следует усвоить. Но вознаграждение от понимания того, как работают компоненты и как они взаимодействуют друг с другом, соизмеримо велико.

4

Память

- *Реализации физической памяти*
- *Архитектура виртуальной памяти*
- *Пейджинг и свопинг*
- *Потребители памяти*
- *Инструменты для измерения производительности памяти*
- *Заключение*

Вот если бы емкость памяти была бесконечной, а любое... слово было бы немедленно доступно...

A. W. Burks, H. H. Goldstine и J. von Neumann
Предварительное обсуждение логической схемы электронного вычислительного прибора, 1946

Физическая память – это совокупность интегральных схем, предназначенных для хранения двоичных данных. Такая память имеет два отличительных свойства. Она *изменяемая* (*transient*), ибо вся хранимая информация исчезает после выключения электропитания. И это память *с произвольной выборкой* (*randomly accessible*), то есть доступ к какому-либо биту так же быстр, как к любому другому. Кроме того, во многих системах реализована виртуальная память, призванная управлять физической памятью и предоставлять разработчикам приложений простой интерфейс. Потребители виртуальной памяти: ядро системы, кэши файловой системы, тесно разделяемая память (*intimately shared memory*)[1] и процессы.

Производительность памяти оказывает влияние на производительность всей системы в двух случаях. Первый случай – когда система не может достаточно быстро извлечь и сохранить данные в физической памяти, или когда системе приходится часто обращаться к оперативной памяти. Такие затруднения можно преодолеть настройкой соответ-

[1] Тесно разделяемая память – это специфичная для Solaris структура, представляющая собой область разделяемой памяти, которую нельзя выгружать на диск. Широко используется в таких СУБД, как Oracle, Sybase, Informix. – *Примеч. науч. ред.*

ствующего алгоритма либо приобрести систему с более быстрым доступом к оперативной памяти. Второй и наиболее вероятный случай – когда объем физической памяти, запрашиваемый всеми одновременно запущенными приложениями, включая ядро, превышает доступный объем. Системе приходится прибегать к *пейджингу* (*paging*), или записи неиспользуемых участков памяти на диск. Если ситуация с нехваткой памяти усугубляется, то на диск записываются все участки памяти, потребляемые процессом (или процессами). Такая операция называется *свопингом* (*swapping*).[1]

В зависимости от емкости памяти возможны четыре варианта поведения системы:

- Памяти достаточно, система работает оптимально.
- Памяти «впритык» (вероятным виновником, особенно на старших моделях Solaris, может быть кэш файловой системы). Система начинает чистить участки памяти, которые не используются активно. Это сказывается на производительности.
- Системе определенно не хватает памяти. Производительность снижается, особенно для интерактивных процессов.
- Скудность памяти критическая. Начинается свопинг. Производительность системы резко падает, а интерактивная производительность просто ужасна.

В этой главе будет описана физическая реализация памяти, а также механизмы, с помощью которых система управляет памятью. Кроме того, будет рассмотрено действие пейджинга и свопинга на уровне системы. Будут представлены инструменты, которые можно использовать для анализа потребления памяти, а также будет объяснено, как работать с пространством для свопинга. Наконец, будут обсуждены механизмы, имеющие отношение к производительности памяти.

Реализации физической памяти

Начнем с рассмотрения того, как в современных системах память реализована физически. Все современные быстрые реализации памяти выполнены на основе полупроводников,[2] которые могут быть двух основных типов: *динамическая память с произвольной выборкой* (*dynamic random access memory*, DRAM) и *статическая память с произвольной выборкой* (*static random access memory*, SRAM). Различие между

[1] Зачастую «пейджинг» и «свопинг» применяются как равнозначнче термины. Однако на практике это совершенно разные механизмы. Поэтому мы не будем их взаимозаменять.

[2] В некоторых случаях, скажем, когда необходима устойчивость при ионизирующем излучении, память на магнитных сердечниках по-прежнему применяется.

ними сводится к конструкции ячейки памяти. Действие динамических ячеек основано на заряде. Каждый бит представлен зарядом, хранимым в крошечном конденсаторе. Заряд истекает за короткий период времени, поэтому такая память должна все время регенерироваться, чтобы избежать потери данных. Акт чтения бита также способствует разрядке конденсатора, поэтому снова прочесть этот бит можно будет только после его регенерации. Статические ячейки, в свою очередь, основаны на логических элементах. Каждый бит хранится в четырех или шести связанных транзисторах. Память SRAM хранит данные, пока есть электропитание. Обновление не требуется. По существу, память DRAM значительно дешевле и предлагает самую высокую плотность ячеек на чип. Она занимает меньше места, потребляет мало энергии и выделяет мало тепла. Однако SRAM на порядок быстрее и потому применяется в высокопроизводительных вычислительных средах. Интересно, что оперативная память суперкомпьютера Cray-1S полностью состояла из SRAM. Тепло, выделяемое подсистемой памяти, стало основной причиной введения в систему жидкостного охлаждения.

Существуют две основные характеристики памяти. Первая представляет собой время, требуемое для чтения или записи определенного участка памяти. Это *время доступа к памяти* (*memory access time*). Вторая описывает, как часто можно повторно обращаться к памяти. Это *время цикла памяти* (*memory cicle time*). Обе характеристики звучат сходно, но часто они достаточно различны в силу таких явлений, как, скажем, необходимость регенерации ячеек DRAM.

Скорость памяти и скорость микропроцессоров несопоставимы. В начале 1980-х время доступа к типичным DRAM составляло около 200 нс. Это было меньше тактовой частоты тогдашнего микропроцессора 4,77 МГц (210 нс). Спустя всего два десятилетия тактовый цикл среднего домашнего микропроцессора упал до наносекунды (1 ГГц), а время доступа к памяти замерло на отметке около 50 нс.

Для улучшения системной производительности было разработано много различных модулей памяти. Вот некоторые из них:

- Самая старшая из ныне используемых – память *режима быстрых страниц* (*fast-page mode*, FPM). Она реализует возможность чтения целой страницы (4 Кбайт или 8 Кбайт) данных за один цикл доступа к памяти.
- Усовершенствование этой технологии вылилось в создание памяти с *расширенным выводом данных* (*extended data output*, EDO). В значительной степени улучшения основаны на электротехнических модификациях.
- Более революционный подход был реализован в *синхронной* (*synchronous*) памяти DRAM (SDRAM), применяющей таймер для синхронизации входа и выхода сигналов. Его частота согласована с частотой CPU, поэтому распределение времени для всех компонентов синхронизировано. Кроме того, в каждом модуле SDRAM реали-

зовано два банка памяти. По сути, это удваивает ее пропускную способность. SDRAM позволяет одновременные множественные запросы к памяти. Разновидность SDRAM, называемая SDRAM *с режимом двойных данных* (*double-data rate*, DDR SDRAM), способна читать данные как на переднем, так и на заднем фронте тактового импульса. Это удваивает скорость передачи данных чипа памяти.

- Модули SDRAM обычно обозначаются PC66, PC100 или PC133. Маркировка соответствует тактовой частоте, на которой они работают: 66 МГц (15 нс), 100 МГц (10 нс) или 133 МГц (8 нс) соответственно. Однако модули DDR SDRAM обычно именуются согласно их максимальной пропускной способности. Например, теоретически модуль PC2100 обладает максимальной пропускной способностью 2,1 Гбайт/с.
- Память *прямого доступа* (Direct Rambus, RDRAM) предлагает совершенно иной подход. В ней реализован узкий (шириной 16 бит), но чрезвычайно быстрый (600–800 МГц) путь к памяти, а также сложная конвейерная обработка операций. Память RDRAM широко распространена в высокопроизводительных встроенных системах (например, в пультах PlayStation 2 Sony и Nintendo64). Модули RDRAM обычно называются PC600, PC700 или PC800, что соответствует их тактовой частоте.

Рабочие станции и серверы стремятся *чередовать* (interleave) память по банкам. Такая концепция подобна чередованию данных на дисках (см. раздел «RAID 0: разбивка на блоки (striping)» в главе 6). Если 128 Мбайт памяти скомпонованы из четырех модулей по 32 Мбайт, то при хранении биты[1] чередуются по модулям вместо того, чтобы хранить первые 32 Мбайт на первом модуле, вторые 32 Мбайт на втором модуле и т. д. За счет этого предотвращаются задержки циклов памяти и существенно повышается производительность. Такое чередование почти всегда выполнено кратно степени двойки.[2]

Это может сбить с толку. Скажем, есть система с семью заполненными банками памяти. Контроллер памяти вполне может решить создать четырех-, двух- и однократное чередование. Это означает, что произво-

[1] Чередуются, конечно, не биты, а ячейки памяти (автор использует термин bits, возможно, чтобы подчеркнуть, что это некий неделимый элемент памяти). Обычно в современных компьютерах минимальной адресуемой ячейкой памяти является байт. Так что при чередовании банков памяти первый байт хранится в первом модуле памяти, второй – во втором и т. д. Двукратное чередование – это когда все четные байты хранит один модуль, все нечетные – другой. Если модулей четыре, то чередуются между собой байты первого и второго модулей и независимо от них – третьего и четвертого. Четырехкратное – когда чередуются байты во всех четырех модулях. – *Примеч. науч. ред.*

[2] Одно исключение – серия Sun Ultra Enterprise с шестиуровневыми чередованиями.

дительность процесса с интенсивным обращением к памяти зависит от его местонахождения в памяти. Лучше всего на вопрос о том, как реализовать оптимальное чередование системной памяти, ответит производитель оборудования.

Архитектура виртуальной памяти

Система виртуальной памяти предоставляет системе средства для управления памятью от имени различных процессов. Система виртуальной памяти дает два преимущества. Разработчики программного обеспечения могут опираться на простую модель памяти. Программист абстрагирован от аппаратной реализации подсистемы памяти. При этом объем используемой памяти может быть значительно больше емкости физической памяти. Кроме того, за счет виртуальной памяти процессы могут иметь нефрагментированное адресное пространство – вне зависимости от того, как физическая память организована или фрагментирована. Для работы такой схемы требуется реализовать четыре ключевых механизма.

Во-первых, за каждым процессом закрепляется отдельное *виртуальное адресное пространство* (*virtual address space*). То есть он может «видеть» и потенциально использовать некий диапазон памяти. Этот диапазон памяти определяется максимальной длиной адреса машины. Например, процесс, запущенный на 32-разрядной системе, будет иметь виртуальное адресное пространство около 4 Гбайт (2^{32}). Система виртуальной памяти отвечает за соотнесение пользовательского участка виртуального адресного пространства и физической памяти.

Во-вторых, несколько процессов могут существенно разделять свои адресные пространства. Пусть запущены два экземпляра командного интерпретатора */bin/csh*. Оба экземпляра имеют отдельные виртуальные адресные пространства. В каждом виртуальном пространстве находится сам запущенный экземпляр, копия библиотеки *libc* и копии других разделяемых ресурсов. Система виртуальной памяти прозрачно отображает эти разделяемые сегменты в одну и ту же область физической памяти. Таким образом, в физической памяти хранится только один экземпляр разделяемого ресурса. Это аналогично созданию в файловой системе жесткой ссылки вместо дублирования файла.

Однако иногда для хранения задействованных участков всех виртуальных адресных пространств физической памяти не хватает. В таком случае система виртуальной памяти выбирает наименее используемые участки памяти и вытесняет их на вспомогательное запоминающее устройство (например, диск). За счет этого оптимизируется использование физической памяти.

Наконец, еще одна функция системы виртуальной памяти подобна одной из ролей учителя начальной школы, который не разрешает своим подопечным перемешивать личные вещи друг друга. Аппаратные

средства диспетчера памяти не позволяют процессу обращаться к памяти вне своего адресного пространства.

Страницы

Подобно энергии в квантовой механике, память квантована. Другими словами, она сформирована из неделимых элементов. Такие элементы называются *страницами* (*pages*). Точный размер страницы варьируется от системы к системе и зависит от реализации *диспетчера памяти процессора* (*memory management unit*, MMU). При больших размерах страниц активность MMU снижается за счет уменьшения количества ошибок из-за отсутствия страниц. Кроме того, экономится память ядра.[1] Однако большие страницы излишне расходуют память, поскольку система имеет дело только со страничными сегментами памяти. В ответ на запрос менее одной страницы памяти будет выделена целая страница. Обычно размер страниц равен 4 Кбайт или 8 Кбайт. В системах Solaris размер страницы можно определить с помощью */usr/bin/pagesize* или `getpagesize(3C)`. В Linux он определен в заголовочном файле ядра *asm/param.h* (параметр `EXEC_PAGESIZE`).

Размеры страниц в различных архитектурах

Обычно встречаются только три конструкции микропроцессора, реализующие страницы размером 8 Кбайт: DEC Alpha, первые процессоры Sun SPARC (например, Ross RT601/Cypress CY7C601/Texas Instruments TMS390C601A, которые были применены в SPARCstation 2) и модели Sun UltraSPARC. Процессоры Intel 80x86, процессоры MIPS, работающие в системах SGI, Motorola/IBM PowerPC и процессоры Sun серий microSPARC и SuperSPARC используют страницы размером 4 Кбайт.

Сегменты

Страницы процесса сгруппированы в несколько сегментов. Каждый процесс имеет по меньшей мере четыре сегмента:

Выполнимый текст

Состоит из рабочих исполняемых команд в двоичном коде. Команды отображены с двоичного образа на диске и доступны только для чтения.

Данные выполнения

Содержит инициализированные переменные, представленные в запущенной программе. Они отображены с двоичного образа на дис-

[1] За счет меньшего размера таблицы распределения ресурсов. – *Примеч. науч. ред.*

ке, однако имеют права чтения/записи/приватности (read/write/private; при приватном отображении изменения этих переменных в ходе выполнения не будут перенесены в дисковый файл или в другие процессы, разделяющие ту же выполняемую программу).

Область динамической памяти («куча»)

Состоит из памяти, выделенной с помощью *malloc(3)*. Такая память называется анонимной, поскольку она не имеет отображения в файловой системе (анонимная память будет обсуждена в разделе «Анонимная память» далее в этой главе).

Стек

Также выделяется из анонимной памяти.

Оценивание необходимой памяти

Порой это сделать просто. Нужно лишь запустить утилиту из коммерческого программного пакета, которая точно скажет, сколько памяти необходимо для оптимальной производительности. К сожалению, на практике так славно бывает нечасто: в типичной вычислительной среде системный администратор имеет сотни различных процессов, о которых следует помнить. Не стоит браться за каждую возможную проблему нехватки памяти. Лучше разработать общий алгоритм, позволяющий определить, сколько памяти необходимо для данной системы.

Самое важное – выяснить, какова реальная рабочая нагрузка на систему. Скажем, есть система, которая обслуживает простых пользователей: они запускают командный интерпретатор, редактор, почтовую программу (наверное, *pine* или *elm*) или, например, программу чтения конференций. Если внимательно контролировать вычислительную среду, то можно определить, сколько процессов и какого вида запускаются в пиковые периоды работы. Когда это сделать трудно или работа начата с нуля, то следует сделать разумное предположение. Пусть в пиковые часы подключаются 50 пользователей, а их действия таковы:

- По 5 вызовов *ksh* и *csh* и 40 вызовов *tcsh*
- 25 процессов *pine* и 10 процессов *elm*
- Другие приложения, не поддающиейся классификации

Вычислим требования к памяти процессов *csh*. Если в качестве аргумента *pmap* задать ID такого процесса, то можно получить такой вывод:

```
% pmap -x 7522
7522:   -csh
Address   Kbytes Resident Shared Private Permissions       Mapped File
0803C000      48       48      -      48 read/write/exec    [ stack ]
08048000     116      116    116       - read/exec         csh
08065000      16       16      -      16 read/write/exec   csh
08069000      72       72      -      72 read/write/exec    [ heap ]
DFF19000     548      444    408      36 read/exec         libc.so.1
DFFA2000      28       28      4      24 read/write/exec   libc.so.1
```

```
DFFA9000       4       4       -       4 read/write/exec    [ anon ]
DFFAB000       4       4       4       - read/exec
libmapmalloc.so.1
DFFAC000       8       8       -       8 read/write/exec
libmapmalloc.so.1
DFFAF000     148     136     136       - read/exec         libcurses.so.1
DFFD4000      28      28       -      28 read/write/exec   libcurses.so.1
DFFDB000      12       -       -       - read/write/exec    [ anon ]
DFFDF000       4       4       -       4 read/write/exec    [ anon ]
DFFE1000       4       4       4       - read/exec         libdl.so.1
DFFE3000     100     100     100       - read/exec         ld.so.1
DFFFC000      12      12       -      12 read/write/exec   ld.so.1
--------  ------  ------  ------  ------
total Kb    1152    1024     772     252
```

Видно, что каждый процесс потребляет около 1150 Кбайт памяти, причем 1024 Кбайт резидентны в памяти. Из них 772 Кбайт разделены с другими процессами и 252 Кбайт являются собственными. Значит, для пяти вызовов потребление памяти составит примерно 2 Мбайт (совместно используется 770 Кбайт плюс пять раз по 250 Кбайт). Это приближенные вычисления, но они прекрасно показывают, сколько памяти необходимо.

Однако следует помнить, что помимо всего прочего память потребляется кэшем файловой системы, тесно разделяемой памятью и ядром. Если в системе не запускается Oracle или другое приложение баз данных, то, возможно, нет нужды беспокоиться о тесно разделяемой памяти. Приближенный подсчет показывает, что следует предусмотреть около 32 Мбайт для ядра[1] и других приложений плюс еще 16 Мбайт при запуске оконной системы. Если пользователи обращаются только к нескольким сотням мегабайт данных, но делают это часто, то памяти должно быть достаточно для кэширования всего набора данных. Тогда производительность ввода-вывода таких данных радикально улучшится.

Размещение адресного пространства

Реализация адресного пространства процессов варьируется в зависимости от архитектуры системы. По существу, в системах Solaris возможны четыре варианта размещения адресного пространства:

- 32-разрядное комбинированное адресное пространство ядра и процессов SPARC V7 (применяется в архитектурах `sun4c`, `sun4m` и `sun4d`)
- 32-разрядное разделенное адресное пространство ядра и процессов SPARC V9 (применяется в машинах `sun4u`)

[1] Это верно не для всех систем UNIX. Например, в Linux и FreeBSD ядру достаточно от 2 до 8 Мбайт оперативной памяти, в зависимости от настроек ядра и конкретной аппаратной конфигурации компьютера. – *Примеч. науч. ред.*

- 64-разрядное разделенное адресное пространство ядра и процессов SPARC V9 (применяется в машинах sun4u)
- 32-разрядное комбинированное адресное пространство ядра и процессов Intel IA-32 (применяется в Solaris на системах Intel)

Комбинированная 32-разрядная модель SPARC V7 отображает адресное пространство ядра в вершину адресного пространства процессов. Это означает, что виртуальное адресное пространство процессов ограничено адресным пространством ядра (256 Мбайт в архитектурах sun4c и sun4m и 512 Мбайт в архитектуре sun4d). Адресное пространство ядра защищено от пользовательских процессов с помощью уровней привилегий процессора. Стек начинается ниже ядра с адреса 0xEFFFC000 (0xDFFFE000 на системах sun4d) и продолжается вниз до адреса 0xEF7EA000 (0xDF7F9000 на sun4d). С этого адреса в память загружаются библиотеки. Рабочие программы и их данные загружаются на дно адресного пространства, а «куча» начинается с верхней границы этих данных и растет к библиотекам.

В архитектуре микропроцессора UltraSPARC ядро может иметь собственное адресное пространство, что устраняет ограничение размеров адресного пространства ядра.[1] Так, для 32-разрядной модели памяти sun4u стек начинается с адреса 0xFFBEC000 и тянется вниз до 0xAA3BC000 (небольшое адресное пространство на самой вершине зарезервировано для OpenBoot PROM). Во всем остальном эта модель сходна с моделями SPARC V7.

Хотя процессор UltraSPARC поддерживает 64-разрядный режим SPARC V9, в реализациях процессоров Ultra SPARC-I и UltraSPARC-II возможны только 44-разрядные физические адреса памяти. Это создает «дыру» в середине виртуального адресного пространства, приходящуюся на пространство с адреса 0xFFFFF7FF.FFFFFFFF до 0x00000800.00000000.

В архитектуре Intel, как и в модели sun4c/sun4m SPARC V7, пользовательское пространство не отделено от пространства ядра. Однако есть существенное различие: стек отображен ниже сегментов рабочих программ (начиная с адреса 0x804800) и может расти вниз до самого дна адресного пространства.

Свободный список

Свободный список – это механизм, с помощью которого система динамически управляет распределением памяти между процессами. Процессы берут память из свободного списка и возвращают ее обратно после завершения процесса. Кроме того, память возвращается сканером страниц. Благодаря ему в системе постоянно есть небольшой объем свободной памяти. Каждый раз, когда запрашивается память, проис-

[1] Это было существенной проблемой в больших системах-предшественницах UltraSPARC, таких как SPARCcenter 2000E.

ходит *страничная ошибка* (*page fault*). В этом разделе будут подробно обсуждены три вида страничных ошибок:

Легкая страничная ошибка (minor page fault)

Происходит каждый раз, когда процессу нужна память или когда процесс пытается получить доступ к странице, которая была изъята сканером страниц, но не использована повторно.

Значительная страничная ошибка (major page fault)

Происходит, когда процесс пытается получить доступ к странице, которая изъята сканером страниц, использована повторно и в текущий момент занята другим процессом. Значительной страничной ошибке всегда предшествует легкая страничная ошибка.

Ошибка копирования при записи (copy-on-write fault)

Вызывается процессом, пытающимся осуществлять запись в страницу памяти, которая используется совместно с другими процессами.

Рассмотрим, как реализовано управление свободным списком в системе Solaris.

При загрузке системы вся память распределяется постранично. Кроме того, создается структура данных ядра, в которой хранятся состояния страниц. Несколько мегабайт памяти ядро резервирует для себя, а оставшееся пространство отходит свободному списку. В какой-то момент, когда процесс запрашивает память, происходит легкая страничная ошибка. Далее из свободного списка извлекается страница, которая обнуляется и поступает в распоряжение процесса. Такая схема, при которой память выдается по принципу «когда нужно», по традиции называется *пейджингом по запросу* (*demand paging*).

Страницы всегда извлекаются с вершины свободного списка.

Когда свободный список сокращается до определенного размера (параметр `lotsfree` задает количество свободных страниц), ядро «пробуждает» сканер страниц (другое название – *демон страниц*). Сканер начинает искать страницы, которые можно изъять для того, чтобы наполнить свободный список. Чтобы избежать изъятия страниц, к которым часто обращаются, сканер страниц работает по двухшаговому алгоритму. Просматривая память в порядке физической памяти, демон страниц очищает бит ссылки MMU для каждой страницы. Этот бит устанавливается, когда идет обращение к странице. Далее сканер страниц делает небольшую паузу, ожидая доступа к страницам и установки их ссылочных битов. Такая задержка зависит от двух параметров:

- `slowscan` – первоначальная частота сканирования. При увеличении этого значения сканер страниц выполняет меньше ненужных заданий, но делает больше работы.

- `fastscan` – частота сканирования, когда свободный список пуст.

Далее демон страниц снова просматривает память. Если ссылочный бит какой-то страницы по-прежнему в исходном состоянии, то значит, к этой странице не обращались. Такая страница изымается для последующего использования. Память можно сравнить с круговым рельсовым путем. По нему двигается поезд. Когда локомотив впереди паровоза проходит шпалу, ссылочный бит этой шпалы очищается. Когда шпалу пересекает служебный вагон, то анализируется ее ссылочный бит. Если бит по-прежнему не установлен, эта страница изымается.

Некоторые страницы не поддаются изъятию. Скажем, страницы, принадлежащие ядру, или страницы, которые совместно используются более чем восемью процессами (за счет этого разделяемые библиотеки хранятся в памяти). Если изъятая страница не содержит кэшированных данных файловой системы, она перемещается в очередь на откачку страниц. Здесь она ожидает ввода-вывода и наконец переписывается в пространство свопинга с кластером других страниц. Если страница используется для кэширования файловой системы, она не записывается на диск. В любом случае эта страница помещается в конец свободного списка, но не очищается. Ядро помнит, что эта страница по-прежнему хранит значимые данные.

Если средний размер свободного списка в течение 30-секундного интервала меньше `desfree`, ядро начинает принимать отчаянные меры для освобождения памяти. Неактивные процессы свопируются, а страницы не собираются в кластеры, а переписываются. Если в течение 5 секунд средняя величина свободной памяти меньше `minfree`, начинается свопинг активных процессов. Когда размер свободного списка поднимется выше `lotsfree`, сканер страниц прекращает поиск и изъятие страниц.

Работой сканера страниц управляет параметр `maxpgio`. Этот параметр ограничивает частоту, с которой происходит ввод-вывод на внешние устройства (диск) при пейджинге. По умолчанию его значение мало (40 страниц в секунду в архитектурах `sun4c`, `sun4m` и `sun4u` и 60 страниц в секунду на `sun4d`). За счет этого предотвращается насыщение вводом-выводом устройств, на которые выгружаются страницы. Для современных систем с быстрыми дисками такая частота недостаточна. Он должен быть в сто раз больше количества дисков,[1] задействованных

[1] Имеются в виду диски как физические устройства, а не разделы диска; Sun рекомендует (*http://www.trw1.btinternet.co.uk/work/#6.1*) несколько иной алгоритм, который приводит к похожим значениям maxpgio: maxpgio = (DISKRPS x 2/3 x NDISKS), где DISKRPS – это число оборотов дисков в секунду, а NDISKS – число таких дисков. Например, если в системе установлены два жестких диска со скоростью вращения 7200 RPM (revolutions per minute – оборотов в минуту), то DISKRPS = (7200/60) = 120, NDISKS =2, maxpgio = (7200/60) × (2/3) × 2 = 160. – *Примеч. науч. ред.*

для свопинга. Если процесс пытается обратиться к странице, помеченной на последующее использование, то происходит еще одна легкая страничная ошибка. Существуют две возможности:

- Страница, которая была помечена на последующее использование, возвращена в свободный список, но еще не была использована. Ядро снова отдает эту страницу процессу.
- Страница была использована, и сейчас с ней работает другой процесс. Происходит значительная страничная ошибка. Откачанные данные считываются в новую страницу, взятую из свободного списка.

Страницы могут разделяться между многими процессами, поэтому для каждого процесса нужно сохранять изменения в странице. Если процесс пытается записывать данные в разделяемую страницу, то происходит ошибка *копирования при записи* (*copy-on-write fault*).[1] В этом случае из свободного списка извлекается страница, и для этого процесса создается копия первоначальной разделяемой страницы. Когда процесс завершается, все его неразделяемые страницы возвращаются в свободный список.

Управление виртуальной памятью в Linux

Рассмотренный алгоритм применяется в большинстве современных систем управления памятью, однако часто он незначительно изменен. Здесь будет обсуждаться ядро Linux 2.2, поскольку на время написания этой книги ядро 2.4 все еще подвергалось существенным исправлениям в области подсистемы виртуальной памяти. Ядро Linux реализует другой набор параметров ядра, поэтому настройка производительности памяти в Linux слегка отличается.

Когда в системе Linux размер свободного списка опускается ниже значения `freepages.high`, система начинает неспешно откачивать страницы. Когда доступная память падает ниже значения `freepages.low`, пейджинг становится интенсивным. А если размер свободного списка опускается ниже `freepages.min`, то только ядро может назначать память. Настраиваемые параметры `freepages` находятся в файле */proc/sys/vm/freepages* (формат `freepages.min freepages.low freepages.high`).

В Linux демон страниц называется *kswapd*. Он очищает столько страниц, сколько необходимо для возврата размера системного свободного списка на отметку выше `freepage.high`. Работой *kswapd* управляют три параметра: `tries_base`, `tries_min` и `swap_cluster`. В таком порядке они расположены в файле */proc/sys/vm/kswapd*. Самый важный параметр – это `swap_cluster`. Он задает количество страниц, которое *kswapd* переписывает за раз. Такое значение должно быть достаточно большим, чтобы *kswapd* производил ввод-вывод большими участками, но в то же

[1] Так называемый COW-сбой; не путать с жвачным животным, падающим в пропасть («cow» – корова. – *Примеч. перев.*).

время и достаточно маленьким, чтобы не перегружать очередь запросов дискового ввода-вывода. Если возникает ситуация с нехваткой памяти и интенсивным пейджингом, то следует поэкспериментировать с увеличением этого значения, чтобы добиться большей пропускной способности страничного пространства.

Когда в Linux происходит страничная ошибка, то для того чтобы избежать множественных коротких обращений к диску, считывается сразу несколько страниц. Точное количество страниц, которые помещаются в память, равно 2^n, где n – значение параметра `page-cluster` (единственное значение в файле настройки */proc/sys/vm/page-cluster*). Устанавливать это значение выше 5 нецелесообразно.

Есть два важных аспекта, касающихся жизненного цикла страниц. Первый – это метод, который применяет сканер страниц, и параметры, определяющие его работу. С помощью управления пейджингом сканер страниц управляет размером свободного списка. Воздействие пейджинга на производительность может быть огромным. Второй аспект – это способ, с помощью которого назначаются и высвобождаются страницы. Легкие страничные ошибки происходят, когда процесс запрашивает страницу памяти. Значительные страничные ошибки случаются, когда запрашиваемая страница не содержит тех данных, которые ожидает процесс. Таблица 4.1 обобщает сведения о параметрах ядра, относящихся к работе системы виртуальной памяти.

Таблица 4.1. Сводка параметров ядра, относящихся к работе памяти

Операционная система	Параметр	Действие
Solaris	`lotsfree`	Целевой размер свободного списка, в страницах. По умолчанию устанавливается равным 1/64 физической памяти, с минимальным размером 512 Кбайт (64 или 128 страниц для страниц размером 8 Кбайт или 4 Кбайт соответственно). Эквивалент в Linux – `freemem/high`.
	`desfree`	Минимальное количество страниц свободной памяти, взятое за интервал 30 секунд, перед началом свопинга. По умолчанию устанавливается равным половине `lotsfree`. Примерный эквивалент в Linux – `freemem/low`.
	`minfree`	Порог минимальной свободной памяти, взятой за интервал 5 секунд, перед началом активного свопинга процессов. Представлена в страницах. По умолчанию устанавливается равным половине `desfree`. Эквивалентного параметра в Linux нет.

Операционная система	Параметр	Действие
	`slowscan`	Самая маленькая частота сканирования (например, в самом начале сканирования) в страницах. По умолчанию устанавливается равным 1/64 размера физической памяти с минимумом 512 Мбайт. То есть 64 страницы на 8 Кбайт размера страницы системы или 128 страниц на 4 Кбайт размера страницы системы. Эквивалента в Linux нет.
	`fastscan`	Самая большая частота сканирования (например, когда свободный список полностью пуст) в страницах. По умолчанию устанавливается равным половине физической памяти с максимумом 64 Мбайт. То есть 8 192 страницы на 8 Кбайт размера страницы системы или 16 384 страницы на 4 Кбайт размера страницы системы. Эквивалента в Linux нет.
	`maxpgio`	Максимальное количество операций ввода-вывода в секунду, которые могут быть поставлены в очередь сканером страниц. По умолчанию устанавливается равным 40 на всех системах, кроме построенных на архитектуре `sun4d`, где значение по умолчанию равно 60. Эквивалента в Linux нет.
Linux (2.2.x)	`freepages.high`	Целевой размер свободного списка в страницах. Когда объем памяти падает ниже этого уровня, система начинает мягко просматривать страницы. Значение по умолчанию – 786. Эквивалент в Solaris – `lotsfree`.
	`freepages.low`	Порог для активного повторного использования памяти в страницах. Значение по умолчанию – 512. Примерный эквивалент в Solaris – `minfree`.
	`freepages.min`	Минимальное количество страниц в свободном списке. Когда достигнуто это значение, только ядро может назначать больше памяти. Значение по умолчанию – 256. Эквивалента в Solaris – нет.
	`page-cluster`	Количество страниц, считываемых в случае страничной ошибки, задаваемое $2^{\text{page-cluster}}$. Значение по умолчанию – 4. Эквивалентного параметра в Solaris нет.

Окрашивание страниц

Организация страниц в кэшах процессора может оказывать разительное влияние на производительность приложений. Оптимальное размещение страниц зависит от потребления памяти приложениями. Одни приложения обращаются к памяти едва ли не произвольно, тогда как другие – в строгом последовательном порядке. Поэтому не существует единого алгоритма, который повсюду давал бы оптимальные результаты.

На самом деле свободный список представлен в виде группы приемников, раскрашенных определенным образом. Количество приемников равно частному от деления размера физического кэша второго уровня на размер страницы (например, система с кэшем второго уровня 64 Кбайт и размером страницы 8 Кбайт будет иметь 8 приемников). Когда страница возвращается в свободный список, ей назначается приемник. Когда запрашивается страница, она берется из определенного окрашенного приемника. Выбор производится на основе виртуального адреса запрашиваемой страницы. (Напомним, что страницы существуют в физической памяти и физически адресуемы, однако процессы понимают только виртуальные адреса. Процесс вызывает страничную ошибку с виртуальным адресом в своем адресном пространстве. После обработки этой ошибки процесс получает страницу с физическим адресом.) Приводимые здесь алгоритмы определяют, как назначаются цвета страницам. В Solaris 7 и в более поздних версиях существуют три алгоритма окрашивания страниц:

- Алгоритм по умолчанию (номер 0) использует алгоритм хеширования виртуального адреса. Цель такого подхода – распределять страницы как можно более равномерно.
- Первый необязательный алгоритм (номер 1) назначает страницам цвета так, чтобы физические адреса напрямую отображались в виртуальные адреса.
- Второй необязательный алгоритм (номер 2) назначает приемники для страниц по круговой системе.

Еще один алгоритм (номер 6), который называется «лучший приемник Кесслера» (Kessler's Best Bin), поддерживался в системах Ultra Enterprise 10000, работавших под управлением Solaris 2.5.1 и 2.6. Он применял исторически сложившийся механизм, назначающий страницы наименее использованным приемникам.

Алгоритм пейджинга, выбранный по умолчанию, таким выбором обязан своей хорошей производительности. Настройка этих параметров при типичной коммерческой рабочей нагрузке вряд ли принесет увеличение производительности. Однако производительность научных приложений может значительно улучшиться. Поэтому необходимо тестирование, которое выявит наиболее эффективный алгоритм исходя из потребления памяти конкретным приложением. Изменить алгоритм можно с помощью присвоения системному параметру `consistent_coloring` значения, равного номеру выбранного алгоритма.

Буферы быстрой переадресации (TLB)

При обсуждении взаимосвязи физических и виртуальных адресов важно упомянуть *буферы быстрой переадресации* (*translation lookaside buffers*, TLB). Одна из обязанностей аппаратного диспетчера памяти – преобразование виртуального адреса, предоставленного операционной системой, в физический адрес. Для этого диспетчер просматривает записи в таблице преобразования страниц. Самые последние преобразования кэшируются в буферах быстрой переадресации. В процессорах Intel и старших процессорах SPARC (предшественниках UltraSPARC) для заполнения TLB применяются аппаратные средства, а в архитектуре UltraSPARC – программные алгоритмы. Следует знать о том, что собой представляют эти буферы и каковы их функции. Однако детальное обсуждение влияния буферов TLB на производительность выходит за рамки этой книги.

Пейджинг и свопинг

Пейджинг и свопинг – это термины, которые зачастую используются попеременно, однако они обозначают совершенно различные механизмы. При *пейджинге* из памяти на диск записываются избранные, нечасто используемые страницы, тогда как при *свопинге* происходит запись целых процессов. Представим человека в автомобиле, в котором совсем мало места для инструментов. Пейджинг эквивалентен перемещению 8-миллиметровой отвертки в ящик с инструментами для того, чтобы иметь достаточно места для плоскогубцев; свопинг подобен перемещению полного комплекта отверток.

Многие администраторы считают, что их системы не должны выполнять неочевидный пейджинг (скажем, в системе, предшествующей Solaris 8, – если только это не связано с работой файловой системы). Важно понимать, что пейджинг и свопинг позволяют системе работать даже в неблагоприятных условиях нехватки памяти. Пейджинг не обязательно является признаком неполадок. С помощью перемещения неактивных страниц на диск сканер страниц увеличивает размер свободного списка. Как правило, 80% времени процесс выполняет 20% своего кода. Так как всему процессу не нужно находиться в памяти одновременно, то перезапись части страниц на диск не оказывает существенного влияния на производительность. Лишь когда нехватка памяти продолжается или нарастает, то производительность начинает падать.

Закат и упадок интерактивной производительности

Исторически в системах UNIX воплощался механизм свопинга, основанный на времени. В соответствии с ним процесс, бездействующий более 20 секунд, выгружался на диск. Теперь такого не происходит. Ныне свопинг используется лишь при самых серьезных нехватках па-

мяти. Если *vmstat* сообщает о ненулевой очереди свопинга, то можно вынести единственное заключение: ранее в какое-то неопределенное время в системе была сильная нехватка памяти, поэтому процесс был выгружен на диск.

Из-за самой сущности механизма восстановления памяти ее нехватка порой кажется серьезней, чем есть на самом деле. Когда система интенсивно свопирует задания, она пытается избежать сильного снижения производительности за счет хранения активных заданий в памяти как можно дольше. Рассмотрим программы, которые напрямую взаимодействуют с пользователями (командные интерпретаторы, редакторы или любые другие, зависящие от данных, вводимых пользователем). К сожалению, пока пользователь не наберет что-либо, эти программы неактивны. В результате такие интерактивные процессы, вероятно, будут выбраны для восстановления памяти. Например, если в ходе набора команды возникла минутная пауза (скажем, при просмотре таблицы процессов и поиске ID процесса, пожирающего всю память, – чтобы избавиться от него), то командный интерпретатор проходит долгий обратный путь с диска в память, прежде чем набранные знаки будут отображены. Хуже того, дисковая подсистема, вероятно, находится под сильной нагрузкой вследствие всех операций пейджинга и свопинга! Вывод таков: при нехватке памяти интерактивная производительность сильно падает.

Если нехватка памяти продолжается, то ситуация ухудшается еще больше. Когда из-за пейджинга и свопинга диски становятся перегруженными, а средняя нагрузка растет по спирали, то работа системы начинает замедляться. Ограничения памяти быстро переходят в ограничения ввода-вывода.

Пространство свопинга

Пространство свопинга (точнее, *пространство пейджинга*, так как почти всегда это пейджинг, а не свопинг) – это тема, порой сбивающая с толку. Пространство свопинга выполняет три функции:

- место для записи страниц собственной (private) памяти (*память для пейджинга*, *paging store*)
- место для хранения анонимной памяти
- место для хранения аварийных дампов[1]

Пространство свопинга назначается как из свободной физической памяти, так и из пространства свопинга на диске, будь то выделенный раздел или файл свопинга.

[1] По умолчанию в Solaris 7 самый крайний участок первого раздела свопинга используется для хранения аварийных дампов ядра.

Анонимная память

Пространство свопинга для анонимной памяти используется в два этапа. Резервирование анонимной памяти происходит из пространства свопинга на диске, но назначения производятся из пространства свопинга физической памяти. Когда запрашивается анонимная память (скажем, через системный вызов `malloc(3C)`), то резервирование выполняется в пространстве свопинга, а отображение – в */dev/zero*. Пространство свопинга на диске используется, пока оно есть. В ином случае будет задействована физическая память. Пространство памяти, которое отображено, но никогда не использовалось, остается в резерве. Это типичный режим работы в больших системах баз данных. Вот почему такие приложения, как Oracle, требуют большого объема дискового пространства свопинга, хотя, скорее всего, они не распределят все зарезервированное пространство.

При первом доступе к зарезервированным страницам физические страницы извлекаются из свободного списка, обнуляются и назначаются, а не резервируются. Если сканер страниц изымает страницу анонимной памяти, то данные записываются в дисковое пространство свопинга (то есть их размещение переносится из памяти на диск), а память становится доступной для повторного использования.

Размер пространства свопинга

Существует немало практических рекомендаций в отношении величины пространства свопинга в системе: от четырехкратной емкости физической памяти до половины этой емкости, и ни одна из этих рекомендаций не является особо хорошей.

В Solaris с помощью */usr/sbin/swap -s* можно получить данные об использовании свопинга в контексте времени. Вот пример рабочей станции, запускающей Solaris 7, с физической памятью 128 Мбайт и с разделом свопинга 384 Мбайт:

```
% swap -s
total: 12000k bytes allocated + 3512k reserved = 15512k used, 468904k available
```

К сожалению, названия полей вводят в заблуждение – если вообще являются правильными. Вот что имеется в виду на самом деле:

```
total: 12000k bytes allocated + 3512k unallocated = 15512k reserved, 468904k
available
```

Когда объем доступного пространства свопинга станет равным нулю, система больше не сможет использовать память (до того как часть объема не высвободится). Полученные данные свидетельствуют, что в системе есть достаточный объем пространства свопинга.

Для того чтобы определить объем задействованного пространства свопинга, можно запустить */usr/sbin/swap -l*. Вот пример с той же системы:

```
% swap -l
swapfile             dev  swaplo blocks   free
/dev/dsk/c1t1d0s1   32,129     16 788384 775136
```

Итак, в системе есть выделенный раздел для свопинга, состоящий из 788 384 блоков размером 512 байт (всего около 384 Мбайт). На момент измерения практически все пространство свопинга было свободно.

Для мониторинга пространства свопинга в контексте времени можно применить */usr/bin/sar -r interval*, где *interval* задается в секундах:

```
% sar -r 3600
SunOS fermat 5.7 Generic sun4u    06/01/99

00:00:00 freemem freeswap
00:00:00     679   935313
01:00:00     680   937184
02:00:00     680   937184
```

sar сообщает данные о свободной памяти в страницах и свободном пространстве свопинга в дисковых блоках (по сути, объем доступного пространства свопинга).

В системах Linux подобную информацию можно получить из файла */proc/meminfo*:

```
% cat /proc/meminfo | head -3 | grep -vi mem
        total:    used:    free:  shared: buffers:  cached:
Swap: 74657792 18825216 55832576
```

По существу, если потребляется более половины пространства свопинга, то его следует расширить. Диски очень дешевы. Поэтому экономия на пространстве свопинга способна лишь повредить системе.

Организация пространства свопинга

Для того чтобы по возможности минимизировать влияние пейджинга,[1] области свопинга следует располагать на самых быстрых дисках. Область свопинга должна располагаться в разделе с маленьким номером (обычно это первый раздел, а файловая система root находится в нулевом разделе). Причина такой разбивки будет объяснена в главе 5. С точки зрения производительности нет никаких оснований помещать области свопинга на медленные диски, или быстрые диски, доступные через медленные контроллеры, или быстрые диски, уже перегруженные вводом-выводом. Также не следует располагать на диске более од-

[1] Речь идет о влиянии на быстродействие. Вы помните, как называется наша книга? – *Примеч. науч. ред.*

ной области свопинга. Лучшая стратегия такова: выделенная область свопинга размещается на нескольких быстрых дисках с быстрыми контроллерами. Если есть возможность, то область свопинга лучше разместить на быстром, незначительно задействованном диске. Если такой возможности нет, то файл свопинга следует разместить на самом нагруженном разделе. За счет этого минимизируется время поиска на диске. Более детальное обсуждение представлено в разделе «Минимизация времени поиска на уровне файловой системы» главы 5.

В Solaris в качестве областей свопинга можно применять файлы, расположенные на удаленных компьютерах (через NFS). В современных системах это не лучший способ с точки зрения производительности. Однако для бездисковой рабочей станции в отсутствие дискового пространства свопинга другого выбора нет. Оценка уровня активности ввода-вывода по разделам и по дискам – это тема, подробно освещаемая в разделе «Инструменты для анализа» главы 5. Самое важное при размещении областей свопинга – это располагать их в разделах с маленькими номерами на быстрых, незначительно загруженных дисках.

Файлы свопинга

Иногда область свопинга необходимо создавать при возникновении «непредвиденных» обстоятельств. В таком случае в системе Solaris лучше всего создать *файл свопинга* (*swapfile*). Файл свопинга – это файл на диске, который система воспринимает как пространство свопинга. Прежде всего, необходимо создать пустой файл с помощью */usr/sbin/mkfile*, а затем применить */usr/bin/swap -a* (аргумент задает файл свопинга). Вот пример:

```
# mkfile 64m /swapfile
# /usr/bin/swap -a /swapfile
# /usr/bin/swap -l
swapfile             dev  swaplo blocks   free
/dev/dsk/c0t0d0s0   32,0      16 262944 232816
/swapfile              -      16  65520  65520
```

В *mkfile* размер файла свопинга можно также указывать в блоках или килобайтах, помечая единицы как `b` и `k` соответственно. Удалить файл из пространства свопинга можно с помощью */usr/bin/swap -d swapfile*. Заметим, что файл свопинга не активизируется автоматически при загрузке системы.

Потребители памяти

В системе память потребляют ядро, кэши файловой системы, процессы и тесно разделяемая память. При запуске часть объема памяти (обычно менее 4 Мбайт) система оставляет для себя. Далее, при динамической загрузке модулей системе нужна дополнительная память, поэтому она запрашивает страницы из свободного списка. Эти страни-

цы блокируются в физической памяти и не могут быть изъяты, за исключением случаев острой нехватки памяти. Иногда, когда в системе случается сильная нехватка памяти, можно услышать особый звук из динамика. Это произошло выключение динамика вследствие выгрузки драйвера звукового устройства из ядра. Однако если процесс использует какое-либо устройство, то его модуль не будет выгружен. Иначе был бы изъят и дисковый драйвер, что вызвало бы затруднения. Тем не менее, изредка в системе происходит *ошибка распределения памяти ядра* (*kernel memory allocation error*). Несмотря на то что существует ограничение на величину памяти ядра[1], такое может произойти при попытке ядра получить память, когда свободный список пуст. Так как ядро не может ждать доступности памяти, то операции скорее не будут выполнены, чем будут отложены. Одна из подсистем, которая не может ждать память, – это подсистема потоковых средств. Если значительное количество пользователей одновременно пытается подключиться к системе, то часть из них может не войти в систему. Начиная с Solaris 2.5.1 в больших системах свободный список расширен для того, чтобы он никогда не становился пуст.

У процессов есть собственная память для хранения пространства стека, кучи и областей данных. Единственный способ определить, сколько памяти процесс активно использует, – это запустить */usr/proc/bin/pmap -x process-id*. Такая возможность есть в Solaris 2.6 и более поздних версиях.

Тесно разделяемая память – это средство, позволяющее разделять низкоуровневые данные ядра о страницах памяти, а не разделять сами страницы. Это существенная оптимизация, так как она устраняет большую часть избыточной информации об отображении. Такой подход непосредственно применяется в приложениях баз данных (таких как Oracle), которые выигрывают от наличия очень большого разделяемого кэша памяти. Отметим три момента, связанные с тесно разделяемой памятью. Во-первых, вся эта память блокирована и не может быть изъята. Во-вторых, структуры управления памятью, которые обычно создаются отдельно для каждого процесса, создаются только один раз и разделяются между всеми процессами. В-третьих, ядро пытается найти большие участки непрерывной физической памяти (4 Мбайт), которые могут использоваться как большие страницы. За счет этого издержки MMU значительно снижаются.

Кэширование в файловой системе

Обычно самый большой потребитель памяти – это механизм кэширования файловой системы. Для того чтобы процесс мог читать файл и записывать в него, файл должен быть помещен в буфер памяти. При этом соответствующие страницы памяти блокируются. После завер-

[1] Это значение обычно очень велико. В системах Solaris на UltraSPARC – около 3,75 Гбайт.

шения операции они разблокируются и помещаются на дно свободного списка. Ядро запоминает страницы, хранящие значимые кэшированные данные. Если данные нужны снова, то они легко доступны в памяти. Это предотвращает дорогостоящее обращение к диску. Когда файл удален или усечен или ядро решает остановить кэширование конкретного индексного дескриптора, то все страницы, кэширующие такие данные, помещаются в верхнюю часть свободного списка для последующего использования. Однако большинство файлов становятся некэшированными только под воздействием сканера страниц. Данные, модифицированные в кэшах памяти, периодически записываются на диск с помощью *fsflush* в Solaris и *bdflush* в Linux, которые будут обсуждены немного позднее.

В Solaris объем пространства, используемый для таких операций, *не* настраивается. Если в памяти необходимо кэшировать большой объем данных файловой системы, то просто нужно приобрести систему с большей емкостью физической памяти. Более того, так как все операции ввода-вывода файловой системы Solaris обслуживает с помощью пейджинга, то значительное количество подкачек (page-in) и откачек (page-out) – это нормально. В ядре Linux 2.2 такое кэширование можно настраивать: для буферизации файловой системы доступен только определенный объем памяти. Параметр `min_percent` задает минимальную емкость системной памяти, доступной для кэширования. Верхний порог не настраивается. Этот параметр можно найти в файле */proc/sys/vm/buffermem*. Формат этого файла: `min_percent max_percent borrow_percent`. Заметим, что `max_percent` и `borrow_percent` не используются.

Записи из кэша файловой системы: fsflush и bdflush

Конечно, кэширование файлов в памяти – это мощное средство повышения производительности. Зачастую кэширование позволяет обращаться к основной памяти (за несколько сотен наносекунд) и избежать повторяющихся обращений к диску (десятки миллисекунд). Так как с содержимым файла в памяти можно работать через кэш файловой системы, то для обеспечения надежности данных важно регулярно записывать измененные данные на диск. Старшие операционные системы UNIX, подобные SunOS 4, записывали измененное содержимое памяти на диск каждые 30 секунд. Solaris и Linux реализуют механизм распределения таких операций с помощью процессов *fsflush* и *bdflush* соответственно.

Такой механизм может оказывать существенное влияние на производительность системы. Кроме того, он объясняет некоторую странную дисковую статистику.

Solaris: fsflush

Максимальное время нахождения модифицированной страницы в памяти задается параметром `autoup`. Его значение по умолчанию равно

30 секундам. При необходимости его можно увеличить до нескольких сотен секунд. Через каждые tune_t_fsflushr секунд (по умолчанию каждые 5 секунд) «пробуждается» *fsflush*, который проверяет долю всей памяти, равную частному деления tune_t_flushr на autoup (то есть по умолчанию 5/30 или 1/6 всей физической памяти системы). Затем он сбрасывает из кэша индексных дескрипторов на диск все найденные модифицированные данные. Такие операции можно запретить, если параметру doiflush присвоить значение нуль. Механизм перемещения страниц можно полностью отключить с помощью присвоения нулевого значения параметру dopageflush, однако это может серьезно отразиться на надежности данных в случае аварии. Заметим, что dopageflush и doiflush – это комплиментарные параметры, а не взаимоисключающие.

Linux: bdflush

В Linux реализован слегка иной механизм, который настраивается через значения в файле */proc/sys/vm/bdflush*. К сожалению, настройка демона *bdflush* существенно изменилась при переходе от ядра 2.2 к ядру 2.4. Рассмотрим каждое из них.

Начнем с Linux 2.2. Если процентное содержание «грязного» буферного кэша файловой системы (то есть доля измененных данных, которые нужно сбросить на диск) превышает bdflush.nfract, то пробуждается *bdflush*. Если значение этого параметра высокое, то перезапись кэша откладывается на какое-то время. Однако в дальнейшем при перезаписи будет выполнен большой объем дискового ввода-вывода. Меньшее значение распределяет дисковую активность более ровно. *Bdflush* перепишет количество буферов, равное bdflush.ndirty. Высокое значение вызовет единичный, пакетный ввод-вывод, а маленькое значение способно привести к нехватке памяти, так как *bdflush* не будет «пробуждаться» достаточно часто. Система будет ждать bdflush.age_bufer или bdflush.age_super (в сотых секунды) перед тем, как записать на диск «грязный» блок данных или метаданных файловой системы. Вот простой скрипт Perl для вывода значений конфигурационного файла *bdflush* в удобном формате:

```
#!/usr/bin/perl
my ($nfract, $ndirty, $nrefill, $nref_dirt, $unused, $age_buffer, $age_super,
$unused, $unused) = split (/\ s+/, `cat /proc/sys/vm/bdflush`, 9);
print "Current settings of bdflush kernel variables:\n";
print "nfract\t\t$nfract\tndirty\t\t$ndirty\tnrefill\t\t$nrefill\n\r";
print "nref_dirt\t$nref_dirt\tage_buffer\t$age_buffer\tage_super\t$age_super\n\r";
```

Чуть ли не единственное, что осталось неизмененным в Linux 2.4, – это то, что *bdflush* по-прежнему «пробуждается» тогда, когда процентное содержание кэша «грязного» буфера файловой системы превышает bdflush.nfract. По умолчанию значение bdflash.nfract (первое в файле) равно 30%. Диапазон – от 0 до 100%. Минимальный интервал

между «пробуждениями» и сбрасываниями на диск определяется параметром `bdflush.interval` (пятый в файле), который выражается в тактах системных часов.[1] По умолчанию он составляет 5 секунд. Минимум равен 0, а максимум – 600. Параметр `bdflush.age_buffer` (шестой в файле) задает максимальное время (в тактах системных часов), в течение которого ядро ждет перед сбрасыванием «грязного» буфера на диск. Значение по умолчанию составляет 30 секунд, минимум равен 1 секунде, а максимум – 6000 секунд. Последний параметр, `bdflush.nfract_sync` (седьмой в файле), задает долю буферного кэша (в процентах), которая должна быть «загрязнена» перед тем, как *bdflush* станет работать синхронно. Другими словами, это жесткий лимит, после которого *bdflush* будет усиленно записывать буферы на диск. Значение по умолчанию равно 60%. Вот скрипт для извлечения значений параметров *bdflush* в Linux 2.4:

```
#!/usr/bin/perl
my ($nfract, $unused, $unused, $unused, $interval, $age_buffer, $nfract_sync, $u
nused, $unused) = split (/\s+/, `cat /proc/sys/vm/bdflush`, 9);
print "Current settings of bdflush kernel variables:\n";
print "nfract $nfract\tinterval $interval\tage_buffer
$age_buffer\tnfract_sync $
nfract_sync\n";
```

Если в системе емкость физической памяти велика, то при «пробуждении» у *fsflush* и *bdflush* (демоны сбрасывания данных, *flushing daemons*) будет много работы. Однако большинство файлов, которые демону сбрасывания следовало бы перезаписать, могут быть закрыты еще до того, как они будут помечены для такой операции. Более того, перезаписи через NFS всегда выполняются синхронно, поэтому демон сбрасывания не требуется. В случае когда система выполняет много ввода-вывода, но не использует прямой ввод-вывод или синхронные перезаписи, производительность демонов сбрасывания становится важной. Для Solaris общее правило таково: если *fsflush* потребил более пяти процентов совокупного процессорного времени (без простоев), то значение `autoup` должно быть увеличено.

Взаимодействия между кэшем файловой системы и памятью

Так как Solaris имеет ненастраиваемый механизм кэширования файловой системы, то в отдельных случаях это может привести к затруднениям. Они связаны с тем, что ядро позволяет кэшу файловой системы расти до уровня, на котором происходит изъятие страниц памяти у пользовательских приложений. Такое поведение не только обкрадывает других потенциальных потребителей памяти, но и означает, что произво-

[1] Обычно происходит 100 тактов системных часов в секунду.

дительность файловой системы напрямую зависит от того, насколько быстро подсистема виртуальной памяти может освободить память.

На помощь могут прийти два механизма: приоритетный пейджинг и цикличный кэш.

Приоритетный пейджинг

Для преодоления означенных трудностей в Solaris 7 введен новый алгоритм пейджинга, названный *приоритетным пейджингом*,[1] который ставит границы в кэше файловой системы.[2] Создан новый параметр ядра, `cachefree`, который действует вместе с `minfree`, `desfree` и `lotsfree`. Система старается хранить `cachefree` доступных страниц памяти, но очищает страницы кэша файловой системы только тогда, когда размер свободного списка находится между `cachefree` и `lotsfree`.

Эффект обычно замечательный. Настольные системы и среды обработки транзакций в режиме онлайн (OLTP) начинают откликаться быстрее, а значительная часть операций свопинга устраняется. Производительность вычислительных программ, которые много записывают на диск,[3] может увеличиться на 300%. По умолчанию приоритетный пейджинг был выключен, чтобы получить отзывы конечных пользователей о его производительности. Вероятно, он станет новым алгоритмом в будущих версиях Solaris. Чтобы применить приоритетный пейджинг, необходимо иметь Solaris 7 либо 2.6 с обновлением ядра 105181-09. Для включения алгоритма следует присвоить параметру `priority_paging` значение 1. На активной 32-разрядной системе это также можно реализовать за счет присвоения параметру `cachefree` значения, равного двойному значению `lotsfree`.

Цикличное кэширование

Более элегантное техническое решение было реализовано в Solaris 8, по существу, благодаря усилиям Ричарда Мак-Дугалла (Richard McDougall), старшего инженера Sun Microsystems. Особых процедур для включения цикличного кэширования не требуется. Сердцевина этого механизма – простое правило: «незагрязненные» страницы, которые нигде не отображены, должны быть в свободном списке. Это правило означает, что теперь свободный список содержит все страницы кэша файловой системы. Такое решение имеет серьезные последствия:

---

[1] Говоря проще, до Solaris 7 было так: система виртуальной памяти при нехватке памяти сначала выгружала на диск страницы процессов, а затем – страницы файлового кэша. Теперь она поступает наоборот: прежде всего старается сохранить страницы процессов, выгружая в первую очередь страницы файлового кэша. – *Примеч. науч. ред.*

[2] Впоследствии алгоритм был портирован «назад» в Solaris 2.6.

[3] Например, программы моделирования объектов или программы расчета прогноза погоды. – *Примеч. науч. ред.*

- Запуск приложений (или другое значительное потребление памяти в короткий период времени) может происходить намного быстрее, так как сканеру страниц не требуется пробуждаться и очищать память.
- Ввод-вывод файловой системы оказывает очень незначительное влияние на другие приложения в системе.
- Операции пейджинга сводятся к нулю, а сканер страниц бездействует, когда памяти достаточно.

В результате в системе Solaris 8 анализ нехватки памяти прост: если сканер страниц *вообще* восстанавливает страницы, то налицо нехватка памяти. Умеренная активность сканера страниц означает, что памяти явно недостаточно.

Взаимодействия между кэшем файловой системы и диском

Когда *fsflush* сбрасывает данные из памяти на диск, то Solaris старается собирать модифицированные страницы, которые смежны друг с другом на диске. Эти страницы могут быть переписаны одним непрерывным участком. Такой схемой управляет параметр ядра `maxphys`. Значение `maxphys` должно быть довольно большим (1 048 576 – хороший выбор; это наибольшее значение, имеющее смысл в современных файловых системах UFS). Как будет обсуждено в разделе «Рецепты RAID» главы 6, значения `maxphys`, равного 1 048 576, с размером чередования 64 Кбайт, достаточно, чтобы 16-дисковый массив RAID 0 с почти максимальной скоростью работал с единичным файлом.

Есть другой случай, когда память и диск взаимодействуют для достижения субоптимальной производительности. Если приложения постоянно записывают и перезаписывают файлы, которые кэшированы в памяти, то кэш файловой системы, находящийся «в памяти», очень эффективен. К сожалению, процесс сбрасывания данных файловой системы на диск не всегда полезен. Например, если рабочее множество составляет 40 Гбайт и полностью помещается в доступную память, а значение `autoup` по умолчанию равно 30, то *fsflush* пытается синхронизировать передачу до 40 Гбайт данных на диск каждые 30 секунд. Большинство дисковых подсистем не могут поддерживать 1,3 Гбайт/с. Это означает, что приложение задыхается и ожидает завершения дискового ввода-вывода несмотря на то, что все рабочее множество находится в памяти!

Существуют три контрольных признака такого случая:

- *vmstat -p* показывает очень низкую активность файловой системы.
- *iostat -xtc* показывает непрерывные дисковые операции записи.
- Приложение имеет большое время ожидания для файловых операций.

Увеличение `autoup` (скажем, до 840) и `tune_t_fsflushr` (до 120) сократит объем данных, посылаемых на диск, увеличивая шансы на единый большой ввод-вывод (вместо множества меньших операций ввода-вывода). Также улучшатся шансы на сокращение операций записи, поскольку не каждая модификация файла будет записываться на диск. Обратная сторона медали – более высокий риск потерять данные в случае аварии сервера.

Инструменты для измерения производительности памяти

Инструменты для анализа производительности памяти можно классифицировать по трем основным областям: исследованию быстродействия памяти, степени нехватки памяти в системе и тому, сколько памяти потребляет конкретный процесс. В этом разделе рассмотрим инструменты для ответа на каждый из этих вопросов.

Измерение производительности памяти

По сути, мониторинг производительности памяти – это мониторинг ограничений памяти. Большей частью инструменты, оценивающие быстродействие подсистемы памяти, интересны в учебных целях, так как маловероятно, что настройкой можно добиться улучшения показателей. Одно исключение из этого правила – пользователи могут осторожно настраивать подходящее чередование. В большинстве систем чередование реализовано исключительно физическими средствами. Поэтому, возможно, следует приобрести дополнительную память. Для получения большей информации необходимо обратиться к документации системных аппаратных средств.[1] Однако при проведении значимых сравнений очень важно знать относительную производительность подсистемы памяти.

STREAM

Тестовая программа *STREAM* проста. Она оценивает время, требуемое для копирования участков памяти. Это измерение «практической» пропускной способности, а не теоретической «максимальной пропускной способности», которую предоставляют большинство производителей компьютеров. Инструментарий *STREAM* был разработан Джоном Мак-Калпином (John McCalpin) во время его пребывания в должности профессора в университете штата Делавэр.

В однопроцессорном режиме запустить тестовую программу достаточно легко (многопроцессорный режим намного более сложный; обрати-

[1] Помните, что обладать недостаточной емкостью чуть более быстрой памяти хуже, чем иметь достаточно чуть более медленной. Будьте внимательны.

тесь к документации тестовой программы для текущих деталей). Вот пример из Ultra 2 Model 2200:

```
$ ./stream
-------------------------------------------------------------
This system uses 8 bytes per DOUBLE PRECISION word.
-------------------------------------------------------------
Array size = 1000000, Offset = 0
Total memory required = 22.9 MB.
Each test is run 10 times, but only
the *best* time for each is used.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 40803 microseconds.
   (= 40803 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function      Rate (MB/s)   RMS time     Min time     Max time
Copy:         226.1804       0.0709       0.0707       0.0716
Scale:        227.6123       0.0704       0.0703       0.0705
Add:          276.5741       0.0869       0.0868       0.0871
Triad:        239.6189       0.1003       0.1002       0.1007
```

Полученные значения соответствуют данным табл. 4.2.

Таблица 4.2. Типы оценок STREAM

Оценка	Операция	Количество байт в итерации
Copy	`a[i] = b[i]`	16
Scale	`a[i] = q * b[i]`	16
Add	`a[i] = b[i] + c[i]`	24
Triad	`a[i] = b[i] + q * c[i]`	24

Также интересно отметить, что есть по крайней мере три типичных способа подсчета объема данных, передаваемых за одну операцию:

Аппаратный метод

Подсчитывает, сколько байт передано физически, так как аппаратные средства могут переместить количество байт, отличное от заданного пользователем. Это может произойти из-за кэша, так как при промахе операции записи в кэше многие системы выполняют *размещение записи*. Это означает, что строка кэша, содержащая такой участок памяти, сохраняется в процессорном кэше.[1]

[1] Это необходимо для обеспечения связности между оперативной памятью и кэшами.

Метод bcopy

Подсчитывает количество байт, переданных из одного участка памяти в другой. Если одна секунда уходит на чтение некоторого количества байт в одном участке и еще одна секунда на запись этого же количества в другой участок, то результирующая пропускная способность – это количество тех же байт, переданных за секунду.

Метод STREAM

Подсчитывает, сколько байт запрошено пользователем для чтения и для записи. Для простого теста «copy» это количество будет ровно в два раза больше, чем количество, полученное методом *bcopy*. Причина состоит в том, что некоторые тесты выполняют арифметические операции, поэтому есть смысл подсчитывать как данные, помещаемые в процессор, так и данные, возвращаемые из него.

Одна из замечательных черт *STREAM* состоит в том, что он всегда использует один и тот же метод подсчета байтов. За счет этого можно проводить достоверные сравнения.

STREAM доступен в исходном коде, поэтому его можно легко компилировать. Список методов измерений также доступен. Домашняя страница *STREAM* расположена по адресу *http://www.streambench.org*.

lmbench

lmbench – другая тестовая программа для измерения производительности памяти. Хотя *lmbench* способна проводить много различных измерений, остановимся на четырех из них. Первые три оценивают пропускную способность: скорость считывания из памяти, скорость записи в память и скорость копирования в памяти (с помощью метода *bcopy*, описанного перед этим). Последнее измерение оценивает время задержки считывания из памяти. Кратко обсудим, что оценивает каждый из этих методов измерений:

Оценка копирования в памяти

Выделяется большой участок памяти, который заполняется нулями. Затем оценивается время, необходимое для копирования первой половины участка памяти во вторую половину. Результаты сообщаются в виде количества мегабайт, переданных за секунду.

Оценка считывания из памяти

Выделяется участок памяти, который заполняется нулями. Затем оценивается время, необходимое для чтения из этой памяти, а именно серия целочисленных операций load и add. Каждое 4-байтное целое число загружается и добавляется к переменной-накопителю.

Оценка записи в память

Выделяется участок памяти, который заполняется нулями. Затем оценивается время, необходимое для записи в эту память, а именно серия из 4-байтных операций сохранения и инкрементирования.

Оценка времени задержки памяти

Измеряется время, требуемое для чтения байта из памяти. Результаты сообщаются в наносекундах на операцию load. Оценивается вся иерархия памяти данных[1], включая задержку в кэшах первого и второго уровня, оперативной памяти и задержки из-за промахов TLB. Для того чтобы извлечь больше информации с помощью *lmbench*, можно построить график зависимости задержки от размера массива, выбранного для теста.

После того как пользователь скомпилирует и запустит программу, *lmbench* задаст серию вопросов, касающихся выбора тестов для выполнения. Этот комплект очень хорошо документирован. Домашняя страница *lmbench*, которая содержит исходный код и более детальные сведения о его работе, расположена по адресу *http://www.bitmover.com/lm/lmbench/*.

Исследование потребления памяти в масштабе всей системы

Понимать производительность системы очень важно. Для этого есть единственный способ – регулярный мониторинг данных.

vmstat

vmstat – это один из самых вездесущих инструментов измерения производительности. Для *vmstat* есть одно главное правило, которое следует понять и никогда не забывать: в первой строке *vmstat* старается представить средние значения с момента загрузки. Первая строка вывода *vmstat* – это мусор, который следует отбросить.

Пример 4.1 показывает вывод *vmstat* в системе Solaris.

Пример 4.1. Вывод vmstat в Solaris

```
# vmstat 5
 procs     memory            page            disk          faults      cpu
 r b w   swap  free    re  mf pi po fr de sr s0 s1 s2 --  in   sy   cs us sy id
 0 0 0  43248 49592   0   1  5  0  0  0  0  0  1  0  0  116  106   30  1  1  99
 0 0 0 275144 56936   0   1  0  0  0  0  0  2  0  0  0  120    5   19  0  1  99
 0 0 0 275144 56936   0   0  0  0  0  0  0  0  0  0  0  104    8   19  0  0 100
 0 0 0 275144 56936   0   0  0  0  0  0  0  0  0  0  0  103    9   20  0  0 100
```

Колонки `r`, `b` и `w` соответственно представляют количество процессов, находящихся в очереди запуска, блокированных в ожидании ввода-вывода (включая пейджинг), и количество процессов, которые запущены, но выгружены на диск. Если в поле `w` встречается ненулевое значение, то можно лишь сделать вывод о том, что ранее в какой-то момент времени

[1] Важно заметить, что любые отдельные кэши команд не оцениваются.

системе не хватило памяти, и поэтому был проведен свопинг. Вот наиболее важные данные, которые можно почерпнуть из вывода *vmstat*:

`swap`

Объем доступного пространства свопинга (в Кбайт).

`free`

Объем свободной памяти, то есть размер свободного списка (в Кбайт). В Solaris 8 эта величина включает объем памяти, используемый для кэша файловой системы. В предшествующих версиях этот объем не учитывается, поэтому значение показателя будет очень маленькое.

`re`

Количество страниц, запрошенных из свободного списка. Страница была изъята у процесса, но затем была затребована процессом еще до того, как она была повторно использована и передана другому процессу.

`mf`

Количество легких страничных ошибок. Пока в свободном списке есть страницы, то обработка таких ошибок быстра, поскольку для нее не нужна подкачка страниц.

`pi`, `po`

Соответственно подкачки и откачки страниц из памяти (в Кбайт/с).

`de`

Краткосрочная нехватка памяти. Если значение не равно нулю, то это означает, что память в последнее время быстро очищалась. Поэтому будет запрошена дополнительная свободная память, так как она может скоро понадобиться.

`sr`

Частота сканирования демона страниц, в страницах в секунду. Это самый серьезный индикатор нехватки памяти. Если в системах-предшественницах Solaris 8 значение `sr` довольно длительное время составляет около 200 страниц в секунду, то системе необходимо больше памяти. Если этот показатель не равен нулю в Solaris 8, то налицо нехватка памяти.

В Linux вывод *vmstat* немного иной:

```
% vmstat 5
 procs                  memory    swap        io     system         cpu
 r b w   swpd   free   buff cache  si  so   bi   bo   in   cs  us  sy  id
 1 0 0 18372   8088 21828 56704   0   0    1    7   13    6   2   1   6
 0 0 0 18368   7900 21828 56708   1   0    0    8  119   42   6   3  91
 1 0 0 18368   7880 21828 56708   0   0    0   14  122   44   6   3  91
 0 0 0 18368   7880 21828 56708   0   0    0    5  113   24   2   2  96
 0 0 0 18368   7876 21828 56708   0   0    0    4  110   27   2   2  97
```

Заметим, что хотя поле w вычисляется, в Linux никогда не происходит интенсивный свопинг. Поля swpd, free, buff и cache соответственно представляют объем используемой виртуальной памяти, объем незадействованной памяти, объем памяти, используемой в буферах, и объем, используемый в памяти кэша. Полезных данных, которые можно извлечь из вывода *vmstat* в Linux, обычно совсем мало. Пожалуй, самые важные из них – это колонки si и so, которые сообщают объем загрузки в своп (swap-ins) и откачки из свопа (swap-out). Если эти значения велики, то, вероятно, необходимо увеличить значение параметра ядра swap_cluster, связанного с работой *kswapd*. За счет этого увеличится пропускная способность при записи в своп и чтении из свопа. Еще один вариант – приобрести больше физической памяти (см. раздел «Управление виртуальной памятью в Linux» ранее в этой главе).

sar

sar (*system activity reporter, генератор отчетов о системной активности*), как и *vmstat*, является вездесущим инструментом мониторинга производительности. Особенно он полезен тем, что его можно настроить для накапливания данных и последующего их рассмотрения, а также для сбора более сфокусированных, краткосрочных данных. Вообще, его вывод сравним с выводом *vmstat*, хотя и размечается по-другому.

По существу, синтаксис для вызова *sar* таков: *sar -flags interval number*. При этом количество данных *number* будет собираться каждые *interval* секунд. Самыми важными ключами для просмотра статистики памяти являются *-g*, *-p* и *-r*. Вот пример полученного вывода:

```
$ sar -gpr 5 100
SunOS islington.london-below.net 5.8 Generic_108528-03 sun4u    02/19/01

11:28:10  pgout/s ppgout/s pgfree/s pgscan/s %ufs_ipf
          atch/s  pgin/s ppgin/s  pflt/s  vflt/s slock/s
         freemem freeswap
...
11:28:50     0.00     0.00     0.00     0.00     0.00
           251.60    5.00    5.20 1148.20 2634.60    0.00
            86319  3304835
```

Наиболее важные поля вывода обобщены в табл. 4.3.

Таблица 4.3. Поля статистики памяти sar

Ключ	Поле	Значение
-g	pgout/s	Количество запросов на откачку страниц из памяти в секунду
	ppgout/s	Количество страниц, откачанных из памяти в секунду
	pgfree/s	Количество страниц, помещенных сканером страниц в свободный список в секунду
	pgscan/s	Количество страниц, сканированных сканером в секунду

Таблица 4.3 (продолжение)

Ключ	Поле	Значение
	%ufs_ipf	Процентное содержание кэшированных страниц файловой системы, изъятых из свободного списка, в то время как они все еще содержали значимые данные. Эти страницы сброшены и не могут быть затребованы (см. «Кэш индексных дескрипторов» в главе 5)
-p	atch/s	Количество страничных ошибок в секунду, которые были удовлетворены запросом страницы из свободного списка (иногда этот процесс называется присоединением)
	pgin/s	Количество запросов на загрузку страниц в память в секунду
	ppgin/s	Количество загруженных страниц в секунду
	pflt/s	Количество страничных ошибок, вызванных ошибками защиты (недопустимый доступ к странице, страничные ошибки copy-on-write), в секунду
-r	Freemem	Средний объем свободной памяти
	freeswap	Количество дисковых блоков в пространстве пейджинга

memstat

Начиная с ядра Solaris 7 Sun реализует ведение новой статистики памяти, призванной помочь в оценке работы системы. Для получения такой статистики необходимо воспользоваться инструментом *memstat*. Хотя в настоящее время он не поддерживается, будем надеяться, что он скоро будет включен в новую версию Solaris. Этот инструмент обладает замечательными функциями. На время написания книги он был доступен по адресу *http://www.sun.com/sun-on-net/performance.html*. Пример 4.2 показывает, какие данные может предоставить *memstat*.

Пример 4.2. memstat

```
# memstat 5
memory ---------- paging---------executable- -anonymous---filesys  --  ---  cpu --
  free  re  mf  pi  po  fr de sr epi epo epf  api apo apf  fpi fpo fpf us sy wt  id
 49584   0   1   5   0   0  0  0   0   0   0    0   0   0    5   0   0  1  1  1  98
 56944   0   1   0   0   0  0  0   0   0   0    0   0   0    0   0   0  0  0  0 100
```

Подобно выводу *vmstat*, первая строка вывода *memstat* не несет никакой смысловой нагрузки и должна быть отброшена. Откачку, подкачку и высвобождение страниц памяти *memstat* разделяет на три различных категории: выполнимые[1], анонимные и файловые операции. В систе-

[1] Имеются в виду страницы выполняющихся процессов или используемых разделяемых библиотек. Более подробные сведения об этом доступны по адресу *http://206.231.101.22/si/tools/vmstat_memstat/index.php*. – *Примеч. науч. ред.*

мах с большим объемом памяти значения `epf` и `apo` должны быть малы. Постоянная активность в этих полях говорит о нехватке памяти.

Однако при запуске Solaris 7 или предшествующей версии и выключенном приоритетном пейджинге, когда объем памяти падает до уровня `lotsfree` или ниже, пейджинг памяти для выполнимых файлов и анонимной памяти будет происходить с еще меньшим объемом ввода-вывода файловой системы.

Сколько памяти потребляют процессы

Очень полезно знать, сколько памяти потребляет конкретный процесс. Адресное пространство каждого процесса, скроенное из многих сегментов, можно оценить по следующим критериям:

- Полный размер адресного пространства процесса (представленный как SZ или SIZE)
- Размер резидентной части адресного пространства процесса (удерживаемой в памяти, RSS)
- Полное разделяемое адресное пространство
- Полное собственное адресное пространство

Для исследования потребления памяти и оценки памяти, требуемой для конкретного процесса, существует много инструментов.

Инструменты Solaris

Один типичный инструмент – */usr/ucb/ps uax*. Вот пример данных, которые он выдает:

```
% /usr/ucb/ps uax
USER       PID %CPU %MEM   SZ  RSS TT       S    START  TIME COMMAND
root     16755  0.1  1.0 1448 1208 pts/0    O 17:33:35  0:00 /usr/ucb/ps uax
root         3  0.1  0.0    0    0 ?        S   May 24  6:19 fsflush
root         1  0.1  0.6 2232  680 ?        S   May 24  3:10 /etc/init -
root       167  0.1  1.3 3288 1536 ?        S   May 24  1:04 /usr/sbin/syslogd
root         0  0.0  0.0    0    0 ?        T   May 24  0:16 sched
root         2  0.0  0.0    0    0 ?        S   May 24  0:00 pageout
gdm      14485  0.0  0.9 1424 1088 pts/0    S 16:17:57  0:00 -csh
```

Заметим, что для демонов ядра, скажем, для *fsflush*, *sched* и *pageout*, значения SZ и RSS равны нулю. Такие процессы полностью выполняются в пространстве ядра и не потребляют память, которая может использоваться для запуска других приложений. Однако *ps* ничего не сообщает об объеме собственной и разделяемой памяти, необходимой процессу, то есть как раз то, что хочется знать.

В Solaris запуск */usr/proc/bin/pmap -x process-id* дает значительно больше данных о потреблении памяти процессами. (В Solaris 8 эта команда перемещена в */usr/bin*.) Рассмотрим процесс *csh* (в продолжение предыдущего примера):

```
% pmap -x 14485
14485:  -csh
Address   Kbytes Resident Shared Private Permissions       Mapped File
00010000     144      144      8     136 read/exec         csh
00042000      16       16      -      16 read/write/exec   csh
00046000     136      112      -     112 read/write/exec    [ heap ]
FF200000     648      608    536      72 read/exec         libc.so.1
FF2B0000      40       40      -      40 read/write/exec   libc.so.1
FF300000      16       16     16       - read/exec         libc_psr.so.1
FF320000       8        8      -       8 read/exec         libmapmalloc.so.1
FF330000       8        8      -       8 read/write/exec   libmapmalloc.so.1
FF340000     168      136      -     136 read/exec         libcurses.so.1
FF378000      40       40      -      40 read/write/exec   libcurses.so.1
FF382000       8        -      -       - read/write/exec    [ anon ]
FF390000       8        8      8       - read/exec         libdl.so.1
FF3A0000       8        8      -       8 read/write/exec    [ anon ]
FF3B0000     120      120    120       - read/exec         ld.so.1
FF3DC000       8        8      -       8 read/write/exec   ld.so.1
FFBE4000      48       48      -      48 read/write         [ stack ]
--------  ------  ------  ------  ------
total Kb    1424     1320     688     632
```

Вывод *pmap* предоставляет сведения по каждому сегменту, а также сведения для процессов в целом.

Инструменты Linux

В Linux команда *ps*, по существу, работает так же, как и в Solaris. Вот пример (добавлена строка заголовка для легкости идентификации отдельных колонок):

```
% ps uax | grep gdm
USER       PID %CPU %MEM  SIZE   RSS TTY STAT START   TIME COMMAND
gdm      12329  0.0  0.7  1548   984  p3 S    18:18   0:00 -csh
gdm      13406  0.0  0.3   856   512  p3 R    18:37   0:00 ps uax
gdm      13407  0.0  0.2   844   344  p3 S    18:37   0:00 grep gdm
```

Хотя команда *ps* вездесуща, она не очень информативна. В Linux эквивалентом *pmap* системы Solaris являются записи в файловой системе */proc*, которые могут рассказать много полезного.

Одна из замечательных черт Linux – это возможность напрямую работать с файловой системой */proc*. Такой доступ дает полезные данные о потреблении памяти конкретными процессами, как и *pmap* в Solaris. Каждый процесс имеет каталог в файловой системе */proc* согласно своему ID. В этом каталоге находится файл *status*. Вот пример содержимого этого файла:

```
% cat /proc/12329/status
Name:   csh
State:  S (sleeping)
```

```
Pid:    12329
PPid:   12327
Uid:    563     563     563     563
Gid:    538     538     538     538
Groups: 538 100
VmSize:     1548 kB
VmLck:         0 kB
VmRSS:       984 kB
VmData:      296 kB
VmStk:        40 kB
VmExe:       244 kB
VmLib:       744 kB
SigPnd: 0000000000000000
SigBlk: 0000000000000002
SigIgn: 0000000000384004
SigCgt: 0000000009812003
CapInh: 00000000fffffeff
CapPrm: 0000000000000000
CapEff: 0000000000000000
```

Налицо большое количество данных. Вот быстрый скрипт Perl для выделения наиболее полезных частей:

```
#!/usr/bin/perl
$pid = $ARGV[0];
@statusArray = split (/\s+/, `grep Vm /proc/$pid/status`);
print "Status of process $pid\n\r";
print "Process total size:\t$statusArray[1]\tKB\n\r";
print "            Locked:\t$statusArray[4]\tKB\n\r";
print "      Resident set:\t$statusArray[7]\tKB\n\r";
print "              Data:\t$statusArray[10]\tKB\n\r";
print "             Stack:\t$statusArray[13]\tKB\n\r";
print " Executable (text):\t$statusArray[16]\tKB\n\r";
print "  Shared libraries:\t$statusArray[19]\tKB\n\r";
```

Заключение

У этой истории простая мораль: требования к памяти сильно зависят от количества пользователей и характера их работы. Большие технические приложения, такие как вычислительная гидрогазодинамика или выяснение структуры белка, могут потребовать немыслимый объем памяти. Легко представить компьютер с 2 Гбайт памяти, который поддерживает только одно такое задание, тогда как та же система вполне комфортно могла бы обеспечивать работу нескольких сотен интерактивных пользователей.

Одно из величайших улучшений в вычислениях за последние двадцать лет – это возможность компоновать большие объемы памяти при относительно низкой их стоимости. Воспользуйтесь таким преимуществом.

5

- *Архитектура диска*
- *Интерфейсы*
- *Общие проблемы производительности*
- *Файловые системы*
- *Инструменты для анализа*
- *Заключение*

Диски

Я думаю, что Кремниевая долина была неверно названа. Если вспомнить доллары, вложенные в изделия за последние десять лет, то выручка от магнитных дисков превысила доход от кремния. Это место следует переименовать в Долину оксида железа.

Al Hoagland, 1982

Традиционно настройка производительности ввода-вывода отставала от настройки более «эффектных» элементов системы. Этот сдвиг и по сей день можно наблюдать на рынке персональных машин. Каждый производитель компьютеров рекламирует тактовую частоту их процессоров. Значительно реже он извещает о частоте вращения или внутренней скорости передачи их жестких дисков. Зачастую наши первые мысли о производительности – это мысли о мощности процессора и потреблении памяти, а не о скорости передачи данных на жесткие диски и с жестких дисков.

Такой подход вызывает сожаление, поскольку ставки в настройке ввода-вывода чрезвычайно высоки. Диски почти на шесть порядков медленнее физической памяти. Поэтому возможность избежать обращений к диску и максимальное ускорение необходимых обращений может оказать огромное влияние на производительность приложений.

Дополнительную важность настройке ввода-вывода придает тот факт, что большинство ярких улучшений в технологии хранения сводились к увеличению объема хранимых данных, а не к повышению скорости доступа к ним. За последние десять лет частота вращения увеличилась примерно в три раза, в то время как емкость запоминающих устройств увеличилась по крайней мере в сто раз.

Эта глава посвящена теме, которой зачастую пренебрегают: анализу производительности и настройке дисковых подсистем. Сначала внимание будет сосредоточено на базовых концепциях дисковых накопителей: физическая компоновка, типы доступа и факторы, влияющие на производительность дисков в различных ситуациях. Кроме того, разговор пойдет об интерфейсах, через которые происходит взаимодействие с дисками, таких как IDE и SCSI. Будут рассмотрены общие проблемы производительности дисков, влияние файловых систем различных типов на производительность, а также инструменты для мониторинга и анализа дисковой производительности.

Архитектура диска

Дисковый накопитель состоит из закрепленных на одной оси металлических дисков, покрытых магнитным слоем, на который происходит запись. Эти диски называются *пластинами* (*platters*).[1] Пластины вращаются вокруг центрального *шпинделя* (*spindle*), обычно со скоростью от 5 400 до 15 000 вращений в минуту. Между этими пластинами находится совокупность подвижных манипуляторов, на которых закреплены маленькие сенсоры,[2] называемые *головками считывания/записи* (*read/write heads*). Для выполнения дисковой операции в заданной части диска манипуляторы должны туда переместиться. Соответствующий участок на пластине должен крутиться ниже головок. При выверенном позиционировании данные могут быть считаны с диска или записаны на диск.[3]

Полезная площадь хранения диска разбита согласно адресной схеме.[4] Кольцо данных, захватываемое головками на заданной пластине при ее вращении, называется *дорожкой* (*track*). Группа дорожек на всех пластинах при заданной позиции манипулятора называется *цилиндром* (*cylinder*). Цилиндры нумеруются с внешнего края диска вовнутрь, то есть цилиндры с наименьшими номерами являются крайними. Каждая дорожка, в свою очередь, делится на *секторы* (*sectors*), обычно размером 512 байт.[5] Количество секторов на дорожку зависит

---

1 На русскоязычном компьютерном жаргоне эти пластины называют «блинами». – *Примеч. науч. ред.*

2 «Сенсор» в данном случае – это устройство, в котором возникает ток, когда мимо него движется намагниченная поверхность. К сенсорным выключателям в телевизорах и т. п., работающим на изменении электрической емкости, этот сенсор отношения не имеет. – *Примеч. науч. ред.*

3 Дисковый накопитель также хранит важную информацию в самом начале диска. Например, метку диска.

4 Часть пространства зарезервирована для внутреннего использования дисковым накопителем. В нем хранятся служебные данные, например данные о выравнивании головок и коррекции ошибок.

5 Этот размер определяется стандартом SCSI.

от размера пластины и плотности записи. Рисунок 5.1 является иллюстрацией архитектуры диска.

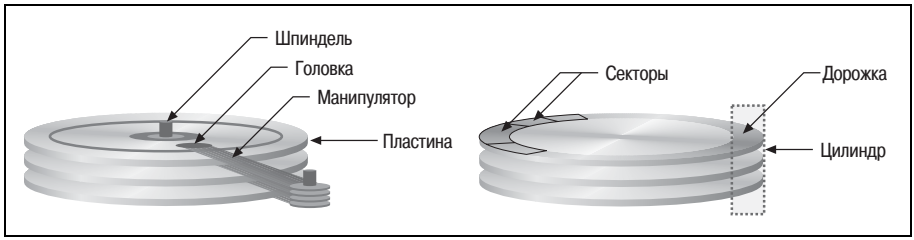


Рис. 5.1. *Архитектура диска*

Позонная организация хранения (ZBR)

Если бы количество секторов на дорожке было постоянным, то данные на дорожках располагались бы менее плотно при продвижении от центра к краю диска. По мере продвижения к краю диска пространство расходовалось бы менее практично. Для преодоления такого затруднения во всех современных дисках используется методика позонной организации хранения (zone bit rate recording, ZBR), согласно которой диск разбивается на зоны, а на крайних дорожках располагается больше секторов.[1] Зоны с меньшими номерами имеют больше секторов на дорожку, чем зоны с более высокими номерами. Таким образом достигается примерно одинаковая плотность данных – в показателях количества бит на дюйм. Кроме того, это приводит к варьированию скорости передачи данных в зависимости от номера цилиндра.

Цилиндр с наименьшим номером – это самый быстрый, самый большой цилиндр с наибольшей плотностью (больше всего бит в цилиндре). Цилиндр с наибольшим номером – это самый медленный, самый маленький цилиндр с наименьшей плотностью (меньше всего бит в цилиндре).

К сожалению, такая схема отчасти усложняет вычисления производительности диска, поскольку скорость передачи варьируется в зависимости от цилиндра, к которому происходит доступ.

Дисковые кэши

SCSI- и IDE-диски обычно имеют локальные кэши. Такие кэши – это просто небольшой объем памяти, обычно между 128 Кбайт и 4 Мбайт, ассоциированный со встроенным контроллером диска. Когда данные считываются с диска, их копия сохраняется в этом кэше. Большинство пауз при обращении к диску связано либо со временем передачи данных с пластин, либо со временем перемещения головок считыва-

[1] В большинстве современных дисков около пятнадцати зон.

ния/записи в нужное положение и ожидания соответствующего сектора, к которому предстоит обратиться. Поэтому обслуживание запроса данных из локального дискового кэша сохраняет немало времени, а при частом обращении к одним и тем же данным локальные кэши могут быть очень эффективны. По умолчанию в большинстве накопителей кэширование чтения включено.

Что обычно не включено, так это возможность использовать локальные кэши для кэширования записи. Обычно накопитель не уведомляет о завершении записи до тех пор, пока данные не будут физически записаны на диск. Если кэширование записи включено, то диск будет уведомлять о завершении записи, когда данные будут сохранены в кэше. Такая память – непостоянная. Это означает, что ее содержимое исчезает, когда питание выключается. При кэшировании чтения это не проблема. Однако из-за неполадок с питанием в неподходящий момент может произойти ошибочное уведомление о записи, так как данные не были физически записаны на диск. Нет абсолютно никакой гарантии в том, что данные, записанные в кэш дискового буфера, когда-нибудь достигнут пластин. Вследствие подобных затруднений с надежностью такие кэши обычно не включены. Их включение может существенно повысить производительность, однако такое улучшение производительности приходит с риском потери данных.

О том, как включить и проверить состояние этих кэшей, будет рассказано в разделе «Включение дисковых кэшей» далее в этой главе.

Типы доступа

Обращения к диску могут либо быть собраны вместе (*последовательный доступ*, *sequential access*), либо происходить без определенного порядка (*произвольный доступ*, *random access*). То, что обращения к диску собраны вместе, не обязательно означает, что они последовательны. Большинство обращений к диску, происходящих друг за другом, оказываются последовательными. Собранные вместе обращения к диску не обязательно, но часто подразумевают доступ к последовательным секторам на диске. К сожалению, такие термины также уместны в других областях. Носители могут быть «прозвольно доступны». Это означает, что «к некому биту можно обратиться так же быстро, как и к любому другому биту». Примером устройства с таким доступом может служить жесткий диск. Либо носитель может быть «последовательно доступен», как, например, накопитель на ленте. При упоминании таких терминов в этой главе речь идет о типах доступа.

Производительность двух типов доступа определяется двумя очень разными правилами:

- При последовательном доступе преобладающим является влияние реального времени передачи с дисковых пластин (внутренняя скорость передачи).

- При произвольном доступе доминируют скорость вращения пластин и время поиска.

Увеличение скорости передачи или преобразование операций произвольного доступа в операции последовательного доступа существенно повышает производительность дисковой подсистемы. *Это одна из наиболее благоприятных областей для приложения усилий по увеличению производительности.*

Считывания

Дисковые считывания обычно синхронные; запрашивающий процесс блокируется до завершения чтения. Это не всегда так, поскольку асинхронное считывание может быть вызвано функцией `aioread(3)`. Когда система считывает данные, они кэшируются на случай, если понадобятся снова. Кэширование использует участок оперативной памяти, что подробно описано в разделе «Кэширование в файловой системе» главы 4. Если ведется дискуссия о повышении производительности считывания и обсуждается, расширить ли оперативную память или инвестировать в хороший дисковый контроллер, то почти всегда лучше выбрать расширение оперативной памяти. В этом случае будет расширен кэш файловой системы, обращение к которому занимает меньше микросекунды. Кроме того, наверняка уменьшится количество считываний, а значит, снизится нагрузка шины и сэкономится время процессора.

Обычно при проектировании дисковых подсистем, ориентированных на чтение, целью является минимизация времени задержки при первоначальном чтении некэшированных байтов и максимальное увеличение пропускной способности при передаче последовательных данных.

Записи

Дисковые записи – это смесь синхронных и асинхронных действий. Асинхронные действия происходят, когда файл записывается в локальную файловую систему: данные записываются в память, и запись немедленно подтверждается. На самом деле эти данные сбрасываются на диск позднее с помощью демона сбрасывания (см. раздел «Записи из кэша файловой системы: fsflush и bdflush» в главе 4). Приложения могут инициировать немедленную запись файла с помощью вызова `fsync(3C)` или закрытия файла. Заметим, что когда процесс завершен, все его открытые файловые дескрипторы закрываются, а данные сбрасываются на диск. Это может занять какое-то время и привести к существенному объему дисковой активности (если одновременно завершаются много процессов).

Перекрывание записи в файловой системе UFS

Для того чтобы предотвратить слишком большое потребление памяти при незавершенных операциях записи, Solaris реализует алгоритм пе-

рекрывания записи (write throttle). Этот алгоритм ограничивает объем данных, ожидающих своей записи в любой заданный файл. Такое ограничение выполняется на попроцессной, пофайловой основе. Для каждого файла ожидать записи могут данные объемом от 256 Кбайт (*нижний предел*, *low-water mark*, задается параметром `ufs:ufs_LW`) до 384 Кбайт (*верхний предел*, *high-water mark*, задается параметром `ufs:ufs_HW`). Когда объем таких данных становится меньше нижнего предела, запись данных осуществляет демон сбрасывания. Когда объем данных, которые ожидают записи на диск, находится между этими двумя пределами, происходит распределение операций записи для сброса данных на диск. Если объем данных превышает верхний предел, то приложению больше не позволяется записывать данные (это будет заблокировано системным вызовом `write()`) до тех пор, пока объем данных, ожидающих записи, снова не упадет ниже отметки нижнего предела.

При высоких скоростях работы с данными асинхронные записи на диск становятся синхронными, а процессы замедляются. Записи являются асинхронными, так как они кэшируются в памяти и немедленно подтверждаются, а сам доступ к диску происходит позднее. Когда буфер записи процесса для конкретного файла полностью заполняется, записи становятся синхронными, а производительность теперь зависит от скорости записи данных на диск, а не от того, насколько быстро данные кэшируются в памяти.

Когда необходимо осуществить быструю запись в файл, а базовой дисковой подсистеме по силам такая пропускная способность, то ее работе с максимальной эффективностью могут помешать преждевременные перекрывания записи. Кроме того, перекрывание записи может оказывать влияние на приложения с большими всплесками активности записи в отдельных интервалах. Лучший диагностик такой ситуации – это счетчик ядра `ufs_throttles`. Его значение инкрементируется при каждом перекрывании записи, поэтому постоянное увеличение этого значения свидетельствует о том, что перекрывание записи следует расширить.

Расширение перекрывания записи может иметь побочный эффект, выражающийся в большой очереди запросов ввода-вывода для файла. Такой способ может увеличить пропускную способность, но также увеличит и среднее время ответа, поскольку очередь будет длиннее. С помощью *ptime*(1) эффект от изменения перекрывания записи можно определить по времени, необходимому для создания файла методом *mkfile*.[1] Для этого теста *не* следует использовать */tmp*. Причина объяс-

[1] Команда *mkfile* есть не в любом UNIX; она точно есть в Solaris и отсутствует в стандартных дистрибутивах Linux. – *Примеч. науч. ред.*

нена в разделе «Временная файловая система (Temporary Filesystem, tmpfs)» далее в этой главе.

На практике значения верхнего и нижнего предела по умолчанию установлены слишком низкими. Отметку верхнего предела следует установить во много раз выше значения параметра `maxphys` (описание этого параметра есть в разделе «Взаимодействия между кэшем файловой системы и диском» главы 4). Отметка нижнего предела должна быть равна половине или двум третям значения верхнего предела. На практике авторы часто присваивают отметке нижнего предела значение, равное 1/32 емкости физической памяти, а отметке верхнего предела – значение, равное 1/16 емкости физической памяти. Это означает, что если есть один процесс, интенсивно записывающий данные в шестнадцать отдельных файлов, то всю системную память теоретически можно использовать как кэш записи. Однако на практике это недостижимо, поскольку в дело вступит демон страниц, который займется сбором кэшированных страниц и сбросом их данных на диск. Для получения большей информации о демоне страниц следует обратиться к разделу «Пейджинг и свопинг» главы 4.

Управление всем механизмом перекрывания записи производится с помощью параметра ядра `ufs:ufs_WRITES`. По умолчанию он равен 1, что означает включение перекрывания. Есть ему присвоить нуль, то перекрывание будет выключено. Потенциальный побочный эффект такого шага – очень высокое потребление памяти.

Характеристики производительности

Одна из сложностей в анализе дисковой производительности – это сбивающий с толку способ, на основе которого большинство поставщиков предоставляет статистику производительности своих устройств.

Один миллион байт – это мегабайт?

Производители дисков, а также спецификация SCSI применяют термин «мегабайт» для обозначения одного миллиона (1 000 000) байт. Однако во всех остальных случаях под мегабайтом понимается 1 024 килобайт, или 2^{20} (1 048 576) байт. Такое пятипроцентное отличие сказывается как на характеристиках производительности, так и на характеристиках хранения:

- Дисковый накопитель со скоростью передачи 8,4 миллиона байт в секунду может передать только 8,0 Мбайт/с.
- Дисковый накопитель с указанной емкостью 9 100 миллионов байт может хранить только 8,67 Гбайт.

Такое соглашение закрепилось во всей индустрии, несмотря на путаницу, которую оно создает.

Пакетная скорость в сравнении с внутренней скоростью передачи

Производители дисков также любят приводить *пакетную скорость* (*burst speed*) встроенного контроллера. Это та скорость, с которой встроенный контроллер может передавать данные на шину. Издержки управления и изменения состояния в расчет не принимаются. Это просто скорость, с которой передаются биты. Такая скорость может быть обеспечена лишь в случае, когда диск способен предоставлять данные с этой скоростью, а запросившее их устройство[1] может справиться с таким потоком. На практике такая скорость достижима, когда данные передаются из локального кэша диска в оперативную память системы. В большинстве же случаев данные не расположены в кэше диска, поэтому операция считывания производится с дисковых пластин. Кроме того, для произвольного ввода-вывода диску, вероятно, необходимо переместить головки на нужную позицию.

Внутренняя скорость передачи (*internal transfer speed*) отражает скорость, с которой биты считываются с дисковых пластин головкой считывания-записи без учета других факторов (таких как поиск). Это измерение показывает лишь скорость, с которой данные могут быть сняты с пластин, и не оценивает производительность любой другой части дискового накопителя. Внутренняя скорость передачи *намного* меньше пакетной скорости. Она может существенно варьироваться в зависимости от того, с какого цилиндра происходит чтение (см. раздел «Позонная организация хранения (ZBR)» ранее в этой главе). В результате внутренняя скорость передачи – это довольно неплохой показатель производительности последовательной передачи данных.

Наиболее важный вывод из различия между пакетной скоростью и внутренней скоростью передачи таков: простой диск с пакетной скоростью 40 Мбайт/с вряд ли будет быстрее другого идентичного диска с пакетной скоростью 20 Мбайт/с.

Внутренняя скорость передачи в сравнении с действительной скоростью

Усложним дело еще больше.

Внутренняя скорость передачи в большей степени отражает истинную производительность дискового накопителя, чем пакетная скорость. Однако внутренняя скорость передачи означает только скорость, с которой все данные проходят под головками считывания-записи, и это измерение включает существенный объем издержек. Большая часть издержек приходится на коды коррекции ошибок, которые

[1] Обычно это контроллер прямого доступа или SCSI-контроллер. – *Примеч. науч. ред.*

предотвращают искажение данных вследствие дисковых ошибок. Более точное измерение производительности простого диска можно получить с помощью следующей формулы:

$$\text{скорость} = \frac{\dfrac{\text{Количество секторов на дорожке} \cdot \text{RMP} \cdot 512}{60\text{ с}}}{1\,000\,000}$$

Вот пример с диском Seagate 18,2 Гбайт Barracuda (ST118273W). Обратимся к листу спецификации производителя и найдем среднее количество секторов на дорожку (237) и частоту вращения диска (7 200 rpm). Теперь можно выполнить вычисление:

$$\text{скорость} = \frac{\dfrac{237 \cdot 7200 \cdot 512}{60}}{1\,000\,000} = 14{,}56\text{ Мбайт/с}$$

Такова скорость, с которой данные могут быть переданы на диск или с диска (предполагается исключительно последовательный доступ).

Среднее время поиска

Среднее время поиска (*average seek time*) также приводится часто, однако этот показатель в высшей степени обманчив. Для производителей дисков среднее время поиска – это частное от деления суммы всех периодов возможных поисков на количество возможных поисков. Хотя такое определение математически верно, механика, вовлеченная в перемещение дисковых манипуляторов, предполагает, что короткие поиски отнимают меньше времени, чем длительные. По существу, «средний» поиск совершенно не похож на «типичный» поиск. При «среднем» поиске перемещение дисковых манипуляторов намного больше, чем при типичном.

Урок таков: минимизация времени поиска при хранении данных оптимизирует производительность. Разбиение одного диска на два раздела, каждый из которых будет занят, является очень плохим решением. При такой конфигурации каждый раз, когда происходит обращение к нетекущему разделу диска, поиск выполняется от средней позиции на одном разделе до средней позиции на другом, что составляет половину максимального времени поиска. Это отнимает много времени. Более верный подход – размещение двух разделов на разных дисках,[1] чтобы минимизировать время поиска.

[1] Как это часто бывает, более дорогое решение оказывается более эффективным. Однако перед покупкой второго диска придется решать: стоит ли получаемый выигрыш нескольких десятков или сотен долларов. – *Примеч. науч. ред.*

Очень легко не придавать значения непомерному времени поиска. Однако если максимальное время поиска находится в районе 20 мс, то каждый такой поиск вызовет задержку, эквивалентную времени выполнения *сотен миллионов* команд.

Емкость запоминающего устройства и эффективность доступа

При операциях с произвольным доступом дисковая производительность определяется частотой вращения пластин и временем поиска. Поэтому производительность накопителей с одной частотой вращения почти одинакова – она очень мало зависит от емкости или физических размеров.[1] При операциях с произвольным доступом наилучший способ увеличить эффективность доступа – это приобрести максимально возможное количество дисков с высокой частотой вращения.

К сожалению, *повышение частоты доступа* (обычно измеряемой как количество операций с произвольным доступом в секунду) не идет в ногу с повышением *емкости запоминающих устройств*. Проиллюстрируем это. Представим диск 5400 rpm 2,1 Гбайт и диск 7200 rpm 9,1 Гбайт. Емкость второго диска более чем в четыре раза больше, но его эффективность доступа больше лишь на 30%. Поэтому наибольшей эффективности доступа при вводе-выводе обычно легче достичь за счет нескольких минимально приемлемых дисков, а не за счет одного большого диска с превосходными характеристиками. Это проиллюстрировано в табл. 5.1.

Таблица 5.1. Сравнение способности доступа при различных реализациях 218 Гбайт

Диски[a] (Гбайт)	Количество	Шины[b]	Количество операций ввода-вывода в секунду на один диск[c]	Общее количество операций ввода-вывода в секунду	Общая скорость передачи (Мбайт/с)
4,3	51	4	108	5 508	160
9,1	24	2	85	2 040	80
18,2	12	1	70	840	40

[a] У всех дисков одна частота вращения.

[b] Минимальное количество шин Ultra Wide (fast-40) SCSI, необходимое для обслуживания дисков.

[c] Это эмпирические значения.

Однако применение меньших дисков обычно более дорого, более усложнено и отнимает больше пространства. Кроме того, необходимо исполь-

[1] Накопители 5,25" имеют чуть более высокое время поиска, но, вероятно, это не оказывает никакого практического влияния.

зовать дисковые массивы, которые будут обсуждены позднее. Назначение дисковых массивов – преодолеть трудности, связанные с повышением интенсивности отказов подсистемы из-за большего количества компонентов. Во многих операционных средах лучше предусмотреть смешанный состав дисков большой и малой емкости ради достижения золотой середины в отношении стоимость–производительность.

Компоновка дисковых подсистем только исходя из соображений емкости обычно является серьезной ошибкой с точки зрения производительности.

Интерфейсы

Интерфейс (*interface*) представляет собой правила, управляющие взаимодействием между главной системой компьютера и совокупностью устройств. Эти правила обычно устанавливаются с помощью хост-адаптера, который обеспечивает связь процессора и других компонентов компьютера с периферийными устройствами (см. раздел «Периферийные соединения» в главе 3).

В основном рынок современных дисков определяется тремя дисковыми протоколами: IDE, SCSI и Fibre Channel. Главным образом, в этой книге будут рассматриваться именно эти протоколы, а также один более старый протокол IPI и некоторые появившиеся интерфейсы устройств хранения (IEEE 1394/Firewire и USB). Важно отметить, что «настоящих» устройств IEEE 1394 и USB не существует. Все это накопители IDE с присоединенными адаптерами, а значит, в них осталось много слабых мест типичных устройств IDE.

IDE

Интерфейсы IDE (встроенный интерфейс устройств, *Integrated Drive Electronics*), или ATA (*AT Attachment*), исторически применялись в недорогих системах, особенно в персональных компьютерах на платформе Intel. Однако благодаря своей дешевизне диски IDE становятся типичными для младших моделей рабочих станций, таких как Sun Ultra 5 и Ultra 10. Спецификация IDE предполагала снижение стоимости интерфейсов за счет размещения управляющей электроники на самом накопителе, а также за счет написания как можно более простого управляющего кода ПЗУ (firmware).[1]

В какой-то мере разработка IDE координируется Техническим комитетом T13, являющимся частью Национального комитета по стандар-

[1] Термин «firmware» мы переводим как «управляющий код ПЗУ», понимая под этим программное обеспечение, которое прошивается в ПЗУ устройства (диска, системной платы и т. п.) на заводе производителя устройств. – *Примеч. науч. ред.*

там информационных технологий (National Committee on Information Technology Standards, NCITS, произносится «инсайтс»). NCITS руководствуется правилами, одобренными Национальным институтом стандартизации США (American National Standards Institute, ANSI). Эти правила призваны обеспечить принятие рекомендательных стандартов, разработанных в индустрии. NCITS разрабатывает стандарты систем обработки информации, в то время как ANSI санкционирует эту деятельность и публикует стандарты. Веб-страница Технического комитета расположена по адресу *http://www.t13.org*.

Вслед за успехом оригинального интерфейса ATA Комитет маленького форм-фактора (Small Form-Factor Committee, организация производителей дисков) разработал обратно-совместимое расширение интерфейса ATA, названное ATA-2 (или FAST-ATA). В этом стандарте добавлены более быстрые режимы PIO и DMA. Вопреки распространенным заверениям, схема адресации логического блока (LBA), переделанная в ATA-2, не имеет ничего общего с преодолением барьера 504 Мбайт. Даже оригинальные спецификации ATA имели предел емкости свыше 100 Гбайт.

Спустя несколько лет вышла следующая версия спецификации, названная ATA-3. Введен усовершенствованный режим управления питанием, повышена надежность, а также добавлено прогнозирование отказов. Ни одного более быстрого режима определено не было. Одно из основных неудобств стандарта ATA состоит в том, что он был разработан только для дисков. Вследствие скачка популярности устройств CD-ROM интерфейс был расширен с помощью пакетного интерфейса ATA (ATA Packet Interface, ATAPI), который облегчил добавление таких устройств. Официально ATAPI был введен в ATA в стандарте ATA-4.

Усовершенствованный IDE (*Enhanced IDE*, EIDE) – это маркетинговый термин, относящийся к реализациям ATA-2 и ATAPI, тогда как Fast-ATA – это конкурирующий маркетинговый термин для обозначения устройств, созданных только на базе стандарта ATA-2.

Одно из последствий снижения стоимости устройства IDE сводится к тому, что процессор перестает играть основную роль в обработке запросов ввода-вывода. По сути, устройства IDE имеют два способа передачи данных процессору: *программируемый ввод-вывод* (*programmed I/O*, PIO) и *прямой доступ к памяти* (*direct memory access*, DMA), который может выполняться на уровне одного или многих слов. В режиме передачи PIO каждую транзакцию с диска в память обслуживает процессор, тогда как в режиме передачи DMA контроллер IDE может записывать данные напрямую в основную память. Накопитель сообщает контроллеру свое состояние готовности, задействуя линию прерывания устройства. Это может быть сделано в двух случаях:

- Накопитель получил команду считывания; эта команда была обслужена, а запрошенные данные находятся в буфере накопителя и готовы к перемещению в память.

- Накопитель получил команду записи; эта команда передана в буфер накопителя (если для накопителя не включено кэширование записи, то линия не будет задействована до тех пор, пока данные не будут физически перенесены на диск).

При обычных считываниях и записях обработка этого прерывания может постоянно докучать процессору и приводить к длительным задержкам в обслуживании накопителя. Составные команды считывания/записи – это команды уровня накопителя, которые предназначены для того, чтобы преодолеть такие трудности. Эти команды позволяют передавать до 128 секторов без промежуточных прерываний, тем самым существенно снижая нагрузку на процессор.

Однословный доступ DMA по сравнению с многословным выполняется очень медленно. По существу, «DMA» обычно предполагает многословную передачу. Поддержка однословного DMA была прекращена в спецификации ATA-3.

Так как быстродействие жестких дисков увеличивалось, то самого быстрого многословного режима DMA (16,7 Мбайт/с) вскоре стало недостаточно. К сожалению, интерфейс был разработан для медленных передач данных (~5 Мбайт/с). Простое повышение тактовой частоты интерфейса вызывало помехи при передаче сигналов. Поэтому был разработан новый вид режима передачи DMA (названный UltraDMA). Он был представлен в стандарте, названном *Ultra-ATA*, который был создан для заполнения пробела между ATA-3 и грядущим ATA-4. Теперь, когда выпущен ATA-4, стандарт Ultra-ATA включен в ATA-4.

Ключевое технологическое улучшение, введенное в UltraDMA, состояло в двойной синхронизации переходов: данные передаются на переднем и заднем фронтах синхронизирующего сигнала. Это та же самая технология, которая была применена для повышения скорости передачи в DDR SDRAM (см. раздел «Реализации физической памяти» в главе 4). Кроме того, для улучшения целостности данных модули UltraDMA применяют алгоритм *проверки циклической избыточности* (*cyclical redundancy checking*, CRC). На основе этого алгоритма для каждого передаваемого блока вычисляется код обнаружения ошибок, который посылается вместе с блоком. Получатель данных может проверить, не повредились ли данные при передаче. Если данные были повреждены, то они передаются повторно.

Несмотря на повышение пропускной способности, достигнутое за счет синхронизации двойных переходов, передача данных со скоростью выше 33 Мбайт/с в конце концов исчерпала возможности стандартного 40-проводного кабеля IDE. Для использования режимов UltraDMA с номером больше 2 требуется специальный 80-проводной кабель.[1]

[1] Такой кабель был описан в спецификации ATA-4 для режимов UltraDMA 0, 1 и 2, но был необязательным.

Этот кабель использует те же 40 контактов, но добавляет еще 40 линий заземления между каждой парой оригинальных линий, чтобы разделить эти линии и избежать наводок.

Обзор этих стандартов представлен в табл. 5.2.

Таблица 5.2. Интерфейсы IDE

Тип	Режим	Время цикла (МГц)[a]	Скорость передачи данных (Мбайт/с)	Поддержано в стандарте
PIO	0	1,67	3,3	ATA
	1	2,61	5,2	ATA
	2	4,17	8,3	ATA
	3	5,56	11,1	ATA-2
	4	8,33	16,6	ATA-2
	5	11,11	22,2	Анонсировано!
DMA (однословный)	0	1,04	2,1	ATA (Убрано в ATA-3)
	1	2,08	4,2	ATA (Убрано в ATA-3)
	2	4,17	8,3	ATA (Убрано в ATA-3)
DMA (многословный)	0	2,08	4,2	ATA
	1	6,67	13,3	ATA-2
	2	8,33	16,6	ATA-2
UltraDMA	0	4,17	16,6	ATA-4
	1	6,25	25,0	ATA-4
	2 (UDMA/33)	8,33	33,3	ATA-4
	3	11,11	44,4	ATA-5
	4 (UDMA/66)	16,67	66,7	ATA-5
	5 (UDMA/100)	25	100,0	ATA-6?

[a] Время цикла ATA обычно представляется в наносекундах, но здесь для легкости сравнения они представлены в мегагерцах.

В табл. 5.3 обобщены реализации IDE и те возможности, которые они поддерживают.

Таблица 5.3. Особенности реализаций IDE

	EIDE	Fast-ATA	Fast-ATA-2	Ultra-ATA
Режимы передачи PIO	До 3	До 3	До 4	Все
Режимы DMA (однословный)	Все	Все	Все	Все
Режимы DMA (многословный)	До 1	До 1	До 2	Все
Множественные операции чтения/записи	Нет	Да	Да	Да
Вычисление контрольной суммы данных	Нет	Нет	Нет	Да

Описанные реализации поддерживаются в ядрах Linux после версии 2.2.0. Для поддержки высокопроизводительного Ultra-ATA можно поставить обновления на более старые ядра. Драйвер IDE ядра Linux, написанный Марком Лордом (Mark Lord), имеет очень маленькие издержки при инициации транзакций. Он замечателен при обслуживании небольших передач данных.

Улучшение производительности IDE в Linux

В Linux существует несколько приемов повышения производительности ввода-вывода IDE, которые в ядре не включены по умолчанию. Например, драйвер устройства IDE может применять 32-разрядный ввод-вывод или взаимодействие с диском через DMA. В ядре 2.2.16 эти возможности по умолчанию выключены. Включать их лучше всего с помощью *hdparm* (см. раздел «hdparm» далее в этой главе).

Проверить, использует ли диск 32-разрядные передачи IDE, можно с помощью ключа *-c*:

```
# hdparm -c /dev/hda
/dev/hda:
I/O support = 0 (default 16-bit)
```

Для включения 32-разрядных передач следует применить *-c 1*:

```
# hdparm -c 1 /dev/hda
/dev/hda:
setting 32-bit I/O support flag to 1
I/O support = 1 (32-bit)
```

Подобным образом с помощью *-d* можно проверить статус режима передачи IDE DMA:

```
# hdparm -d /dev/hda
/dev/hda:
using_dma = 0 (off)
```

Для включения IDE DMA следует применить *-d 1*:

```
# hdparm -d 1 /dev/hda
/dev/hda:
setting using_dma to 1 (on)
using_dma = 1 (on)
```

Некоторого увеличения производительности можно добиться за счет тонкой настройки количества секторов, передаваемых за одно прерывание (устанавливается с помощью ключа *-m* утилиты *hdparm*). Как обычно, изменения следует протестировать, чтобы проверить, улучшилась ли производительность. Когда определены установки, которые нужно сохранить для интерфейса IDE, следует выполнить *hdparm -k /dev/hdX*.

Ограничения устройств IDE

Интерфейс IDE достаточен для небольших приложений, которые не требуют высокой производительности, особенно если они не выполняют большого количества параллельных дисковых операций ввода-вывода, а общая нагрузка на дисковую подсистему относительно невелика.

IDE – это привлекательное, недорогое решение. На время написания этих строк (середина 2001 года) можно отметить, что при одинаковой производительности цена устройств IDE составляет примерно половину стоимости накопителей SCSI, а емкость устройств IDE приблизительно в два раза выше. К сожалению, IDE имеет несколько ограничений. Самые главные из них – это очень ограниченная возможность расширения в расчете на один канал, а также однопоточность каналов (даже с двумя дисками на одном канале перекрытие выполнения команд не поддерживается). В свою очередь, SCSI предлагает гибкое присоединение устройств, а также поддержку почти всех типов периферии и перекрытие выполнения команд. Высокопроизводительные устройства с интерфейсом SCSI обычно появляются значительно раньше, чем устройства IDE.

IPI

После разработки стандарта SCSI был определен второй стандарт массового хранения с высокой производительностью и высокой стоимостью инсталляций, известный как интеллектуальный интерфейс периферийных устройств (Intelligent Peripheral Interface, IPI). Сейчас ясно, что у IPI нет будущего: он был оттеснен своим младшим собратом SCSI, ставшим доминирующим высокопроизводительным дисковым интерфейсом. IPI здесь обсуждается по историческим причинам, а также ради тех, кто имеет существенную базу установленных устройств IPI. Интерфейс IPI, в дополнение к Fibre Channel, подпадает под управление Технического комитета T11 (*http://www.t11.org*).

Стандарт IPI был разработан для того, чтобы максимально облегчить оптимизацию, направленную на достижение самой высокой производительности. С этой целью в протоколе была внедрена глобальная оптимизация. Полная модель IPI состоит из четырех уровней:

- Уровень 0 определяет механические и электротехнические соглашения, такие как кабели и разъемы.
- Уровень 1 определяет основные подробности передачи, такие как шинные протоколы.
- Уровень 2 определяет протоколы дисковых цепочек, которые описывают специфичные для устройств команды, синхронизацию и адресацию томов.
- Уровень 3 определяет протоколы каналов ввода-вывода, которые включают логические команды, правила командных очередей, системы буферизации и т. д.

Полностью скомпонованная система IPI состоит из некоторого количества *хост-адаптеров* IPI-3 (*host adapter*; другое название – *аппаратура*, *facilities*), которые подсоединены к главной системе. Хост-адаптер – это полный компьютер ввода-вывода, отвечающий за управление дисковыми *цепочками* (*string*). Каждая дисковая цепочка включает в себя от одного до восьми дисковых накопителей, которые соединены с хост-адаптером с помощью *контроллера цепочки* IPI-2 (*string controller*). Диск может быть присоединен к нескольким цепочкам. Такое условие известно как *многопортовость* (*multiporting*). Через высокопроизводительную шину IPI-3 с хост-адаптером могут взаимодействовать до шестнадцати контроллеров цепочек. Подобно хост-адаптеру, контроллер цепочки облачен солидными логическими функциями. Он отвечает за оптимизацию доступа к своей цепочке. Шины IPI-3 и IPI-2 можно расширять до пятидесяти метров.

Все логические функции IPI реализованы в хост-адаптере и контроллере цепочки; даже такие простые операции, как отображение неисправных блоков, выполняются контроллером цепочки. У дисковых накопителей интеллект полностью отсутствует. В них может даже не быть локальных кэшей. В итоге, для того чтобы IPI-диск мог осуществить передачу, необходимо выполнить нескольких условий:

- Запрашиваемая цепочка не должна быть занята.
- Контроллер цепочки должен иметь свободный буфер для хранения данных.
- Головка считывания/записи должна быть правильно спозиционирована.
- Данные должны чуть ли не летать под головкой.

Так как шина используется практически в любой передаче и разделяется между всеми устройствами цепочки, то конфликты шины – это серьезная забота для IPI. Вот почему на одной цепочке IPI-2 следует

размещать не более трех дисков. Если скомпоновать все восемь возможных дисков, то производительность значительно снизится.

Для того чтобы предоставить контроллеру цепочки достаточно информации для оптимизации очереди запросов, диски IPI сообщают координату поворота и радиальную координату головок считывания/записи. Каждый контролллер цепочки сортирует серию запросов, выстраивая их в оптимизированном порядке. Иногда контроллеру цепочки нужно извлечь данные с двух дисков, которые готовы к передаче. Тогда канал IPI-3 может передать запрос другой цепочке в случае, если выполняются два условия: диск подсоединен к нескольким цепочкам, а операционная система поддерживает такую возможность.[1] Кстати, большинство реализаций IPI не соответствует спецификации в полной мере. Например, Sun не полностью реализует аппаратуру IPI-3, а вместо этого считает, что каждый контроллер цепочки спарен с одним хост-адаптером. Лучше всего IPI работает под сильной нагрузкой, когда формируются длинные очереди для каждого устройства ввода-вывода. Если очереди короткие или их нет, то логические функции в цепочке IPI остаются невостребованными.

SCSI

Спецификация Small Computer System Interface (SCSI, произносится «скази»)[2] являлась дисковым стандартом для рабочих станций среднего класса последние пятнадцать лет. Ясно, что это доминирующий стандарт ввода-вывода на рынке современных рабочих станций. В настоящее время спецификация SCSI координируется Техническим комитетом T10 (*http://www.t10.org*).

Стандарт SCSI определяет модель периферийной шины, которая соединяет все физические устройства, будучи полностью отделенной от любой другой основной шины, за исключением одной точки пересечения. Эта точка пересечения – *хост-адаптер* (*host adapter*), часто называемый *контроллером SCSI*. Здесь будет применяться термин «хост-адаптер» по причинам, которые скоро будут ясны.

Каждое устройство на шине SCSI имеет уникальный адрес, состоящий из двух частей: *целевого идентификатора* (*target identifier*) и *кода логического устройства* (*logical unit number*, LUN). Аналогом могут служить почтовые ящики. Большинство почтовых ящиков имеет отдельный идентификационный номер (целевой идентификатор), однако в некоторых почтовых ящиках есть отделения, различаемые с помощью

[1] Solaris не поддерживает.

[2] Среди системных администраторов, воюющих с неполадками в последовательности SCSI, ходит шутка: «Этот потрепанный интерфейс снова портачит». («Scuzzy» (англ.) означает «грязный, потрепанный». – *Примеч. перев.*)

LUN (например, обычная совокупность почтовых ящиков для многоквартирного дома имеет один идентификатор и, вероятно, много отдельных почтовых ящиков внутри себя). Скажем, адрес «Некая улица, дом 123, квартира 918» состоит из целевого идентификатора «Некая улица, дом 123» и LUN «918». Целевой объект запроса определяется наличием сигнала на отдельной линии. В результате 8-разрядная шина SCSI имеет восемь возможных целевых идентификаторов.

Изначально предназначенный для небольших систем, протокол SCSI был разработан с расчетом на оптимальную производительность за низкую цену. В отличие от таких конструкций, как IPI, где большинство логических функций делегированы контроллеру, SCSI размещает их на самих периферийных устройствах. Это значительно упрощает периферийные передачи данных. Как правило, они обслуживаются микропроцессором, расположенным на каждом устройстве. Такой микропроцессор называется *встроенным контроллером* (*embedded controller*). Поскольку при такой схеме периферийное устройство осуществляет самоконтроль, а внешний контроллер отсутствует, то нет смысла называть хост-адаптер контроллером SCSI.

Один из участков, где периферийные логические функции значительно упростили операционные процедуры, – это *управление дефектами* (*defect management*). Дисковые накопители представляют собой очень сложные электромеханические устройства, и неисправные блоки неизбежны. В системах с централизованными логическими функциями, таких как IPI, от управляющей аппаратуры требовалось распознавание дисковых ошибок и управление списком дефектов для каждого физического диска. В SCSI такой мониторинг возложен на дисковый уровень. Кроме того, диск может отслеживать некие внутренние события, например определять, когда код управления дефектами уже не может обработать ошибку за счет свободных блоков, и сообщать о надвигающемся отказе хост-адаптеру. Один побочный эффект конструкций с периферийной логикой заключается в том, что сокровенные знания об устройствах расположены на самих устройствах, а не на хост-адаптере. В результате оптимизация должна проводиться на встроенном контроллере, что может быть затруднительным. С точки зрения хост-адаптера, итоговая степень абстракции позволяет произвольно расширять массив устройств, взаимодействующих через SCSI, но затрудняет оптимизацию в больших конфигурациях.

Если два устройства захотят одновременно задействовать шину, то потребуется арбитраж. SCSI применяет простую схему, в которой целевому объекту с большим номером отдается предпочтение. По этой причине хост-адаптеры почти всегда имеют номер 7.[1] Также обычной практикой является назначение медленным устройствам, таким как

[1] Это правда (по причинам обратной совместимости) даже в реализациях Wide SCSI, которые имеют 16 целевых идентификаторов.

ленты или CD-ROM, высоких целевых номеров, чтобы не лишать их полосы пропускания из-за обслуживания быстрых устройств. Однако в реальном мире такой подход не распространен.

Многоинициаторный SCSI

Несмотря на то что SCSI является стандартом, компоновать хост-адаптер как седьмой целевой объект вовсе не обязательно. Это приводит к интересному случаю, называемому *многоинициаторным SCSI*, где шина SCSI разделяется между несколькими системами. Такой подход можно применять для разделения дискового пула между двумя системами, что является полезным для систем с высокой степенью готовности. К сожалению, здесь есть несколько недостатков:

- В стандарте SCSI не описан механизм обслуживания ожидающих операций, которые возникают при сбросе (reset) шины (например, если один инициатор решает выполнить сброс шины, то в системе произойдет беспорядок или неустойчивая электротехническая неисправность). Это означает, что состояние дисков становится неопределенным, а данные могут быть повреждены.
- Хотя для хост-адаптеров стандарт определяет методы «владения» устройством (с помощью команд SCSI *reserve* и *release*), для прерывания резервирования не существует однозначного способа.

Кроме того, существует возможность применять многоинициаторные SCSI для взаимодействия система–система на коротких участках. Это более надежно, чем разделение дисков с высокой степенью готовности, для которого обычно используют многоинициаторные схемы. В Linux такая возможность реализована как модуль ядра.

Шинные транзакции

Операции на шине SCSI достаточно абстрактны. Они представляют собой несколько операций, каждая из которых занимает полосу пропускания. Когда целевой объект[1] намерен инициировать запрос, он должен *захватить* (*arbitrate*) шину. Когда контроль получен, инициатор выбирает исполнителя и затем выдает *команду*. Теперь исполнитель должен решить, сможет ли он немедленно обслужить запрос. Если да (например, запрос для чтения, который можно обслужить из локального кэша диска), то данные пересылаются с *входящим* или *исходящим сообщением*. Если запрос не может быть обслужен немедленно (запрос для чтения, сводящийся к считыванию с пластин), то исполнитель *отсоединяется*. Спустя время исполнитель *захватывает* шину, *восстанавливает соединение* с инициатором, и (если это задано) пере-

[1] Как правило, это хост-адаптер или устройство, желающее взаимодействовать с хост-адаптером, но так бывает не всегда. Некоторые системы (в основном, предназначенные для работы в реальном масштабе времени) позволяют операции внутри устройств.

дает данные с помощью *входящих* или *исходящих сообщений*. Далее исполнитель передает сигнал завершения с помощью *сообщения-статуса* и затем *отсоединяется*. Такая схема включает в себя от шести до восьми изменений состояния на шине. При этом шина занята, за исключением времени между отсоединением и восстановлением соединения. Обычно служебная часть транзакции занимает около 140 мкс. Для сравнения, передача 2 Кбайт данных занимает около 100 мкс. Издержки довольно высоки, так как с предельной пакетной скоростью передаются только данные. Команды и статус всегда передаются в более медленном асинхронном режиме.

Сравнение синхронных и асинхронных передач

Стандарт SCSI определяет два варианта передачи данных: синхронный и асинхронный. Это относится к способу, с помощью которого хост-адаптер и его целевой объект координируют передачу данных. При асинхронных передачах данные могут быть переданы в произвольное время. Получатель и отправитель не должны быть синхронизированы по тактовой частоте. В такой передаче необходимо подтверждать прием каждого байта, что налагает практическое ограничение на скорость передачи. В синхронном режиме передачи могут начинаться только в особой фазе синхронизации, но это ослабляет проблему подтверждения, которая существует при асинхронной передаче. Хост-адаптер старается установить наилучшее соединение с устройствами, когда целевые объекты установлены. Если по какой-то причине синхронное соединение с исполнителем не может быть установлено, то хост-адаптер приступает к асинхронным передачам. В Solaris определить установленную скорость синхронной передачи можно с помощью *prtconf -v*, обратив внимание на драйвер устройства SCSI. Следующий пример приведен для устройства Fast SCSI-2, подсоединенного к рабочей станции Ultra 1:

```
% prtconf -v
...
            esp, instance #0
               Driver properties:
                   name <target0-sync-speed> length <4>
                       value <0x00002710>.
...
```

В этом случае установленная скорость синхронной передачи приведена для устройства, скомпонованного как целевой объект 0. Скорость взаимодействия с контроллером составляет 0x00002710 Кбайт/с. Это число сообщается в шестнадцатеричном формате. Представив его в десятичном формате и разделив на тысячу, получим установленную скорость 10 Мбайт/с.

Зачастую стоит проводить контроль. Применение кабелей невысокого качества или неподходящего оконечного оборудования нередко приводит к откату на более низкую скорость. Именно на эти участки следует

обратить внимание в первую очередь, если производительность последовательности SCSI не соответствует ожидаемой.

Терминаторы

Работа SCSI основана на электротехнических принципах и осуществляется непосредственно через шинный механизм. Трафик от одного устройства «исходит» на шину. Устройство-получатель ответственно за «снятие» с шины тех данных, которые предназначены для него. Однако данные, проходящие мимо последнего устройства последовательности SCSI, могут быть «отражены» обратно по шине. Такое явление отражения вызывает серьезные сложности, которые преодолеваются с помощью терминаторов. Вкратце, терминатор – это дополнительный набор резисторов, добавляемый к устройству с целью предотвращения отражения сигнала. Терминаторы «поглощают» данные, которые в противном случае были бы отражены.[1] Теоретически правила для погашения просты: обрывать распространение на первом и последнем устройствах шины SCSI. Многие современные диски могут при необходимости осуществлять эту операцию автоматически. О проблемах погашения обычно свидетельствует низкая скорость передачи данных или наличие устройств, чья работа не кажется надежной (если они вообще являются надежными). Заметим, что вследствие различий в электротехнических конструкциях дифференциальные и односторонние терминаторы несовместимы (более подробные сведения об их различиях можно получить в разделе «Дифференциальная передача сигналов» далее в этой главе).

Организация очереди команд

Когда разрабатывалась первоначальная спецификация SCSI, производительность рассматривалась во вторую очередь: рынок, для которого предназначался стандарт, был больше заинтересован в низкой стоимости, нежели в высокой производительности. Ко времени выпуска версии SCSI-2 тема производительности стала намного более важной. Для того чтобы оптимизировать запросы (то есть определить оптимальную последовательность их обслуживания), нужно было что-то предпринять.

Однако поскольку каждое устройство SCSI обладает высокой степенью автономности, у хост-адаптера нет достаточных данных для принятия реальных решений по оптимизации. Очевидно, что в такой схеме глобальная оптимизация была невозможна (она реализована в других стандартах, таких как IPI). Положение ухудшилось с принятием стандарта SCSI-1, в котором позволялась только одна команда, ожидающая заданного целевого объекта. Если ожидать обработки может только один запрос, то какая тут может быть оптимизация!

---

[1] Оконечное оборудование – это довольно сложная тема. Полное обсуждение выходит за рамки этой книги.

Для преодоления таких трудностей в SCSI-2 была введена *очередь команд*, в которой целевой объект может накапливать несколько запросов. Далее эти запросы обрабатываются в наиболее полезном порядке. Каждому запросу назначается *тег* (отсюда синоним *очередь тегов*), который служит идентификатором. С помощью тегов каждый объект может ассоциировать заданное взаимодействие с запросом. Такой подход улучшает производительность за счет совмещения шинных транзакций и физических операций. При этом сберегается существенный объем времени, особенно в приложениях со множеством небольших дисковых записей. К сожалению, очередь команд наиболее эффективна при сильной нагрузке на диск. Обычно от распределения нагрузки на диски можно добиться большего эффекта, чем от оптимизации очереди команд. Влияние очереди команд на пропускную способность однопоточных задач, типичных для рабочих станций, ограничено.

Дифференциальная передача сигналов

Представим двух друзей, стоящих на морском берегу, причем один из них хочет послать сообщение другому. У него есть фонарик. Если он держит фонарик на уровне ног, то это означает передачу нулевого сигнала, а если у головы, то передачу единицы. Такая схема хорошо работает, если второй человек может четко определить, где «высоко», а где «низко». Если же он возьмет лодку и отплывет на милю, то определять, когда фонарик вверху, а когда внизу, станет намного сложнее. Однако если первый человек возьмет два фонарика и будет держать их на одной и той же высоте для отображения нуля, а для единицы поднимет один фонарь вверх, а другой разместит внизу, то второму человеку будет намного легче. Первый режим передачи сигналов (с одним фонариком) называется односторонним, а второй – дифференциальным.

Низковольтная дифференциальная (*low-voltage differential*, LVD) передача сигналов – это новый электротехнический стандарт, введенный со спецификацией SCSI-3. По существу, низковольтная дифференциальная передача выполняет то же самое, что и дифференциальная, но за меньшую стоимость.

Дифференциальная передача данных разработана с целью смягчить ограничение диапазона передачи в одностороннем SCSI: односторонние интерфейсы 20 Мбайт/с имеют ограничение по длине до полутора метров. Дифференциальные интерфейсы могут передавать данные на 25 метров. Поскольку такие расстояния подразумевают полный путь прохождения сигнала, включая кабели между блоками, участок на встроенном контроллере и около фута дополнительно для каждого разъема[1], то заданная длина кабеля не всегда настолько велика, как можно подумать.

[1] Вследствие импеданса коннекторов.

Дифференциальная передача сигналов сама по себе не приводит к повышению производительности. В случае когда единственное различие между двумя шинами SCSI состоит в том, что одна применяет одностороннюю, а другая дифференциальную передачу сигналов, эти шины будут работать одинаково.

Использование шины

Обычная рекомендация по производительности – не нагружать диск более чем на 40%. Такой совет основан на дисковых структурах ранних мэйнфреймов, в которых, как правило, отсутствовало кэширование дисков. Поэтому для осуществления транзакции головка считывания/записи должна обращаться к контроллеру – чтобы вообще выполнить любую передачу. Если шина занята, когда данные проходят под головкой, то для выполнения ввода-вывода система должна ждать следующего окна – по меньшей мере 8,3 мс на диске 7200 rpm. Такая ситуация известна как промах *при определении полярной координаты* (*rotational position sense*, RPS). Поскольку в объектах SCSI всегда есть локальные кэши, то можно достичь намного большей интенсивности работы шины до того, как производительность начнет снижаться. Вспомним, что SCSI – это архитектура с большими издержками; при каждом запросе может быть от шести до восьми изменений состояния. При небольших операциях ввода-вывода такие издержки начинают значительно влиять на производительность (занимая 60% общего времени). Поэтому возникает важный вопрос: как высокие накладные расходы SCSI воздействуют на производительность.

Если говорить о производительности в последовательной операции, то время поиска и цикл обращения не существенны – ключевыми являются проблемы использования шины. Для операций ввода-вывода 2 Кбайт на цепочке 20 Мбайт/с производительность находится на уровне 8 Мбайт/с вне зависимости от количества скомпонованных дисков. Шине по силам намного большая производительность. Операции ввода-вывода 64 Кбайт приводят к 95-процентному снижению количества выданных команд и повышению производительности до 17 Мбайт/с. Вот общие рекомендации для конфигурирования дисковых подсистем SCSI для последовательного доступа:

- Компоновать цепочки 20 Мбайт/с следует так, чтобы одновременно было активно не более 4–8 устройств. Большие (~64 Кбайт) операции ввода-вывода насытят шину с 4 устройствами. При маленьких (~2 Кбайт) операциях ввода-вывода в цепочке допустимо 8 устройств. (Следует помнить, что при вводе-выводе 64 Кбайт/с будет более высокая пропускная способность, чем при 2 Кбайт/с. Объем данных, которые может переносить шина, есть сумма объемов данных и команд. При 2 Кбайт/с большую часть передаваемых бит будут составлять команды!)

Дисковый массив, состоящий из нескольких дисковых накопителей, эквивалентен такому же количеству дисковых устройств, напрямую подсоединенных к хост-адаптеру.

Последовательная пропускная способность будет значительно ниже номинальной скорости шины, особенно при вводе-выводе небольших порций данных.

В операциях с произвольным доступом наблюдается совершенно другое явление. Если доступ действительно случайный, то диски должны проводить поиск для каждой операции ввода-вывода. В этом случае пропускная способность дисков *разительно* падает (до уровня порядка 250 Кбайт/с). Очевидно, что с добавлением дополнительных дисков производительность возрастает линейно. Конечно, если для каждого из пятнадцати дисковых накопителей пропускная способность составляет 250 Кбайт/с, то в целом можно достичь лишь около 3,7 Мбайт/с. Поэтому будет удивительно, если возникнут какие-то ограничения пропускной способности шины.

- Если доступ к данным действительно случайный, то производительность шины вообще не будет ограничивающим фактором. Можно безопасно компоновать ~80 устройств на одной шине с пропускной способностью 20 Мбайт/с.
- На практике в условиях произвольного доступа на шине с пропускной способностью 20 Мбайт/с можно компоновать 1 или 2 дисковых массива (от 16 до 64 устройств).

Путаница с различными скоростями устройств SCSI

Обычное разногласие, возникающее при обсуждении SCSI, связано с тем, как на более быстрых устройствах сказывается наличие на шине более медленных устройств. Такой вопрос обычно встает, когда кто-нибудь инсталлирует новый, быстрый диск SCSI последовательно со старым, более медленным диском. Стандарт SCSI разработан тщательно, поэтому управление отдельным устройством максимально изолировано от его окружения. Для каждого устройства хост-адаптер индивидуально устанавливает подходящую скорость пакетной передачи, и работа шины определяется ограничениями для каждого устройства. Однако возникают некоторые сложности. Например, использование шины варьируется в зависимости от скорости передачи, особенно если это асинхронная передача. Более медленные устройства способны оккупировать шину, так как они не могут избрать высокую скорость пакетной передачи для целесообразной работы с шиной. Такое почти никогда не происходит на рабочих станциях. Однако когда в системе присутствует много устройств, то будет хорошим решением поместить медленные устройства на отдельную шину, обеспечивая тем самым хорошую производительность на других участках.

Реализации SCSI

Весной 1979 года компания Shugart Associtates приступила к работе над *Shugart Associates System Interface* (SASI), который впоследствии был представлен широкой публике для того, чтобы производители дисковых контроллеров смогли применять этот протокол. В 1980 году стандарт SASI был представлен комитету X3T9.3 ANSI для того, чтобы SASI был принят вместо IPI (см. раздел «IPI» ранее в этой главе).

Летом 1981 года компания NCR начала исследование возможности применения SASI вместо собственного, недавно утвержденного интерфейса. Эта идея провалилась вследствие опасений, что технические усовершенствования могут вызвать сумятицу на рынке. Однако ранней осенью 1981 года Optimem (дочерняя компания Shugart) предложила те же технические нововведения, что и NCR. В стремлении примириться с неизбежным Shugart повторно обратилась к NCR. Было проведено несколько встреч. В результате SASI стал лучше документирован, и к нему были добавлены несколько возможностей из патентованного интерфейса NCR. В декабре 1981 года NCR и Shugart совместно ходатайствовали об участии X3T9.3 на их следующей встрече в феврале 1982 года, чтобы рассмотреть их совместное предложение. Ответом X3T9.3 явилось разделение встречи на две параллельные сессии: одна для IP и одна для SASI. По завершении этого собрания сессии воссоединились. Было решено, что SASI необходимо передать одной из существующих исследовательских групп, X3T9.2, которая не имела текущего проекта. В апреле 1982 года состоялось совещание X3T9.2. Результатом явилось официальное предложение по проекту Small Computer System Interface (SCSI), которое было основано на SASI. С 1982 по 1984 год встречи X3T9.2 проводились регулярно. SCSI был полностью документирован с участием NCR (предложившей систему команд для лент, процессоров и принтеров) и Optimem (предложившей систему команд для оптической однократной записи и многократного чтения (write-once, read-many, WORM)). В апреле 1983 года NCR анонсировала NCR-5385, первый чип с реализацией протокола SCSI. Вслед за ним в апреле 1984 года был выпущен чип NCR-5380, в который была добавлена существенная аппаратная поддержка для недорогих реализаций SCSI.[1] Также в апреле 1984 года группа X3T9.3 проголосовала за передачу SCSI в ее родительский комитет, чтобы начать долгий процесс утверждения стандарта в ANSI. Утверждение произошло в июне 1986 года, и SCSI официально стал стандартом ANSI X3.131-1986.

В июле 1985 года специальная рабочая группа начала серию встреч в целях определения расширений для системы команд SCSI (для дисков), позднее известных как Common Command Set (CCS). В июне 1986 года комитет X3T9.2 начал проект SCSI-2, призванный добавить под-

[1] NCR-5380 был чипом, первоначально примененным Apple Computer для шины SCSI в Macintosh Plus.

держку для CCS и внести другие улучшения в протокол SCSI. Период с 1986 по 1989 год был временем интенсивной работы. Стандарт SCSI вырос с 200 страниц почти до 600. В феврале 1989 года SCSI-2 был направлен для утверждения в ANSI и утвержден в сентябре 1990 года.

В ходе дальнейшей разработки SCSI, а именно при создании SCSI-3, усилия были распределены между несколькими исследовательскими группами, каждая из которых сосредоточилась на отдельной части протокола.

Хотя SCSI-2 полностью обратно-совместим с SCSI-1, стандарт SCSI-3 не является полностью совместимым со SCSI-2 ввиду наличия существенных аппаратных расширений. Однако программное обеспечение не изменилось.

Стандарт SCSI-3 был принят по частям. Многие производители уже реализовали наиболее совместимые части. Для обзора реализаций SCSI, пожалуйста, обратитесь к табл. 5.4.

Таблица 5.4. Обзор реализаций SCSI

	SCSI-1	**SCSI-2**	**SCSI-3**
Носитель	Медь	Медь	Медь Оптоволокно
Топология	Дерево	Дерево Многоинициаторное дерево	Дерево Многоинициаторное дерево Управляемая петля Оптическая коммутация (fabric)
Передача сигналов	Односторонняя	Односторонняя Дифференциальная	Односторонняя Дифференциальная Низковольтовая дифференциальная
Очередь команд	Нет	Необязательная	Необязательная
Режимы передачи	Синхронный Асинхронный	Синхронный Асинхронный	Синхронный Асинхронный
Синхронизация передачи	5 МГц	5 МГц 10 МГц	5 МГц 10 МГц 20 МГц
Ширина передачи	8-разрядный	8-разрядный 16-разрядный	8-разрядный 16-разрядный

Каково быстродействие устройств, удовлетворяющих этим стандартам? Авторам известны люди, которые строили таблицы, очень похожие на табл. 5.5 и табл. 5.6. Возможно, эти таблицы будут полезны.

Таблица 5.5. Расшифровка вариантов SCSI

	Стандарт	Частота (МГц)	Ширина (разрядность)	Максимальная скорость (Мбайт/с)	Максимальное количество устройств	Максимальная длина[a]
SCSI-1	SCSI-1	5	8	5	8	6.0
Fast	SCSI-2	10	8	10	8	3.0
Wide	SCSI-2	5	16	10	16	3.0
Fast/Wide	SCSI-2	10	16	10	16	3.0
Ultra (fast-20)	SCSI-3	20	8	20	8	Од[b]: 1.5 Д: 12.5
Ultra Wide (fast-40)	SCSI-3	20	16	40	16	Од: 1.5 Д: 12.5
Ultra2	SCSI-3	40	8	40	16	Од: 1.5 Д: 12.5
Ultra2 Wide (fast-80)	SCSI-3	40	16	80	16	Од: 1.5 Д: 12.5
Ultra160	SCSI-3	80	16	160	16	Д: 12.5

[a] В метрах, для синхронных операций. Если иное не указано, только для односторонних конфигураций.

[b] Од – односторонняя передача сигнала, Д – дифференциальная передача сигнала. – *Примеч. перев.*

Таблица 5.6. Расшифровка разъемов SCSI

Разъем	Выводы	Форм-фактор	Применяется для
SCSI-1	50	Разъем D-shell 3 ряда выводов	SCSI-1
DB-25	25	Разъем D-shell 2 ряда выводов	SCSI-1, узкий SCSI-2 Первоначально использовался в Макинтоше
Centronix 50	50	Разъем Centronix	SCSI-1, узкий SCSI-2
Micro 50	50	Разъем micro-type; 2 ряда выводов	Узкий SCSI-2, узкий SCSI-3
Micro 68	68	Разъем micro-type; 2 ряда выводов	Широкий SCSI-2, узкий SCSI-3
SCA	80	Разъем micro-type; 2 ряда выводов	SCSI-2, SCSI-3 Только внутренние присоединения «Универсальный» коннектор SCSI: интегрировано питание и выбор ID
VHDC	68	Разъем очень высокой плотности	Дифференциальный SCSI-3

Таблица 5.6 (продолжение)

Разъем	Выводы	Форм-фактор	Применяется для
HD-60	60	Разъем micro-type с панелью вместо выводов	Узкий SCSI-2 Только старые адаптеры SCSI IBM
HD-68	68	Разъем micro-type с панелью вместо выводов	Широкий SCSI-2 Только старые адаптеры IBM SCSI

Fibre Channel

Спецификация *волоконно-оптического канала* (*Fibre Channel*, FC), определенная ANSI X3T9.3, использует команды, подобные SCSI. Обычно применяется волоконно-оптический кабель, а не медные провода. Стандарты Fibre Channel находятся под эгидой Технического комитета T11 (*http://www.t11.org*).

Fibre Channel – дуплексный носитель. Это означает, что он может поддерживать двунаправленный поток данных. В такой схеме необходима волоконная нить для каждого направления. Хотя Fibre Channel определен собственным стандартом, он взят под крылышко SCSI-3. Fibre Channel имеет три очень различных топологии:

Точка-точка

Одно устройство соединяется только с одним устройством.

Управляемая петля

Часто называемая FC-AL. Устройства и один или несколько хостов соединены в кольцо на базе множества связей точка–точка.

Сетка

Для создания сложной сети применены коммутаторы и хабы. В сети может быть множество путей от заданного хоста до заданного устройства. Такая схема известна как *сеть области хранения* (*storage-area networking*). Вероятно, именно в этом направлении будут развиваться хранилища класса предприятия в будущем. К сожалению, обсуждение производительности сети области хранения (SANS) является слишком сложным для этой книги.

Устройствам Fibre Channel назначаются уникальные адреса, называемые *всемирными именами* (*world wide name*, WWN). Поскольку устройства могут быть соединены в произвольные сетки, обычной практикой является назначение каждому устройству полностью уникального WWN, подобно тому как назначаются адреса в Ethernet. Стандарт Fibre Channel описывает три класса передачи сигналов, которые *не* являются взаимозаменяемыми. Обычно они задаются в терминах скорости передачи данных: 25 Мбайт/с (FC-25), 50 Мбайт/с (FC-50)

или 100 Мбайт/с (FC-100). Первоначально индустрия средств хранения задерживала принятие Fibre Channel до появления реализаций FC-100, однако теперь Fibre Channel широко используется. По существу, диски Fibre Channel должны работать как диски SCSI. При конфигурации следует тщательно выполнять все правила, предписанные производителем.

Хост-адаптеры Fibre Channel обычно имеют два оптических модуля связи, поэтому хост-адаптер может соединяться с двумя независимымы шинами Fibre Channel. Соединением управляет *последовательный оптический коммутатор* (*serial-optical coupler*), или SOC (обычно называемый GBIC). Типичный SOC может обслуживать до 255 ожидающих запросов, что изредка может стать ограничением для интерфейсов с одним соединением и является узким местом для двухпортовых интерфейсов. Для преодоления такого ограничения важно, чтобы в устройствах с интерфейсом FC ввод-вывод осуществлялся большими порциями данных (8–64 Кбайт). В свете этого факта ко второму порту удобно присоединить архиватор или другое мало используемое устройство.

Одно из самых полезных свойств Fibre Channel – это устранение ограничений на протяженность участков, обусловленных пределами медных проводов. Стандарт FC позволяет применять кабель до десяти километров. Благодаря этому можно территориально рассредоточить хранилища, что неоценимо в случае восстановления после аварии.

IEEE 1394 (FireWire)

Подобно Fibre Channel, спецификация *IEEE 1394* оказалась под крылышком SCSI-3. Стандарт IEEE 1394, также известный как *FireWire*, первоначально был разработан в Apple Computer. Он быстро распространился на рынке настольных систем среднего класса.

Безусловно, самое важное при оценке периферии IEEE 1394 – учитывать, что скорость шины приводится в мегабитах в секунду, а не в мегабайтах в секунду, как в сетевых соединениях. В настоящее время существуют варианты 100, 200 и 400 Мбит/с. Стандарт IEEE 1394 поддерживает присоединение до 63 устройств на одну цепочку. Кроме того, он обеспечивает возможность подключения устройств без выключения системы.

IEEE 1394 поддерживает *изохронные* (*isochronous*) передачи данных, которые отличаются от более типичных асинхронных передач данных тем, что в них гарантируется своевременная доставка информации. Когда устройство с изохронной передачей данных, скажем, цифровая видеокамера, подсоединено к шине IEEE 1394, то это устройство захватывает определенную часть шины. Кроме того, шина резервирует 20% всей полосы пропускания для выполнения последовательных команд. Когда происходит полная загрузка шины, эта шина больше не распознает другие устройства, вне зависимости от того, передают ли

эти устройства данные. Кроме того, когда определенное количество устройств подключается к шине, то возникает ситуация, известная как *издержки нескольких устройств* (*multiple device overhead*). Влияние таких издержек растет с увеличением количества подключаемых устройств. Задержка при взаимодействии с каждым устройством, обусловленная реализацией самоконфигурирования и горячей замены, повышается с увеличением их количества.

По существу, интерфейс IEEE 1394 – это замечательное средство для подсоединения устройств к рабочим станциям, особенно если они восприимчивы ко времени (например, цифровое видеопроизводство). Однако IEEE 1394 не в полной мере подходит даже для хранилищ среднего размера, что связано с увеличением издержек из-за наличия множества устройств на шине.

Universal Serial Bus (USB)

Универсальная последовательная шина (Universal Serial Bus), или USB, была разработана в целях замещения последовательного и параллельного портов. В этом контексте благодаря шине USB достигается большое усиление производительности с 250 Кбит/с до 12 Мбит/с, а также улучшение легкости применения, поскольку USB поддерживает «горячие замены». Кроме того, USB имеет хорошее управление питанием, а ее архитектура, основанная на концентраторах, способна поддерживать до 127 устройств.

Вследствие своей относительно высокой максимальной скорости передачи данных (1,5 Мбайт/с) USB подходит для соединения устройств с относительно невысокой полосой пропускания, например для принтеров, флоппи-дисководов, дисководов Zip и маленьких сканеров. Довольно простая конструкция, благодаря которой устройства USB так недороги, также обуславливает их централизованную структуру. Микропроцессор такого устройства должен инициировать каждую транзакцию. В итоге шина USB недостаточно хорошо масштабируема, поэтому не следует применять ее где-либо, кроме как для подсоединения простых настольных периферийных устройств.

Общие проблемы производительности

Существуют три общие проблемы дисковой производительности: дисбаланс ввода-вывода, перегрузка диска вследствие пейджинга и необъяснимо большое время обслуживания бездействующих накопителей.

Сильный дисбаланс ввода-вывода

Диск легко перегрузить очень большим количеством обращений к нему. Ситуация, называемая *сильным дисбалансом ввода-вывода*, про-

исходит, когда на часть дисков ложится больше нагрузки, чем на другие. Влияние избыточного использования ресурсов трудно переоценить. Субъективная диагностика смещения ввода-вывода не представляет сложности. Для этого можно воспользоваться инструментом *iostat*, который подробно обсужден в разделе «Применение iostat» далее в этой главе. В выводе, выдаваемом *iostat*, следует обратить внимание на широкий диапазон времени обслуживания и процентное соотношение занятости дисков в системе.

Теоретически преодолеть трудности из-за дисбаланса ввода-вывода очень просто – необходимо разделить рабочую нагрузку между несколькими дисками. Сделать это можно несколькими способами:

- Разделить рабочую нагрузку на функциональной основе. Скажем, если перегруженный диск содержит две большие базы данных, то одну из них следует переместить на другой диск. Такой подход предполагает постоянную рабочую нагрузку и зачастую неприменим на практике.
- Разделить рабочую нагрузку по пользователям. Если перегруженный диск содержит домашние каталоги группы инженеров и группы документалистов, то каталоги одной из групп следует переместить на другой диск. Как и при функциональной разбивке, такой подход предполагает устойчивую рабочую нагрузку.
- Назначать дисковое пространство по круговой системе. Такая схема работает хорошо, но требует поддержки на уровне приложений. Ядро Solaris использует этот метод для распределения обращений среди нескольких пространств свопинга.
- Объединить несколько дисков. В этой схеме будет перегружен один «горячий» диск, а оставшаяся часть данных будет более доступна.
- Скомпоновать несколько дисков в один блок (stripe). Так реализован массив RAID 0, который будет обсужден более подробно в разделе «RAID 0: разбивка на блоки (striping)» главы 6. Размер чередующихся блоков будет в большей степени определять производительность. Слишком большое чередование не подходит для распределения данных среди многих дисков. Слишком маленькое приведет к тому, что небольшие обращения к данным будут активизировать несколько дисков. Однако такие блоки (stripes) не могут быть увеличены в режиме «горячего подключения», что ограничивает будущее расширение набора дисков. Кроме того, они уязвимы при дисковых неполадках.
- Скомпоновать диски в комбинированный дисковый массив зеркало/блок. Это самая быстрая и надежная конфигурация хранения. Но она может быть очень дорогой.
- Скомпоновать массив с защитой по четности (обычно RAID 5; см. раздел «RAID 5: разбивка на блоки с участками четности, распределенными по ширине блока» в главе 6). Это дешевый и надежный

способ, однако он может вызвать существенное снижение производительности.

У этой проблемы нет простого решения. Все зависит от конкретной системы и от бюджета.

Взаимодействия память–диск

Пейджинг и свопинг предназначены для высвобождения памяти за счет использования дискового пространства в качестве вспомогательного места хранения. По существу, проблема памяти с непосредственной выборкой (immediate memory) сводится к возможным трудностям при конфигурировании дисков. Для большей информации о механизме пейджинга см. раздел «Пейджинг и свопинг» главы 4. Память значительно быстрее диска, поэтому в ситуации, когда системе необходимо интенсивно использовать пространство свопинга, относительно медленная дисковая подсистема может стать серьезным узким местом. В идеале пространство свопинга следует разместить на быстром диске с изолированным контроллером, которые больше ни для чего не используются.

При наличии нескольких областей свопинга ядро Solaris предоставляет простой, автоматический и эффективный алгоритм круговой системы, призванный распределять ввод-вывод по нескольким дискам. Подобную оптимизацию ядро Linux выполняет среди областей свопинга равного приоритета. Файл */etc/fstab* в Linux обуславливает разбиение пространства свопинга на три части:

```
/dev/sda1       swap      swap      pri=1
/dev/sdb1       swap      swap      pri=1
/dev/sdc1       swap      swap      pri=1
```

Однако зеркалирование дискового пространства может дать существенное повышение надежности. Такое усиление надежности отражается на производительности благодаря большему количеству операций записи.

Длительное время обслуживания

Иногда диски ведут себя очень странно. Диск с невысокой нагрузкой зачастую может иметь чрезвычайно длительное время обслуживания (сотни миллисекунд). Диск никак не может быть перегружен. Тогда в чем же причина? Все сводится к ошибкам округления на низких уровнях активности, термическим повторным калибровкам или размещению файловых систем. Адриан Кокрофт (Adrian Cockroft) из Sun Microsystem подробно проанализировал ситуацию. Выяснилось, что основная причина лежит в процессах *fsflush* и *bdflush*, сбрасывающих данные из памяти на диск (см. раздел «Записи из кэша файловой системы: fsflush и bdflush» в главе 4). Хотя это и выглядит необычным, такие явления не доставляют хлопот.

Файловые системы

Существуют два вида современных файловых систем: системы, *основанные на блоках* (*block-based*) и *на экстентах* (*extent-based*). Косвенная файловая система, основанная на блоках, разбивает все файлы на *блоки данных* (*data blocks*) по 8 Кбайт и распределяет их по диску. Дополнительные 8 Кбайт *косвенных блоков* (*indirect blocks*) хранят информацию о расположении блоков данных. Для файлов размером более чем несколько мегабайт сведения о косвенных блоках хранятся в *двойных косвенных блоках* (*doubly indirect blocks*). Когда создается файл, используется минимальное количество блоков. Когда файл расширяется, дополнительные блоки берутся из совокупности неиспользованных. Дисковая подсистема размещения старается оптимизировать такие пространственные соотношения, поэтому большинство обращений к файлам являются смежными и не требуют широкого поиска. Однако каждый раз, когда назначается блок, данные должны быть записаны на диск. Метаданные всегда записываются синхронно безо всякого кэширования. Операции с метаданными могут оказывать существенное влияние на производительность системы ввода-вывода именно потому, что они синхронные.

Файловая система, основанная на экстентах, определяет файлы в контексте *экстентов*. Экстент содержит начальную точку и размер. Если записывается файл размером 1 Гбайт, то это один экстент размером 1 Гбайт. Косвенных блоков нет: последовательно читается экстент, а затем все данные. Это означает, что при последовательном доступе к файлам издержки с метаданными очень малы. К сожалению, файловая система, основанная на экстентах, подвержена серьезным осложнениям из-за фрагментации. После создания и удаления множества файлов трудно найти пространство для размещения большого файла. Поэтому для всех файловых систем существуют инструменты для дефрагментирования. Заметим, что дефрагментирование файловой системы по понятным причинам сводится к интенсивному вводу-выводу. Значит, время проведения такой операции следует выбирать, учитывая наличие критических производственных задач. На рис. 5.2 показано сравнение файловых систем, основанных на блоках и на экстентах.

vnodes, inodes и rnodes

Обычно в системах UNIX *индексные дескрипторы* (*inodes*) используются для хранения информации, необходимой для доступа к файлу (скажем, размер файла, владелец, права доступа, расположение блоков данных и т. д.). Системы System V, включая Solaris, применяют абстракцию более высокого уровня, называемую *виртуальным индексным дескриптором* (*virtual node*), или *vnode*. Это упрощает реализацию новых файловых систем: ядро работает с виртуальными дескрипторами, каждый из которых содержит какой-либо индексный

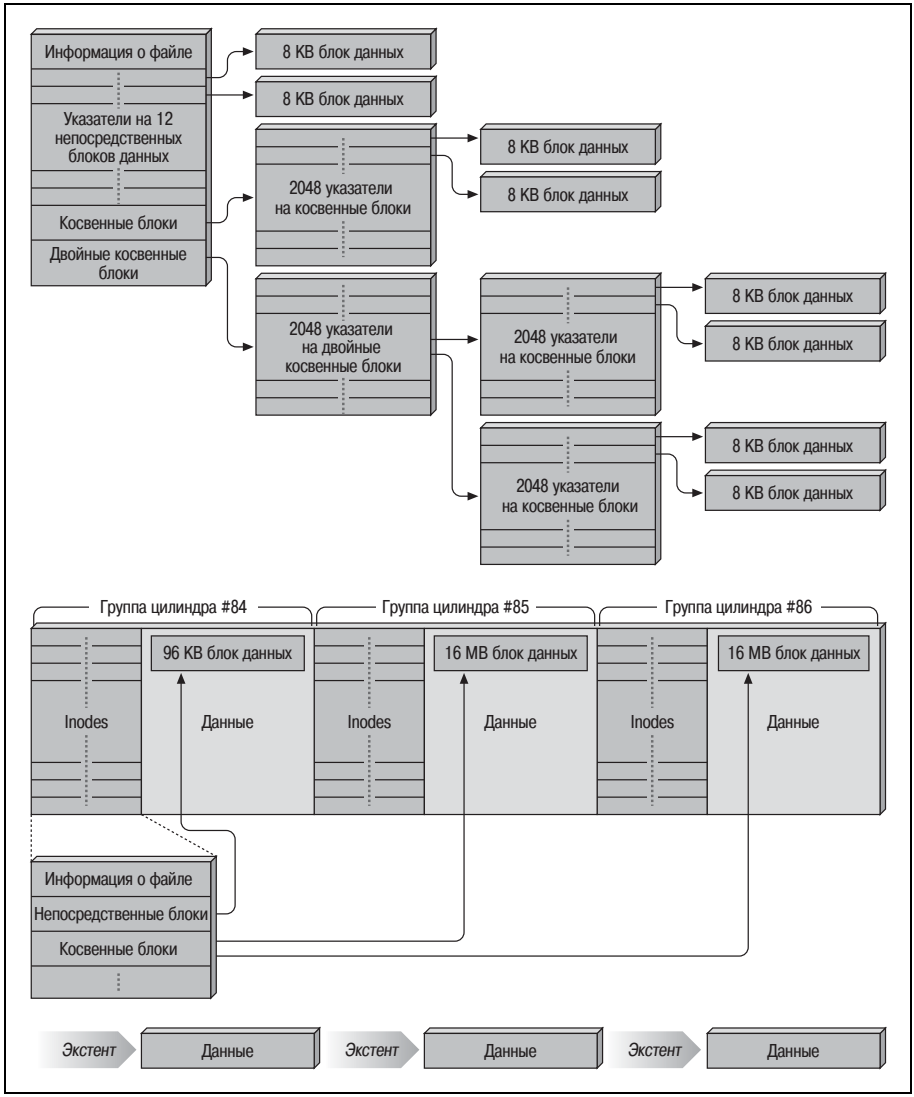


Рис. 5.2. *Сравнение файловых систем, основанных на блоках и на экстентах*

дескриптор с базовым временем файловой системы. В UFS такие структуры по-прежнему называются индексными дескрипторами. В NFS они называются *удаленными дескрипторами* (*rnode*).

Кэш поиска имен каталогов (DNLC)

Каждый раз при открытии файла в дело вступает *кэш поиска имен каталогов* (*directory name lookup cache*), или DNLC. Это кэш низкого уровня в памяти ядра, который хранит данные о соответствии между

именами файлов и виртуальными индексными дескрипторами. При открытии файла на основе его имени DNLC вычисляет соответствующий индексный дескриптор. Если имя находится в кэше, то оно быстро находится, поскольку не происходит сканирования каталогов. Каждая запись DNLC имеет фиксированный размер, поэтому пространство для путевого имени (например, */etc* и *passwd* – отдельные компоненты путевого имени */etc/passwd*) не может превышать 30 знаков, а более длинные имена файлов не кэшируются. При наличии каталогов с тысячами записей поиск конкретного файла часто отнимает много времени. Поэтому если открыто много файлов, а каталоги достаточно большие, то высокая частота успешных обращений к DNLC очень важна. В итоге следует стремиться к тому, чтобы названия часто используемых каталогов и символических ссылок были менее 30 знаков. Обычно на практике это не представляет трудности.

Как правило, настраивать DNLC нет необходимости, если речь не идет о службе NFS на машине с памятью менее 512 Мбайт. В таком случае хорошим выбором будет значение 8000 или около того. Максимальное протестированное значение равно 34906, что соответствует значению 2048 параметра `maxusers`. Для заданной системы значение по умолчанию определяется как (`maxusers` × 17) + 90 с максимальным значением 17498 без явной настройки. (Параметры, на которые оказывает влияние `maxusers`, обсуждены в разделе «Параметры maxusers и pt_cnt» главы 2).

Команда *vmstat -s* показывает частоту успешных попаданий в DNLC с начала работы системы. По существу, показатель менее 90% требует обратить на себя внимание:

```
% vmstat -s |grep 'name lookups'
 34706339 total name lookups (cache hits 98%)
```

Количество проведенных поисков в секунду можно определить, если проанализировать поле `namei/s` в выводе команды *sar -a*:

```
% sar -a 1 5
...
16:44:54  iget/s namei/s dirbk/s
16:44:55     111     603     203
```

Для индексных и удаленных дескрипторов существует два подобных кэша. Они будут обсуждены в разделах, относящихся к соответствующим файловым системам (раздел «Кэш индексных дескрипторов» далее в этой главе и раздел «Кэш удаленных дескрипторов» в главе 7).

UNIX Filesystem (UFS)

UNIX Filesystem (UFS) – это файловая система Solaris по умолчанию. Это косвенная файловая система, основанная на блоках. Каждый

файл, хранимый в файловой системе, на самом деле состоит из нескольких компонентов:

- *Запись каталога* (*directory entry*), содержащая имя файла, его маску прав доступа и время создания, модификации и доступа к файлу.
- *Блоки данных* (*data blocks*), содержащие данные.
- *Записи контрольного листа доступа* (*access control list entries*, необязательные), которые предлагают улучшенный механизм для контроля доступа по сравнению с типичной моделью прав доступа UNIX.

При создании файловой системы для индексных дескрипторов выделяется отдельное пространство. Индексные дескрипторы хранят записи каталога и контрольного листа доступа. Количество индексных дескрипторов – это статическая величина, которая не может быть увеличена без пересоздания файловой системы. Количество свободных индексных дескрипторов можно определить, проанализировав вывод команды *df -e*:

```
% df -e
Filesystem              ifree
/proc                     862
/dev/dsk/c0t0d0s0      294838
fd                          0
/dev/dsk/c0t0d0s4      246965
swap                     9796
/dev/dsk/c1t0d0s0     3256077
```

Плотность индексных дескрипторов

По умолчанию файловая система UFS назначает один индексный дескриптор на каждые 2–8 Кбайт используемого дискового пространства (табл. 5.7).

Таблица 5.7. Плотность индексных дескрипторов по умолчанию

Размер файловой системы (Гбайт)	Количество Кбайт на индексный дескриптор
< 1	2 048
< 2	4 096
< 3	6 144
> 3	8 192

Каждый индексный дескриптор потребляет небольшой объем дискового пространства (512 Кбайт), однако общий объем быстро нарастает. Один индексный дескриптор на 16 Кбайт дискового пространства – вот самая рациональная разбивка по умолчанию в большинстве случаев. Если файловая система хранит очень большие файлы, например файлы, относящиеся к научным задачам, то подойдет значительно мень-

шая плотность – до одного индексного дескриптора на мегабайт дискового пространства. Подобным образом, в файловых системах с большим количеством очень маленьких файлов размещение, заданное по умолчанию, будет хорошим решением. Замечательный пример – файловая система, содержащая спул Usenet, в которой один файл создается для каждой статьи новостей (обычно очень короткой).

Плотность индексных дескрипторов можно указать, если запустить *newfs* с ключом *-i bytes-per-inode*.

Размер кластера файловой системы

Когда файл расширяется, файловая система при любой возможности группирует данные блоками по 8 Кбайт. При таком алгоритме за одну операцию на диск можно записать целый кластер. Это привносит два важных преимущества:

- С точки зрения операционной системы одна операция записи более эффективна, так как затраты на ввод-вывод постоянны вне зависимости от того, сколько данных записывается.
- При следующем чтении файла за одну операцию можно прочитать целый кластер. Это снижает издержки ввода-вывода, и, что более важно, снижает время поиска по диску между извлечениями каждого блока.

Количество смежных блоков файловой системы, которые будут сохранены на диске для одного файла, определяется параметром файловой системы `maxcontig`. В версиях, предшествующих Solaris 8, этот параметр по умолчанию составлял семь дисковых блоков, или 56 Кбайт. Такое значение по умолчанию связано с ограничениями конструкции в более раннем оборудовании Sun.[1] В Solaris 8 значение по умолчанию изменилось и составляет 128 Кбайт (16 блоков). Изменение этого параметра сильно влияет на последовательные операции. Однако в файловых системах с произвольным доступом этот параметр не оказывает серьезного влияния, так как при каждом доступе целые кластеры обычно не считываются. Но даже если файловая система характеризуется произвольным доступом, то следует помнить, что процесс резервного копирования данных по сути является последовательным: большой размер кластера может существенно ускорить резервное копирование. Наибольшее значение `maxconfig` достигнуто в устройствах RAID 5, так как при значении по умолчанию каждая запись фактически обеспечивает очень дорогую операцию чтения/модификации/записи (см. раздел «RAID 5: разбивка на блоки с участками четности, распределенными по ширине блока» в главе 6).

[1] А именно системы, основанные на архитектуре `sun4`, не способны передавать за одну операцию более 64 Кбайт. В архитектурах `sun4c`, `sun4m`, `sun4d` и `sun4u` таких ограничений нет.

Размер кластера файловой системы задается с помощью ключа *-C blocks* команды *newfs* при создании файловой системы. Для существующей файловой системы его можно изменить с помощью ключа *-a* команды *tunefs*.

Настройка параметра `maxcontig` после создания файловой системы не дает того же результата, что и настройка `maxcontig` при ее создании. При настройке после создания файловой системы блоки, которые уже были записаны, не будут перекомпонованы.

По существу, значение `maxcontig` следует установить кратным размеру блока (stripe), если файловые системы расположены на устройствах с чередованием данных или массивах RAID 5.[1] Например, в 6-дисковом томе RAID 5 с размером чередования 32 Кбайт ширина блока равна 160 Кбайт (так как один диск используется для контроля четности), а рациональное значение `maxcontig` составит 160 или 320 Кбайт (20 или 40 блоков соответственно). В файловых системах, содержащих очень большие файлы с последовательным доступом, значение `maxcontig` следует установить равным по крайней мере двойной ширине данных устройства хранения.

Кроме того, чтобы извлечь наибольшую пользу от установки значения `maxcontig` более 128 Кбайт, необходимо настроить максимальный размер передачи ввода-вывода. Для этого нужно установить значения параметров ядра `maxphys`, `md_maxphys` и `vxio:vol_maxio` в */etc/system*. Первые два параметра задаются в байтах, а `vxio:vol_maxio` — в единицах по 512 байт.

При рабочей нагрузке с произвольным доступом применимы другие правила. В целом, необходимо соотнести размеры ввода-вывода и кластера файловой системы, а затем предусмотреть самый большой кэш файловой системы (если этот параметр настраивается) и запретить опережающую выборку и считывание.

Минимальное свободное пространство

При создании файловой системы некоторая часть пространства хранения резервируется для будущих непредвиденных обстоятельств. Эта доля задается параметром `minfree` и исторически составляет 10% файловой системы. Такая схема восходит к компьютерному прошлому. Кроме предоставления разумного пространства на случай «переполнения», такой участок также обеспечивал возможность системы легко найти свободный дисковый блок, когда диск почти заполнен. В современных средах, где файловые системы размером 200 Гбайт вполне обычны, 10% свободного объема составляет 20% дискового про-

[1] Следует помнить, что ширина данных одного диска в массиве RAID 5 меньше, чем количество дисков в массиве.

странства. Так как свободные блоки легко разместить даже в области 20 Мбайт, то первоначальные концепции производительности также являются предметом дискуссии.

Чтобы минимизировать такие издержки, значение `minfree` следует установить равным 1%. При создании файловой системы минимальное свободное пространство можно задать с помощью ключа *-m percentage* команды *newfs*.

Задержка поворота

Параметр задержки поворота (*rotational delay parameter*) – это возврат к дням, когда дисковые контроллеры не могли обрабатывать данные с той скоростью, с которой эти данные могли предоставить дисковые пластины. Для таких случаев была разработана стратегия нестандартного физического хранения блоков, при которой два блока сохранялись с достаточным радиальным расстоянием между ними. Таким образом, головка диска подходила к расположению второго блока лишь после завершения обработки первого блока.

Например, рассмотрим диск, который вращается с оборотом 11 мс (около 5400 rpm), причем контроллеру требуется 2 мс для обработки каждого блока. Если блоки данных последовательны, то к тому времени, когда первый блок считывается и обрабатывается, второй блок уже прошел под головками, а значит, приходится ждать, когда пластина снова вернется в ту же позицию: обработка считывания занимает свыше 20 мс. Если блоки физически удалены друг от друга на четверть оборота (2,75 мс), то контроллер будет обрабатывать второй блок, как только он появится, а полная операция займет около 9 мс. В наши дни контроллеры достаточно быстры, чтобы не думать о таких сложностях, поэтому лучше всего параметру `rotdelay` присвоить нуль, что и сделано по умолчанию.[1]

Задержку поворота можно установить с помощью ключа *-d rotational-delay* команды *newfs* при создании файловой системы.

fstyp и tunefs

Сведения о конкретной файловой системе можно получить с помощью команды *fstyp*. Вывод этой команды довольно большой, полезными являются первые 18 строк:

```
% fstyp -v /dev/dsk/c0t0d0s3 | head -18
ufs
magic   11954   format  dynamic time    Fri Feb 16 15:01:02 2001
sblkno  16      cblkno  24      iblkno  32      dblkno  456
sbsize  5120    cgsize  5120    cgoffset 72     cgmask  0xffffffe0
```

[1] При использовании магнитооптических дисков или других медленных носителей установка `rotdelay` может понадобиться.

```
ncg     272     size    7790674 blocks  7670973
bsize   8192    shift   13      mask    0xffffe000
fsize   1024    shift   10      mask    0xfffffc00
frag    8       shift   3       fsbtodb 1
minfree 1%      maxbpg  2048    optim   time
maxcontig 16    rotdelay 0ms    rps     120
csaddr  456     cssize  5120    shift   9       mask    0xfffffe00
ntrak   27      nsect   133     spc     3591    ncyl    4339
cpg     16      bpg     3591    fpg     28728   ipg     3392
nindir  2048    inopb   64      nspf    2
nbfree  886360  ndir    2149    nifree  888732  nffree  6320
cgrotor 249     fmod    0       ronly   0       logbno  1216
fs_reclaim FS_RECLAIM
file system state is valid, fsclean is -3
```

Наиболее важными являются поля, обсужденные ранее: `minfree`, `maxcontig` и `rotdelay`.

С помощью команды *tunefs* можно настраивать файловую систему после ее создания.[1] По существу, можно настраивать все три ранее обсужденных атрибута: `maxcontig` через ключ *-a*, `rotdelay` через ключ *-d* и `minfree` с помощью *-m*. Например, изменим максимальное количество последовательных блоков в файловой системе:

```
# fstyp -v /dev/dsk/c0t0d0s3 | grep maxcontig
maxcontig 16    rotdelay 0ms    rps     120
# tunefs -a 32 /dev/dsk/c0t0d0s3
maximum contiguous block count changes from 16 to 32
# fstyp -v /dev/dsk/c0t0d0s3 | grep maxcontig
maxcontig 32    rotdelay 0ms    rps     120
```

Выключение кэширования файлов в памяти

Если последовательный ввод-вывод выполняется на очень больших файлах (диапазон от сотен мегабайт до гигабайт) или существует совершенно произвольная модель ввода-вывода на маленьких файлах, то можно отключить буферизацию, обычно проводимую кодом файловой системы. В файловой системе UFS выключить буферизацию можно с помощью параметра `directio` команды *mount_ufs(1M)*. Однако в UFS по-прежнему существуют ограничения косвенных блоков, поэтому применение `directio` не настолько эффективно, как настоящий прямой доступ. Так как непосредственный ввод-вывод обходит кэш файловой системы, то опережающее чтение файловой системы тоже выключено. Небольшие считывания и записи сводятся ко многим раздельным запросам ввода-вывода, обращенным к устройствам хранения, а не группируются в более крупные запросы. Другой побочный эффект непосредственного ввода-вывода состоит в том, что он не налагает на-

[1] «Вы можете настраивать файловую систему, но не можете настраивать рыбу (fish)» – *BSD tunefs manpage*.

грузки на подсистему памяти Solaris и устраняет операции пейджинга.[1] Это было типично для систем, предшествующих Solaris 8, в которых не было приоритетного пейджинга (см. раздел «Взаимодействия между кэшем файловой системы и памятью» главы 4).

Кэш индексных дескрипторов

Одним из интересных кэшей ядра, имеющих отношение к файловой системе UFS, является кэш индексных дескрипторов. При любой операции над файлом, расположенным в файловой системе UFS, необходимо прочитать индексный дескриптор. Частота чтений индексных дескрипторов сообщается как `iget/s` в выводе команды *sar -a*. Данные чтения индексных дескрипторов с диска кэшируются на случай, если они будут снова нужны. Когда файл не используется, его данные кэшируются в памяти с помощью неактивной ячейки кэша индексного дескриптора (каждая ячейка этого кэша имеет страницы для соответствующего файла, связанного с ней). Если неактивная ячейка с присоединенными к ней страницами запрошена для последующего использования, то эти страницы высвобождаются и помещаются в свободный список. Такие сведения представлены в поле `%ufs_ipf` вывода команды *sar -g*. Любое сообщенное ненулевое значение указывает, что кэш индексных дескрипторов слишком мал.

Алгоритм кэширования индексных дескрипторов реализован на основе хранения «списка повторного использования» пустых индексных дескрипторов. Они доступны в любой момент. Количество активных индексных дескрипторов не ограничено, а количество бездействующих дескрипторов (неактивных, но кэшированных) находится в диапазоне между значением `ufs_inode` и 75% значения `ufs_inode` за счет очистки потока ядра. Единственный верхний предел – это объем памяти ядра, потребляемой индексными дескрипторами. Тестированный верхний предел составляет 34 906, что соответствует значению `maxusers`, равному 2 048. Для того чтобы получить приблизительную статистику, касающуюся кэша индексных дескрипторов, следует запустить *netstat -k inode_cache* или *kstat -n ufs:0:inode_cache*:

```
% netstat -k inode_cache
inode_cache:
size 22207 maxsize 68156 hits 1044055 misses 23085 kmem allocs 22207 kmem frees0
maxsize reached 22218 puts at frontlist 95088 puts at backlist 544
queues to free 45284 scans 2558333385 thread idles 0 lookup idles 0 vget idles 0
cache allocs 23085 cache frees 880 pushes at close 0
```

Если значение `maxsize_reached` больше, чем `maxsize` (который равен `ufs_ninode`), то это означает, что количество активных индексных де-

[1] Напомним, что в системах Solaris до версии 7 включительно при нехватке оперативной памяти для файлового кэша производился пейджинг для освобождения нужного количества страниц памяти. – *Примеч. науч. ред.*

скрипторов в какой-то предшествующий момент превысило размер кэша. Вероятно, в таком случае стоит увеличить `ufs_ninode`.

Буферный кэш

Второй механизм кэширования, применяемый в UFS, называется *буферным кэшем*. В клонах BSD UNIX буферный кэш используется для кэширования всего дискового ввода-вывода. В системах UNIX SVR 4, таких как Solaris, такой кэш применяется только для кэширования индексных дескрипторов, косвенных блоков и дискового ввода-вывода, связанного с группами цилиндров.[1] Параметр `nbuf` управляет количеством выделенных буферов страничного размера, а параметр `p_nbuf` задает количество буферов, выделяемых за раз. По умолчанию параметр `p_nbuf` равен 100. Параметр `bufhwm`, значение по умолчанию которого составляет 2% системной памяти, определяет максимальный объем памяти, который может использовать буфер. Размер буферного кэша в системе можно увидеть с помощью */usr/bin/sysdef | grep bufhwm*:

```
# /usr/sbin/sysdef | grep bufhwm
20914176        maximum memory allowed in buffer cache (bufhwm)
```

Чтобы получить статистику работы буферного кэша, следует запустить *kstat unix:0:biostats*:[2]

```
% kstat unix:0:biostats
module: unix                            instance: 0
name:   biostats                        class:    misc
        buffer_cache_hits               40596836
        buffer_cache_lookups            40707195
        buffers_locked_by_someone       8221
        crtime                          49.729349302
        duplicate_buffers_found         0
        new_buffer_requests             0
        snaptime                        3630491.75512563
        waits_for_buffer_allocs         0
```

Сравнение удачных попаданий в буферный кэш и количества поисков свидетельствует, что показатель удачных попаданий с начала работы системы составляет 99,981%, что вполне разумно. Вопреки советам в некоторых прошлых публикациях Sun, параметры буферного кэша не следует настраивать.

---

[1] Структуры групп цилиндров есть не во всех типах файловых систем. Получить более подробную информацию об их применении можно, например, в статье по адресу *http://www.osp.ru/os/1999/07-08/13.htm*. – *Примеч. науч. ред.*

[2] Команда *kstat* добавлена в Solaris недавно. Исторически для *netstat* существовал недокументированный ключ *-k*, позволявший получить такую статистику. При отсутствии в системе *kstat* можно попробовать *netstat -k biostats*.

Если коэффициент удачных попаданий в буферный кэш меньше 90%, а размер файловой системы более чем в 40 000 раз превышает емкость физической памяти (скажем, есть 1 Гбайт памяти, а размер файловой системы ~40 Тбайт), то величину буферного кэша следует увеличить.

Файловые системы с протоколированием

Для того чтобы обладать запасом прочности, требуемым в современных системах, файловая система должна надежно хранить приложения, а в случае аварии необходимо быстро восстановить исходное состояние информации. Такое случалось не всегда: аварии системы обычно приводили к неустойчивому состоянию файловых систем, а восстановление их целостности занимало длительное время.[1] Один из подходов к преодолению таких трудностей – это применение механизма *протоколирования* (*logging*; другое название – *журналирование*, *journaling*). Такая методика предотвращает разрушение файловой системы за счет принятия лишь тех изменений, которые фиксируются в последовательном протоколе.

Протоколирование позволяет поддерживать верное состояние файловой системы, поэтому вне зависимости от неполадок с питанием или других аварий состояние файловой системы в точности известно. Вместо того чтобы сканировать всю файловую систему (с помощью *fsck*), достаточно проверить протокол и, если необходимо, внести поправки согласно последним нескольким записям протокола. Это может означать разницу между 20 секундами и 48 часами проверки целостности файловой системы, во время которой систему почти никогда нельзя использовать. Однако протоколирование не даруется просто так. Оно обуславливает большое количество записей, проводимых синхронно. Протоколирование метаданных требует по крайней мере трех записей при каждом обновлении файла. Если думать только о производительности (к черту шансы потерять данные, да здравствует предельная скорость!), то протоколирование не есть лучший выбор. Однако при создании высокопроизводительного сервера с групповыми транзакциями надежность файловых систем заслуживает компромисса с производительностью.

Существует три типичных способа протоколирования:

Протоколирование метаданных

В протокол записываются только изменения структуры файловой системы. Это самый распространенный механизм.

Протоколирование файлов и метаданных

В протокол записываются все изменения в файловой системе.

[1] Для больших файловых систем со множеством файлов это занимало много часов.

Протоколирование структуры

Вся файловая система есть протокол.

В файловой системе без протоколирования несколько разорванных синхронных записей способны привести к ее изменению. Если авария произошла в середине операции, то состояние файловой системы неизвестно. Необходимо проверять целостность всей файловой системы. Протоколирование метаданных основано на циклическом обновлении протокола только за счет новых записей, когда в протокол записывается состояние каждой транзакции. Перед модификацией любой дисковой структуры в протокол попадает запись о «намерении изменить данные». Затем изменяется структура каталога, а запись об окончании помещается в протокол. Так как каждое изменение состояния при обработке любой транзакции записывается в протокол, то целостность файловой системы можно проверить, просканировав протокол, – проверка всей файловой системы не требуется. Если найдена запись о намерении изменить данные и нет соответствующей записи об окончании операции, то нужно проверить задействованный блок и внести поправки, если это необходимо. Одни алгоритмы позволяют вести такой протокол на диске, отделенном от файловой системы, а другие размещают его в том же разделе, где находятся данные.

Несколько файловых систем позволяют хранить в протоколе не только метаданные, но и сами данные. Данные записываются в протокол, а затем *воспроизводятся* в файловой системе. Такой метод обеспечивает целостность данных вплоть до записи последнего блока и способствует производительности при небольших записях. Маленькая запись в системах без протоколирования данных подразумевает два поиска и одну запись: первый поиск для самих данных и второй для записи в протокол. Помещение данных в протокол снижает количество шагов до одного.

Файловые системы с протоколом структуры (также известные как файловые системы *с повсеместной записью*, *write-anywhere filesystems*) всю структуру файловой системы реализуют в виде протокола. В такой среде запись в файловую систему добавляет блоки данных к концу протокола. После того как это происходит, более ранние блоки помечаются как недействительные. Такой подход позволяет последовательно записывать каждый файл вне зависимости от порядка блоков, что делает запись *очень* быстрой. К сожалению, при этом снижается производительность считывания, так как блоки расположены в порядке записи. Это может означать, что файлы в большей степени раскиданы по диску. Кроме того, в файловых системах с протоколом структуры нужен механизм для сканирования файловой системы и удаления недействительных блоков. Также необходимо кэшировать положение блоков данных, так как они могут быть в произвольном порядке. Это кэширование очень сложное. Такие файловые системы плодотворны в средах с интенсивной обработкой метаданных и не эффек-

тивны в средах с интенсивной обработкой самих данных. По существу, в настоящее время файловые системы с протоколом структуры авторам неизвестны.

Solstice:DiskSuite

Пакет Solstice:DiskSuite компании Sun предлагает расширение, известное как устройство *metatrans*, которое добавляет механизм протоколирования транзакций к существующей структуре файловой системы. Он предоставляет возможность ведения только протокола метаданных. Протокол можно отделить от файловой системы. В средах без энергонезависимого кэша этот пакет может стать дешевым (хотя и более медленным) эквивалентом такого кэша.

Solaris

В системах Solaris начиная с версии 7 существует способ реализовать протоколирование без применения пакета DiskSuite. Такой механизм просто использовать, и он не требует никаких изменений в файловой системе. Нужно лишь смонтировать файловую систему с помощью команды *mount*, запускаемой с ключом *-o logging*:

```
# mount -F ufs -o logging /dev/dsk/c1t0d0s0 /mnt
```

Также это можно делать во время загрузки:

```
# device          device     mount   fs       fsck  mount    mount
# to-mount        to fsck    point   type     pass  at boot  options
/dev/dsk/c1t0d0s0 /dev/rdsk/c1t0d0s0
                             /mnt    ufs      2     yes      logging
```

Вторая расширенная файловая система (EXT2)

Вторая расширенная файловая система (Second Extended Filesystem), изобретенная Реми Кардом (Remy Card), на момент написания этих строк используется во всех поставляемых дистрибутивах Linux. Это косвенная файловая система, основанная на блоках, которая довольно похожа на UFS.

Количество свободных индексных дескрипторов в разделе ext2fs можно определить с помощью *df -i*:

```
% df -i
Filesystem            Inodes   IUsed   IFree IUse% Mounted on
/dev/hda1             513024  111953  401071   22% /
```

Единственная и самая большая сложность с ext2fs состоит в том, что по умолчанию она записывает метаданные асинхронно. Это означает, что неполадки с питанием и другие внезапные, необычные остановы системы часто вызывают существенное повреждение файловой системы. На это налагается отсутствие в ext2fs возможности вести протоко-

лы. Если протоколирование необходимо, то следует применять файловую систему JFS IBM, третью расширенную файловую систему (ext3) или файловую систему Рейзера (Reiser).

Третья расширенная файловая система (ext3)

Третья расширенная файловая система (Third Extended Filesystem, ext3fs) – это переработанная версия ext2fs. Наиболее существенное улучшение связано с поддержкой журналирования. Поддерживается возможность журналирования только метаданных, а также возможность журналирования файловых данных. Extfs3 может мирно сосуществовать с ext2: файловые системы ext3 могут быть монтированы как файловые системы ext2 (правда, без поддержки журналирования). Кроме того, несмотря на запись части данных более одного раза, файловая система ext3 демонстрирует более высокую по сравнению с ext2 пропускную способность, так как процесс журналирования оптимизирует движение головок считывания/записи жесткого диска. Как и следовало ожидать, внимание здесь будет сосредоточено на настройке производительности ext3.

Вследствие журналирования всех изменений метаданных в файловой системе ext3, затраты на обновления могут оказывать сильное влияние на время доступа к данным. Отключить такие обновления можно, если смонтировать файловую систему с ключом *noatime*.

Настройка алгоритма элеватора

Большинство драйверов блочных устройств Linux применяют общий алгоритм «элеватора» для планирования ввода-вывода. Этот алгоритм старается минимизировать количество необходимых перемещений дисковых головок считывания/записи. Программу */sbin/elvtune* можно использовать для повышения пропускной способности за счет времени ожидания, или наоборот. У *elvtune* есть два ключевых параметра: максимальное время задержки чтения (*-r value*) и максимальное время задержки записи (*-w value*). По умолчанию эти значения равны 8 192 и 16 384. Например, */sbin/elvtune -r 4096 -w 8192 /dev/sdc* изменит настройки элеватора для устройства */dev/sdc*. По существу, задержка записи должна примерно в два раза превышать задержку чтения. Для определения наилучших значений таких параметров необходимо провести эксперименты. Опыт показал, что если уменьшить эти значения наполовину, то это будет хорошим началом для файловых систем ext3. Когда подходящие значения найдены, то вызовы *elvtune* следует добавить в конец скрипта *rc.local*; установки алгоритма элеватора не сохраняются при перезагрузках.

В некоторых случаях попытки добиться максимальной пропускной способности за счет времени задержки (то есть установка больших значений задержки чтения и записи через *elvtune*) могут неблагоприятно

отразиться на реальной пропускной способности, а время задержки будет большим. Одна из возможных причин состоит в том, что в файловой системе ext3 операции записи запланированы каждые 5 секунд в противоположность 30-секундным задержкам в ext2. Поэтому эффект от журналирования транзакций весьма умеренный.

Выбор режима журналирования

Файловая система ext3 поддерживает три различных типа журналирования. Они устанавливаются с помощью параметра `data` вызова *mount*:

data=journal

Журналируются все данные (файловые данные и метаданные). Этот параметр предоставляет максимальные гарантии целостности данных, но также имеет самые высокие издержки производительности.

data=ordered

В протокол записываются только метаданные. Файловые данные записываются перед метаданными. Это означает, что все метаданные должны указывать на достоверные данные. Если в системе происходит авария, когда данные добавляются в файл, расположенный на файловой системе ext3 с монтированием `data=ordered`, то может произойти следующее. Блоки данных будут записаны, а метаданные, соответствующие расширению файла, – нет. Тогда блоки данных не будут принадлежать никакому файлу. Если запись происходит в середине файла (это встречается намного реже, чем запись в конец файла; даже перезапись файла выполняется с помощью усечения файла и последующего перезаписывания его), то возможно повреждение данных.

data= writeback

В протокол записываются только метаданные, а файловые данные не обрабатываются. В таком режиме работает большинство файловых систем с журналированием. Файловая система будет целостной, но в файлах после некорректного останова могут появляться старые данные. Такой режим журналирования при определенных рабочих нагрузках может послужить увеличению скорости обработки, так как обеспечивается гарантия целостности ослабленных данных (weakened data). В рабочей нагрузке преобладают интенсивные синхронные записи или создание и удаление множества маленьких файлов. Если при переключении с ext2 на ext3 существенно снижается производительность, то параметр `data=writeback` может способствовать ее значительному увеличению.

Переход с ext2 на ext3

К счастью, перейти с ext2 на ext3 легко: переформатирование не требуется. Нужно лишь запустить *tune2fs -j /dev/hda2* (или указать любое другое устройство) и в соответствующей строке файла */etc/fstab* изме-

нить ‘ext2’ на ‘ext3’. Для файловой системы root такой переход более сложен. Более свежую информацию можно почерпнуть в документации, прилагаемой к дистрибутиву Linux.

Файловая система Рейзера (ReiserFS)

ReiserFS 3.6.x (включена в состав ядер Linux 2.4.x) – это файловая система с журналированием, придуманная и разработанная Хансом Рейзером (Hans Reiser) и Namesys. Основой разработки ReiserFS было стремление создать отдельную, разделяемую среду («пространство имен»), в которой приложения могли бы взаимодействовать. Такая среда позволяла бы пользователям напрямую взаимодействовать с файловой системой, а не строить уровни, специально предназначенные для работы в верхнем слое файловой системы (такие как внутренние структуры базы данных для общения с приложениями).

Первоначально RaiserFS была сфокусирована на производительности работы с небольшими файлами (меньше нескольких тысяч байт) – обычно невысокой в файловых системах, подобных ext2 или UFS. ReiserFS в десять раз быстрее, чем ext2, обрабатывает файлы размером менее 1 Кбайт. Такая скорость достигается за счет сбалансированной древовидной системы, применяемой для организации метаданных файловой системы. Эта схема снимает искусственные ограничения на уровне файловой системы, а приятным побочным эффектом является назначение индексных дескрипторов только при необходимости – вместо создания фиксированного набора дескрипторов во время генерации файловой системы.

Компоновка остатков

В ReiserFS есть отличительная особенность, называемая *компоновкой остатков* (*tail packing*). В ReiserFS файл размером меньше, чем блок файловой системы (4 Кбайт), называется *остатком* (*tail*). Одной из предпосылок высокой производительности ReiserFS при работе с небольшими файлами является возможность напрямую включать эти остатки в файловые структуры данных. Таким образом, остатки находятся вблизи метаданных. Однако остатки не занимают много дискового пространства. Поскольку они не заполняют весь блок файловой системы, дисковое пространство может расходоваться впустую. Например, файл размером 100 байт займет целый блок данных (4 Кбайт) файловой системы. Эффективность хранения составит около 2,4%. Для преодоления этих трудностей ReiserFS проводит компоновку остатков, благодаря которой остатки хранятся вместе в блоке файловой системы. Эффективность хранения повышается. К сожалению, компоновка остатков вызывает существенное снижение производительности. Если необходимо пожертвовать емкостью хранения ради производительности, то при монтировании файловой системы следует применить ключ `notail`. Компоновка остатков будет отключена, а быстродействие увеличится.

Файловые системы ReiserFS начинают работать медленно, когда они заполнены более чем на 90%.

Как и в любой новой технологии, в ReiserFS есть свои особенности размещения и сервиса. Например, исторически она не взаимодействовала с NFS. ReiserFS получает поддержку, стремится быть мощной и устойчивой файловой системой и по-прежнему находится в стадии разработки. Дополнительные сведения о ReiserFS можно получить по адресу *http://www.namesys.com/*.

Journaled Filesystem (JFS)

Журналируемая файловая система (JFS) – это конструкция, основанная на экстентах. Первоначально она была разработана IBM для своей операционной системы AIX. Как и другие файловые системы с протоколированием, JFS сочетает повышенную надежность и возможность быстрого восстановления.

JFS представляет собой *агрегатную* (*aggregate*) файловую систему, состоящую из массива дисковых блоков со специальным форматом. Каждый агрегат включает в себя суперблок и карту размещения. Суперблок определяет раздел как агрегат JFS, а карта размещения описывает структуру размещения каждого блока данных. Агрегат вместе с управляющими структурами, которые необходимы для его задания, называется *набором файлов* (*fileset*). Агрегат является монтируемым. Каждый агрегат ведет протокол, в котором записываются изменения метаданных.

IBM перенесла JFS в Linux вместе с обновлениями, доступными для различных ядер. Веб-сайт JFS для Linux – *http://www.ibm.com/developerworks/oss/jfs/*.

Временная файловая система (Temporary Filesystem, tmpfs)

Solaris поддерживает особый тип файловой системы, называемый *tmpfs*. Если в системе есть свободная физическая память, то *tmpfs* предпочитает хранить данные в памяти, а не на диске. Когда памяти начинает не хватать, то файлы записываются в пространство свопинга. Таким образом, файлы в томе *tmpfs* не потребляют всю физическую память системы. К сожалению, это также означает, что файлы, хранимые в томе *tmpfs*, будут потеряны после перезагрузки. Наблюдаемое в таких файловых системах увеличение производительности *не* является результатом кэширования (равноценное кэширование выполняется стандартным механизмом Solaris). Причина в другом: файлы *tmpfs* записываются на диск лишь тогда, когда остается совсем мало свободной памяти. Поэтому применение *tmpfs* для ускорения приложений с преобладающим считыванием не является разумным, так как данные уже помещаются в буфер памяти файловой системой.

```
% df -k /tmp
Filesystem            kbytes    used   avail capacity  Mounted on
swap                 1707512  107472 1600040     7%    /tmp
```

По умолчанию системы Solaris применяют *tmpfs* для */tmp*. При этом избегается существенный объем ввода-вывода, так как */tmp* используется для временных файлов компилятора, редактора и т. д. В итоге обычная практика создания отдельного раздела для */tmp* не является действенной в Solaris. Один хороший побочный эффект состоит в том, что с помощью *df* можно посмотреть, сколько пространства свопинга свободно.

Veritas VxFS

Veritas Software продает файловую систему VxFS, основанную на экстентах. Эта файловая система обладает возможностями, благодаря которым она хорошо подходит для обслуживания больших приложений. Одно из самых существенных свойств – это возможность использования прямого ввода-вывода, минуя расположенный в памяти кэш файловой системы. Такой подход более эффективен, чем ключ `directio` UFS. Внутренняя организация VxFS способствует оптимизации производительности. Кроме того, эта файловая система факультативно ведет протокол данных, тогда как в UFS+ реализовано только протоколирование метаданных. Однако в такой среде устройство протоколирования должно иметь характеристики производительности, которые соответствуют быстродействию подсистемы оперативной памяти. Иначе диск протокола станет узким местом. К сожалению, VxFS сложна в использовании, тем самым отражая сложность современных хранилищ данных.

Доступ к документации VxFS в режиме онлайн можно получить по адресу *http://support.veritas.com*.

Кэширующие файловые системы (CacheFS)

В показателях производительности и сетевого трафика обращение к файлу на диске менее дорого, чем доступ к файлу, находящемуся на сетевом томе (например, через NFS). CacheFS была введена в Solaris 2.3 и доступна на многих других платформах, включая HP-UX и IRIX. Она призвана ускорить доступ к файлам томов NFS. Всецело работая на клиентской стороне, она не требует поддержки сервера NFS. В CacheFS каталог кэширования называется *клиентским* (*frontend*), а каталог, разделяемый через NFS, является *выходным* (*backend*). Для хранения локальных копий удаленно расположенных файлов CacheFS использует каталог кэширования. Это позволяет избегать обращений к удаленно размещенным файлам, когда в этом нет необходимости.

Первым делом нужно задать каталог кэширования с помощью *cdsadmin(1M)*. По умолчанию файлы кэшируются в блоки размером по

64 Кбайт, причем кэшируются только файлы размером до 3 Мбайт. Файлы с большим размером не кэшируются.

cfsadmin – это универсальное приложение: оно создает, удаляет, составляет список, проверяет целостность и обновляет параметры конфигурации. В табл. 5.8 обобщены доступные команды.

Таблица 5.8. Ключи cfsadmin

Ключ	Применение
`-c cache-directory`	Создает указанный каталог кэширования
`-d cache-id`	Удаляет указанный каталог кэширования. Значение *cache-id* заданного каталога показано в *cfsadmin -l*
`-l cache-directory`	Приводит список файловых систем, кэшируемых в указанном каталоге кэширования
`-s mount-point`	Выполняет проверку целостности кэшированной файловой системы, если файловая система была монтирована с ключом `demandconst`
`-u cache-directory`	Увеличивает параметры. Если указано без *-o options*, то устанавливает параметры по умолчанию, если такое возможно

Тест на целостность проверяет только доступность файлов, поэтому *cfsadmin -s* не приводит к большому объему тестирования. Ключи *-c* и *-u* также имеют параметры, задаваемые с помощью *-o options*. Параметры приведены в табл. 5.9.

Таблица 5.9. Параметры cfsadmin

Параметр	Описание	По умолчанию
`maxblocks`	Наивысшее процентное содержание блоков в каталоге кэширования, которое указанный кэш может использовать	90
`minblocks`	Процентное содержание блоков в каталоге кэширования, которое указанный кэш может использовать без ограничений	0
`threshblocks`	Когда процентное содержание использованных блоков превысит этот показатель, кэш не сможет запрашивать ресурсы, пока потребление блоков не достигнет уровня, заданного параметром `minblocks`	85
`maxfiles`	Наивысшее процентное содержание файлов в каталоге кэширования, которые указанный кэш может использовать	90
`minfiles`	Процентное содержание индексных дескрипторов в каталоге кэширования, которые указанный кэш может использовать без ограничений	0

Таблица 5.9 (продолжение)

Параметр	Описание	По умолчанию
threshfiles	Когда количество использованных индексных дескрипторов превысит это процентное содержание, кэш не сможет запрашивать ресурсы, пока потребление индексных дескрипторов не достигнет уровня, заданного параметром minfiles	85
maxfilesize	Этот параметр определяет самый большой файл, который можно кэшировать, в мегабайтах. *Файлы большего размера кэшироваться не будут*	3

Рассмотрим примеры применения *cfsadmin*. Начнем с создания кэша с параметрами по умолчанию и монтирования файловой системы в кэше:

```
# cfsadmin -c /cache
# mount -F cachefs -o ro,backfstype=nfs,cachedir=/cache \
  docsun.cso.uiuc.edu:/services/patches /mnt
# df -k /mnt
Filesystem            kbytes    used   avail capacity  Mounted on
/cache/.cfs_mnt_points/docsun.cso.uiuc.edu:_services_patches
                     8187339 2529123 5658216    31%    /mnt
```

Как видим, существует несколько специализированных параметров *mount* для использования с CacheFS, которые обобщены в табл. 5.10.

Таблица 5.10. Параметры mount для файловой системы CacheFS

Параметр	Описание	По умолчанию
acdirmax	Время хранения атрибутов каталога кэширования перед удалением из кэша (в секундах)	30
acdirmin	Время (в секундах), в течение которого атрибутам каталога кэширования гарантировано место в кэше. По истечении этого времени CacheFS проверяет, изменилось ли время модификации в выходной файловой системе. Если да, то данные из кэша удаляются и загружается новый экземпляр	30
acregmax	Время, в течение которого атрибуты кэшированного файла хранятся после модификации файла до их удаления (в секундах)	30
acregmin	Время (в секундах), в течение которого атрибутам кэшированного файла гарантировано место в кэше после модификации файла. По истечении этого времени, если время модификации файла на выходной файловой системе изменилось, то данные из кэша удаляются и загружается новый экземпляр	30

Параметр	Описание	По умолчанию
actimeo	Служебный параметр. Параметры acdirmax, acdirmin, acregmax и acregmin будут равны этому значению	30
backfstype	Задает тип выходной файловой системы. Обычно *nfs*	Нет
backpath	Задает точку монтирования выходной файловой системы. Если этот аргумент не задан, то CacheFS выбирает подходящее значение	Авто
cachedir	Название каталога кэширования	Нет
cacheid	Строка, определяющая конкретное пространство кэширования. Если не указана, то CacheFS подберет его сама	Авто
demandconst	Когда этот флаг установлен, целостность кэша проверяется только по запросу, а не автоматически. Это полезно для файловых систем, изменяющихся очень редко. Несовместим с noconst	Переключатель (Выключено)
local-access	Для определения прав доступа используется клиентская файловая система. Для проверки этих данных выходная файловая система не используется. Установка этого параметра может привести к уязвимости защиты	Переключатель (Выключено)
noconst	Отключает автоматическую проверку целостности кэша. Следует устанавливать лишь в том случае, когда известно, что выходная файловая система не будет изменяться. Несовместим с demandconst	Переключатель (Выключено)
purge	Удалить все кэшированные данные	Нет
ro	Монтирование только для чтения. Несовместим с rw	Переключатель (Выключено)
rw	Монтирование для чтения/записи. Это состояние по умолчанию. Несовместим с ro	Переключатель (Выключено)
suid	Разрешает выполнение *suid*-приложений. Это состояние по умолчанию. Несовместим с nosuid	Переключатель (Выключено)
nosuid	Запрещает выполнение *suid*-приложений. Несовместим с suid	Переключатель (Выключено)
write-around	Операции записи происходят так, как будто они выполняются на обычной NFS. Записи производятся в выходной файловой системе, а данные кэша становятся недействительными. Это состояние по умолчанию. Несовместим с non-shared	Переключатель (Выключено)

Таблица 5.10 (продолжение)

Параметр	Описание	По умолчанию
non-shared	Операции записи выполняются как на выходной, так и на клиентской файловой системе. Этот параметр следует использовать лишь при уверенности, что больше никто не выполняет запись в файловую систему. Несовместим с write-around	Переключатель (Выключено)

Посмотрим на кэш:

```
# cfsadmin -l /cache
cfsadmin: list cache FS information
   maxblocks     90%
   minblocks      0%
   threshblocks  85%
   maxfiles      90%
   minfiles       0%
   threshfiles   85%
   maxfilesize    3MB
  docsun.cso.uiuc.edu:_services_patches:_mnt
```

Кажется, неплохо. Чтобы изменить параметры кэша без удаления и пересоздания каталога кэширования, можно воспользоваться *cfsadmin -u* после того, как были размонтированы все кэшированные файловые системы:

```
# umount /mnt
# cfsadmin -u -o maxblocks=95,minblocks=65,threshblocks=90,\
  maxfiles=95,minfiles=80,threshfiles=85,maxfilesize=10 /cache
# cfsadmin -l /cache
cfsadmin: list cache FS information
   maxblocks     95%
   minblocks     65%
   threshblocks  90%
   maxfiles      95%
   minfiles      80%
   threshfiles   85%
   maxfilesize   10MB
  docsun.cso.uiuc.edu:_services_patches:_mnt
```

Для очистки и удаления монтированной файловой системы и ее кэша следует выполнить:

```
# umount /mnt
# cfsadmin -d docsun.cso.uiuc.edu:_services_patches:_mnt /cache
# cfsadmin -l /cache
```

Для полного удаления кэша файловой системы можно воспользоваться *cfsadmin -d all*:

```
# cfsadmin -d all /cache
```

Если используется CacheFS, то для того чтобы сетевая файловая система автоматически монтировалась при загрузке, в */etc/vfstab* следует добавить строку, подобную нижеприведенной:

```
# device          device     mount   fs       fsck  mount    mount
# to-mount        to fsck    point   type     pass  at boot  options
nfs-server:/apps  /cache     /apps   cachefs  3     yes
ro,backfstype=nfs,cachedir=/cache
```

Когда выполняется новое считывание, запрос разбивается на части по 64 Кбайт и передается на сервер NFS. Копия полученных данных записывается в CacheFS на локальном диске. Запись кэшированных данных делает недостоверными данные в кэше, поэтому данные с сервера приходится загружать повторно. На самом деле это не настолько плохо, поскольку такое считывание, вероятно, будет быстро выполнено из кэша памяти.

Вполне вероятно, что при незначительно нагруженной сети 100 Мбит и быстром сервере NFS промахи кэша могут обрабатываться более быстро, чем попадания в кэш на сильно нагруженном локальном диске! Если локальный диск также обслуживает операции пейджинга, то операции пейджинга и кэширования могут быть синхронизированы, особенно при начале работы приложения.

Лучшие файловые системы для кэширования – это системы «только для чтения» или с преобладающим чтением, расположенные в загруженных сетях с низкой пропускной способностью. Частоту успешных попаданий в кэш можно проверить с помощью команды *cachefsstat*, доступной в Solaris старше версии 2.4:[1]

```
# cachefsstat

    /mnt
                cache hit rate:    90% (54 hits, 6 misses)
            consistency checks:     31 (31 pass, 0 fail)
                      modifies:      0
            garbage collection:      0
```

Эти данные могут рассказать, насколько в операционной среде эффективно кэширование. В системах «только для чтения» авторы наблюдали, что частота успешных попаданий превышала 95%.

Минимизация времени поиска на уровне файловой системы

В этой главе периодически упоминается необходимость минимизации времени поиска, особенно для приложений с произвольным доступом.

[1] При работе с Solaris 2.3 и 2.4 получить такие данные нет возможности. Может помочь обновление операционной системы.

Само собой разумеется, что чем больше данных вмещает цилиндр, тем меньше требуется операций поиска. Этого можно достигнуть за счет распределения данных по дискам (striping), то есть нужно создать логические цилиндры, размер которых больше размера физического цилиндра (который, конечно, фиксирован). Однако подобного результата можно достигнуть даже на одном диске. Вследствие механизма ZBR (см. раздел «Позонная организация хранения (ZBR)» ранее в этой главе) внешние цилиндры являются более быстрыми и содержат больше данных. Так как все цилиндры большие, то поиск на крайних цилиндрах происходит быстрее, чем поиск на внутренних! В итоге использование внешних цилиндров (там, где это возможно) приведет к снижению времени поиска.

Тот факт, что длинные поиски продолжительнее коротких, приводит к другому важному правилу: при конфигурировании каждому диску следует определять одно назначение. Размещение на диске двух «горячих» областей, к которым обращаются часто, приводит к тому, что бульшая часть времени тратится на поиск между этими областями, а не на непосредственное обслуживание запросов.

По историческим причинам многие системные администраторы разбивают основной диск на четыре раздела в таком порядке: root, swap, */usr* и */home*. Подобное распределение далеко от оптимального. Вероятно, обращения к разделу свопинга будут происходить наиболее часто, а к */home* будут обращаться чаще, чем к */usr*. Если происходит поиск вне текущего раздела, то само собой разумеется, что дисковому манипулятору необходимо много времени, чтобы при поиске перейти из swap к */home*, минуя */usr*. Простое изменение порядка разделов, а именно root, swap, */home* и */usr*, снизит среднее время поиска на диске!

Это кажется нарушением первого принципа настройки производительности: что-то получено бесплатно! Однако в этом случае стоимость скорее интеллектуальная, чем денежная. Затраты на получение такого преимущества – это усилия, направленные на понимание процесса доступа к диску на базовом уровне и применение этих знаний на практике.

Инструменты для анализа

Теперь, после обсуждения работы дисков, их места в системе, а также файловых систем, служащих промежуточным слоем между низкоуровневыми компонентами и пользователями, можно перейти к инструментам для анализа и повышения производительности дисковых подсистем. В этом разделе обсуждается: включение дисковых кэшей, инструменты измерения дисковой производительности, типичная ловушка в анализе дисковой производительности и способ ее избежать, а также инструменты для мониторинга производительности устройств (*sar* и *iostat*) и средства мониторинга низкоуровневого дискового ввода-вывода.

Включение дисковых кэшей

Сущность кэширования записи на диск уже обсуждалась. Прежде чем приступить к этой процедуре, следует прочитать раздел «Дисковые кэши» ранее в этой главе. Бездумное включение кэшей записи может сильно повлиять на надежность данных. Помня об этом предостережении, рассмотрим пример 5.1. В нем показано, как запускать *format* в режиме, позволяющем менять стратегию кэширования для каждого диска.

Пример 5.1. Включение кэширования записи на диск в Solaris

```
# format -e
Searching for disks...done

AVAILABLE DISK SELECTIONS:
       0. c0t0d0 <SUN2.1G cyl 2733 alt 2 hd 19 sec 80>
          /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0
Specify disk (enter its number): 0
...
format> cache

CACHE MENU:
      write_cache - display or modify write cache settings
      read_cache  - display or modify read cache settings
      !<cmd>      - execute <cmd>, then return
      Quit
cache> write_cache

WRITE_CACHE MENU:
        display   - display current setting of write cache
        enable    - enable write cache
        disable   - disable write cache
        !<cmd>    - execute <cmd>, then return
        Quit
write_cache> display
Write Cache is disabled
write_cache> enable
write_cache> display
Write Cache is enabled
```

В Linux стратегию кэширования диска лучше всего менять с помощью утилиты *hdparm*, которая доступна по адресу *ftp://metalab.unc.edu/pub/Linux/system/hardware/hdparm-4.1.tar.gz*. Вот как можно включить кэширование записи для */dev/hda*:

```
# hdparm -W1 /dev/hda

/dev/hda:
 setting drive write-caching to 1 (on)
```

Если надежность данных предпочтительна, а скорость менее важна, то кэширование записи можно отключить:

```
# hdparm -W0 /dev/hda
```

```
/dev/hda:
 setting drive write-caching to 0 (off)
```

Измерение дисковой производительности

После того как были обсуждены все методы «подгонки» дисковых подсистем к рабочей нагрузке и улучшения дисковой производительности, будет очень полезно поговорить о производительности подсистемы ввода-вывода. К счастью, для измерения производительности есть несколько хороших инструментов. Здесь будут рассмотрены *hdparm*, *tiobench* и *iozone*.

hdparm

Команда *hdparm* – это замечательное, компактное средство для приблизительного тестирования производительности устройств в Linux. *hdparm* позволяет провести несколько простых тестов производительности при последовательной передаче данных. С помощью ключей *-Tt* можно провести довольно хорошую оценку производительности одного диска. Это очень полезно, так как в среде PC-совместимого оборудования присутствует большое количество сильно различающихся дисковых накопителей.

```
# hdparm -Tt /dev/hda
/dev/hda:
 Timing buffer-cache reads:   128 MB in  3.94 seconds = 32.49 MB/sec
 Timing buffered disk reads:  64 MB in 15.31 seconds =  4.18 MB/sec
```

tiobench

tiobench – это открытый программный продукт, являющийся инструментом оценки потоковой дисковой производительности. *tiobench* пришел на смену старому средству измерения *bonnie*. Он выдает более полные сведения, чем *hdparm*, поскольку учитывает последовательный и произвольный типы доступа, а также информирует о времени задержки.

Запускать эту тестовую программу очень просто, особенно в Linux. Приводимые ниже данные были получены в системе с 300 МГц Celeron, запускающей ядро Linux 2.4. Дисковое устройство – диск IDE 7 200 rpm.

```
% tiotest
Tiotest latency results:
.-------------------------------------------------------------------------.
| Item         | Average latency | Maximum latency | % >2 sec | % >10 sec |
+--------------+-----------------+-----------------+----------+-----------+
| Write        |        1.827 ms |     1421.003 ms |  0.00000 |   0.00000 |
| Random Write |        0.453 ms |      409.878 ms |  0.00000 |   0.00000 |
| Read         |        0.241 ms |      262.779 ms |  0.00000 |   0.00000 |
| Random Read  |        0.087 ms |      107.313 ms |  0.00000 |   0.00000 |
|--------------+-----------------+-----------------+----------+-----------|
```

```
| Total        |        0.819 ms |     1421.003 ms |  0.00000 |   0.00000 |
`--------------+-----------------+-----------------+----------+-----------'

Tiotest results for 4 concurrent io threads:
,-----------------------------------------------------------------------.
| Item                  | Time     | Rate         | Usr CPU  | Sys CPU |
+-----------------------+----------+--------------+----------+---------+
| Write          40 MBs |   14.1 s |   2.834 MB/s |   1.1 %  |  19.5 % |
| Random Write   16 MBs |   10.2 s |   1.533 MB/s |   0.2 %  |  15.3 % |
| Read           40 MBs |    1.1 s |  37.924 MB/s |   3.8 %  |  19.0 % |
| Random Read    16 MBs |    0.5 s |  32.338 MB/s |   4.1 %  |  14.5 % |
`-----------------------------------------------------------------------'
```

О дополнительных ключах можно узнать, если внимательно просмотреть вывод *tiotest -h*. Особый интерес представляют ключи *-S* и *-W*. При задании -S все записи будут проводиться синхронно, а при *-W* – последовательно. Вместе с *tiobench* поставляется скрипт Perl, который можно применить для автоматизации запуска тестов.

На момент написания этих строк *tiobench* доступен по адресу *http://sourceforge.net/projects/tiobench/*.

iozone

Вполне возможно, что тестовая программа *iozone* – это один из самых полных инструментов, когда-либо разработанных для оценки работы диска и файловой системы. К сожалению, он чрезвычайно сложен (побочный эффект его полноты) и выходит за рамки текущего обсуждения. Однако на момент написания этих строк исчерпывающая документация доступна в режиме онлайн по адресу *http://www.iozone.org*.

Второй раз быстрее?

Рассмотрим простые измерения пропускной способности:

```
% mkfile -v 10m test-file
test-file 10485760 bytes
% ptime cat test-file > /tmp/glop

real         1.728
user         0.004
sys          0.273
% rm /tmp/glop
% ptime cat test-file > /tmp/glop

real         0.504
user         0.004
sys          0.211
```

Проницательный администратор может спросить, почему второй запуск намного быстрее первого. Ответ прост: это фиктивные измерения. Первый запуск законно скопировал данные с диска во временное про-

странство, которое на этой системе представлено файловой системой *tmpfs*.[1] Другое дело второй запуск: файл из *расположенного в памяти кэша файловой системы* (*in-memory filesystem cache*) копируется во временное пространство. При запуске программы следует убедиться, что измеряется именно дисковая производительность, а не производительность кэша файловой системы, расположенного в памяти. Обеспечить достоверность измерений можно за счет размонтирования и повторного монтирования файловой системы между испытаниями.

Применение iostat

Одним из самых полезных инструментов для мониторинга дисковой производительности является *iostat*. К сожалению, *iostat* может сбить с толку. Его вывод может быть представлен во многих различных формах. Кроме того, в некоторых случаях он необычно ведет расчеты (по историческим причинам).

Возможно, самыми полезными ключами версии *iostat* в Solaris являются *-x* (для расширенной статистики), *-n* (для печати названий в более наглядном формате `cXtXdXsX`)[2] и *-P* (для статистики отдельно по каждому разделу). Другие интересные ключи: *-C* вызывает объединение согласно идентификаторам контроллеров, *-m* сообщает точки монтирования файловых систем, а *-z* подавляет вывод строк, состоящих из одних нулей (например, для бездействующих дисков).

```
% iostat -mnPxz 30
...
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.0    0.2    0.0    0.8  0.0  0.0    0.0    9.5   0   0 c0t0d0s3 (/)
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.0    1.4    0.0   10.6  0.0  0.0    0.0   13.0   0   1 c0t0d0s3 (/)
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.0    2.6    0.0   22.4  0.0  0.0    0.0   12.1   0   1 c0t0d0s3 (/)
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    3.2    0.8   22.2    6.4  0.0  0.0    0.0    3.9   0   2 c0t0d0s3 (/)
```

[1] По существу, данные просто не были записаны на диск. Этот тест измерил время, требуемое для чтения файла с диска и сохранения его в памяти.

[2] Формат `cXtXdXsX` позволяет легко соотнести путевое имя раздела с тем, как раздел присоединен к системе. Число, следующее за `c`, обозначает номер контроллера. Число, следующее за `t`, обозначает целевой номер устройства. Число, следующее за `d`, обозначает код логического устройства (LUN). И число, следующее за `s`, означает номер раздела. Например, `c1t5d0s3` означает третий раздел (`s3`) на диске с ID 5 (`t5`), присоединенном к контроллеру 1 (`c1`). У дисков IDE нет значения `d`. Они представлены в виде `cXtXsX`.

```
                    extended device statistics
 r/s    w/s   kr/s    kw/s wait actv wsvc_t asvc_t  %w  %b device
 0.2    0.2    1.4     1.6  0.0  0.0    0.0    9.1   0   0 c0t0d0s3 (/)
 0.0    2.2    0.0   206.4  0.0  0.0    0.0   21.7   0   1 c0t1d0s0
                                                        (/export/home)
```

Поля вывода в порядке их следования:

- `r/s` и `kr/s` сообщают количество операций чтения в секунду и количество прочитанных килобайт в секунду. `w/s` и `kw/s` сообщают ту же информацию, только для записи.
- `wait` – среднее количество транзакций, ожидающих обслуживания.
- `actv` – количество запросов, активно обслуживаемых в настоящее время.
- `wsvc_t` – среднее время обслуживания в очереди ожидания в миллисекундах.
- `%w` представляет процент времени, в ходе которого имелись транзакции, ожидающие обслуживания.
- `%b` представляет процент времени, в ходе которого диск активно обслуживал транзакции.

В Linux для *iostat* следует применять ключи *-d -x interval*. Тогда команда будет выдавать расширенный вывод и отображать только данные о диске (без статистики процессора):

```
$ iostat -d -x 30
Linux 2.4.2-2 (aiua)      07/29/2001

Device: rrqm/s wrqm/s   r/s   w/s  rsec/s  wsec/s avgrq-sz avgqu-sz  await svctm  %util
hde       2.88   1.32 52.90  2.23   99.06   28.50     2.31     0.65  11.82  8.34  4.60
hde1      0.00   0.00 49.60  0.00   49.61    0.00     1.00     0.28   5.64  5.64  2.80
hde2      2.88   1.32  3.30  2.23   49.44   28.50    14.10     0.37  67.26 33.29  1.84
hde5      0.00   0.00  0.00  0.00    0.01    0.00     8.00     0.00 100.00100.00  0.00
hdg       5.89   1.61 34.27  3.27  233.97   39.41     7.28     5.17 137.62 31.69 11.90
hdg1      0.00   0.00 12.47  0.00   12.47    0.00     1.00     0.05   4.21  4.21  0.53
hdg2      5.89   1.61 21.80  3.27  221.50   39.41    10.40     5.11 203.98 45.35 11.37
```

Перед временным интервалом можно указать конкретное устройство, если статистика нужна только для него.

Исторические ограничения: iostat и терминология очередей

В прошлом, когда такие интерфейсы, как IPI, доминировали на рынке, дисковые контроллеры действительно напрямую управляли дисками. В таких средах термины *использование* (*utilization*), *время обслуживания* (*service time*) и *время ожидания* (*wait time*) имели четко определенные значения. Простые вычисления на основе теории массового обслуживания позволяли получить эти значения с помощью набора базовых измерений. В тот же период появился *iostat*. Сегодня

дисковые контроллеры в дисковых подсистемах не настолько вездесущи: современные диски чрезвычайно умны. Простая модель, реализованная *iostat*, и особенно ее терминология, могут сбить с толку в современных операционных средах.

В старых системах, где главную роль играли контроллеры, было так. Когда дисковый драйвер направлял запрос диску, то он знал, что диск бесхитростно обслужит этот запрос. Было известно, что для этого потребуется время, равное времени обслуживания. В нынешних системах время обслуживания варьируется в зависимости от дисковой нагрузки и длины очереди, поэтому нельзя провести прямое сравнение с более ранними измерениями времени обслуживания. Но можно создать две особые очереди: очередь ожидания для команд, образующих очередь дискового драйвера, и активную очередь для команд, стоящих в очереди к самому диску. К сожалению, *iostat* пытается сообщать новые измерения, используя старую терминологию. «Время ожидания обслуживания» («wait service time») – это, по сути, время, потраченное в очереди «ожидания», которое при измерениях не имеет смысла.

Значение, которое *iostat* называет «временем обслуживания», определяется как длина очереди, разделенная на пропускную способность. На самом деле это не является временем обслуживания. Скорее это период времени, за который будет обработана первая команда. То время, которое *iostat* называет «временем обслуживания», лучше формально назвать «временем отклика».

Применение sar

С помощью запуска команды *sar* с ключом *-d* можно получить различную статистику по дисковым операциям для каждого раздела:

```
% sar -d 5 100
SunOS islington 5.8 Generic_108528-03 sun4u                    02/16/01

14:58:51   device        %busy   avque   r+w/s  blks/s  avwait  avserv
...
14:59:06   fd0               0     0.0       0       0     0.0     0.0
           nfs1              0     0.0       0       0     0.0     0.0
           sd0              93     2.6      81   17934     0.0    31.9
           sd0,a             0     0.0       0       0     0.0     0.0
           sd0,c             0     0.0       0       0     0.0     0.0
           sd0,d            93     2.6      81   17934     0.0    31.9
           sd1               0     0.0       0       0     0.0     0.0
           sd1,a             0     0.0       0       0     0.0     0.0
           sd1,c             0     0.0       0       0     0.0     0.0
           sd6               0     0.0       0       0     0.0     0.0
...
```

Первая колонка, `%busy`, сообщает о периоде времени, когда устройство было занято обслуживанием запроса передачи в интервале выборки;

avque дает среднее количество запросов, ожидающих выполнения в тот же период. Количество операций чтения, записи и переданных байтов (в единицах по 512 байт) приводится в колонках read/s, write/s и blks/s. В колонке avwait выдается среднее время ожидания, в миллисекундах. Колонка avserv сообщает среднее время обслуживания, так же в миллисекундах. Для более подробной информации о *sar* следует обратиться к разделу «sar» главы 4.

Мониторинг ввода-вывода

Было бы здорово, если бы вывод *iostat* рассказал еще больше. Например, при наличии высокого коэффициента использования одного раздела было бы очень полезно узнать, какие процессы явились причиной такого поведения. Однако для того чтобы это выяснить, необходимо применить более сложные инструменты: средства мониторинга, предоставляемые ядром.

В системах UNIX мониторинг средствами ядра существовал всегда. Однако изначально он предусматривался только для разработчиков. Активация такой возможности в рабочих ядрах обычно не проводилась, поскольку мониторинг приводил к небольшим издержкам производительности. В ядре AIX IBM решила оставить мониторинг средствами ядра включенным. С тех пор Sun ходит в ту же масть.

Мониторинг очень полезен. У него есть несколько славных черт. Например, трассировочный файл – самоописываемый, поэтому внешние описания не требуются. Инструмент мониторинга состоит из трех утилит: *prex(1)*, контролирующей выполнение пробы,[1] *tnfxtract(1)*, выводящей данные буфера трассировки ядра, и *tnfdump(2)*, отображающей данные в удобном для восприятия формате. Однако не все так просто. Например, пробы уровня пользователя могут записывать данные напрямую в трассировочные файлы, а пробы ядра записывают данные в кольцевой буфер, который реализован как отдельный буфер для каждого потока ядра. Сделано это с целью предотвращения блокировок и конфликтов в многопроцессорных системах. Сложность состоит в том, что трудно определить размер буфера. Одна очень активная проба может неоднократно повторно заполнять буфер, в то время как другие только начинают работать. К счастью, *tnfxtract* фиксирует состояние буфера, поэтому второй снимок состояния (snapshot) будет включать неиспользованные данные. Кроме того, *tnfdump* очень хорошо сортирует данные в хронологическом порядке. Однако несмотря на то что это чрезвычайно мощные утилиты, здесь будут обсуждаться только возможности, касающиеся мониторинга ввода-вывода.

[1] Для англ. слова «probe» в этом контексте для краткости избран буквальный перевод – «проба». Речь идет о программах измерения, которые в данном случае встроены в ядро. – *Примеч. перев.*

Применение проб ядра

При запуске команда *prex* переходит в интерактивный командный режим, где можно задать детали той трассировки, которую необходимо выполнить. Здесь использован ключ *-k*, поэтому происходит подключение к внутренним пробам ядра:

```
# prex -k
Type "help" for help ...
prex> buffer alloc 2m
Buffer of size 2097152 bytes allocated
prex> enable io
prex> trace io
prex> ktrace on
```

Эти команды выделяют 2 Мбайт буфера для хранения данных[1] и задают набор проб `io` для трассировки.[2] Команда *ktrace on* включает «рубильник» («master switch») пробы ядра, контролирующий время сбора данных. В этой точке следует выждать момент, когда произойдет интересующее событие, и выключить «рубильник»:

```
prex> ktrace off
```

Затем можно выполнить дамп записей трассировки в файл и проанализировать их:

```
# mkdir /tmp/trace
# cd /tmp/trace
# tnfxtract iotrace.tnf
# tnfdump iotrace.tnf | more
```

Итоговый вывод можно разбить на две секции: заголовок и содержимое. Обе очень обширны, поэтому они были безжалостно отредактированы, чтобы вместить их на страницу. Заголовок проиллюстрирован в примере 5.2.

Пример 5.2. Заголовок вывода tnfdump при трассировке средствами ядра

```
probe tnf_name: "strategy" tnf_string: "... blockio;file .../driver.c;line 305;"
probe tnf_name: "pageout" tnf_string: "... pageio io;file .../vm_pvn.c;line 552;"
probe tnf_name: "biodone" tnf_string: "... blockio;file .../bio.c;line 1189;"
```

Строки заголовка подробно описывают пробы, заданные для трассировки. Проба `strategy` записывает данные о происходящем вводе-выводе, а проба `biodone` записывает данные о вводе-выводе, который был за-

[1] Нет необходимости явно указывать размер буфера – простого *buffer alloc* будет достаточно.

[2] Отважный читатель может включить *все* пробы ядра. Это можно достичь с помощью команд *enable $all* и далее *trace $all*. Следует учитывать, что это очень подробный режим и, вероятно, не для слабонервных.

вершен. Другой механизм, который может вызвать ввод-вывод, – это пейджинг. Данные о нем записываются с помощью пробы `pageout`, поэтому она включена в набор проб `io`. Кроме того, заголовок сообщает, к каким наборам принадлежит проба (например, `io` и `blockio`), а также файл исходного кода и строку, на которой проба теперь находится.

Содержимое трассировки соответствует формату, показанному в примере 5.3, который так же сильно усечен.

Пример 5.3. Содержимое вывода tnfdump при трассировке средствами ядра

```
Elapsed  Delta  PID LWPID TID CPU Probe   Data / Description . . .

 0.000   0.000 2040     1 ... 1 strategy ...  buf: 0x707b45a0 flags: 4194313
16.911  16.911    0     0 ... 0 biodone  ...  buf: 0x707b45a0
17.027   0.116 2040     1 ... 1 strategy ...  buf: 0x706d8410 flags: 524569
26.101   9.074    0     0 ... 0 biodone  ...  buf: 0x706d8410
26.106   0.005    0     0 ... 0 biodone  ...  buf: 0x706d8410
26.158   0.051 2040     1 ... 1 pageout  ...
```

Следующая секция, содержащая данные трассировки, в хронологическом порядке показывает, какие запросы ввода-вывода были обработаны. Она *не* отображает запросы ввода-вывода, которые были обслужены средствами расположенного в памяти кэша файловой системы. Первая колонка сообщает время (в мс), прошедшее с начала работы первой пробы. Вторая представляет время (в мс), затраченное на выполнение операции. Следующие три колонки относятся к ID процесса, ID легковесного процесса и ID потока процесса, инициировавшего запрос ввода-вывода. Колонка CPU сообщает, какой процессор обработал запрос. Описываемая трассировка проводилась на двухпроцессорной машине Sun Ultra 2. Это объясняет присутствие CPU 0 и CPU 1. Последняя колонка описывает данные, генерированные пробой. Ясно, что задача точного отслеживания тех данных, которые представляет проба, может быть нетривиальной. На помощь может прийти трассировка запущенного процесса посредством *truss(1)*. Авторы считают запуск *truss -lDdaf command* очень полезным.

Заметим, что поле флагов переведено из двоичного формата в шестнадцатеричный. Поле флагов образовано логическим сложением флагов. Стандартный список можно найти в */usr/include/sys/buf.h*. Самые полезные флаги представлены в табл. 5.11.

Таблица 5.11. Флаги трассировки ввода-вывода

Флаг	Название	Описание
0x000001	BUSY	Запрос занят (см. WANTED)
0x000002	DONE	Транзакция завершена
0x000004	ERROR	Транзакция прекращена

Таблица 5.11 (продолжение)

Флаг	Название	Описание
0x000008	KERNBUF	Обращение к буферу ядра
0x000010	PAGEIO	Страничный ввод-вывод
0x000040	READ	Запрос на чтение
0x000080	WANTED	Запуск после снятия флага BUSY
0x000100	WRITE	Запрос на запись
0x000400	ASYNC	Асинхронный; не ожидает завершения ввода-вывода
0x002000	DONTNEED	Нет кэширования после записи
0x008000	FREE	Освобождение страницы после работы с ней
0x010000	INVAL	Запрос содержит неверные данные
0x080000	NOCACHE	Не кэшировать блок, когда он освобожден
0x400000	RETRYWRI	Повторять запись до успешной попытки

В приведенном примере анализировались все запросы ввода-вывода для краткосрочного процесса. Сырые данные из примера 5.3 представлены в табл. 5.12 в более ясном формате.

Таблица 5.12. Реальные операции, видимые в выводе tntdump

Буфер	Время (мс)	Операция	Содержимое поля флагов
0x707b45a0	0.000	Выпущен запрос ввода-вывода	0x400009
	16.911	Запрос ввода-вывода завершен	
0x706d8410	17.027	Выпущен запрос ввода-вывода	0x080119
	26.102	Запрос ввода-вывода завершен	

Исследуем флаги. Первый ввод-вывод – это запрос 8 Кбайт к буферу ядра, обработка которого заняла 16,911 мс. Второй – для некэшированной записи 1 Кбайт в страницу ввода-вывода, буферизованную в ядре (это заняло 9,074 мс). Первый запрос – это обновление индексного дескриптора, второй – запись данных.

Описанный метод отслеживает все операции ввода-вывода в системе. Он способен дать интересные результаты, но может стать и настоящим испытанием. При необходимости в нагруженной системе можно проводить трассировку ввода-вывода отдельного процесса вместо анализа состояния всей системы. Для этого существует свой способ.

Применение фильтрации процессов

prex реализует средства выборочной трассировки, называемые *фильтрацией процессов* (*process filtering*). Если включен процесс фильтрации, то трассировка в ядре сводится только к событиям, которые были инициированы от имени процессов из заданного списка. Процесс 0 представляет все потоки, не ассоциированные с процессом.

Когда команда *pfilter* инструмента *prex* запущена без аргументов, она отображает режим фильтрации текущих процессов (включена или выключена), а также список идентификаторов процессов. Список идентификаторов процессов можно модифицировать вне зависимости от состояния фильтрации процессов.

Для того чтобы идентификатор процесса можно было добавить в список идентификаторов, такой процесс должен существовать. Список автоматически обновляется, а несуществующие процессы удаляются. Если при включенном режиме фильтрации список становится пустым, то *prex* выдает предупреждение. Режим фильтрации процессов сохраняется между вызовами *prex*, поэтому при необходимости его следует выключать вручную.

Рассмотрим пример, в котором представлена трассировка всего ввода-вывода, порожденная сессией *vi* (ID процесса 19786):

```
# prex -k
Type "help" for help ...
prex> buffer alloc
Buffer of size 393216 bytes allocated
prex> pfilter add 19786
prex> pfilter on
prex> pfilter
Process filtering is on
Process filter set is {19786}
prex> trace io
prex> enable io
prex> ktrace on
...in a separate window, exit the vi session (:wq)...
prex> ktrace off
Warning: Process filtering on, but pid filter list is empty
prex> pfilter off
prex> quit
# tnfxtract /tmp/vi.trc
# tnfdump /tmp/vi.trc
...
```

Заметим, что *prex* выдал предупреждение, когда выполнение процесса *vi* завершилось и не осталось процессов для трассировки.

Перезапуск prex

Если перезапустить трассировку средствами ядра спустя короткое время после предыдущего запуска, то пользователь, еще не приступивший к работе, увидит на экране старые данные. Такое явление можно легко устранить за счет высвобождения и повторного назначения буфера:

```
prex> buffer dealloc
prex> buffer alloc
```

Такие примеры – лишь штрихи к *prex*. Это сложный инструмент, способный решать многие задачи. Для проведения дальнейших исследований можно воспользоваться другими важными наборами проб. Один из них – это *vm*. Он содержит все пробы, относящиеся к виртуальной памяти.

Заключение

В этой главе были обсуждены основные вопросы, касающиеся настройки производительности ввода-вывода: базовые концепции дисковых накопителей, такие как физическое размещение, типы доступа и параметры, управляющие дисковой производительностью; интерфейсы, связанные с этими дисками; различные типы файловых систем, а также инструменты мониторинга производительности дисков и подсистем ввода-вывода.

Во многих отношениях диски занимают уникальный пьедестал в архитектуре компьютера. Диски – это единственный блок системы, работа которого зависит от механических параметров и движущихся частей. Диски являются последним прибежищем для хранения информации в режиме онлайн. Это королевство без очарования дизайна микропроцессора, где подсчитывается каждый цикл. В отличие от памяти, диски не рассматриваются как основная часть конструкции современных компьютеров. Вероятно, именно по этим причинам при настройке производительности системные администраторы уделяют дискам мало внимания.

Настройка диска подобна темному озеру в лесной глуши, наполненному чем-то таинственным и необычным. Оно не выглядит прекрасным и порой может устрашать. Но на дне его, всего на несколько футов ниже поверхности, скрываются жемчужины значительной производительности. Не упустите шанс добыть их.

- *Терминология*
- *Уровни RAID*
- *Сравнение программных и аппаратных реализаций RAID*
- *Итог по конструкциям дисковых массивов*
- *Программные реализации RAID*
- *Рецепты RAID*
- *Заключение*

6

Дисковые массивы

Безусловно, ввод-вывод отставал последние десять лет.

Seymour Cray, 1976

Почти все улучшения в технологии дисков сводились к увеличению отношения емкость–стоимость. Несмотря на то что хранилища большой емкости значительно подешевели, такие изменения вызвали два достаточно серьезных осложнения:

- Когда простой диск хранит десятки гигабайт данных, надежность отдельного диска становится важным делом, поскольку неполадки с отдельным диском могут привести к потере большого объема данных.
- Повышение дисковой производительности чрезмерно отставало от улучшения показателя емкость–стоимость.

Для преодоления таких трудностей было предпринято много усилий. В результате появились методы организации набора дисков, разработанные для повышения надежности и производительности. Такие наборы дисков стали известны как RAID, что означает либо «массив недорогих устройств с избыточностью» («Redundant Array of Inexpensive Disks»), либо «массив независимых дисковых накопителей с избыточностью» («Redundant Array of Independent Disks»). Можно встретить оба варианта. Существует семь уровней RAID. В каждом из них реализован свой подход для преодоления описанных трудностей. Типы RAID представлены в табл. 6.1.

Таблица 6.1. Сводка уровней RAID

Уровень RAID	Организация	Сильные стороны	Слабые стороны
RAID 0	Разбивка на блоки (striping)	Очень быстрые, простые	Низкая надежность
RAID 1	Зеркалирование (mirroring)	Быстрые, простые	Дорогие
RAID 2	Надежность через коды Хэмминга (Hamming)	Надежные	Повышенные требования к конструкции
RAID 3	Контроль по четности	Быстрый последовательный доступ, надежные	Медленный произвольный доступ, сложность реализации
RAID 4	Контроль по четности	Средняя производительность, надежные	Узкие места на дисках, предназначенных для контроля четности
RAID 5	Контроль по четности	Хорошая производительность, надежные, дешевые	Медленная запись, большие требования к кэшу
RAID 10	Чередующиеся зеркала	Очень быстрые, простые	Дорогие

В компоновке дисковых массивов существует один основной компромисс. По существу, это классический пример одного из принципов настройки производительности (см. раздел «Принцип 1: БСНБ!» в главе 1). Компромисс можно представить так:

Быстро, *дешево*, *надежно*.

Из трех этих свойств можно выбрать только два.

В этой главе будут обсуждены основы объединения множества дисков в единый логический блок, а именно элементарная терминология, различия между программными и аппаратными дисковыми массивами, рецепты для их разработки и много других аспектов реализации современных массивов.

Терминология

Перед тем как начать обсуждение RAID, необходимо ввести несколько терминов:

- Так как дисковый массив содержит множество физических дисков, то термин *логический том* (*logical volume*) будет использоваться для различения тома, составленного из множества дисков.

- *Средняя наработка на отказ* (*mean time between failure*, MTBF) – это среднее время между отказами компонента. Например, значение MTBF заданного дискового накопителя может быть указано как 500 000 часов. Это означает, что типичный диск этой модели будет работать полмиллиона часов перед отказом. Так как это всего лишь статистическая оценка, то отдельный дисковый накопитель вполне может находиться пятьдесят (или миллион) часов в ремонте. Такие оценки отдельных компонентов можно применить ко всей системе. Например, для массива емкостью 9 Тбайт, собранного из тысячи дисков по 9 Гбайт с MTBF, равным 1,2 миллиона часов, общее значение MTBF составит 1 200 часов (около пяти дней).

$$\frac{\text{MTBF компонента}}{\text{Количество компонентов}} = \text{MTBF системы}$$

- *Среднее время до потери данных* (*mean time to data loss*, MTTDL) – период времени, в течение которого система может работать до возникновения существенных неполадок, связанных с потерей данных. Если все компоненты работают независимо, то значение MTTDL эквивалентно MTBF наименее надежного компонента. Однако разработчики приложили немало усилий для того, чтобы увеличить MTTDL – несмотря на то, что значения MTBF-компонентов неизменны. Например, в зеркальной паре дисков данные могут быть потеряны лишь в случае, если возникнут неполадки с обоими дисками. Вероятность такого события намного меньше вероятности неполадок с одним диском. Важно отметить, что высокое значение MTTDL не означает непрерывную работоспособность: неполадки с хост-контроллером могут сделать зеркальную пару дисков недоступной, а данные по-прежнему будут целы.
- *Среднее время до недоступности данных* (*mean time to data inaccessibility*, MTTDI) формально не определено. Однако становится все более важным иметь постоянный доступ к данным, а не просто предотвращать их потерю. MTTDI – это период времени, в течение которого система может работать до возникновения существенных неполадок, связанных с недоступностью данных.
- Хотя это не будет здесь обсуждаться, *среднее время до восстановления* (*mean time to repair*, MTTR) часто является важным показателем, который обязательно следует отслеживать. Как будет показано далее, потеря диска в дисковом массиве может привести к снижению производительности и к опасности потери данных. Знание того, сколько времени необходимо для восстановления системы после отказа диска (включая время, затрачиваемое на замену неисправного диска), может быть весьма полезно.

Уровни RAID

RAID определен на нескольких уровнях, каждый из которых описывает различные механизмы повышения производительности и надежности. Уровни нумеруются от нуля до пяти (заметим, что RAID 10 не включен; это комбинация двух уже описанных уровней RAID). Первый уровень RAID, RAID 0, приносит только увеличение производительности, совершенно не повышая надежность, тогда как RAID 1 большей частью дает улучшение надежности. RAID 2 использует сложный алгебраический метод для достижения надежности данных. Другие уровни применяют механизмы защиты по четности для повышения надежности, а также увеличения производительности.

Сердцевиной систем RAID с защитой по четности (уровни 3, 4 и 5) является математически простое вычисление реверсивного равенства. Каждая коммерческая реализация RAID с защитой по четности использует свои вариации этого алгоритма. Сам алгоритм сводится к представлению данных в виде единой строки и применению к ней *побитовой функции исключающего ИЛИ* (*bitwise exclusive-or function*, XOR) с целью получения бита четности:

$$P = d_0 \oplus d_1 \oplus \ldots d_{n\text{-}1} \oplus d_n$$

Функция XOR обладает свойством реверсивности. Если ее применить дважды, то будут получены первоначальные данные. Когда отказывает любой диск-участник системы, данные могут быть вычислены на основе данных, извлеченных с работоспособных дисков:

$$d_{\text{отсутствующий}} = P \oplus d_0 \oplus d_1 \oplus d_2 \ldots d_{\text{отсутствующий-1}} \oplus d_{\text{отсутствующий+1}} \oplus \ldots \oplus d_n$$

Такие тома могут вынести отказ любого отдельного диска. Значение надежности массива с защитой по четности с n дисками-участниками может быть получено так:

$$\text{MTFB}_{\text{массива с защитой по четности}} = \frac{\text{MTBF}^2_{\text{диска-участника}}}{n-1}$$

Вот одно очень типичное неверное представление: в операциях чтения системы RAID с защитой по четности используют механизм чтения и сравнения (read-and-compare). Обычно это не так.[1] На уровне диска и протокола существуют устойчивые механизмы обнаружения ошибок, а издержки системы чтения и сравнения были бы значительны.

В некоторых конфигурациях RAID надежность можно повысить за счет добавления томов «горячего резервирования», так как в случае

[1] Некоторые очень параноидные отказоустойчивые приложения проводят вычисление контрольной суммы данных на уровне драйвера устройства.

отказа одного диска резерв немедленно доступен (MTTR существенно снижается). Такой подход снижает статистическую вероятность потери данных. К сожалению, одноканальные массивы с защитой по четности могут испытывать сложности из-за отказа контроллера. В основной системе это воспринимается как отказ совокупности дисковых накопителей. Дисковые массивы, скомпонованные с избыточными контроллерами, устойчивы к таким событиям.

RAID 0: разбивка на блоки (striping)

На первом уровне RAID физические дисковые накопители разбиты на *блоки* (*stripes*), совокупность которых представляет собой единый логический блок. Каждый дисковый накопитель разделен на элементы, называемые *участками* (*chunks*). Соседние логические участки хранятся на последовательных дисках. Размер участка часто называют *размером элемента чередования* (*interlace unit size*) или *размером элемента блока* (*stripe unit size*). *Ширина блока* (*stripe width*) – это произведение размера участка на количество дисков в массиве. Рисунок 6.1 показывает, как в массиве RAID 0 данные распределяются по физическим дискам.

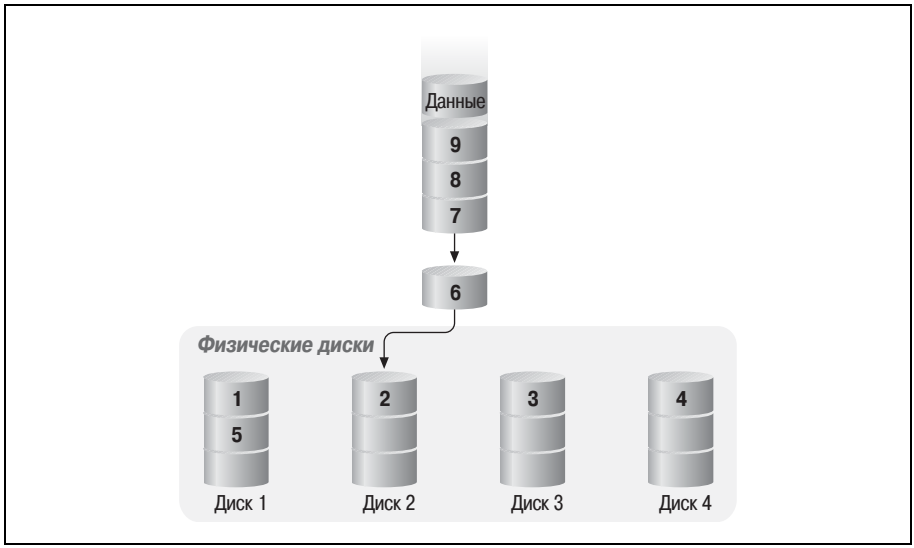


Рис. 6.1. *Организация дисков в RAID 0*

Такая схема размещения означает, что во многих случаях запрос обслуживается путем обращения к нескольким дискам. Последовательная производительность в большой степени определяется временем, требуемым для передачи данных с пластин. Поэтому производительность будет оптимальна, когда размер запрашиваемых данных будет равен ширине блока. Дополнительным преимуществом такой схемы

является равномерное распределение дисковой нагрузки среди всех дисков. При этом снижается использование (utilization) каждого устройства массива и улучшается производительность произвольного доступа. Напомним, что производительность произвольного доступа является функцией времени поиска. Чем короче поиск, тем скорее будет обработан следующий запрос. Ясно, что массивы RAID 0 повсеместно увеличивают производительность дисковой подсистемы.

Типичным вымыслом является то, что разбивка на блоки увеличивает только последовательную производительность. Это не так. Производительность произвольного доступа существенно улучшается, так как в массиве из N устройств использование каждого диска снижается в 1/N раз. Чем меньше использование, тем выше производительность ввода-вывода.

В четырехдисковом массиве с размером чередующихся участков 32 Кбайт запрос данных объемом 128 Кбайт будет обработан за счет чтения четырех последовательных участков, каждый из которых находится на отдельном физическом диске. Последовательная производительность логических томов, образованных разбивкой на блоки, приближается к величине, равной произведению пропускной способности каждого диска-участника на количество дисков в массиве. Таким образом, на последовательную производительность не накладываются ограничения операционной системы, хост-адаптеров SCSI, контроллера массива и т. д. Если каждый накопитель обладает внутренней скоростью передачи 6 Мбайт/с, то массив должен поддерживать скорость около 24 Мбайт/с. В однопоточной операции производительность произвольного доступа существенно не повышается, так как производительность будет определяться временем поворота и временем поиска, которые характерны для дисков-участников. В многопоточной последовательной операции запросы к дискам-участникам производятся в произвольном порядке, поэтому в массиве, по существу, присутствует произвольный тип доступа.

Как правило, в RAID 0 производительность не увеличивается линейно, поскольку для дисков-участников не характерна *синхронизация шпинделей*: вращение каждого диска и поиск на нем осуществляются индивидуально. Более подробно синхронизация шпинделей будет обсуждена позднее, когда речь пойдет о массивах RAID 3. Кроме того, в устройствах RAID 0 производительность не повышается, если обращение к «горячим участкам» происходит намного чаще, чем к остальной части тома. Это приводит к неравномерному распределению ввода-вывода среди дисков-участников. Вот один пример – *именованный канал* (*named pipe*). При постоянной записи данных по этому каналу происходит почти непрерывное обновление времени последней модификации, что приводит к выраженному «горячему участку». Один из хитрых способов обойти такие трудности – это установка *sticky bit* (с помо-

щью *chmod +t filename*), что приведет к задержке обновления соответствующего поля в записи каталога. Это немного опасный способ, поскольку в случае аварии системы последняя дата модификации будет неверной. Однако ни данные, посылаемые по каналу, ни целостность файловой системы риску не подвергнутся.

Наихудшая производительность массива RAID 0 бывает в том случае, когда размер запроса очень маленький, причем запрос попадает на границу между участками. Производительность блока почти всегда будет ниже производительности одного диска, если размер запроса равен 1 Кбайт, а данные находятся в двух участках. В этом случае будут задействованы два накопителя – время задержки повысится, а пропускная способность не увеличится.

Выбор размера элемента блока (stripe unit size) в какой-то мере сложен. Он может оказать существенное влияние на производительность. В средах с произвольным доступом производительность оптимизируется за счет вовлечения максимально возможного количества дисков. Поскольку в обработку любого запроса вовлечены только один или два диска, то использование дисков минимизируется, когда их количество максимально. В этом случае размер участка должен быть скорее большим – около 128 Кбайт. Кажется, что это противоречит интуиции, однако следует предусмотреть как можно большее количество активных дисков, каждый из которых вовлечен в обработку отдельного запроса. Производительность в средах с последовательным доступом оптимизируется, когда размер запроса ввода-вывода равен ширине блока, а размер участка достаточно велик, чтобы издержки ввода-вывода того стоили.

Прирост производительности в RAID 0 не даруется просто так. Распределение данных по дискам означает, что целостность логического тома зависит от каждого диска-участника. В случае отказа одного диска весь том становится непригодным. Регенерировать том можно только за счет замены отказавшего компонента и восстановления данных с резервного носителя.

RAID 1: зеркалирование

RAID 1 был разработан с целью преодолеть проблемы надежности, характерные для массивов RAID 0 и отдельных дисков. Задействованная методика, часто называемая зеркалированием (mirroring) или горячим резервированием (shadowing), основана на том, что отдельные диски вполне надежны.[1] Однако в больших системах, где дисковое хозяйство может содержать 1500 дисков с показателем MTBF, равным 1 000 000 часов для каждого диска, система столкнется с отказом дис-

[1] В современном диске типичное значение MTBF составляет около 1 000 000 часов, или примерно 114 лет и 56 дней.

ка примерно через 28 дней. Для информационного центра такой надежности обычно недостаточно. Центральный принцип RAID 1 – располагать данные таким образом, чтобы отказ отдельного диска не вызвал потерю данных. В схеме зеркалирования для каждого диска, содержащего данные, резервируется по крайней мере один дополнительный диск. Такие диски называются *субзеркалами* (*submirrors*). Во время операции записи данные раздельно записываются на основной диск и на каждый из резервных. При чтении данные могут быть извлечены с любого из дисков. Если отказывает один из дисков-участников, то данные восстанавливаются с работоспособных субзеркал. Рисунок 6.2. иллюстрирует организацию дисков в RAID 1.

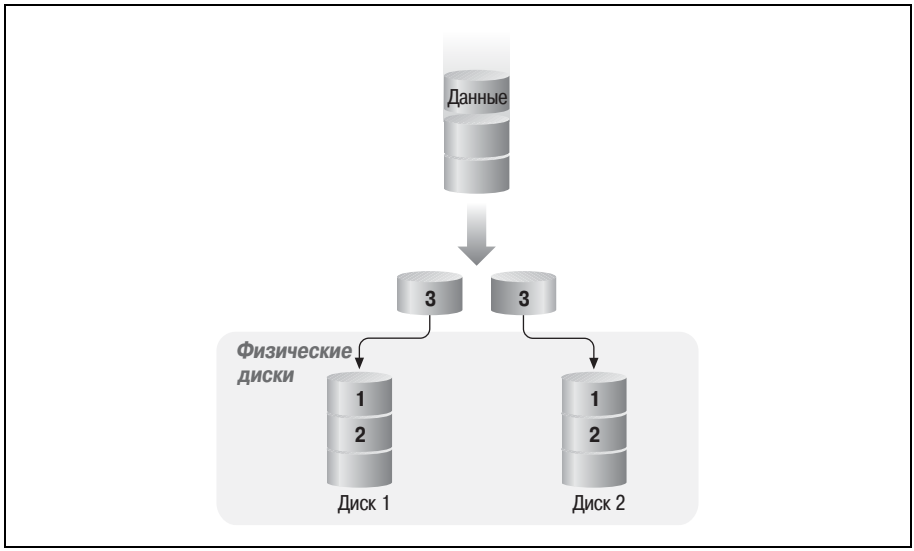


Рис. 6.2. *Организация дисков в RAID 1*

По существу, двойного зеркалирования вполне достаточно. Однако в некоторых случаях полезно тройное зеркалирование, например для сохранения текущих данных без того, чтобы подвергать риску надежность рабочего тома. Несмотря на то что некоторые реализации позволяют и большее зеркалирование, это избыточно. Благодаря зеркалированию надежность значительно повышается: двойное зеркалирование дисков с показателем MTBF, равным 1 000 000 часов для каждого диска, дает значение MTTDL, равное $5{,}0 \times 10^{11}$ часов.

Производительность зеркалированного устройства различна. Операции записи должны выполняться на всех дисках-участниках, поэтому даже в самом лучшем случае производительность слегка уменьшается по сравнению с производительностью отдельного диска. Большинство механизмов зеркалирования предлагает две стратегии для управления операциями записи:

Стратегия параллельной записи на каждом диске-участнике

Вызывает последовательную отправку запросов на запись, однако эти запросы обслуживаются параллельно. Так как продолжительность отправки намного меньше времени физической записи, то записи на зеркала, по существу, происходят параллельно. При двойном зеркалировании производительность операции записи составляет 80% производительности той же операции на отдельном диске, а при тройном зеркалировании – 65%. Снижение происходит из-за задержек поиска и вращения.

Стратегия последовательной записи на каждом диске-участнике

Представляет собой отправку первого запроса на запись, ожидание завершения его обработки, а затем выдачу второго запроса на запись. При N зеркалах операция записи продолжается в N раз дольше, чем запись на отдельный диск.

Производительность операции чтения с зеркала существенно отличается. В контексте однопоточной активности (как произвольной, так и последовательной) производительность примерно такая же, как и производительность отдельного диска-участника. В «параноидной» реализации данные могут считываться со всех дисков-участников и затем сравниваться, однако это делается лишь в полностью отказоустойчивых системах. Наилучшая производительность достигается при чтении с произвольным доступом, где для обработки запроса система может выбрать наименее занятый диск. Интенсивность использования каждого диска при наличии N зеркал сводится к 1/N.

В случае отказа зеркалированного диска он должен быть со временем заменен. Процесс доведения заменяющего диска до отображения текущего состояния данных называется *ресинхронизацией* (*resynchronization*). Самый прямолинейный способ – просто скопировать все биты с работоспособных дисков на диск-заменитель. Именно так и следует поступить, если требуется полная ресинхронизация. Для того чтобы минимизировать влияние ресинхронизации на работу системы, такой процесс проводится на сниженной скорости (около 1 Мбайт/с/диск). Однако полная ресинхронизация не всегда необходима. Иногда субзеркало изымается временно, например для резервного сохранения текущего состояния данных. В этом случае нужно скопировать лишь измененные блоки. Такой процесс обычно проводится с помощью *протокола «грязных» участков* (*dirty region log*, DRL), в котором хранится битовый образ каждого диска. Каждый бит определяет, был ли дисковый участок изменен с момента изъятия субзеркала. Типичный участок имеет размер 32 Кбайт, поэтому размер DRL вполне мал, а его наличие оказывает минимальное влияние на производительность.

RAID 2: массивы с кодом Хэмминга (Hamming Code)

Один типичный алгоритм для обнаружения поврежденных данных[1] сводится к применению кода четности, когда в заданной последовательности данных подсчитывается количество единиц. Бит четности равен 1, если количество 1 нечетно, и равен 0, если четно. Единичный бит четности может обнаружить ошибку, если какой-нибудь бит неверен, но не покажет, какой именно. Такая однобитная схема контроля четности официально называется *кодом с расстоянием 2*. Изменение одного бита приведет к тому, что контрольный бит четности тоже изменится. Конечно, если изменить два бита (два бита данных либо один бит данных и бит четности), то четность будет соответствовать данным, а ошибка не будет обнаружена: между правильными комбинациями четности и данных есть расстояние в два шага (изменения). Если нужно найти более одной ошибки или исправить ошибку, то необходима схема *код с расстоянием 3*, в которой расстояние между комбинациями «правильные данные/четность» равно трем изменениям.[2] Такое расстояние позволяет исправить ошибку в одном бите и определить, но не исправить ошибки двух бит. Количество бит четности, требуемое для схемы «код с расстоянием 3», растет медленно по сравнению с увеличением количества бит данных. Например, для 64 бит данных необходимо 7 бит четности, а для 128 бит – 8 бит четности. Такой тип кодирования называется *кодом Хэмминга* в честь Р. Хэмминга (R. Hamming), который описал метод синтезирования такого кода.

В RAID 2 этот алгебраический метод применяется для генерации соответствующей информации о четности, которая позволяет восстанавливать данные в случае дискового сбоя. Однако RAID 2 никогда не был широко распространен вследствие сложности алгоритмов генерации кода Хэмминга. Такие алгоритмы подразумевают аппаратную поддержку и налагают ограничения на количество и организацию дисков в массиве.

RAID 3: разбивка на блоки с защитой по четности

В RAID 3 предпринята попытка преодолеть сложности, связанные с низкой надежностью RAID 0 и высокой стоимостью RAID 1. RAID 3 основан на идее разбивки на блоки, но использует дополнительный дисковый накопитель для хранения вычисленных данных четности. Система вычисляет контрольный бит четности для каждого бита по

---

[1] В русскоязычной специальной литературе также принято употреблять термин «обнаружение ошибок», например «корректирующие коды с обнаружением ошибок». – *Примеч. науч. ред.*

[2] Существует разница между «кодами контроля четности» и «кодами исправления ошибок» (error-correction codes, ECC). Защита по четности позволяет обнаружить повреждение данных. Помимо этого ECC позволяет их исправить.

ширине блока и сохраняет данные четности на соответствующем участке дополнительного диска. Реализовать RAID 3 технически очень трудно, так как размер чередования изменяется от 1 до 32 бит, а современные диски SCSI используют блоки 512 байт. Большинство производителей, заявляющих о реализации схемы «RAID 3», на самом деле воплощают схему RAID 5, но с данными о четности, хранимыми на отдельном диске.

Поскольку массив RAID 3 основан на структуре RAID 0, он обладает теми же преимуществами: очень хорошая производительность операций с данными большого размера и низкая производительность маленьких операций (то есть тех операций, в которые вовлечены только два диска). Несмотря на то что само по себе вычисление четности не приводит к существенному (>5%) снижению производительности, диск четности вовлекается во все операции записи. В операциях чтения контроль четности не задействован, поэтому производительность системы примерно та же, что и в случае RAID 0. То же самое можно сказать про однопоточные операции записи. Однако в случае многопоточных операций записи могут возникнуть серьезные трудности, так как диск, применяемый для контроля четности, становится перегруженным. Многопоточные операции записи, осуществляемые в томе RAID 3, очень схожи с записями на один диск. В некоторых средах это приемлемо. Для большинства универсальных серверов RAID 3 не является хорошим выбором.

RAID 3 предназначен для оптимизации запросов, которые задействуют все устройства в массиве. Так как каждый запрос к туму RAID 3 должен ждать завершения всех физических вспомогательных запросов, то задержку можно снизить за счет приведения всех дисков-участников в одну и ту же позицию. Такой процесс, называемый *синхронизацией шпинделей* (*spindle synchronization*), в значительной степени оптимизирует последовательные однопоточные запросы. Производительность операций с произвольным доступом ужасна: массив ведет себя как один очень большой, очень надежный дисковый накопитель с той же произвольной частотой запросов к логическому туму, что и частота запросов к диску-участнику. По всем этим причинам RAID 3, по сути, никогда не используется.

RAID 4: разбивка на блоки с защитой по четности и независимыми дисками

Массив RAID 4 является расширением RAID 3. Вместо того чтобы применять маленькие участки, их размеры выбраны в соответствии с размерами в массивах RAID 0 (между 2 и 128 Кбайт). Кроме того, в RAID 4 не нужно синхронизировать шпиндели дисков-участников. Поэтому если речь идет об обычных дисках SCSI, то реализовать массив RAID 4 намного легче, чем RAID 3. Практически все устройства,

объявленные как «RAID 3», на самом деле реализованы как массивы RAID 4.

По существу, RAID 4 имеет те же сильные и слабые стороны, что и RAID 3. Он превосходен в последовательной пропускной способности и считываниях, однако производительность операций записи так же мала, как и у отдельного диска-участника.

RAID 5: разбивка на блоки с участками четности, распределенными по ширине блока

Слабость конструктивных особенностей RAID 3 состоит в том, что он не в полной мере учитывает специфику многозадачных сред. В массиве RAID 5 для данных четности не выделяется отдельный диск или участки нескольких дисков. Вместо этого блоки данных для контроля четности и блоки реальных данных в RAID 5 перемешаны (рис. 6.3).

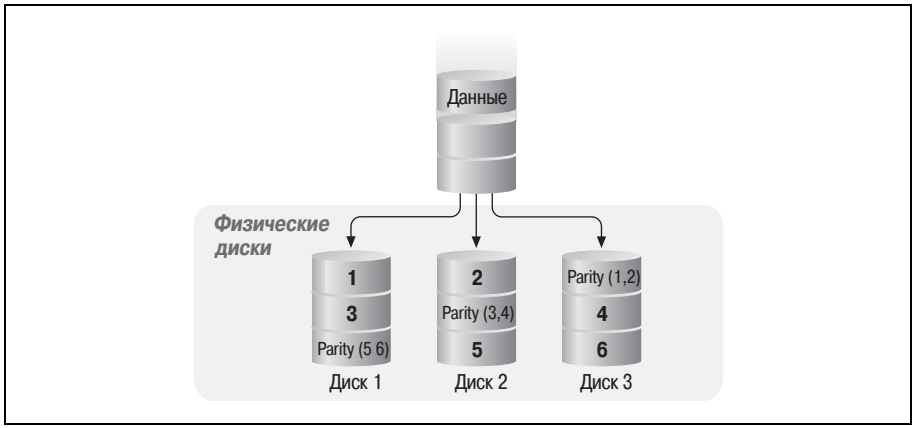


Рис. 6.3. *Массив RAID 5*

Одна из ключевых истин, касающихся всех конструкций RAID с контролем четности, состоит в том, что они преобразуют логические операции записи во множественные физические операции. Для поддержки целостности данных такие операции записи должны совершаться атомарно, иначе вычисленные данные контроля четности и реальные данные могут потерять синхронизацию, что приведет к повреждению данных. Наиболее типичное решение основано на двухэтапном процессе фиксации. Когда выдан запрос на запись, вычисляется блок четности, который записывается в буфер. При этом обновляется протокол – появляется отметка о том, что проводится запись данных. Как только данные оказываются в надежном энергонезависимом хранилище, программное средство управления массивом записывает обновленные данные и данные четности на диски-участники, а в протоколе отражается окончательный статус таких блоков. Если этот процесс был где-то

прерван, то программа управления массивом может восстановить состояние тома, предшествовавшее записи (consistent state). Протокол обновляется частыми операциями записи, но поскольку они ожидают выполнения лишь несколько миллисекунд, протокол очень короток.

Зачастую производительность массивов RAID 5 трактуется неверно, особенно если речь идет о производительности операций записи. Но сначала рассмотрим простую операцию чтения. Однопоточное чтение в массиве RAID 5 происходит почти с такой же скоростью, как и в блоке (stripe), в котором количество дисков на единицу *меньше*. Например, быстродействие шестидискового массива RAID 5 соответствует быстродействию пятидискового массива RAID 0. Причина состоит в том, что в RAID 5 один диск предназначен для данных четности. Максимальная производительность достигается тогда, когда запрос выровнен по границе блока, а размер запроса кратен его ширине. Многопоточные операции чтения (то есть множественные одновременные считывания) в томах RAID 5 выполняются быстрее тех же считываний в томах RAID 0. Благодаря дополнительному диску, задействованному в RAID 5, нагрузка распределяется на большее количество дисков. Как и в массивах RAID 0 и RAID 3, производительность RAID 5 минимальна, когда размер запроса очень мал и захватывает два диска-участника – при этом в работу вовлечены три диска (два диска данных и диск четности).

К сожалению, тема производительности операций записи более сложна. Первая трудность состоит в том, что операции записи обычно содержат цикл чтение–модификация–запись (read-modify-write). Для того чтобы вычислить данные четности, необходимо прочитать остальные данные блока (*контекст четности*, *parity context*). Далее вставляются новые данные, а затем данные и информация о четности записываются обратно на диск. При этом возможны три ситуации:

- Операция записи изменяет данные по всей ширине блока. Старый контекст четности не считывается, поскольку он все равно будет перезаписан. Поскольку происходит запись сведений о четности, то величина производительности, соответствующая одному блоку, вычисляется так:

$$1 - \frac{n-1}{n}$$

- Операция записи изменяет данные более чем на одном диске данных, но не по всей ширине блока. Необходимо восстановить контекст четности, что сводится к циклу чтение–запись–модификация: два считывания, две записи данных и одна запись сведений о четности. Производительность составляет около 25% производительности отдельного диска.
- Операция записи изменяет данные только на одном диске. Восстанавливать контекст четности нет необходимости, поскольку его мож-

но вычислить – как в случае отказа диска. Операция порождает одну запись данных и одну запись сведений о четности. Производительность составляет около 50% производительности отдельного диска.

К сожалению, на практике дело обстоит еще сложнее. Так как для достижения надежности операции записи проводятся в два этапа, то необходимы три дополнительных операции: две для обновления протокола «грязных» участков и одна для записи обновленных данных и сведений о четности. Хуже того, записи в протокол и записи данных должны проводиться в строго последовательном порядке.

Массив RAID 5 сложен, и принятие решения о варианте его конфигурации может быть трудным. Важнее всего стремиться к тому, чтобы операции записи проводились по всей ширине блока. Если в работе приложения преобладает какая-то одна транзакция записи, то конструкцию массива следует согласовать с размерами данных, задействованных в такой операции. Для этого ширина данных тома должна соответствовать типичному размеру ввода-вывода. Если типичный размер ввода-вывода составляет 128 Кбайт, а в массиве RAID 5 есть пять дисков, то размер участка должен быть 26 Кбайт:

$$\text{размер участка} = \frac{\text{типичный размер ввода-вывода}}{\text{количество дисков} - 1}$$

Слабым местом RAID 5 является производительность операций записи, особенно операций с маленькими файлами. Это становится особенно заметным при использовании тома RAID 5, в котором много операций записи производится через NFS V2. Причина состоит в том, что в NFS V2 возможны маленькие размеры ввода-вывода. Более подробно это будет обсуждено в разделе «NFS» главы 7.

Когда в массиве происходят неполадки, показатели производительности существенно изменяются. В обычном режиме в операции чтения не вовлекаются все диски. Кроме того, такие операции не выполняют вычислений четности. Когда же произошел отказ диска, то для того чтобы восстановить информацию, необходимо прочитать данные всех дисков-участников и применить механизм контроля четности (описанный в разделе «Уровни RAID» ранее в этой главе). Когда массив RAID 5 работает в замедленном режиме, одно дисковое считывание требует $2(n-1)/n$ операций чтения, в отличие от n операций чтения в полном рабочем режиме. Интенсивность использования каждого диска почти удваивается. Хорошая новость состоит в том, что отказ диска не оказывает существенного влияния на производительность операций записи.

RAID 10: зеркалированная разбивка на блоки

В настоящее время простые массивы RAID 1 довольно редки, так как требования к хранилищам превзошли пределы емкости отдельных

физических дисков. В результате общепринятой практикой на сегодня является создание зеркал, чьими субзеркалами являются блоки (stripes). Такие массивы часто называют RAID 10, RAID 1+0 или RAID 0+1. Рисунок 6.4 иллюстрирует организацию RAID 1+0.

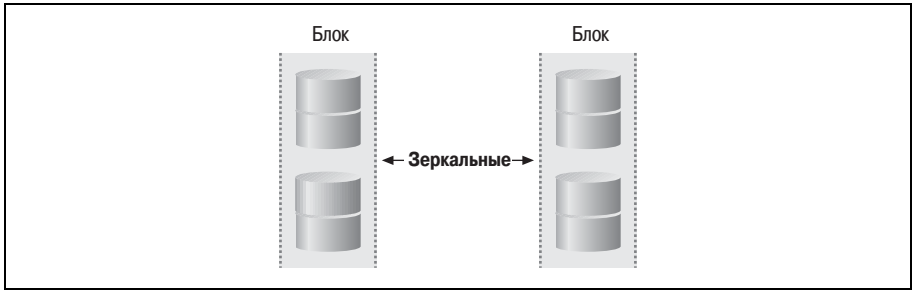


Рис. 6.4. *Организация RAID 1+0*

В массиве RAID 10 производительность операций чтения повышается как из-за разбивки на блоки, так и из-за зеркалирования, однако улучшение произвольного многопоточного считывания происходит только вследствие зеркалирования. В RAID 10 операции записи обычно на 30% медленнее по сравнению с массивами, в которых тома организованы исключительно из блоков, что объясняется издержками при записи двух копий.

Порядок создания массива RAID 10 очень важен. Существует два механизма построения таких устройств (рассмотрим пример с четырьмя дисками):

- Создаются два блока,[1] каждый из двух дисков. Затем они объединяются для создания зеркала. Получается одно зеркало, чьими субзеркалами являются блоки. Это схема RAID 0+1 (рис. 6.5).
- Создаются два двойных зеркала, а затем они объединяются в блок. Получается блок, компонентами которого являются зеркала. Это схема RAID 1+0 (рис. 6.6).

Такое разграничение может показаться теоретическим, однако последствия при отказе диска в этих двух схемах существенно отличаются. Когда неполадки с диском происходят в RAID 0+1, перестает быть активным все подзеркало. Любой последующий отказ подвергает опасности весь массив. В массиве RAID 1+0 отказ отдельного диска ухудшает только этот компонент блока: от потери данных массив из N

[1] В таком контексте в литературе можно вместо термина «блок» встретить термин «том», так как блоком привычнее называть часть диска, а логическое объединение нескольких дисков или частей нескольких дисков привычнее называть томом. Когда речь идет о блоке в RAID5, блок (stripe) является частью диска, а в RAID10 вырастает до набора частей разных дисков. – *Примеч. науч. ред.*

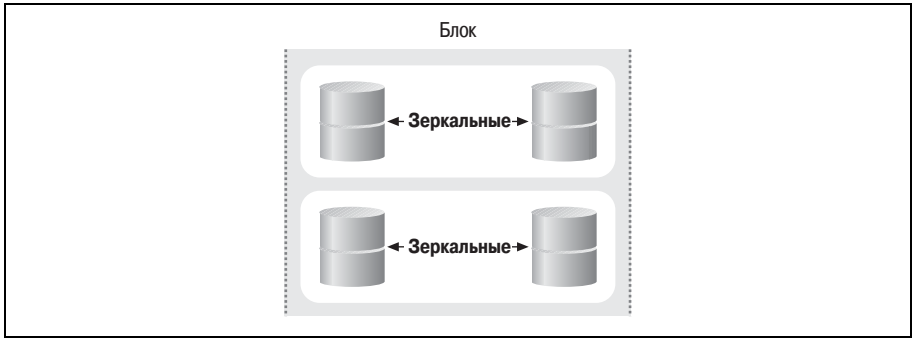


Рис. 6.5. *RAID 0+1*

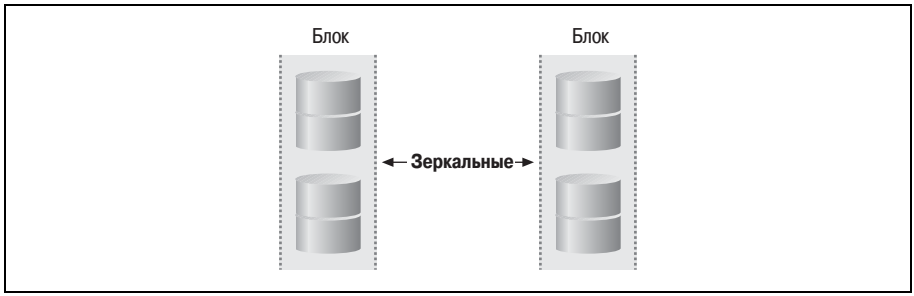


Рис. 6.6. *RAID 1+0*

устройств отделяют от двух (в худшем случае) до N/2 (в лучшем случае) отказов. К сожалению, ответы производителей на вопрос о варианте конструкции зачастую весьма туманны. Их информацию необходимо проверять.

Пакет Solstice DiskSuite примечателен тем, что он заставляет сначала создать блоки, а затем их зеркалировать (RAID 0+1). Такой подход кажется неверным, однако на самом деле Solstice DiskSuite реализует правильную схему (RAID 1+0).

Сравнение программных и аппаратных реализаций RAID

Обычно массивы RAID реализуются либо на базе программного обеспечения, либо с помощью выделенного аппаратного контроллера массива. Конечно, производители аппаратных RAID утверждают, что их аппаратные контроллеры лучше. В свою очередь, компании-разработчики программных средств говорят, что лучше их программные продукты. Несмотря на то что обе схемы выполняют одни и те же функции, различия между ними существенны.

Программное обеспечение

При программном управлении массивом базовые физические диски видимы для администратора. Функции RAID реализованы через драйвер виртуального устройства. Бонусом этой схемы является гибкость: такой массив может использовать любой доступный диск, вне зависимости от компоновки и расположения. Произвольное размещение дисков может быть полезным при необходимости построить массив, устойчивый к стихийным неполадкам (например, возгорание или затопление). Обычно такую схему критикуют за низкое быстродействие по сравнению с RAID на базе контроллера, особенно при вычислениях четности в массиве RAID 5. Хотя программные массивы действительно медленнее аппаратных, это не связано с производительностью вычислений четности. Типичный контроллер массива эквивалентен процессору Intel 486 66 МГц, который может выполнить восемь вычислений четности за пятнадцать циклов. Современные высокопроизводительные микропроцессоры рабочих станций могут совершать два вычисления четности за каждый тактовый цикл и имеют тактовую частоту в восемь раз выше! Кроме того, нагрузка, ложащаяся на систему при вычислении четности, незначительна.

Аппаратные средства

В RAID на базе аппаратных средств управление массивом осуществляет выделенный процессор. Нагрузка на систему по сравнению с программными RAID значительно снижается, так как программный массив должен управлять двумя группами операций ввода-вывода: логическими операциями и операциями с дисками-участниками. Операции с дисками-участниками могут быть достаточно большими, так как каждый ввод-вывод SCSI требует существенной доли времени процессора. Однако самая важная причина для выбора RAID на базе контроллеров заключается в том, что такие массивы обладают архитектурными преимуществами в использовании энергонезависимой памяти для кэширования протокола записи RAID 5. Контроллер может выполнить логическую операцию записи в память и уведомить об этом систему еще до того, как данные будут помещены на диск. Таким образом значительно снижается время задержки. Кроме того, в аппаратных массивах реализована внутренняя оптимизация, улучшающая работу медленных участков RAID 5, таких как объединение операций чтения–записи–модификации в операцию по всей ширине блока. Такая схема приносит значительные улучшения, если в системе с RAID 5 преобладают последовательные операции записи.

Преимущество кэшей NVRAM в массивах на базе контроллеров заключается в ускорении операций записи. Попытки ускорить операции чтения за счет размещения большого количества модулей NVRAM бессмысленны.

Если речь не идет о RAID 5, то в массивах на базе контроллеров преимущество от кэша NVRAM сводится к быстрым операциям записи. Контроллер может записывать данные в NVRAM, уведомлять о записи и завершать операцию на досуге. Для этой цели 16 Мбайт кэша вполне достаточно. В RAID 5 для эффективного и полного кэширования блоков необходимо 64–128 Мбайт.

Совмещение уровней RAID

Несмотря на то что реализации RAID на базе контроллеров превосходны для операций записи в RAID 5, такие аппаратные схемы менее значимы для других уровней RAID, особенно с тех пор, как издержки процессора при запуске RAID 0 или RAID 1 стали очень малы. Для достижения большего эффекта можно комбинировать программные средства и аппаратные контроллеры. Например, взять группу аппаратных RAID 5 и объединить их в блоки на основе программного RAID 0. Такой подход принесет высокую надежность данных (за счет уровня RAID 5) и замечательную производительность чтения (за счет RAID 0).

Итог по конструкциям дисковых массивов

По существу, разработка дискового массива предполагает создание списка наиболее важных свойств, характерных для имеющейся вычислительной среды. Понимание ее особенностей совершенно необходимо. Этому будут способствовать ответы на следующие вопросы:

- Преобладают ли в рабочей нагрузке операции чтения, записи или они сбалансированы?
- Какие типы доступа характерны? Большей частью это произвольный или последовательный доступ?
- Каков необходимый уровень устойчивости к неполадкам? Сколько дисковых отказов должно выдерживаться? Насколько восприимчива среда к снижению производительности в случае дискового отказа?

При разработке сложных дисковых подсистем такие вопросы следует внимательно обдумать.

Выбор уровня RAID

Критерий для выбора конструкции RAID прост и важен. Хотя об этом уже говорилось, повторим еще раз.

Быстро, дешево, надежно.

Из трех этих свойств можно выбрать только два.

Один достаточно типичный подход при выборе уровня RAID – это выбор двух наиболее значимых свойств в порядке их важности. После этого можно построить карту выбора, показанную на рис. 6.7.

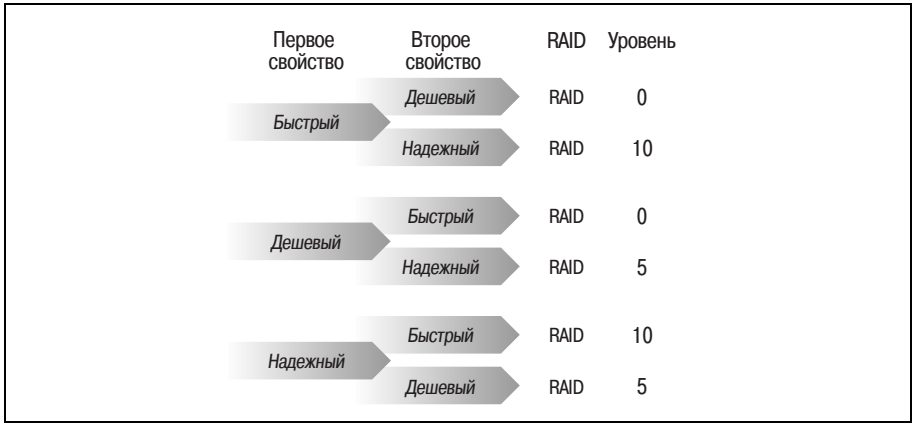


Рис. 6.7. *Карта RAID*

Программные реализации RAID

Многие администраторы с пренебрежением относятся к программным реализациям RAID, ссылаясь на их невысокие показатели производительности и надежности. Вообще говоря, авторы считают, что это не всегда так. Хотя программные реализации RAID зачастую довольно своеобразны и требуют внимания при установке, обычно они не настолько медленны, как утверждает молва, особенно дисковые массивы без контроля четности. Вероятно, их главный недостаток – это невозможность выполнять кэширование на уровне устройств. Кратко обсудим два самых распространенных программных продукта для создания массива: Solstice DiskSuite и пакет Linux *md*.

Solaris: Solstice DiskSuite

Solstice DiskSuite основан на концепции *метаустройств* (*metadevices*). Метаустройство – это группа физических участков (slices), которую система считает единым логическим устройством. Такие метаустройства рассматриваются как физические устройства. Метаустройство начинается с буквы *d*. Блочные метаустройства расположены в */dev/md/dsk* (например, */dev/md/dsk/d0*), а групповые метаустройства – в */dev/md/rdsk* (например, */dev/md/rdsk/d0*). По умолчанию в системе возможно 128 метаустройств, однако предел программного обеспечения равен 1024.[1] Авторы считают полезной практику, когда в имени

[1] Увеличить количество метаустройств можно путем изменения параметра `nmd` в */kernel/drv/md.conf*.

метаустройства кодируется расположение физического устройства. Например, диск с ID 4, закрепленный за контроллером 1, образует метаустройство *d14*. Составные запросы ввода-вывода физических устройств обрабатываются *драйвером метадиска* (*metadisk driver*). Поскольку такие метаустройства трактуются как физические устройства, то после того как они будут созданы, для генерации файловой системы необходимо воспользоваться утилитой *newfs*.

В SDS есть два различных способа решения одних и тех же задач. Существует графический интерфейс, вызываемый *metatool*, а также утилиты командной строки, на которых здесь будет сосредоточено внимание. Однако SDS – это чрезвычайно сложный продукт. Обсуждаемые здесь особенности являются лишь сердцевиной его конфигурации. По SDS компания Sun издает два замечательных технических руководства: «Solstice™ DiskSuite™ 4.2 Reference Guide» и «Solstice™ DiskSuite™ 4.2 User's Guide».

Базы данных состояния

В SDS сведения о конфигурации хранятся в *базе данных состояния* (*state database*), также называемой *базой метаинформации* (*metainformation database*). Поскольку такие сведения необходимы для надлежащей работы продукта, то на диске хранятся несколько копий (*реплик*, *replicas*). При записи данных каждая копия обновляется одна за другой. В системе должно существовать от трех до пяти реплик. По умолчанию размер каждой реплики равен 517 Кбайт.

В целях обеспечения целостности данных для реплик реализован *алгоритм согласованности большинства* (*majority consensus algorithm*). Согласно этому алгоритму, для работы системы должна быть доступна бульшая часть реплик (половина реплик, округленная в меньшую сторону, плюс одна). Если этот критерий не выполняется, то база данных считается *недействительной* (*stale*), а работа останавливается:

- Система работает нормально, если активна по крайней мере половина реплик.
- Работа системы нарушается, если доступно менее половины реплик.
- Система не перезагрузится, если большинство реплик недоступно.

В итоге реплики должны быть распределены среди максимально возможного количества дисков и контроллеров. Это снизит вероятность того, что отказ отдельного компонента вызовет недействительность всей базы данных.

Обычно реплики размещаются либо на выделенных разделах, либо на участках, являющихся частями метаустройств.[1] Авторы всегда разме-

[1] Если размещать реплику на участке, который планируется использовать для метаустройства, то реплику на участок необходимо добавлять до добавления участка к метаустройству.

щали их на участке 3 или участке 7, но для этого нет технических оснований. Общим правилом могут служить рекомендации табл. 6.2.

Таблица 6.2. Выбор количества реплик

Количество дисков	Количество реплик	Размещение
1	3	Все на одном участке
2–4	4–8	2 на диск
Более 5	5+	1 на диск

Управлять репликами базы данных состояния можно с помощью команды *metadb*. Это довольно сложная команда. Ее работа обобщена в табл. 6.3.

Таблица 6.3. Сводка по использованию metadb

Команда	Описание
`metadb`	Выводит список всех сконфигурированных реплик
`metadb -i`	Выводит список неактивных реплик
`metadb -a -f` *slice*	Создает новые реплики
`metadb -a` *slice*	Создает дополнительные реплики
`metadb -d -f` *slice*	Удаляет неисправную реплику

Для того чтобы создать более одной реплики на участке, при генерации реплик можно указать *-c number-of-copies*. Если размер реплики по умолчанию слишком мал, то можно воспользоваться опцией *-l size* команды *metadb* для того, чтобы задать больший размер.

Запуск *metadb* без аргументов сообщит сведения обо всех репликах:

```
# metadb
        flags           first blk       block count
     a m  pc luo        16              1034            /dev/dsk/c0t0d0s7
     a    pc luo        1050            1034            /dev/dsk/c0t0d0s7
     a    pc luo        16              1034            /dev/dsk/c0t8d0s7
     a    pc luo        1050            1034            /dev/dsk/c0t8d0s7
```

Флаги, представленные строчными буквами, означают состояние «Все в порядке» («Okay»). Флаги, представленные заглавными буквами, свидетельствуют о проблемах. Информация о возможных флагах обобщена в табл. 6.4.

Таблица 6.4. Флаги состояния metadb

Флаг	Описание
o	Реплика активна до последнего изменения конфигурации
u	Реплика синхронна (up to date)

Таблица 6.4 (продолжение)

Флаг	Описание
l	Локатор реплики был прочтен успешно
c	Реплика размещалась в */etc/opt/SUNWmd/mddb.cf*
p	Реплика размещалась в */etc/system*
a	Реплика активна
m	Реплика является мастер-репликой
W	Реплика имеет ошибки записи
R	Реплика имеет ошибки чтения
M	Реплика имеет проблемы с мастер-блоками
D	Реплика имеет проблемы с блоками данных
F	Реплика имеет проблемы формата
S	Реплика слишком мала, чтобы хранить текущую базу данных состояния

Неактивные реплики базы данных можно легко найти с помощью *metadb -i*. Если база данных состояния недействительна (обычно из-за отказа диска), то ее работоспособность необходимо восстановить. Это можно сделать так. Сначала с помощью *metadb -i* получить список неактивных реплик, обратив внимание на количество копий на каждом участке. Затем применить *metadb -d -f replica-slice* для устранения неактивных реплик. Далее нужно выключить систему и заменить отказавший диск. Когда система снова загрузится, следует перечитать реплики с помощью *metadb -a -c number-of-copies replica-slice*.

Для первоначального создания базы данных состояния нужно запустить *metadb* с ключами *-a* и *-f*. Если в системе менее пяти дисков, то следует создать несколько копий на участок с помощью *-c number-of-copies*. Это необходимо для поддержания кворума в случае отказа диска. Если в разделе уже существует файловая система, то в нем нельзя создать базу данных состояния.

Вот пример того, как первоначально создать реплики базы данных состояния в системе с тремя дисками, а также две копии на каждый участок:

```
# metadb -a -f -c 2 c0t0d0s7 c0t1d0s7 c0t2d0s7
# metadb
        flags           first blk       block count
     a m  pc luo        16              1034            /dev/dsk/c0t0d0s7
     a    pc luo        1050            1034            /dev/dsk/c0t0d0s7
     a    pc luo        16              1034            /dev/dsk/c0t1d0s7
     a    pc luo        1050            1034            /dev/dsk/c0t1d0s7
     a    pc luo        16              1034            /dev/dsk/c0t2d0s7
     a    pc luo        1050            1034            /dev/dsk/c0t2d0s7
```

После того как база данных состояния создана, можно добавлять дополнительные реплики. Для этого при запуске команды не нужно указывать ключ *-f*. Общий формат команды таков: *metadb -a -c number-of-copies replica-slice*.

Как только база данных состояния создана, можно работать с метаустройствами.

RAID 0: блоки (stripes)

Блок может обслуживать только ту файловую систему, которая *не* используется во время загрузки системы и *не* участвует в процессе обновления/установки. То есть из рассмотрения следует исключить такие системы, как /, swap, */usr*, */var* или */opt*. Необходимым условием создания блока является наличие по крайней мере трех стабильных реплик базы данных состояния. Создать блок можно с помощью команды *metainit*:

```
# metainit d0 1 2 c0t1d0s0 c0t2d0s0 -i 64k
d0: Concat/Stripe is setup
```

В этом примере создано метаустройство *d0*, представляющее собой одиночный (1) блок из двух (2) участков, а именно *c0t1d0s0* и *c0t2d0s0*. Необязательный ключ *-i* устанавливает размер чередования 64 Кбайт. По умолчанию размер чередования равен 16 Кбайт. Вот пример создания блока из четырех устройств:

```
# metainit d1 1 4 c0t1t0s0 c0t2d0s0 c0t3d0s0 c0t4d0s0
d1: Concat/Stripe is setup
```

Так как значение размера чередования не было указано, то в этом массиве размер чередования равен 16 Кбайт.

Одним интересным применением блока является создание метаустройств, состоящих только из одного физического устройства:

```
# metainit d2 1 1 c1t0d0s0
d1: Concat/Stripe is setup
```

При зеркалировании полезна инкапсуляция, которая будет обсуждена далее.

RAID 1: зеркала

Зеркало включает в себя от одного до трех *субзеркал*. Субзеркало состоит из одного и более метаустройств, в которых данные распределены по блокам. Полезность инкапсуляции состоит в том, что можно создать субзеркало в виде блока с одним устройством. Зеркало (на жаргоне DiskSuite – *метазеркало*) можно использовать для любой файловой системы. Как и в случае с блоком, для зеркала необходимо наличие по крайней мере трех реплик базы данных состояния. В отличие от блока, зеркало можно применять для любой файловой системы.

Есть четыре случая, когда создание зеркала может понадобиться:

- в самом начале (неиспользованные участки);
- для файловой системы, которая может быть размонтирована;
- для файловой системы, которая не может быть размонтирована;
- для файловой системы root.

Действия в таких случаях слегка различны.

При создании зеркала с нуля необходимы три шага «высокого уровня». Первый – создание двух инкапсуляций с помощью *metainit*. Второй – применение *metainit* для конфигурирования метазеркала. Третий – использование *metattach* для добавления второго субзеркала к метазеркалу:

```
# metainit d10 1 1 c1t0d0s0
d10: Concat/Stripe is setup
# metainit d11 1 1 c1t1d0s0
d11: Concat/Stripe is setup
# metainit d50 -m d10
d50: Mirror is setup
# metattach d50 d11
d50: Submirror d11 is attached
```

Команда *metattach* присоединяет метаустройство *d11* к метазеркалу *d50*. При этом создается двойное зеркало и происходит ресинхронизация зеркал. Во время ресинхронизации все данные нового субзеркала перезаписываются на мастер-субзеркало. Следить за прогрессом ресинхронизации можно с помощью *metastat*.

Если необходимо создать зеркало для существующей файловой системы (скажем, */data* на *c2t0d0s0*) без резервного сохранения и восстановления, то это можно сделать достаточно быстро. Суть идеи сводится к такой последовательности действий: создать две инкапсуляции, затем создать зеркало, размонтировать файловую систему, принуждая систему обращаться к устройству-зеркалу, а не к физическому устройству, затем смонтировать файловую систему и, наконец, присоединить второе субзеркало:

```
# metainit -f d20 1 1 c2t0d0s0
d20: Concat/Stripe is setup
# metainit d21 1 1 c2t1d0s0
d21: Concat/Stripe is setup
# metainit d51 -m d20
d51: Mirror is setup
# umount /data
(Необходимо отредактировать файл /etc/vfstab. В нем нужно отразить, что
/data монтируется как /dev/md/dsk/md51, а не как /dev/dsk/c2t0d0s0 )
# mount /data
# metattach d51 d21
d51: Submirror d21 is attached
```

Ключ *-f* команды *metainit* вызывает создание метаустройства без разрушения существующей монтированной файловой системы. После выполнения команды *metattach* содержимое метаустройства *d21* уничтожается в ходе операции ресинхронизации.

Иногда файловая система не может быть размонтирована во время нормальной системной операции (например, */usr* на *c3t0d0s0*). Процедура зеркалирования такой файловой системы по существу такая же, как и зеркалирование размонтированной файловой системы, за исключением того, что вместо размонтирования и повторного монтирования выполняется перезагрузка:

```
# metainit -f d30 1 1 c3t0d0s0
d30: Concat/Stripe is setup
# metainit d31 1 1 c3t1d0s0
d31: Concat/Stripe is setup
# metainit d52 -m d30
d52: Mirror is setup
(Необходимо отредактировать файл /etc/vfstab. В нем нужно отразить, что
/data монтируется как /dev/md/dsk/md51, а не как /dev/dsk/c2t0d0s0)
# reboot
...
# metattach d52 d31
d52: Submirror d31 is attached
```

Зеркалирование файловой системы root (в этом примере на *c4t0d0s0*) вызывает дополнительные трудности. А именно следует воспользоваться специальной командой *metaroot*. Кроме того, перед тем как провести перезагрузку, необходимо сбросить все ожидающие, отмеченные в протоколе транзакции всех файловых систем:[1]

```
# metainit -f d40 1 1 c4t0d0s0
d40: Concat/Stripe is setup
# metainit d41 1 1 c4t1d0s0
d41: Concat/Stripe is setup
# metainit d53 -m d40
d53: Mirror is setup
# metaroot d53
# lockfs -fa
# reboot
...
# metattach d53 d41
d53: Submirror d41 is attached
```

Массивы RAID 5

Вот что следует иметь в виду при создании метаустройств RAID 5:

- Метаустройство должно состоять по меньшей мере из трех участков.

[1] Такая процедура будет работать только на системах Solaris/SPARC. При работе с Solaris/i86pc следует обратиться к документации Sun, описывающей эту процедуру.

- Значение размера чередования *критично* для хорошей производительности RAID 5; для более подробной информации следует обратиться к разделу «RAID 5: разбивка на блоки с участками четности, распределенными по ширине блока» ранее в этой главе.
- Чем больше участков содержит метаустройство RAID 5, тем дольше будут выполняться операции при сбое участка.
- В SDS метаустройства RAID 5 не могут быть разбиты на блоки или зеркалированы.

Хорошая новость – создавать метаустройства RAID 5 очень просто:

```
# metainit d60 -r c5t0d0s0 c5t1d0s0 c5t2d0s0 c5t3d0s0 c5t4d0s0
d60: RAID is setup
```

Так как размер чередования не был указан с помощью ключа *-i*, то в этом массиве размер чередования равен 16 Кбайт. Далее команда *metainit* начинает процесс инициализации, который можно наблюдать с помощью *metastat*. Прежде чем создавать новое метаустройство, необходимо дождаться завершения создания предыдущего.

Пулы горячего резервирования

Пул горячего резервирования (*hot spare pool*) – это совокупность участков, зарезервированных в качестве автоматической замены в случае отказа субзеркала или элемента RAID 5. Устройство горячего резервирования должно состоять из участков, а не из метаустройств. Хотя участок горячего резервирования может быть ассоциирован с одним или несколькими пулами, а пул может быть разделен между несколькими субзеркалами и массивами RAID 5, тем не менее каждое подзеркало или устройство RAID 5 может быть ассоциировано только с одним пулом горячего резервирования. Кроме того, участок горячего резервирования не может содержать реплику базы данных состояния.

Конфигурирование пула горячего резервирования включает в себя два этапа: создание пула горячего резервирования и ассоциирование с ним метаустройств:

```
# metainit hsp001 c6t0d0s0 c6t1d0s0
hsp001: Hotspare pool is set up
```

Как только создан пул горячего резервирования, с ним можно ассоциировать устройства:

```
# metaparam -h hsp001 d10
# metaparam -h hsp001 d11
```

После того как пул горячего резервирования первоначально сконфигурирован, к нему можно добавить дополнительные участки с помощью команды *metahs*:

```
# metahs -a hsp001 c6t2d0s0
```

Если необходимо добавить участок ко всем пулам горячего резервирования, присутствующим в системе, то вместо названия пула следует указать *-all*.

Linux: md

В Linux поддержка программных дисковых массивов осуществляется с помощью пакета драйвера многокомпонентного устройства (multiple device, *md*). Обсудим новую версию программного обеспечения RAID, которая интегрирована в рабочие ядра 2.4. Если такой вариант RAID необходимо организовать в системах с ядрами 2.0 или 2.4, то на эти ядра следует установить обновления, так как в этих ядрах нет прямой поддержки программного обеспечения RAID. В Linux пакет *md* поддерживает линейную схему организации дисков (или жесткое объединение дисков), а также RAID 0, 1, 5 и 0+1.

Прежде чем начать обсуждение поддерживаемых уровней RAID и их реализацию, кратко представим два базовых принципа, лежащих в ее основе: постоянный суперблок (persistent superblock) и размер участка (chunk size).

Постоянные суперблоки

В предшествующих версиях пакета *md* программный инструментарий RAID считывал конфигурационный файл */etc/raidtab* и инициализировал массив. Однако такой подход требует, чтобы файловая система, в которой находится файл */etc/raidtab*, была монтирована. Это вызывает затруднения, если предполагается загружаться с диска. В таком случае на помощь приходит постоянный суперблок. Если в файле */etc/raidtab* присутствует опция «постоянный суперблок» (persistent-superblock), то при инициализации массива в начало каждого диска-участника записывается специальный суперблок. За счет этого ядро может считывать конфигурацию устройства RAID с самих дисков, а не обращаться к файлу конфигурации, который может быть недоступен.

Размер участка

Понятие размера участка аналогично понятию размера чередования в DiskSuite. Это наименьшая «атомарная» порция, которая может быть записана на устройство. Пусть в устройстве RAID 0 с двумя дисками размер участка равен 4 Кбайт. Тогда при записи 16 Кбайт данных первый и третий участки размером 4 Кбайт будут записаны на первый диск массива, а второй и четвертый – на второй диск. Размер участка должен быть задан для всех уровней RAID, в том числе для линейной организации дисков. Размер участка следует задавать в килобайтах. В табл. 6.5 представлено четкое описание того, на что влияет размер участка при конфигурировании дисковых массивов всех типов.

Таблица 6.5. Описание влияния размера участка на дисковые массивы

Уровень RAID	Влияние
Линейная организация	Не оказывает влияния, какой бы размер ни был
RAID 0	Порции данных с размером участка последовательно записываются на каждый диск. Например, запись 24 Кбайт в массив из двух дисков с размером участка 8 Кбайт означает, что по 8 Кбайт данных параллельно записываются на оба диска, а затем оставшиеся 8 Кбайт записываются на первый диск
RAID 1	Для операций записи размер участка не важен, поскольку все данные должны быть записаны на все диски. В этом случае размер участка определяет, сколько данных будет считано с дисков-участников. Так как все активные диски содержат одни и те же данные, то операции чтения можно распараллелить, подобно тому как это делается в массиве дисков с блоками
RAID 5	В RAID 5 операции записи должны ко всему прочему обновлять информацию о четности. Размер участка – это размер данных четности. Если один байт записывается с массива RAID 5, то порции данных будут считаны со всех дисков данных (размер порции равен размеру участка). Затем будет подсчитана четность, и данные четности будут записаны на диск четности

В зависимости от задачи размер участка следует выбирать от 32 до 128 Кбайт.

Все дисковые массивы конфигурируются с помощью записи строк в */etc/raidtab*. Рассмотрим несколько примеров того, что представляют собой такие записи в конфигурационном файле для каждого типа RAID. Далее обсудим, как создать само устройство.

Линейная организация дисков

Устройство с линейной организацией дисков – это простое объединение существующих устройств. Оно не поддерживает никаких расширенных особенностей RAID, но может быть полезно, если приложение пользователя в нем нуждается. Иллюстрация строк файла */etc/raidtab* для линейного массива приведена в примере 6.1.

Пример 6.1. Запись /etc/raidtab для линейного массива

```
raiddev /dev/md0
   raid-level              linear
   nr-raid-disks           2
   chunk-size              32
   persistent-superblock   1
   device                  /dev/sdb1
   raid-disk               0
   device                  /dev/sdc1
   raid-disk               1
```

Поскольку пример конфигурации приводится в первый раз, рассмотрим эти строки пошагово:

- Первая строка определяет создаваемое устройство, а именно */dev/md0*.
- Строка `raid-level` описывает уровень RAID этого массива.
- Строка `nr-raid-disk` определяет, сколько дисков находится в массиве.
- Параметр `chunk-size` устанавливает размер участка (в Кбайт).
- Строка `persistent-superblock` действует как булев переключатель – следует ли создавать постоянный суперблок. Этот параметр должен быть всегда равен 1.
- Следующие четыре строки – это пары устройство–номер диска. Они должны всегда существовать в парах. Строка `device` – это путь к разделу диска, тогда как `raid-disk` – это номер диска в массиве целиком.

RAID 0: блоки (stripes)

По существу, блоки конфигурируются так же, как и линейные массивы:

```
raiddev /dev/md0
   raid-level              0
   nr-raid-disks           2
   chunk-size              32
   persistent-superblock   1
   device                  /dev/sdb1
   raid-disk               0
   device                  /dev/sdc1
   raid-disk               1
```

RAID 1: зеркала

Строки в конфигурационном файле для массива с зеркалами довольно похожи на строки для блоков и линейных массивов. Но есть одно ухищрение:

```
raiddev /dev/md0
   raid-level              1
   nr-raid-disks           2
   nr-spare-disks          1
   chunk-size              32
   persistent-superblock   1
   device                  /dev/sdb1
   raid-disk               0
   device                  /dev/sdc1
   raid-disk               1
   device                  /dev/sdd1
   spare-disk              0
```

Здесь есть два дополнительных параметра, непосредственно связанных друг с другом (эти строки выделены):

- Строка `spare-disk` следует за строкой устройства и определяет, что указанный диск будет использоваться для горячего резервирования. При отказе одного из дисков зеркала данные будут восстановлены на этот резервный диск.
- Строка `nr-spare-disks` определяет, сколько резервных дисков присоединено к массиву.

Массивы RAID 5

Для построения массива RAID 5 необходимо иметь по крайней мере три диска. Строки в */etc/raidtab* могут выглядеть так:

```
raiddev /dev/md0
   raid-level              5
   nr-raid-disks           7
   nr-spare-disks          0
   chunk-size              128
   persistent-superblock   1
   parity-algorithm        left-symmetric
   device                  /dev/sdb1
   raid-disk               0
   device                  /dev/sdc1
   raid-disk               1
   device                  /dev/sdd1
   raid-disk               2
   device                  /dev/sde1
   raid-disk               3
   device                  /dev/sdf1
   raid-disk               4
   device                  /dev/sdg1
   raid-disk               5
   device                  /dev/sdh1
   raid-disk               6
```

Для массива RAID 5 добавлена новая строка, а именно строка `parity-algorithm`, в которой следует оставить значение `left-symmetric`. При желании можно добавить резервные диски.

Создание массива

После завершения редактирования файла */etc/raidtab* можно создать сам массив. Для этого нужно запустить *mkraid device*, указав устройство для массива. Например, чтобы создать любой из массивов, обсужденных выше, следует применить *mkraid /dev/md0*. Дисковые массивы, требующие сложного процесса инициализации (RAID 1 и 5), начнут ресинхронизацию. Это не оказывает никакого влияния на работу системы (можно применить *mke2fs* для создания новой файловой системы, работать с ней и т. д.). Информацию о состоянии любого из уст-

ройств RAID можно извлечь из файла */proc/mdstat*. Как только массив стал работать надлежащим образом, можно воспользоваться командами *raidstop* и *raidstart* для управления его работой.

При создании файловой системы следует учитывать два важных параметра, задаваемых для *mke2fs*. Первый параметр – это размер блока (block size) в байтах, указываемый с помощью *-b*. Его значение должно быть не меньше 4 096 (4 Кбайт). Второй параметр, относящийся только к устройствам RAID 5, – это параметр *-R stride=N*. Применение таких параметров позволяет лучшим образом размещать структуры данных на устройстве. Например, если размер участка равен 64 Кбайт, то это означает, что 64 Кбайт смежных данных будут располагаться на одном диске. Если при построении файловой системы задается размер блока 4 Кбайт, то в одном участке массива будут находиться 16 блоков файловой системы. Вот как может выглядеть запуск утилиты *mke2fs*:

```
# mke2fs -b 4096 -R stride=16 /dev/md0
```

Производительность RAID 5 во многом зависит от этих ключей.

Автоматическое обнаружение

Автообнаружение позволяет ядру автоматически обнаруживать и запускать устройства RAID во время загрузки системы. Для того чтобы автоматическое обнаружение работало, необходимо, чтобы система отвечала трем критериям:

1. Поддержка автоматического обнаружения должна быть учтена при компиляции ядра.
2. Постоянные суперблоки в устройствах RAID должны быть включены.
3. Типы разделов RAID должны быть установлены в 0xfd.

Если все эти критерии выполняются, то можно просто перезагрузить систему – автоматическое обнаружение должно работать нормально (для того, чтобы в этом убедиться, можно запустить *cat /proc/mdstat*). Устройства, запущенные таким образом, автоматически остановятся при выключении системы. Теперь к устройствам */dev/md* можно обращаться так же, как и к любым другим устройствам.

Загрузка с устройства-массива

Если необходимо загружаться с устройства-массива (array device), то есть кое-что, о чем следует позаботиться. Первое условие – ядро, используемое при загрузке, должно иметь доступ к модулю RAID. Для этого нужно либо скомпилировать новое ядро, в котором модули RAID скомпонованы напрямую (а не собраны как загружаемые модули), либо указать LILO использовать RAM-диск, содержащий все модули ядра, необходимые для монтирования раздела root. Это можно сделать с помощью команды *mkinitrd -with=module ramdis-name kernel*. Напри-

мер, если раздел root расположен на устройстве RAID 5, то необходимо запустить *mkinitrd -with=raid5 raid5-ramdisk 2.4.5*.

Существует другая сложность, а именно: LILO 0.21 (текущая версия) не понимает, что делать с устройством RAID. В результате ядро не может быть прочитано с устройства RAID, а файловая система */boot* должна располагаться на устройстве, отличном от RAID. В Redhat 6.1 включена «заплата» для LILO, позволяющая проводить загрузку ядра, расположенного *только* на массиве RAID 1. Эту «заплату» можно получить из *dist/redhat-6.1/ SRPMS/SRPMS/lilo-0.21-10.src.rpm* на любом сервере RedHat. В версии LILO с наложенной «заплатой» в файле *lilo.conf* можно задать *boot=/dev/md0*, причем загрузка возможна с любого диска в зеркале.

Теперь необходимо установить файловую систему root на устройство RAID. Для этого необходимо наличие резервного диска, на который можно инсталлировать систему, причем этот диск не должен быть частью конфигурируемого RAID. После инсталляции системы, когда поддержка RAID обеспечена, а само устройство RAID, созданное с помощью стандартных процедур, устойчиво работает после перезагрузки, можно запустить *mke2fs* для создания файловой системы на новом устройстве-массиве, а затем смонтировать ее (скажем, в */newroot*). Далее нужно скопировать содержимое текущей файловой системы root (на резервном диске) в новую файловую систему root (в массиве). Лучше всего это сделать с помощью *find* и *cpio*:

```
# cd /
# find . -xdev | cpio -pm /newroot
```

Теперь необходимо отредактировать файл */newroot/etc/fstab* для поддержки правильного устройства загрузки.

Последний шаг – обновление LILO. Сначала нужно размонтировать текущую файловую систему */boot* и смонтировать устройство загрузки на */newroot/boot*. Далее следует изменить */newroot/etc/lilo.conf*, указав правильные устройства. Заметим, что устройство загрузки попрежнему должно быть обычным устройством (не RAID), однако устройство root должно указывать на новый RAID. После выполнения этих действий необходимо запустить *lilo -r /newroot*, а затем перезапустить систему, которая загрузится с нового устройства RAID.

Рецепты RAID

Здесь представлены четыре «рецепта дисковых массивов», адресующих наиболее типичные варианты их компоновки. Такие рецепты не являются «серебряными пулями». Скорее они могут послужить отправной точкой в работе. Далее будет разобран конкретный случай для иллюстрации обсужденных принципов.

Домашние каталоги, отличающиеся интенсивным обращением к их атрибутам

Под эту категорию подпадает большинство операций, в которые вовлечены домашние каталоги.

Такое обращение к данным массива почти всецело представляет собой произвольный доступ. Когда файлы малы, то характеристики доступа к данным зависят от извлечения записей каталогов, данных индексных дескрипторов и т. п. Такие операции требуют существенного объема поиска. Как правило, в обращении к данным преобладает считывание, а операции записи относительно редки.

Поскольку рабочая нагрузка носит произвольный характер, то шины, вероятно, используются неинтенсивно. Если необходимо максимально возможное количество дисковых накопителей, то наилучшая стратегия сводится к размещению большого количества маленьких дисков на среднем количестве контроллеров. Можно попробовать скомпоновать 10 активных дисков на одну цепочку SCSI 40 Мбайт/с.

Если работа приложений не приводит к высокой загрузке каналов (особенно из-за операций записи), то RAID 5 является замечательным выбором. Если необходимы более быстрые операции записи, а средства это позволяют, то массив RAID 0+1 более предпочтителен.

Файловые системы нуждаются в настройке. Вообще, необходимо предусмотреть 1% свободного пространства, а размер индексного дескриптора файловой системы должен составлять 16 Кбайт. Кроме того, следует задать размер кластера (см. раздел «Размер кластера файловой системы» в главе 5).

Если домашние каталоги разделяются через NFS, то производительность операций записи становится более важной. Поэтому отличным решением будет применение NVRAM и других энергонезависимых устройств хранения. Приблизительный подход таков: конфигурировать три накопителя на каждое активное соединение NFS. Ведение протокола также является отличной идеей. По возможности следует использовать протоколирование, предлагаемое *metatrans* Solstice DiskSuite или Veritas. Такое протоколирование основано на интегрированных механизмах протоколирования файловой системы.

Домашние каталоги, вовлеченные в обработку большого объема данных

Домашние каталоги, вовлеченные в обработку большого объема данных, представляют иную сложность. Для них характерна последовательная рабочая нагрузка, в которой операции чтения уже не доминируют.

По мнению авторов, в этом случае будет плодотворен такой подход. В блоках (stripes) следует разместить максимальное количество самых быстрых дисков (то есть дисков с «наивысшей внутренней скоростью передачи»), причем количество контроллеров должно быть достаточно велико. На один контроллер не следует компоновать более четырех дисков.

Серьезным затруднением является то, что множество клиентов, обращающихся к одному диску, могут создать узкое место в системе. Поэтому нужно конфигурировать маленькие дисковые массивы (скажем, один на два нагруженных дисковых контроллера; лучше использовать RAID 0+1, но RAID 5 тоже приемлем, если есть большой объем NVRAM для кэширования записи), а затем объединять эти устройства. Для каждых трех-пяти активных клиентов следует компоновать одно устройство RAID.

К протоколированию следует подойти с особым вниманием. Если для протоколирования планируется применять специальные диски, то следует создать маленький том RAID 0+1 и использовать его как устройство протоколирования. Напомним: если доступ к рабочему пространству осуществляется через NFS, то для каждого клиента, соединенного через Fast Ethernet, может понадобиться полоса пропускания до 10 Мбайт/с.

Высокопроизводительные вычисления

Для сред с высокопроизводительными вычислениями обычно характерно примерно равное распределение операций чтения и записи[1] и последовательный тип доступа. Кроме того, производительность ставится превыше всего, поскольку данные часто регенерируются.

Ключевая идея состоит в перенесении нагрузки ввода-вывода, требующей высокой производительности, из домашних каталогов в отдельное «рабочее пространство» («workspace»). Вообще говоря, такую дисковую подсистему следует скомпоновать с большим количеством быстрых дисковых контроллеров (40 Мбайт/с), каждый из которых обслуживает два-три самых быстрых диска. Самое главное – не перенасытить контроллеры дисками, потому что последовательная рабочая нагрузка может легко перегрузить шину.

Следует остерегаться ограничений файловой системы, вызванных перекрыванием записи в UFS (см. раздел «Перекрывание записи в файловой системе UFS» в главе 5) или механизмом виртуальной подсистемы памяти. Необходимо обновить систему до версии Solaris 8 и вклю-

[1] Вообще, все зависит от приложения, однако для большинства научных приложений, с которыми работали авторы, расклад таков: либо преобладают считывания, либо операции записи и чтения составляют примерно равные доли.

чить приоритетный пейджинг, если он не включен. Бульшая часть такой настройки невозможна в Linux, однако нужно стремиться работать со свежей версией ядра для того, чтобы извлечь пользу из последних достижений в разработке подсистемы виртуальной памяти.

Протоколирование может стать существенной преградой для производительности. Поскольку данные можно восстановить быстро, а перезагрузка влечет за собой перезапуск вычислений, то типичным решением является создание максимально возможного количества томов RAID 0 с шестью дисками на каждый массив, а затем их объединение.

Базы данных

При конфигурировании дисковых массивов для баз данных необходимо учитывать множество условий. Одни операции чтения, например индексные поиски, могут выполняться маленькими последовательными блоками. Другие считывания, такие как сканирование всей таблицы, могут проводиться большими последовательными участками. Операции записи обычно небольшие и синхронные. Например, в Oracle размер блока по умолчанию равен 2 Кбайт. За счет этого дисковое время обслуживания небольших произвольных операций мало, однако при считывании больших таблиц может быть одновременно затребовано много блоков.

Конфигурирование в контроллере RAID большого объема энергонезависимой памяти – это отличная идея в силу двух причин. Первая – операции записи обычно синхронны и находятся на «критическом пути» задач пользователя. Вторая – большие операции записи обычно представляют собой длинный поток маленьких записей, которые поддаются объединению.

Конфликты виртуальной памяти могут создавать серьезные трудности. Хорошим решением для файловых систем UFS будет применение параметра `directio` (см. раздел «Выключение кэширования файлов в памяти» в главе 5), поскольку программное обеспечение баз данных предусматривает интенсивное внутреннее кэширование. Следует идти в ногу с выпусками новых версий базового программного обеспечения Solaris и Linux. Включение приоритетного пейджинга или переход на Solaris 8 будут очень полезны.

Несмотря на то что целостность данных является первостепенным условием, некоторые части базы данных могут регенерироваться «на лету». По этой причине есть смысл разделить базу данных на две части:

- Временные табличные области и индексы можно расположить на больших, незащищенных, быстрых дисковых участках, реализованных в виде объединенных массивов RAID 0. На контроллере с пропускной способностью 40 Мбайт/с следует располагать от шести до восьми быстрых дисков.

- Области, в которых надежность первостепенна, обычно предназначены для преимущественного считывания. Такие области можно реализовать с помощью томов RAID 5 или, если производительность важнее стоимости, на основе RAID 0+1.

Конкретный случай: приложения, выполняющие большой ввод-вывод

Рассмотрим сценарий обработки транзакций. Рабочая нагрузка приложения состоит из нескольких процессов, берущих информацию из сети. В обработку этих данных вовлечена память, а после обработки данные сбрасываются на диск блоками по 16 Кбайт. Пользователи жалуются на низкую пропускную способность дисковой подсистемы.

В этом случае можно ожидать, что *iostat -xtc 1* покажет непрерывную дисковую активность. Известно, что приложения выполняют большие дисковые операции (в данном случае 16 Кбайт). Для того чтобы дисковая пропускная способность была максимальна, запись данных на каждый диск должна осуществляться как можно более крупными порциями. Согласно эмпирическим наблюдениям, диск может поддерживать около 150 физических операций ввода-вывода в секунду и интенсивность обмена данными около 15 Мбайт/с. Отсюда следует, что нужно стремиться к тому, чтобы объем данных каждой операции с дисками составлял 100 Кбайт. Вот что необходимо предпринять:

1. Задать размер чередования при проектировании тома (напомним, что размер чередования – это порция смежных данных, которые размещаются на одном диске; следующая порция смежных данных размещается на другом диске и т. д). Размер чередования следует выбрать из диапазона 64–128 Кбайт.
2. При построении файловой системы большие участки файла следует размещать на блоках со смежными номерами. Размещением смежных данных управляет параметр `maxcontig` (устанавливается с помощью ключа *-C* команды *newfs* при создании новой системы или с помощью *tunefs -a* для существующей файловой системы). Хорошим значением для `maxcontig` будет 128 (в блоках по 8 Кбайт). За счет этого файловая система будет размещать файлы блоками, содержащими по 1 Мбайт смежных данных.
3. Наконец, когда задачи работают с файловой системой, ядру необходимо сбрасывать данные на диск. Для этого ядро находит измененные страницы, которые расположены рядом в файловой системе, и сбрасывает их на диск в ходе большой, объединенной операции ввода-вывода. Максимальный объем ввода-вывода задается параметром `maxphys`, поэтому его следует установить равным 1 048 576.

Вероятно, следует обратить внимание на перекрывание записи в UFS, если оно является преградой для производительности.

Заключение

Технологии дисковых массивов помогают преодолеть одну из самых существенных преград на пути к производительности и надежности: вращение пластин, покрытых магнитной пленкой. Эта удивительно простая идея позволяет если не убить, то оглушить двух птиц одним камнем. К сожалению, методы, призванные обеспечить оптимальную производительность дисковых массивов в различных средах, окружены вымыслами и заблуждениями. Зачастую бывает трудно определить, что происходит в рассматриваемой среде.

Основной итог этой главы таков: разработка дискового массива – это процесс проб и ошибок. Необходимо определить рабочую нагрузку, которую можно воспроизводить, и замерить ее. Далее следует применить материал этой главы для выбора правильного пути, а затем экспериментировать: изменять параметры по одному, повторять измерения и находить наилучшее решение. Это не та область, где один параметр решает все.

7

Сети

- *Основы сетей*
- *Физические носители*
- *Сетевые интерфейсы*
- *Сетевые протоколы*
- *NFS*
- *CIFS и UNIX*
- *Заключение*

Раз-два, раз-два, вперед и вперед
От IMP к IMP шел ACK и NACK,
А когда пришел RFNM, он сказал:
«Мой черед!» и обратно отправил ответ.

Так вступил ли ты в ARPANET?
Иди ко мне, мой бедный друг!
Звони скорее NIC! Шли RFCs!
А он лишь засмеялся вдруг.

Цитата из «APRAWOCKY», D. L.Covill,
май 1973 (RFC 527)

Зачастую дистанция между *системным администратором* и *сетевым администратором* довольно велика. Обычно системный администратор отвечает за несколько сотен машин, возможно, расположенных в нескольких зданиях. Однако сетевой администратор запросто может координировать сетевую инфраструктуру, покрывающую города и государства – от высокопроизводительных магистралей до разъемов на чьем-то рабочем столе. Чрезвычайно редко можно встретить кого-либо, одинаково хорошо справляющегося с двумя такими ролями.

Главным образом, эта глава посвящена системным администраторам, имеющим небольшой опыт работы с сетями, которые поймали себя на мысли, что сложности с производительностью исходят от сети. Такое положение может быть затруднительным, особенно если поблизости нет сетевого инженера! Когда незнакомые акронимы неожиданно возникают из эфира (ether), то бывает трудно определить, что же происходит. Основа настройки производительности лежит в понимании того, как работает система. Эта глава дает обзор принципов работы сети, от физических носителей до протоколов TCP и UDP. Далее будут об-

суждены два наиболее типичных механизма совместного использования файлов, NFS и CIFS. Однако прежде всего, важное замечание о терминологии.

В мире сетей скорость передачи данных измеряется в мега*битах* в секунду (Мбит/с), а не в мега*байтах* в секунду (Мбайт/с), как обычно принято в обсуждении работы компьютера. Для того чтобы выполнить правильное сравнение, нужно количество Мбит/с разделить на восемь. Например, если скорость передачи данных по сети Ethernet равна 10 Мбит/с, то по ней передается самое большее 1,25 Мбайт/с.

Основы сетей

Сеть – это структура, разработанная для совместного использования данных устройствами, которые подсоединены к разделяемому носителю. Такой разделяемый носитель называется *каналом передачи данных* (*data link*). Кроме того, в сети существуют определенные правила, описывающие, как именно он разделяется, – подобно тому как у постояльцев в квартире есть правила, кто что берет из холодильника. Такой набор правил называется *управлением доступа к среде* (*media access control*, MAC). Кроме того, устройства в сети должны иметь уникальные идентификаторы. Благодаря таким идентификаторам устройства не получают данные, которые для них не предназначены. Сами данные инкапсулируются в «двоичный конверт», называемый *кадром* (*frame*), в котором заданы адреса получателя и отправителя (эквивалент обратного адреса). Формат этих адресов зависит от используемого протокола MAC. Ethernet, самый распространенный канал передачи данных, разделяет типичный формат с Token Ring и FDDI. Адрес Ethernet представляет собой 48-битный код, разработанный таким образом, чтобы каждое сетевое устройство на планете имело уникальный адрес. Такой код, называемый *MAC-адресом* (*MAC address*), *аппаратным адресом* (*hardware address*), *адресом канала передачи данных* (*data link address*) или *физическим адресом* (*physical address*), служит для уникальной идентификации устройств.[1]

По мере расширения сетей возникает потребность в дополнительных устройствах. Каждое из них решает задачи, связанные с различными издержками роста сети. К числу наиболее важных устройств относятся *повторители* (*repeaters*), борющиеся с затуханием сигнала, а также *мосты* (*bridges*) и *маршрутизаторы* (*routers*), необходимые для расширения сети. *Коммутаторы* (*switches*) призваны решать обе задачи.

[1] Хотя в некоторых сетевых устройствах возможно изменение этого значения, суть в том, что все идентификаторы в сети являются уникальными.

При передаче данных по любому физическому носителю сигнал затухает. Затухание усиливается с увеличением расстояния. *Повторитель* служит для преобразования «зашумленного» сигнала в новый сигнал, такой же, но чистый. Название этого устройства происходит от «повторения» сетевого трафика. Кроме того, здесь будет использоваться термин *хаб* (*hub*). Это повторитель, который имеет много портов. Любой трафик, посылаемый из одного порта, повторяется на всех других портах (заметим, что «хаб» может означать и многое другое: это может быть сетевой центр, сетевой шкаф, то есть место, где сходятся много сетей, и т. д.).

Когда слишком много устройств пытаются общаться одновременно, а трафик в сети становится более интенсивным, то возможна перегруженность каналов. Одно из решений – разбить одну большую сеть на несколько частей, связанных между собой с помощью устройства, называемого *мостом*. По существу, мост работает так: прослушивает весь доступный ему трафик, а затем строит *таблицу соединений* (*bridging table*), которая говорит, какие хосты «живы» и в каких сетях. В таблицу включаются хосты, данные от которых мост видит на своих входах. Эту таблицу мост использует для пропускания трафика через свои входы таким образом, чтобы доставить данные до их получателя (поэтому он не пропустит трафик сквозь себя, когда пункт назначения находится на том же входе, что и приемник). Такой мост называется *сквозным* (*transparent*). Однако существует много других видов мостов, полное обсуждение которых выходит за рамки этой книги.

К сожалению, мосты не обеспечивают бесконечной расширяемости. Некоторые кадры нельзя локализовать в отдельной сети. Такие *широковещательные* (*broadcast*) кадры должны быть доставлены всем хостам. По мере увеличения количества хостов в сети, построенной исключительно на мостах, широковещательный трафик в конце концов вызовет перегрузку сети.

Когда речь заходит о расширяемости сети, то в какой-то момент становится необходимо организовать совокупности сетей внутри большой сети, более известной как объединенная сеть (internetwork). Разграничение таких сетей находится в компетенции *маршрутизатора*, названия которого могут быть различны. В дни, прешествующие зарождению Интернета, маршрутизаторы ARPANET назывались *межсетевыми процессорами сообщений* (*internet message processors*, IMP). Сейчас их называют *шлюзами* (*gateways*)[1] и *промежуточными системами* (*intermediate systems*). Маршрутизатор служит для передачи информации по специальному пути между двумя сетями, причем современ-

[1] Это другой пример небрежного использования слов. Технически шлюз – это система, преобразовывающая протоколы (например, LocalTalk в IP). Маршрутизатор этого не делает. Шлюз может быть маршрутизатором, но маршрутизатор не может быть шлюзом.

ные маршрутизаторы зачастую выбирают наилучший путь из нескольких возможных. Механизм выбора наилучшего маршрута, а также способ разделения информации с другими маршрутизаторами в объединенной сети образуют *протокол маршрутизации* (*routing protocol*). Поскольку маршрутизаторы предназначены только для доставки данных до конкретной сети назначения, то они обычно не обращают внимания на другие устройства, не являющиеся маршрутизаторами. Когда маршрутизатор видит, что входящие данные предназначены для его сети, то он действует как хост и использует идентификатор хоста назначения для доставки данных. Маршрутизатор может перенаправлять пакеты даже в том случае, когда целевой объект не расположен в сети, напрямую связанной с маршрутизатором. Такие пакеты передаются другому маршрутизатору, который находится ближе к хосту назначения.

В большинстве сетей основная часть трафика – это передача данных от одного хоста к другому, то есть *однонаправленный* (*unicast*), а не широковещательный трафик. Другим хостам нет необходимости видеть эти данные, поскольку кадр адресован не им. Кроме того, в среде рабочей группы на повторители возлагаются обязанности, связанные с безопасностью. Рассмотрим случай с тремя системами, подключенными к хабу. Одна машина, `alice`, открывает сессию *telnet* на другой машине, `bob`. В сети с повторителями канал передачи данных, ведущий к третьей машине (`spy`[1]), содержит весь трафик, который передается между `alice` и `bob`. Большинство сетевых интерфейсов могут работать в *«беспорядочном» режиме* (*promiscuous mode*), в котором они прослушивают все передаваемые пакеты, а не только пакеты, адресованные исключительно им. Несмотря на то что такой режим незаменим, когда необходимо отлаживать работу сети, это также означает, что `spy` может читать все данные, передаваемые между `alice` и `bob`. *Коммутатор*, подобно повторителю, регенерирует сигналы, но кроме этого он фактически создает «виртуальные сети» между хостами. Поэтому любой трафик, исходящий из одного порта, доставляется только в порт назначения. Тогда исполнить описанный опасный трюк будет труднее, поскольку `spy` больше не сможет напрямую видеть трафик между `alice` и `bob`.[2] Кроме того, такой подход способен значительно улучшить производительность, поскольку каждый хост фактически получает выделенный носитель 10 Мбит/с, а не разделяет один носитель 10 Мбайт/с со всеми другими хостами.

---

[1] Spy (англ.) – шпион. – *Примеч. перев.*

[2] Однако для целей диагностики большинство коммутаторов могут быть сконфигурированы таким образом, чтобы весь трафик со всех интерфейсов повторялся в одном порту. По этой причине, если на коммутаторе безопасность снижена, то коммутируемая сеть по-прежнему восприимчива к атакам такого рода. Также существует много других атак, не связанных с компромиссами в работе коммутатора.

Существуют два типа коммутаторов – коммутаторы *с промежуточным хранением* (*store-and-forward*) и *коммутаторы без буферизации* (*cut-through*). Коммутатор с промежуточным хранением буферизует в памяти входящий кадр, а затем перенаправляет его в соответствующий пункт назначения, тогда как коммутатор без буферизации использует метод «горячей картошки» (скорейшей передачи). В прошлом, вследствие высокой стоимости буферной памяти, предпочтение отдавалось коммутаторам без буферизации. Одно из преимуществ таких коммутаторов состоит в том, что они вызывают очень маленькую дополнительную задержку при передаче пакетов, тогда как задержка на коммутаторах с промежуточным хранением может изменяться. Коммутаторы с промежуточным хранением изящно обрабатывают коллизии (см. раздел «Коллизии» далее в этой главе), а коммутаторам без буферизации коллизии доставляют серьезные хлопоты. В большинстве современных коммутаторов применяется промежуточное хранение, а методы коммутации без буферизации нашли свою нишу в специализированных сетях.

Один из классических трудов, посвященных объединению сетей, – это книга «Соединения: мосты и маршрутизаторы» («Interconnections: Bridges and Routers») Радиа Перлмана (Radia Perlman), изданная Addison-Wesley.

Модель OSI

Подобно наличию уровней представления, описанных при рассмотрении архитектуры компьютера (см. раздел «Уровни представления» в главе 1), для сетевой архитектуры существует аналогичное формальное описание. Оно представлено в *модели OSI*, которая состоит из семи уровней:

Физический уровень

Описывает физические характеристики носителя. Разбит на четыре категории: *электротехнические/оптические* протоколы, описывающие особенности передачи сигнала по проводу; *механические* протоколы, описывающие типы используемых разъемов или диаметр провода; *функциональные* протоколы, описывающие значения конкретных сигналов (например, то, что закодировано на каждом выводе), и *процедурные* протоколы, объясняющие принципы работы физического носителя (например, какой вольтаж соответствует двоичному разряду).

Канальный уровень

Описывает протоколы, управляющие физическим уровнем. Например, они определяют механизмы идентификации устройств, а также образование кадров из данных. Примером канального протокола может служить Ethernet.

Сетевой уровень

Описывает выбор маршрута передачи данных в логических каналах, а также формирование сетевых адресов.

Транспортный уровень

Описывает протоколы, управляющие сетевым уровнем, – подобно тому как канальные протоколы управляют физическим уровнем. Протоколы транспортного уровня управляют трафиком в логическом канале, который представляет собой сквозное соединение устройств и осуществляет согласование каналов передачи данных.

Сеансовый уровень, уровень представления и уровень приложений

Описывают предоставление услуг на уровне пользовательских приложений.

В этой главе сеть будет рассмотрена с самых основ. То есть вначале будет обсужден самый низкий уровень модели OSI, физическая сеть, а затем будем подниматься выше до программного обеспечения сетевой инфраструктуры.

Физические носители

По существу, в современных сетях встречаются два типа физических носителей: неэкранированная витая пара (UTP) и волоконно-оптические кабели. В целом, для оптоволокна характерна более высокая скорость передачи данных на большие расстояния. Передача данных по оптоволокну намного безопаснее, однако его стоимость существенно выше.[1] Существуют три трудности, связанные с передачей данных по физическим каналам: поглощение, искажение и интерференция.

Поглощение происходит из-за того, что на преодоление сопротивления провода при «проталкивании» сигнала требуется энергия. В результате происходит постепенное затухание сигнала, поскольку по мере продвижения по проводу сигнал теряет энергию. *Искажение* связано со способом перемещения сигнала в носителе, так как задержка при передаче различных частот сигнала неодинакова. Наконец, *интерференция* происходит вследствие внешних факторов, приносящих шумы в сигнал.

UTP

Неэкранированная витая пара делится на несколько категорий с различными характеристиками кабеля. Кабели категорий с высокими

[1] Хотя само по себе оптоволокно относительно дешево, его цена определяется высокой стоимостью разъемов и работ, выполняемых с оптоволокном. Например, соединить два медных провода очень легко. В значительной степени труднее соединить две стеклянные нити. Вот почему оптоволокно еще и более безопасно.

номерами способны противодействовать шумам на высоких частотах, а значит, более приспособлены к противодействию внешней интерференции. Классификация категорий UTP представлена в табл. 7.1.

Таблица 7.1. Категории UTP

Категория	Максимальная скорость	Применение
1	56 Кбит/с	Передача речевых сигналов
2	4 Мбит/с	Цифровая сеть связи с комплексными услугами (Integrated Services Digital Network, ISDN)
3	16 Мбит/с	Носитель данных для локальных сетей: Ethernet 10 Мбит/с, Token Ring
4	20 Мбит/с	Локальные сети, раскинутые на большие расстояния
5	Свыше 100 Мбит/с	Носители данных для Ethernet 100 Мбит/с, ATM и FDDI по UTP, а также для Ethernet 1 Гбит/с на небольших расстояниях

Кроме того, появился новый стандарт кабеля, называемый категорией 5E, или категорией 6. Внутри этого кабеля есть жесткая основа, позволяющая осуществлять скрутку сигнальных проводов. Такой стандарт особенно полезен в средах с высокой полосой пропускания (таких как Gigabit Ethernet на медном носителе).

Для кабеля UTP обычно применяют два типа разъемов: RJ-11 и RJ-45. RJ-11 – это стандартный четырехконтактный разъем, используемый в телефонных розетках и почти не применяющийся в современных сетевых схемах. RJ-45 фактически определяет назначение контактов, однако этот термин обычно используется для обозначения стандартного восьмиконтактного разъема, повсеместно применяемого в сетях.

Замечание по терминологии: штепсель и розетка

Штепсель (*plug*) – это вилка разъема, располагаемая на конце кабеля; *розетка* (*jack*) – это гнездо соединителя в коммутационной панели или в плате в стене. Если смотреть на розетку в стене (зажим «смотрит вниз»), то контакт 1 – крайний слева. Если смотреть «сверху вниз» на кабель (зажим на штепселе «смотрит вниз»), то контакт 1 – крайний справа.

Оптоволокно

Оптоволоконный кабель состоит из трех частей. Стеклянная или пластиковая *основа* (*core*), по которой распространяется свет, окружена *оболочкой* (*cladding*), которая так же выполнена из стекла или пластика, однако обладает существенно иными оптическими свойствами.

Основа и оболочка окружены *рубашкой* (*jacket*), защищающей кабель от воздействия окружающей среды. Свет от источника проникает в основу. При этом те лучи, которые поступают под небольшим углом (почти вровень с основой), отражаются оболочкой и передаются вдоль основы, а лучи, поступающие под большим углом, поглощаются наружными слоями. Луч, поступивший в оптоволокно, называется *модой* (*mode*), поскольку оптоволокно очень чувствительно к отклонениям радиуса кабеля. Если участок кабеля изогнуть слишком тугой петлей, то часть передаваемого света будет потеряна. Такое явление называется *поглощением*. И хотя поглощение не приводит к потере сигнала, амплитуда сигнала снижается. Рисунок 7.1 показывает оптоволоконный кабель в поперечном разрезе.

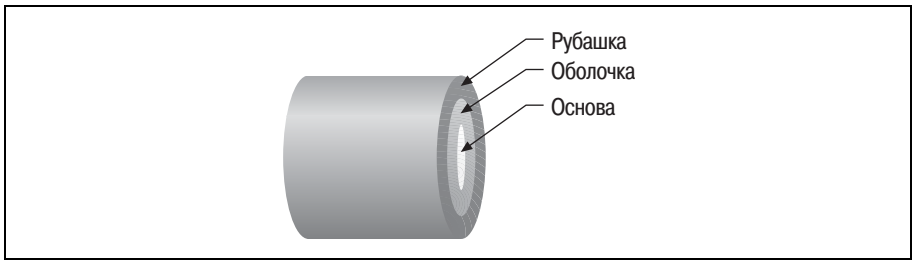


Рис. 7.1. *Оптоволоконный кабель в поперечном разрезе*

Поскольку существует много возможных мод, которые можно одновременно посылать по оптоволокну, такой тип распространения света называется *многомодовым* (частое сокращение – MMF, *multimode fiber*). Моды могут распространяться с различными скоростями. Размер основы многомодового оптоволокна обычно составляет 50 или 62,5 микрона. Лучи света проходят по оптоволокну различными путями, поэтому они прибывают на приемный конец волокна в разное время. «Тянучка» пропорциональна расстоянию. Наконец, сильное искажение импульсов влияет на целостность данных. Такой эффект допустим, поскольку многомодовое оптоволокно стоит недорого.

Существуют две разновидности многомодового оптоволокна: *со ступенчатым коэффициентом* (*step-index*) и *с плавно изменяющимся коэффициентом* (*graded-index*). В оптоволокне первого типа показатели преломления в основе и в оболочке различны, однако коэффициент преломления один и тот же по всей основе. В итоге, когда длина пути света увеличивается с расстоянием, происходит дисперсия. В оптоволокне с плавно изменяющимся коэффициентом показатель преломления варьируется в поперечном сечении. За счет этого различие в длинах путей сглаживается, а возможное расстояние передачи моды соответственно увеличивается.

Одномодовое (*single-mode*) оптоволокно, или SMF, имеет очень тонкую основу, обычно от двух до десяти микронов. Это означает, что вдоль осевых линий может распространяться только одна мода, поэтому

сложности с внутренним отражением, характерные для многомодового оптоволокна, здесь отсутствуют. Одномодовое оптоволокно может иметь очень высокую протяженность (километры). Рисунок 7.2 иллюстрирует оптоволокно со ступенчатым и плавно изменяющимся коэффициентом, а также одномодовое волокно.

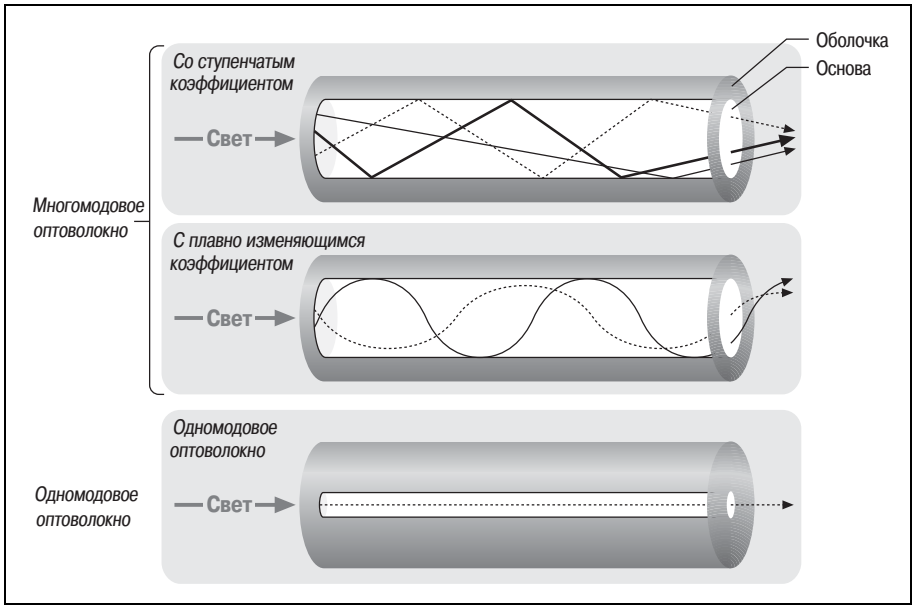


Рис. 7.2. *Многомодовое волокно с пошаговым и ступенчатым коэффициентом и одномодовое оптоволокно*

При монтаже оптоволокна применяется много различных разъемов. Наиболее часто используются три из них. FSD, или *Fixed Shroud Device*, обычно применяют в FDDI. ST – это типичный штыковой разъем, использующий один соединитель на нить. SC является международным стандартом. Это разъем с зажимом. Его преимущество над ST состоит в том, что это единый разъем, вмещающий как передающее, так и получающее оптоволокно. Обычный способ запоминания вариантов разъемов сводится к запоминанию «стыковка с кручением или стыковка щелчком». Недавно были предприняты усилия создать новый стандарт разъемов. Наиболее вероятным претендентом на право быть признанным в качестве стандарта является разъем LC, разработанный в Lucent Technologies.

Сетевые интерфейсы

В этом разделе обсуждаются физические уровни, а именно то, как сетевые устройства взаимодействуют друг с другом. Главным образом, речь пойдет о протоколе Ethernet.

Ethernet

Зарождение Ethernet восходит к 1973 году, когда этот интерфейс был изобретен Бобом Меткалфом[1] (Bob Metcalfe) и Дэвидом Богсом (David Boggs) и впоследствии доработан в Xerox PARC (Palo Alto Research Center). В первой реализации скорость передачи данных составляла 3 Мбит/с. С этим интерфейсом было создано около 5 000 машин. Основная их часть осталась в Xerox, однако некоторые из них приобрели другие корпорации (особенно Boeing) и учебные заведения. Первоначальный стандарт Ethernet был разработан консорциумом Xerox, Digital Equipment Corporation и Intel, сформированном в 1979 году. Результаты были направлены в новую рабочую группу IEEE, впоследствии названную IEEE Project 802. Эта рабочая группа разделилась на три группы:

- Группа интерфейса высокого уровня (High Level Interface, HILI), сосредоточившаяся на протоколах межсетевого обмена и управления сетью (позднее группа 802.1).
- Группа управления логической связью (Logical Link Control, LLC), акцентировавшая внимание на возможностях сквозных соединений и уровнях, независимых от носителя (позднее группа 802.2).
- Группа управления каналом передачи данных и контроля доступа в сети (Data Link and Medium Access Control, DLMAC), отвечающая за сам протокол доступа.

В 1982 году группа DLMAC, в свою очередь, разбилась на три комитета: 802.3 для Ethernet, ведомый Xerox, DEC и Intel; 802.4 для Token Bus, поддерживаемый Burroughs, Honeywell и Western Digital, и 802.5 для Token Ring, территории IBM. В итоге комитет стандартов Ethernet 802.3 разработал серию спецификаций для Ethernet 10 Мбит/с, поддерживающих различные виды физических носителей. Первоначально стандарт Ethernet был реализован на толстом коаксиальном кабеле, однако позднее стали применять тонкий коаксиальный кабель, неэкранированную витую пару и оптоволокно.

При разработке Ethernet производительности уделялось мало внимания. На то было две причины. Первая заключалась в том, что сети, как правило, были маленькие, а трафик не достигал пределов пропускной способности, поэтому мониторинг производительности не требовался. Вторая причина была обусловлена трудностью организации такого мониторинга, поскольку внедрение средств измерения и управления привнесло бы в интерфейс дополнительную сложность и увеличило его стоимость.

Когда стало ясно, что скорость передачи 10 Мбайт/с слишком мала, были предприняты усилия увеличить ее на порядок. Однако в силу по-

[1] Впоследствии он основал 3Com Corporation и проповедовал Ethernet как стандарт для индустрии.

литических соображений две сформированные группы избрали различные подходы. Первая группа стремилась сохранить существующий протокол и проводила только незначительные изменения интерфейса, изначально сосредоточившись на кабеле UTP категории 5. Это усилило репутацию Ethernet, а также поддержало накопленные инженерные наработки. Эта группа, получившая большую маркетинговую и общественную поддержку, стала официально именоваться 802.3u Task Force. Другая группа приступила к разработке нового протокола. Целью являлось создание более эффективного метода передачи данных, а внимание было сфокусировано на повторном использовании существующего кабеля UTP категории 3. Эта группа получила название 802.12 Task Force. В ходе разработки стандартов обе группы воевали на техническом и маркетинговом фронтах. Наконец, 13 июня 1995 года Совет по стандартам IEEE утвердил реализации 802.3u и 802.12 в качестве официальных стандартов. Каким бы ни было преимущество стандарта 802.12, чаша весов склонилась к 802.3u. Практически все основные производители оборудования LAN приняли стандарт 802.3u.

Можно было предположить, что выход сетевых продуктов со скоростью передачи 100 Мбайт/с немедленно уничтожит рынок Ethernet 10 Мбит/с. Однако в силу нескольких причин этого не произошло. Более высокая пропускная способность требовалась далеко не всегда. Кроме того, имелось достаточно много устройств со скоростью передачи 10 Мбит/с, а первоначальная переплата за новые устройства была значительна.

Вскоре после принятия стандарта Fast Ethernet публика стала проявлять интерес к повышению производительности Ethernet еще на порядок. Такая потребность была обусловлена заторами в широко распространенных сетях 100BASE-T, в которых нагрузка, приходящаяся на магистрали, была очень велика. В то время уже существовали технологии с намного бульшими скоростями передачи данных, например ATM (622 Мбит/с; в настоящее время скорость передачи близка к 2,4 Гбит/с), однако для увязки таких технологий с широко распространенным оборудованием Ethernet требовалось изменение форматов кадра. Такая работа стоит недешево, если ее выполнять быстро. Поэтому усилия по достижению скорости передачи 1 Гбит/с были предприняты в самой модели Ethernet. Разработка стандарта была завершена в июне 1998 года, а в 2000 году началось внедрение этой технологии.

Основы передачи сигналов в Ethernet

В Ethernet 10 Мбит/с передача сигналов по проводам основана на методе, называемом *манчестерским кодированием* (*Manchester encoding*), который позволяет передавать сигнал синхронизации и данные в одном логическом пакете. Такая передача данных напоминает дифференциальную передачу сигналов (см. раздел «Дифференциальная передача сигналов» в главе 5), но вместо одного провода задействованы два. Этот пакет, формально называемый *битом-символом* (*bit-symbol*),

образован логическим инверсным значением кодированного бита, за которым следует его реальное значение. Таким образом, в середине бита-символа всегда есть переход сигнала. Например, согласно манчестерскому кодированию, бит «0» будет закодирован как бит-символ «01». Такой подход кажется нелогичным, поскольку это выглядит как двойная работа, необходимая для посылки бита данных. Однако, как и дифференциальная передача сигналов, такой способ полезен при передаче данных на большие расстояния. Самый значительный недостаток этого способа заключается в том, что изменение сигнала происходит в два раза быстрее, чем скорость передачи данных, поэтому такая схема более восприимчива к интерференции. В итоге в Ethernet 100 Мбит/с манчестерское кодирование не применяется.

Инкапсуляция данных в Ethernet нетривиальна. Пакет данных размером от 46 до 1500 байт[1] имеет заголовок размером 23 байт, а также четырехбайтный замыкающий блок. Содержимое заголовка и концевика представлено в табл. 7.2.

Таблица 7.2. Заголовок Ethernet

Биты	Содержимое
0–55	Преамбула
56–63	Разделитель начала кадра
64–111	Адрес данных
112–159	Адрес источника
160–175	Тип
	Данные
Последние 32 бит	FCS

Преамбула представляет собой последовательность перемежающихся единиц и нулей. Это единственная частота (последовательность передаваемых битов), на которую может ориентироваться хост-получатель для того, чтобы «захватить» входящий поток битов и не пропустить его. Очевидно, что преамбула играет важнейшую роль в обнаружении коллизий. Если два устройства одновременно передают данные, то какой бы ни был уровень волны сигнала, интерференция будет обнаружена, поскольку преамбула всегда представляет собой участок чередующихся нулей и единиц. За преамбулой следует признак начала кадра, называемый разделителем начала кадра (Start Frame Delimeter, SFD). Адрес данных – это 48-битный аппаратный адрес получателя (см. раздел «Основы сети» ранее в этой главе). Адрес источника – это 48-битный адрес отправителя. Поле типа описывает применяемый протокол.

[1] Размер MTU Ethernet (1500 байт) включает только полезную часть кадра Ethernet. В объем полезной нагрузки, равный 1500 байт, заголовки Ethernet не входят.

Топологические схемы

Каждая из существующих топологических схем Ethernet обозначается по-разному. Первоначальный стандарт, использующий толстый коаксиальный кабель, получил название 10BASE5. Число 10 обозначает сетевую скорость передачи данных в Мбит/с, «BASE» связано с использованием метода узкополосной передачи сигналов (baseband), а число 5 описывает максимальную длину сегмента, задаваемую в 100-метровых участках. Коаксиальный кабель представляет собой линейную шину, к которой подсоединяются все узлы. Центральный провод задействован для передачи сигнала, а оболочка (оплетка) используется как «земля». Коаксиальный кабель, используемый в 10BASE5, достаточно толст (около сантиметра) и не очень гибок. Само соединение с узлом осуществляется с помощью кабеля интерфейса подключаемых устройств (attachment unit interface, AUI). Максимальная длина такого кабеля составляет 50 метров, а в соединении задействован обычный 15-контактный разъем D-типа. Кабель AUI присоединен к *устройству подключения к среде* (*medium attachment unit*, MAU), которое служит интерфейсом между узлом и коаксиальным кабелем. Такое соединение можно создать двумя способами. Первая схема (с выступающим ответвителем) подразумевает фиксацию MAU на толстом коаксиальном кабеле и вставку в него ножа.[1] Вторая схема (с врезанным ответвителем) предполагает разрезание коаксиального кабеля и вставку MAU в середину. Между каждой парой MAU должно быть не менее 2,5 метров, а само расстояние должно быть кратно 2,5 метрам. В современных сетях работать с такой топологией ужасно трудно: незначительные изменения предполагают повторную трассировку кабеля, а пронзающие ответвители славятся трудностью установки и неполадками в работе. Врезанные ответвители приводят к простоям в сети. Несмотря на эти ограничения такую топологию еще можно встретить.

Стандарт 10BASE2, призванный смягчить недостатки 10BASE5, позволяет использовать тонкий (менее 5 мм) коаксиальный кабель, обычно называемый «тонкой сетью» («Thinnet»). Тонкий коаксиальный кабель более гибок и с ним легче работать. Подсоединение устройств стало намного удобнее – для этого требуется простой разъем BNC.[2] Однако применение недорогого носителя снизило максимальную длину кабеля до 185 метров. Вследствие ненадежности[3] в начале 1990-х годов стандарт 10BASE2 был вытеснен стандартом 10BASE-T.

---

[1] Так называемый «*зуб вампира*» (пронзающий ответвитель).

[2] Название происходит от Bayonet-Neill-Concelman.

[3] Ненадежность – это, например, случайные перегибы кабеля, ведущие к потере работоспособности сети до разгибания кабеля. Они могли произойти при неудачной прокладке кабеля, при неосторожной установке стула или шкафа на кабель и т. п. – *Примеч. науч. ред.*

10BASE-T

Стандарт 10BASE-T, более известный как «витая пара Ethernet», позволяет применять стандартные двухпарные кабели категории 3 (одна пара для передачи данных, а другая для приема – всего четыре активных провода). Поскольку в такой кабельной системе подразумеваются раздельные пути для передачи и получения данных, то необходим контроль физического канала, осуществляемый на каждой стороне соединения. Такая проверка называется *испытанием целостности связи* (*Link Integrity test*). Когда нет данных для отправки, каждая сторона соединения посылает импульс *испытания связи* (*link test*). Если импульс испытания связи не получен, то интерфейс Ethernet помечает линию как *поврежденную* (*link fail*) и отказывается передавать данные (в теории).

В отличие от 10BASE2 и 10BASE5, топология 10BASE-T не является последовательной. Она представляет собой топологию звезды, где каждый сетевой хост соединен с центральным хабом, действующим как повторитель. Длина каждого соединения может быть до 100 метров (то есть расстояние от любого хоста до хаба не должно превышать 100 м).

100BASE-T4

Протокол 100BASE-T4 – это плод усилий, направленных на достижение скорости передачи данных 100 Мбит/с по широко распространенным кабелям категории 3. 100BASE-T4 основан на топологии «звезда» и требует постоянного использования четырех пар проводов. Три пары предназначены для передачи данных, а четвертая пара – для прослушивания одновременной активности на другой стороне соединения (именно так обнаруживаются коллизии). Поскольку три из четырех пар проводов необходимы для передачи данных, то 100BASE-T4 является только полудуплексным носителем. Одна из сложностей такой схемы передачи сигналов связана с явлением, называемым *расфазировкой пары* (*pair skew*). Расфазировка пары означает, что каждый из сигналов, посланных по каждой из трех пар, распространяется по проводам за неодинаковое время. Небольшое расхождение по времени обусловлено неодинаковой длиной кабелей. Несмотря на то что в аппаратуре 100BASE-T4 предусмотрена защита от расфазировки, важно, чтобы все пары кабеля были в одной и той же рубашке. Хотя стандарту 100BASE-T4 оказывается хорошая поддержка, он не нашел широкого применения.

100BASE-TX

Топология 100BASE-TX предоставляет скорость передачи данных 100 Мбит/с на расстояния до 100 метров по неэкранированной витой паре категории 5. В целом, этот стандарт похож на другие стандарты витой пары. Он отличается только типом необходимого кабеля и тон-

костями передачи сигналов, которые не будут здесь обсуждаться. Стандарт 100BASE-TX быстро стал доминирующей технологией на рынке Ethernet 100 Мбит/с.

Топологические схемы Gigabit Ethernet

В настоящее время для Gigabit Ethernet существуют четыре топологических стандарта: 1000BASE-SX, коротковолновый свет лазера по многомодовому оптоволокну; 1000BASE-LX, длинноволновый свет лазера по многомодовому и одномодовому оптоволокну; 1000BASE-CX, медный провод для ближней связи, и 1000BASE-T, использующий кабель UTP категории 5. Как показано в табл. 7.3, стандарты 1000BASE-SX и 1000BASE-LX в основном различаются расстояниями, на которые передаются данные.

Таблица 7.3. Спецификация расстояний Gigabit Ethernet по оптоволокну

Тип	MMF (50 м)	MMF (62,5 м)	SMF (10 м)
1000BASE-SX	~500 м	~220 м	Нет
1000BASE-LX	~500 м	~500 м	~5 км

Стандарт 1000BASE-CX предназначен для недорогих соединений между устройствами, расположенными рядом друг с другом, например в монтажном шкафу. Длина соединения 1000BASE-CX может быть до 25 м. Стандарт 1000BASE-T схож с 100BASE-T, однако его реализация технически более сложна из-за интерференции, возникающей при использовании медных проводов на таких высоких скоростях передачи.

Правило 5–4–3

Правило 5-4-3 – это правило IEEE, описывающее, как сегменты можно соединять друг с другом с помощью повторителей. Оно определяет максимальный размер немаршрутизируемой/некоммутируемой сети. Согласно правилу 5-4-3, между любыми двумя хостами Ethernet допустимо не более пяти сегментов, соединенных повторителями, и не более четырех повторителей, причем из этих пяти сегментов нагруженными могут быть только три (сегмент считается нагруженным, если кроме мостов к нему присоединены другие устройства). Такие ограничения могут значительно усложнить монтаж сети. Поскольку коммутаторы регенерируют пакеты и заново образуют кадры, то это правило к ним не относится.

Коллизии

Коллизия (конфликт) происходит, когда два устройства в сети, определив, что сеть простаивает, пытаются одновременно посылать данные. Коллизию можно обнаружить только во время передачи преамбулы

(см. раздел «Основы передачи сигналов в Ethernet» ранее в этой главе). Так как в любой момент передавать данные может только одно устройство, оба устройства «отступают» и пытаются начать передачу заново. Согласно алгоритму повторной передачи, каждое устройство ждет случайный отрезок времени перед тем, как сделать еще одну попытку, а значит, маловероятно, что повторные передачи от двух устройств наложатся друг на друга. Если пакет «сталкивается» 16 раз подряд, то его передача отменяется. В сети с сегментами, соединенными повторителями, какое-то количество коллизий допустимо, однако чрезмерные коллизии могут серьезно снизить производительность. К сожалению, хорошего практического правила, говорящего: «Это количество коллизий слишком велико», не существует. Данные табл. 7.4 могут служить отправной точкой в оценке количества коллизий.

Таблица 7.4. Оценка коэффициента коллизий

Тип сети	Уровень предупреждения	Серьезный уровень
Некоммутируемая	5–10%	15%
Коммутируемая	2–4%	5%

Для того чтобы справиться со значительными коллизиями в некоммутируемых сетях, лучше всего задействовать коммутаторы.[1] Что касается коммутируемых сетей, то наиболее вероятной причиной высокого коэффициента коллизий являются дуплексные несоответствия между коммутатором и хостами. Другими словами, коммутатор предполагает, что связь дуплексная, а хост работает в полудуплексном режиме, или наоборот.

Поздние коллизии – это коллизии, которые не обнаруживаются к тому времени, когда они должны разрешаться согласно правилам протокола. Дело в том, что время, необходимое для посылки данных по сети, больше времени «проталкивания» данных по проводу, поэтому два устройства, создавшие коллизию, никогда не увидят данные друг друга до тех пор, пока они не «выложат» все свои данные в сеть. Поздние коллизии приводят к серьезному снижению производительности, особенно в средах NFS версии 2. Причинами возникновения поздних коллизий могут быть чрезмерно длинные сегменты, поврежденные разъемы или неисправные устройства.

Наличие поздних коллизий – это *очень* весомое свидетельство серьезных проблем в сети. Обычные коллизии не оказывают существенного влияния на производительность, если не превышают 10% в некоммутируемых сетях и 5% в коммутируемых.

[1] То есть превратить сети в коммутируемые. – *Примеч. науч. ред.*

Автоматическое согласование

Когда производители приступили к выпуску устройств Ethernet 100 Мбит/с, то такие устройства, особенно сетевые адаптеры и коммутаторы, должны были поддерживать старый стандарт 10 Мбит/с.

Стандарт 100BASE-T описывает механизм *автоматического согласования* (*auto-negotiation*)[1], благодаря которому устройства могут работать как на скорости 10 Мбит/с, так и на скорости 100 Мбит/с. Кроме того, автоматическое согласование определяет, способно ли устройство работать в дуплексном режиме или только в полудуплексном.

Таким образом, механизм автоматического согласования ищет ответ на два вопроса, причем на каждый из них возможно два ответа:

- На какой скорости может работать интерфейс? (10 или 100 Мбит/с.)
- Какой режим может использовать интерфейс? (Дуплексный или полудуплексный.)

К сожалению, несмотря на то что механизм автоматического согласования замечательно выглядит в теории, его еще нужно реализовать. Хотя в окончательном варианте стандарта 100BASE-T механизм автоматического согласования и был описан, тем не менее это был один из тех разделов, который дорабатывался в последнюю очередь, а многие производители выпустили продукты 100 Мбит/с еще до завершения разработки стандарта.

Рассмотрим пример, где автоматическое согласование работает должным образом. Пусть в локальном сетевом интерфейсе и сетевом интерфейсе коммутатора (например, коммутатора рабочей группы) включено автоматическое согласование. В начальной фазе соединения каждый из интерфейсов передает список режимов, в которых он может рабо-

Дуплексный и полудуплексный режимы

Понятие дуплексного и полудуплексного режима заключается в том, могут ли обе стороны передавать данные одновременно. В *дуплексном* соединении одновременные передачи данных возможны. За счет этого в устойчивом соединении пропускная способность фактически удваивается.[a] В *полудуплексном* соединении в любой момент времени передавать данные может только одно устройство.

[a] Отношение объема переданных данных к объему отправленных примерно равно 1.

[1] Иногда такой механизм называют *автоопознаванием*, что вводит в заблуждение.

тать. С учетом этих данных каждая сторона может независимо выбрать наиболее подходящий режим. Важно отметить, что автоматическое согласование – это активный метод определения режима связи. Ожидается, что каждый интерфейс передаст данные о режимах, в которых он может работать (в особом формате). Если интерфейс такой информации не получает, то он предполагает, что другая сторона не поддерживает автоматического согласования. Кроме того, следует понимать, что механизм автоматического согласования предполагает, что каждое устройство сделает одинаковый выбор из списка типичных режимов. *Обычно* так и происходит, однако надежность не составляет 100%.

Большинство интерфейсов, выпущенных до принятия стандарта, используют автоопознавание. В таких интерфейсах на основании входящих данных делается предположение о том, на что способна вторая сторона соединения. Непосредственного взаимодействия интерфейсов для выяснения взаимоприемлемого режима не происходит. Скорость соединения определить легко, поскольку каждый пакет Ethernet начинается с последовательности перемежающихся единиц и нулей. Для того чтобы определить скорость, интерфейс просто замеряет время между этими изменениями состояния. Когда один интерфейс переходит с самого быстрого режима работы на более медленные, то другой интерфейс может выбрать первую приемлемую для него скорость. К сожалению, надежно определить поддержку дуплексного режима с помощью пассивных методов почти невозможно. Кроме того, соединение Ethernet будет работоспособно,[1] даже если дуплексные режимы интерфейсов не совпадают.

Для того чтобы преодолеть такие трудности, в интерфейсах обычно устанавливают конкретный режим и не рассчитывают на автоматическое согласование или автоопознавание. Однако такая практика может вызвать осложнения. Вот классический пример. В интерфейсе коммутатора установлен дуплексный режим 100 Мбит/с, а в интерфейсе рабочей станции на другом конце соединения задан полудуплексный режим 100 Мбит/с. В этом случае возникает много сетевых ошибок. Причина состоит в том, что в большинстве новых интерфейсов используется автоматическое согласование. Пусть автоматическое согласование на стороне коммутатора выключено, а интерфейс на другой стороне не посылает свои данные автоматического согласования. Поскольку коммутатор не посылает таких данных, то хост на другой стороне предположит, что интерфейс коммутатора не способен работать в дуплексном режиме и задействует пассивные методы автоопознавания для выбора скорости передачи.

[1] Хотя скорость передачи данных уменьшится в несколько, а то и в десятки раз. В реальных условиях на моих глазах это приводило к снижению скорости передачи данных до 30 Кбайт/с, а это примерно в 25 раз медленнее, чем обычно. – *Примеч. науч. ред.*

В итоге нужно либо доверять результативности автоматического согласования, либо вручную конфигурировать обе стороны соединения, особенно если необходим дуплексный режим. Если у одной из сторон автоматическое согласование отключено, то другая сторона соединения всегда найдет правильную скорость, однако по умолчанию перейдет в полудуплексный режим.

Просмотр и установка режимов

В системах Solaris, применяющих драйвер Ethernet *hme*[1], текущий статус интерфейса Ethernet можно узнать с помощью *ndd -get device flag*:

```
# ndd -get /dev/hme link_status
1
# ndd -get /dev/hme link_speed
1
# ndd -get /dev/hme link_mode
1
```

Смысл выдаваемых значений разъяснен в табл. 7.5.

Таблица 7.5. Возвращаемые значения ndd

Флаг	Смысл «0»	Смысл «1»
link_status	Интерфейс неактивен	Интерфейс активен
link_speed	10 Мбит/с	100 Мбит/с
link_mode	Полудуплекс	Дуплекс

Параметры конфигурации интерфейса проще всего изменять с помощью утилиты *hmeconfig*, которую коллега авторов написал во время их работы в Университете штата Иллинойс. Эта программа доступна по адресу *http://arrakis.cso.uiuc.edu/jak/code/hmeconfig.html* и самодокументирована.

Кроме того, для установки интерфейса в конкретный режим можно воспользоваться *ndd*. Как показано в следующем примере, такую настройку можно выполнять во время работы интерфейса. Однако следует отметить, что порядок выполнения этих команд является важным. Вот последовательность команд, устанавливающая интерфейс *hme* в дуплексный режим 100 Мбит/с:

```
# ndd -set /dev/hme instance 0
# ndd -set /dev/hme adv_100T4_cap 0
# ndd -set /dev/hme adv_100fdx_cap 1
# ndd -set /dev/hme adv_100hdx_cap 0
```

[1] На самом деле hme – это акроним для Happy Meal Ethernet. Предшественник адаптера 10/100 имел название be, что означает BigMac Ethernet (а не Broken Ethernet, как принято шутить).

```
# ndd -set /dev/hme adv_10fdx_cap 0
# ndd -set /dev/hme adv_10hdx_cap 0
# ndd -set /dev/hme adv_autoneg_cap 0
```

Если необходимо, чтобы изменения сохранились после перезагрузки, то следует отредактировать */etc/system*:

```
set hme:hme_adv_autoneg_cap=0
set hme:hme_adv_100T4_cap=0
set hme:hme_adv_100fdx_cap=1
set hme:hme_adv_100hdx_cap=0
set hme:hme_adv_10fdx_cap=0
set hme:hme_adv_10hdx_cap=0
```

Каждый из этих параметров имеет четко определенное значение. Параметры и их значения приведены в табл. 7.6.

Таблица 7.6. Параметры сетевого интерфейса hme

Переменная	Значение[a]
adv_autoneg_cap	Использование автоматического согласования
adv_100T4_cap	Устройство способно работать согласно кабельному стандарту T4
adv_100fdx_cap	Устройство способно работать в дуплексном режиме 100 Мбит/с
adv_100hdx_cap	Устройство способно работать в дуплексном режиме 100 Мбит/с
adv_10fdx_cap	Устройство способно работать в дуплексном режиме 10 Мбит/с
adv_10hdx_cap	Устройство способно работать в полудуплексном режиме 10 Мбит/с

[a] Для всех параметров значение 0 означает «выключено» («off»), а значение 1 – «включено» («on»).

Например, для того чтобы перевести интерфейс в полудуплексный режим 10 Мбит/с, необходимо всем этим параметрам, за исключением adv_10hdx_cap, присвоить нуль. Параметру adv_10hdx_cap следует присвоить 1.

Узнать характеристики интерфейса на другой стороне соединения можно с помощью следующих команд:

```
# ndd -get /dev/hme lp_autoneg_cap
1
# ndd -get /dev/hme lp_100fdx_cap
1
# ndd -get /dev/hme lp_100hdx_cap
1
```

```
# ndd -get /dev/hme lp_10fdx_cap
1
# ndd -get /dev/hme lp_10hdx_cap
1
```

В этом случае хост на другой стороне способен к автоматическому согласованию и использованию всех возможных сетевых режимов.

К сожалению, в Linux нет общего метода настройки интерфейса вследствие поддержки большого количества различных сетевых адаптеров. Поэтому необходимо внимательно изучить документацию для каждого драйвера.

В многомодульных коммутаторах Cisco, запускающих Switching System Version 5.5, например в коммутаторах серии Catalyst 5500, для отображения состояния конкретного порта можно воспользоваться командой ***show port*** *card/number*:

```
Switch> (enable) show port status 2/6
Port  Name               Status     Vlan       Level  Duplex Speed Type
----- ------------------ ---------- ---------- ------ ------ ----- ----------
 2/6  Sample Connection  connected  trunk      normal full     100 100BaseTX
```

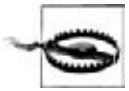
Переключение между дуплексным и полудуплексным режимами порта можно осуществлять с помощью команды ***set port duplex*** *card/number value*, где *value* – либо «half» (полудуплексный режим), либо «full» (дуплексный). Задать скорость порта либо установить автоматическое согласование можно с помощью команды ***set port speed*** *card/number value*, где *value* может быть «100», «10» либо «auto» (автоматическое согласование).[1] Вот пример установки порта 4 в адаптере 3 в полудуплексный режим 10 Мбит/с:

```
Switch> (enable) set port speed 4/3 10
Port 4/3 speed set to 10 Mbps.
Switch> (enable) set port duplex 4/3 half
Port 2/1 set to half-duplex.
```

В коммутаторах Cisco класса рабочей группы, таких как коммутаторы серий Catalyst 2900 и 3900, применяется другой интерфейс конфигурации, очень напоминающий операционную систему IOS, которая работает на большинстве маршрутизаторов Cisco. Изменить параметры порта можно с помощью следующей последовательности команд:

```
Switch# configure terminal
configure# interface interface
configure# speed 10|100|auto
configure# duplex half|full|auto
configure# end
```

[1] Если в интерфейсе установлено, что скорость определится автоматическим согласованием, то таким же образом определится и его дуплексный режим.



Изменение параметров интерфейса обычно предполагает останов сетевого интерфейса, а затем его повторное включение. Такие действия могут быть опасны, если через этот интерфейс система подсоединена к сети...

FDDI

Распределенный интерфейс передачи данных по оптоволокну (Fiber Distributed Data Interface, FDDI), разработанный ANSI в середине 1980-х годов, явился ответом на растущее давление, вызванное нехваткой полосы пропускания в существовавших технологиях. Стало ясно, что необходим новый носитель, способный поддерживать современные, высокопроизводительные рабочие станции, а также обладающий высоким уровнем сетевой надежности. Эти две характеристики, производительность и надежность, стали ориентиром в создании нового протокола.

FDDI – это структура 100 Мбит/с, в которой две непрерывных петли, или *кольца*, подсоединены к каждому хосту в сети. Однако направления передачи в этих кольцах противоположны: трафик в каждом кольце идет навстречу друг другу. Одно из колец назначается первичным. В ходе обычной операции по первичному кольцу проходит весь трафик, а другое кольцо остается бездействующим. Для такой схемы характерен высокий уровень отказоустойчивости.

Если в станции, присоединенной к обоим кольцам, происходят неполадки или повреждается кабель, то кольцо автоматически сворачивается (то есть «свободные» концы на любой стороне повреждения «соединяются» аппаратурой FDDI). При этом образуется единое кольцо. Сетевая производительность абсолютно не изменяется. Однако FDDI способно противостоять только одной неисправности. Если происходит два или более повреждения, то кольцо будет сегментировано на несколько независимых колец, которые не смогут взаимодействовать. Здесь выручает *оптический обходной коммутатор* (*optical bypass switch*), который, по существу, «сидит» на кольце вместо хоста. С помощью зеркал этот коммутатор ретранслирует сигналы, предназначенные хосту, по кольцу FDDI. В случае неполадок оптический обходной коммутатор пропустит свет через себя и обеспечит целостность кольца.

Однако не все устройства нужно подсоединять к обоим кольцам. Одна из уникальных особенностей FDDI состоит в том, что существуют несколько способов соединения устройств. Есть три вида устройств: *концентратор* (*concentrator*), *однопортовая станция* (*single-attachment station*, SAS) и *двухпортовая станция* (*dual-attachment station*, DAS). Двухпортовая станция имеет два порта и присоединена к обоим кольцам. В тех случаях, когда питание двухпортовой станции выключается, в кольце происходит сбой, однако современные интерфейсы FDDI имеют встроенные оптические обходные коммутаторы.

Концентратор, также называемый *двухпортовым концентратором* (*dual-attachment concentrator*, DAC), присоединен к обоим кольцам. Благодаря ему сбой на любой присоединенной станции не подвергает кольцо опасности. По существу, концентратор эквивалентен хабу Ethernet. Однопортовая станция имеет единственный порт, с помощью которого она подсоединяется только к одному кольцу через концентратор. Основное преимущество такого соединения состоит в том, что в случае отсоединения устройства или выключения питания целостность кольца не будет нарушена.

Формат кадра FDDI в значительной степени перенял особенности кадра Token Ring. Размер кадра FDDI может быть до 4500 байт. Здесь протокол FDDI подробно обсуждаться не будет. Отметим лишь, что основная идея сводится к тому, что в любой момент времени только одна станция (с «маркером») может передавать данные. Когда станция завершает передачу, она передает маркер следующему хосту. Кроме того, в FDDI реализованы очень мощные протоколы управления сетью.

В FDDI, как явствует из его названия, обычно задействовано оптоволокно. Однако протокол FDDI может работать и по медным кабелям. Официально такой интерфейс именуется распределенным интерфейсом передачи данных по витой паре (Twisted Pair Distributed Data Interface, TP-DDI), однако чаще его называют медным распределенным интерфейсом передачи данных (Copper Distributed Data Interface, CDDI). В CDDI расстояние между концентратором и хостом может быть до 100 м, тогда как в FDDI по многомодовому оптоволокну – до 2 км. Если применять одномодовое оптоволокно, то возможны и бульшие расстояния.

В начале 1990-х годов FDDI широко использовался в магистралях, поскольку в то время он рассматривался как быстрый интерфейс, способный работать на длинных расстояниях и обладающий отказоустойчивостью. Однако когда появились более дешевые ATM и Fast Ethernet, FDDI перестал пользоваться спросом.

ATM

Еще одна типичная высокопроизводительная структура сети – это *режим асинхронной передачи* (*Asynchronous Transfer Mode*, ATM). Сеть ATM задумывалась как настоящая универсальная сеть с комплексными услугами, где голос, данные и видеосигнал могут передаваться по единой инфраструктуре, тем самым обеспечивая рентабельность сети. К сожалению, в структуре ATM есть недостатки, поэтому она оказалась в тени сети Gigabit Ethernet, предлагающей более быструю скорость передачи данных и более простую конфигурацию.

ATM – это технология на основе соединения (она устанавливает непосредственные соединения между конечными точками; такие соединения называются *контурами*, *circuits*). Для передачи информации при-

меняются *ячейки* (*cells*) размером 53 байт. Такая ячейка состоит из 5 байт заголовка и 48 байт данных.[1] Вопреки названию сети ATM пакеты в ней не переносятся асинхронно. Ячейки передаются непрерывно, даже когда данные не посылаются. Когда трафика нет, в каждой передаваемой ячейке присутствует специальная последовательность битов, свидетельствующая о том, что ячейка пуста. Слово «асинхронная» в названии ATM вызвано тем, что время отправки первой ячейки не задается. Благодаря таким маленьким ячейкам фиксированной длины аппаратная коммутация является простой и быстрой, а буферы могут точно размещаться в сегментах, что предотвращает потери памяти.

Сеть ATM можно реализовать на многих физических носителях с различными скоростями передачи данных (табл. 7.7).

Таблица 7.7. Физические носители ATM с различными скоростями передачи данных

Скорость передачи данных	Носитель
25 Мбит/с	UTP категорий 3, 4 и 5
155 Мбит/с	UTP категории 5, многомодовое оптоволокно
622 Мбит/с	Многомодовое и одномодовое оптоволокно

Одним из главных преимуществ технологии ATM (по крайней мере, в теории) является поддержка всестороннего *качества обслуживания* (*quality of service*, QoS). Благодаря этому для приложений возможно резервирование определенной части полосы пропускания, а их ячейки могут обслуживаться с более высоким приоритетом.

Существуют три различных подхода к передаче пакетов IP поверх ATM: классический IP поверх ATM, LAN Emulation (LANE) и Multiprotocol over ATM (MPOA). Классический IP преобразует адреса IP в адреса ATM, что позволяет устройствам ATM передавать трафик IP. К сожалению, классический IP поверх ATM поддерживает только сам протокол IP (нет надлежащей поддержки для IPX, AppleTalk и любых других транспортных протоколов). Кроме того, такие сети сложно расширять. Для того чтобы снизить время развертывания оборудования ATM, в LANE применен другой подход. Сеть ATM делается «невидимой» для существующей LAN Ethernet, то есть интерфейс LANE выглядит как стандартный интерфейс Ethernet. К сожалению, поскольк-

[1] Читатель может заинтересоваться, почему размер данных равен 48 байт. Напомним, что сеть ATM разрабатывалась как для передачи данных, так и для передачи голоса. Ходят слухи, что было выбрано именно это число, поскольку оно являлось средним между требованиями инженеров, отвечающих за передачу голоса (32 байт), и предпочтениями инженеров, отвечающих за передачу данных (по меньшей мере, 64 байт). Обычная присказка к таким слухам – «...и каждый остался с носом».

ку LANE так «прозрачен», он не может использовать преимущества ATM как сервисного протокола.

Сравнение Ethernet и ATM/FDDI

До того как коммутируемое оборудование Fast Ethernet получило широкое распространение, ATM и FDDI могли похвастаться двумя большими преимуществами. Их производительность под нагрузкой угасает более плавно, нежели нагрузка на участках Ethernet, соединенных повторителями. Кроме того, в них нет коллизий. Однако сегодня коммутируемые сети Ethernet свели на нет почти все эти преимущества. В итоге сеть FDDI почти полностью вышла из употребления, а сеть ATM была низведена с пьедестала и занимает небольшие ниши на рынке. Если рассматривать показатель цена–производительность, то побить Fast или Gigabit Ethternet трудно. Однако в сети ATM есть особые технические преимущества, которые могут быть полезны в некоторых операционных средах.

Сетевые протоколы

Если физические сети Ethernet, FDDI и ATM представить как шоссе, то сетевые протоколы – это правила на дорогах. Мало просто иметь полосу асфальта. Нужно нанести линии, чтобы регулировать трафик. Как из Сан-Франциско добраться до Бостона? Кто имеет право проехать первым, если автомобили одновременно застыли на красном сигнале светофора?[1] Эти правила – зона действия двух главных сетевых протоколов: *межсетевого протокола* (*Internet protocol*, IP) и *протокола управления передачей* (*Transmission Control Protocol*, TCP). Если продолжить аналогию с шоссе, то протокол IP аналогичен правилам для прокладки междугородних автомобильных дорог – он описывает, как из одного города добраться до другого (следующего). Протокол TCP – это совокупность правил о том, как путешествовать по дорогам.

IP

IP – это протокол сетевого уровня (см. раздел «Модель OSI» ранее в этой главе), изобретенный в начале 1970-х годов Винтом Серфом (Vint Cerf) и Бобом Коэном (Bob Kahn). Вот другие примеры протоколов сетевого уровня: IPX и AppleTalk. Однако здеь они обсуждаться не будут.

Протокол IP определяет заголовок, присоединяемый к началу каждой передаваемой порции данных. Заголовок и данные образуют *пакет* (*packet*). В табл. 7.8 представлен формат заголовка IP.

---

[1] Как будет далее видно для протокола TCP, автомобили вполне могут столкнуться, а вместо них отправляются новые автомобили. Разве не было бы здорово так легко справляться с автокатастрофами?

Таблица 7.8. Заголовок IP

Биты	Содержимое
0–3	Версия
4–7	Длина заголовка
8–15	Тип сервиса
16–31	Общая длина
32–47	Идентификатор
48	Не используется
49	Флаг: не фрагментировать (DF)
50	Флаг: больше фрагментов (MF)
51–63	Смещение фрагмента
64–71	Время жизни
72–79	Протокол
80–95	Контрольная сумма заголовка
96–127	Адрес источника
128–159	Адрес назначения
160–191	Опции и дозаполнение

Поле версии обычно равно 4, так как версия 4 (Ipv4) – это текущая версия протокола IP, получившая широкое распространение. Версия 6 (Ipv6) медленно продвигается вперед, однако пока широко не применяется. Производительность передачи данных в IPV6 еще не была хорошо изучена, поэтому эта версия здесь обсуждаться не будет. Длина заголовка указывает длину поля заголовка в 32-битных словах. Поле типа сервиса определяет специальные требования к обслуживанию пакета, реализуемые протоколами обеспечения качества обслуживания (Quality of Service). Такими требованиями могут быть сортировка и приоритетное обслуживание. Поле общей длины определяет длину всего пакета в байтах, включая заголовок.[1]

Фрагментация

Поле идентификатора вместе с полем смещения фрагмента и двумя битами-флагами управляет *фрагментацией* пакета (*fragmentation*). Если его первоначальная длина превышает величину *максимальной единицы передачи* (*maximum transmission unit*, MTU), установленной для базового канала передачи данных, то такой пакет должен быть разбит на

[1] Максимальное десятичное число, представляемое 16 бит (размер поля общей длины) равно 65 535. В итоге максимально возможный размер пакета IP составляет 65 535 байт.

множество маленьких пакетов. Например, пакет размером 6500 байт, путешествующий по сети Ethernet с MTU 1500 байт, будет разбит на несколько пакетов, каждый из которых будет меньше 1500 байт. В тех случаях, когда необходима фрагментация пакета, но установлен флаг «не фрагментировать», пакет не подлежит фрагментированию, а потому будет отброшен с сообщением об ошибке. Когда маршрутизатор фрагментирует пакет, то во всех фрагментах, кроме последнего, биту-флагу «больше фрагментов» присваивается 1. По этому признаку получатель узнает о прибытии последнего фрагмента. В поле смещения фрагмента закодирована информация, необходимая для последующей сборки фрагментов в правильном порядке.

Время жизни

Полю времени жизни присваивается определенное жесткое числовое значение (обычно 64, но 15 и 32 так же часто встречаются). По мере прохождения пакета через маршрутизаторы это число уменьшается. Если оно станет равным нулю, то пакет будет отброшен, а к источнику вернется сообщение об ошибке. Благодаря этому пакеты-«зомби», которые не могут найти свой пункт назначения, не блуждают по сети, наслаждаясь жизнью.

Протоколы

Поле протокола описывает тип передаваемых данных. Существуют более 100 номеров, закрепленных за различными протоколами. Наиболее типичные номера приведены в табл. 7.9.

Таблица 7.9. Типичные номера протоколов

Номер протокола	Описание
1	Протокол управляющих сообщений в сети Интернет (Internet Control Message Protocol, ICMP)
6	Протокол управления передачей (Transmission Control Protocol, TCP)
17	Пользовательский протокол данных (User Datagram Protocol, UDP)
88	Протокол маршрутизации внутреннего шлюза (Internet Gateway Routing Protocol, IGRP)
89	Первоочередное открытие кратчайших маршрутов (Open Shortest Path First, OSPF, протокол маршрутизации)

Очевидно, что контрольная сумма заголовка содержит контрольную сумму для заголовка. Однако это поле должно пересчитываться на каждом маршрутизаторе, поскольку значение времени жизни пакета уменьшается. Адреса источника и назначения – это 32-битные адреса протокола IP.

Адреса IP

Адрес протокола IP – это 32-битное двоичное значение, состоящее из *сетевой части* (*network portion*), уникально определяющей сеть, и *машинной части* (*host portion*),[1] которая уникально определяет хост в этой сети. Поскольку записывать двоичные значения затруднительно, IP-адреса часто представляют в «десятичном представлении с разделительными точками». Каждый из четырех однобайтных участков, разделенных точками, представляет собой десятичное число от 0 до 255. Десятичное представление с разделительными точками служит лишь для удобства чтения IP-адресов. Все сетевые аппаратные средства всегда работают с 32-битными двоичными адресами.

Одна из самых интересных особенностей адреса IP состоит в том, что размеры сетевой и машинной частей могут варьироваться в зависимости от размера сети. Разработчики IP сделали этот протокол очень гибким, поэтому он может поддерживать как огромные сети с миллионами хостов, так и маленькие сети, содержащие лишь несколько машин. Вот почему управлять IP-адресами в какой-то мере трудно.

Классовая адресация

Первая схема, разработанная для управления IP-адресами, предполагала разделение сетей на три широких группы, или класса:

Сеть класса A

Очень большая, много хостов. Сетей класса A существует немного.

Сеть класса B

Среднего размера, среднее количество хостов. Сети класса B более распространены, чем сети класса A.

Сеть класса C

Очень маленькая (менее 250 присоединенных хостов). Сети класса C обычны.

В итоге сетевая и машинная части адреса делятся согласно табл. 7.10.

Таблица 7.10. Сводка классов сетей

Класс	Сетевая часть (в байтах)	Возможное количество сетей[a]	Машинная часть (в байтах)	Возможное количество хостов в сети
A	1	127	3	16 777 216
B	2	32 767	2	65 536
C	3	8 388 607	1	256

[a] Это не то, что можно было ожидать, так как часть адресного пространства класса A используется для адресов класса B и C.

[1] Эти части нередко называют «номером сети» и «номером компьютера». – *Примеч. науч. ред.*

IP-адрес, в котором в машинной части стоят все нули, указывает на саму сеть в целом и называется *сетевым адресом* (*network address*). Определить, к какой сети принадлежит хост, можно с помощью *маски адреса* (*address mask*). Маска адреса – это 32-битное двоичное число, в котором все байты, относящиеся к сетевой части, установлены в 1. Результатом операции «поразрядного И», операндами которой являются любой IP-адрес и его маска, всегда будет сетевой адрес. Для примера обратимся к табл. 7.11.

Таблица 7.11. Адреса IP, сетевые маски и сетевые адреса в сети класса B

	Десятичное представление с разделительными точками	1-й байт	2-й байт	3-й байт	4-й байт
IP-адрес хоста	176.16.146.38	10101100	00010000	10010010	00100110
Сетевая маска	255.255.0.0	11111111	11111111	00000000	00000000
Сетевой адрес (поразрядное И)	176.16.0.0	10101100	00010000	00000000	00000000

Сравнение общего и частного адресных пространств

Согласно RFC 1918, часть IP-адресов зарезервирована для «частного» применения. То есть такие адреса никогда не могут встречаться в сетях общего пользования. К зарезервированным адресам относятся: 10.0.0.0–10.255.255.255, 172.16.0.0–172.31.255.255 и 192.168.0.0–192.168.255.255.

Частные адреса применяются в сетях IP, которые не присоединены к Интернету. Примерами могут служить соединения «точка–точка» в центрах обработки данных или в сетях, которые «скрыты» от Интернета с помощью брандмауэра или механизма трансляции сетевых адресов (Network Address Translation, NAT).

Подсети классовых сетей

Работа с подсетями – это сложная тема, которую многие администраторы понимают недостаточно хорошо. Однако прежде чем приступить, дадим совет.

При работе с подсетями лучше применять двоичное представление адресов. Десятичное представление с разделительными точками существует только для облегчения восприятия человеком. Машины мыслят двоичными категориями.

Подсети применяются для повышения эффективности назначения IP-адресов. *Выделение подсетей* (*subnetting*) предполагает разбивку еди-

ной большой сети на множество более мелких сетей путем расширения сетевой части адреса на величину, не превышающую полный байт. То есть сетевая часть вторгается в машинную часть.

Пусть необходимо сегментировать сеть 172.16.0.0 класса B. Если расширить сетевую маску на 8 бит, то есть с 2 до 3 байт, то в рамках одного адреса класса B можно будет назначить до 256 подсетей. Обратимся к табл. 7.12, чтобы увидеть, как это выглядит в двоичном виде.

Таблица 7.12. Разбивка на подсети в сети класса B

	Десятичное представление с разделительными точками	**1-й байт**	**2-й байт**	**3-й байт**	**4-й байт**
IP-адрес хоста	176.16.146.38	10101100	00010000	10010010	00100110
Маска подсети	255.255.255.0	11111111	11111111	11111111	00000000
Адрес подсети (поразрядное И)	176.16.146.0	10101100	00010000	10010010	00000000

Дополнительные 8 бит в маске подсети – это часть подсети, в добавление к уже описанным машинной и сетевой частям адреса. Заметим, что сетевой адрес по-прежнему остается тем же – 176.16.0.0!

Маски подсети могут быть представлены тремя способами: десятичное представление с разделительными точками (см. табл. 7.12); шестнадцатеричная форма, полученная из 32-битного двоичного значения (например, в табл. 7.12 маска подсети в шестнадцатеричной форме будет выглядеть так: 0xFFFFFF00), и форма с указанием битов, в которой после сетевого адреса ставится косая черта и указывается количество бит, используемых для маски подсети (если вернуться к примеру из табл. 7.12, то адрес и маска будут выглядеть так: 172.16.0.0/24).

При назначении адресов в подсети следует придерживаться следующих правил:

- В каждой подсети есть два «неиспользуемых» адреса IP: сам адрес подсети и широковещательный адрес.
- Количество подсетей, возможных для сети с заданной маской подсети, равно $2^{ls}-2$, где ls – это длина части подсети в битах.
- Количество возможных адресов хостов в подсети равно $2^{lh}-2$, где lh – это длина части хоста в битах.

В рассмотренном примере пространство подсети представлено целым байтом. Так бывает не всегда, причем такая разбивка не является преимущественной. Пусть есть сеть 192.168.144.0 класса C. Необходимо организовать работу 6 отделов, в каждом из которых 25 систем. Это легко сделать, если часть подсети составляет 3 бит (2^3-2 дает 6 возможных подсетей – будем надеяться, что добавлять отделы не понадобится!). Тогда в каждой подсети на машинную часть придется по 5 бит, по-

этому в каждой сети может быть 30 хостов. Как найти маску подсети для такой разбивки? Рассмотрим табл. 7.13.

Таблица 7.13. Разбивка на подсети сети класса C

	Десятичное представление с разделительными точками	1-й байт	2-й байт	3-й байт	4-й байт
Сетевой адрес	192.168.144.0	11000000	10101000	10010000	00000000
Маска подсети	255.255.255.224	11111111	11111111	11111111	11100000

Если часть подсети представлена с помощью 3 бит, то вот какими будут все возможные комбинации: 000, 001, 010, 011, 100, 101, 110 и 111. Кроме того, известно, что первый и последний адреса, 000 и 111, задействовать нельзя. Вычислив все возможные комбинации для 5 бит машинной части адреса, можно определить диапазон возможных адресов для каждой подсети. Вычисления обобщены в табл. 7.14.

Таблица 7.14. Вычисление диапазонов адресов для подсетей в сети класса C

Подсеть	Сетевой адрес	Широковещательный адрес	Диапазон адресов хостов
001	192.168.144.32	192.168.144.63	192.168.144.33–192.168.144.62
010	192.168.144.64	192.168.144.95	192.168.144.65–192.168.144.94
011	192.168.144.96	192.168.144.127	192.168.144.97–192.168.144.126
100	192.168.144.128	192.168.144.159	192.168.144.129–192.168.144.158
101	192.168.144.160	192.168.144.191	192.168.144.161–192.168.144.190
110	192.168.144.192	192.168.144.223	192.168.144.193–192.168.144.222

Теперь ясно (возможно, мучительно ясно), почему перед началом обсуждения подсетей был дан совет использовать двоичное представление адресов. По виду адреса, представленного в десятичной форме, трудно определить, является ли он адресом подсети, широковещательным адресом или адресом хоста. Мало помогает даже маска подсети. Поэтому при работе с подсетями все адреса следует рассматривать в двоичном виде.

Переход к бесклассовому миру

К 1990 году Интернет столкнулся с двумя серьезными проблемами роста. Поскольку популярность Интернета росла, появилось море новых классовых сетей. Каждую из них нужно было включить в таблицы маршрутизации. В итоге маршрутизаторам стало не хватать памяти, а разрешение адресов стало отнимать слишком много времени. Кроме того, становилось ясно, что спрос на новые сети класса B скоро исчерпает имеющиеся в наличии адреса. Инженерная комиссия Интернета (Internet Engineering Task Force, IETF) разработала решение,

известное как *бесклассовая междоменная маршрутизация* (*classless interdomain routing*, CIDR), или просто *бесклассовая маршрутизация*, или *supernetting*.

Технология CIDR основана на расширении уже обсужденной схемы организации подсетей. CIDR предполагает перенос границы между машинной и сетевой частями адреса влево, а не вправо. Группы соседних классовых сетей могут быть объединены в единые элементы таблицы маршрутизации. Группы сетей класса C могут назначаться пачками по 2, 4, 8 или 16 – для того, чтобы обеспечить потребности организаций, которые в ином случае потребовали бы сети класса B.

Для того чтобы различать машинную и сетевую части адреса, в бесклассовой адресации вместо маски подсети применяется признак длины. Этот признак, состоящий из косой черты и числа, которые ставятся после сетевого адреса, называется *маской подсети с переменной длиной* (*variable length subnet mask*), или VLSM. Число после адреса определяет, сколько бит занимает сетевая часть адреса. Например, в адресе 192.168.214.0/21 сетевая часть составляет 21 бит. В стандартном представлении маска подсети была бы такой: 255.255.248.0.

Большим преимуществом бесклассовой адресации является повышение эффективности назначения IP-адресов. Например, если организации назначен адрес класса B, то у нее есть 16 384 адреса. Если в этой организации только 1000 хостов, то оставшаяся часть адресов резервируется, но не используется. В бесклассовом мире такой организации можно было бы просто назначить одну сеть с VLSM из 22 бит, что дает 1024 адреса (эквивалент четырех «объединенных» сетей класса C). Эквивалентная сетевая маска была бы такой: 255.255.252.0.

Маршрутизация

В этой книге не будут обсуждаться средства, с помощью которых маршрутизаторы опознают друг друга и обмениваются информацией, тем самым содействуя соединениям в IP-сетях. При наличии интереса к этой теме следует обратиться к замечательному (объемному) справочнику Джефа Дойла (Jeff Doyle) «Routing TCP/IP» (Маршрутизация TCP/IP), Cisco Press.

TCP

Протокол управления передачей, или TCP, представляет собой надежный коммуникационный протокол на основе соединений. TCP предусматривает установление двухточечного соединения, обладающего двумя особенностями: есть только один маршрут от источника до пункта назначения, а пакеты приходят в том порядке, в котором были посланы. Однако поскольку базовый механизм доставки (IP) – это механизм без установления прямого соединения, то TCP может предоставить только видимость соединения. Подобно телефонному соедине-

нию, модуль TCP должен установить соединение, передать данные, а после завершения передачи данных разорвать соединение. Для достижения этих целей в TCP применяются три механизма:

- В первом байте каждого пакета содержится порядковый номер, поэтому сторона-получатель может правильно объединить пакеты, пришедшие не по порядку.
- Применение системы уведомлений, контрольных сумм и таймеров обеспечивает надежность. Например, сторона-получатель может уведомить отправителя, что пакет не прибыл. Отправитель повторно посылает любой пакет, не подтвержденный стороной-получателем в течение некоторого периода времени.
- Существует механизм управления потоком пакетов, называемый *оконным*. Его применение снижает вероятность пропуска данных вследствие переполнения буфера получаемых пакетов.

В TCP к отправляемым данным присоединяется заголовок, содержащий поля адресов приложения-источника и приложения-получателя (порты), а также все контрольные данные TCP. После этого данные плюс заголовок TCP инкапсулируются в IP-пакет для доставки. Заголовок TCP довольно сложный (табл. 7.15).

Таблица 7.15. Заголовок протокола управления передачей (TCP)

Биты	Содержимое
0–15	Порт источника (source port)
16–31	Порт пункта назначения (destination port)
32–63	Порядковый номер (sequence number)
64–95	Номер уведомления (acknowledgement number)
96–99	Длина заголовка (header length)
100–105	Зарезервировано (reserved)
106	Флаг: срочный (URG)
107	Флаг: уведомление (ACK)
108	Флаг: немедленная отправка данных (PSH)
109	Флаг: необходимость уничтожения канала (RST)
110	Флаг: синхронизация (SYN)
111	Флаг: закрытие канала (FIN)
112–127	Размер окна (window size)
128–143	Контрольная сумма (checksum)
144–159	Указатель срочности (urgent pointer)
160–191+	Опции и дозаполнение нулями (options and padding)

Просмотр и установка параметров TCP: ndd

ndd – это инструмент Solaris, позволяющий просматривать и устанавливать параметры без перезагрузки системы. Несмотря на то что он может применяться для большинства устройств, имеющих файл устройства в */dev* и модуль ядра, здесь он будет рассматриваться как инструмент настройки TCP.

Работать с *ndd* можно в интерактивном режиме:

```
# ndd /dev/tcp
name to get/set ? tcp_slow_start_initial
value ?
length ?
1
name to get/set ? ^D
```

Утилиту *ndd* можно применить для отображения всех доступных параметров:

```
# ndd /dev/icmp \?
?                               (read only)
icmp_wroff_extra                (read and write)
icmp_ipv4_ttl                   (read and write)
icmp_ipv6_hoplimit              (read and write)
icmp_bsd_compat                 (read and write)
icmp_xmit_hiwat                 (read and write)
icmp_xmit_lowat                 (read and write)
icmp_recv_hiway                 (read and write)
icmp_max_buf                    (read only)
icmp_status                     (read only)
```

Кроме того, можно запросить значения нескольких параметров за один раз:

```
# ndd -get /dev/hme link_status link_speed link_mode
1

0

0
```

Наконец, *ndd* можно применять для настройки параметров:

```
# ndd -set /dev/tcp tcp_slow_start_initial 2
```

В Linux параметры TCP обычно назначаются жестко – путем редактирования и перекомпилирования исходного кода ядра. Во время работы системы параметры TCP также можно регулировать – посредством внесения изменений в файлы в */proc/sys/net/ipv4*.

Порядковый номер определяет позицию инкапсулированных байтов данных в потоке. Например, если сегмент имеет порядковый номер 5748 и содержит 256 байт данных, то следующий сегмент будет иметь порядковый номер 6004. Такое значение «вероятного следующего порядкового номера» сообщается в поле номера уведомления. Таким образом, хост-отправитель может узнать не только о потере пакетов, но и о том, что это были за пакеты.

Поле длины заголовка определяет длину заголовка в 32-битных словах. В зарезервированном поле всегда стоят шесть нулей. Размер окна применяется для управления потоком. В этом поле указывается количество байт, которое адресат готов принять, – начиная с байта, указанного в поле номера уведомления, который отправитель пакета получит от другой стороны соединения до выдачи запроса о подтверждении.

Контрольная сумма – это 16-битная общая контрольная сумма заголовка и инкапсулированных данных, предназначенная для исправления ошибок. Если установлен флаг срочности, то применяется и поле указателя срочности. Совместно с порядковым номером оно сообщает о завершении передачи срочных данных. Поле опций содержит данные, определяемые приложением. Наиболее типичное использование этого поля – отправка размера максимального сегмента, который сообщает получателю о самом большом размере сегмента, который отправитель может принимать. Остаток поля заполняется нулями до достижения 32-битной длины этого поля.

Следует заметить, что в ходе дальнейшего обсуждения протокола TCP будет подробно рассматриваться его внутренняя реализация. Читатели, имеющие опыт в этой области, сразу заметят, что некоторые вещи упрощены. TCP – это сложный протокол, и его полная трактовка выходит за рамки этой книги. Безусловно, лучшим справочником по TCP является книга У. Ричарда Стивенса (W.Richard Stevens) «TCP/IP Illustrated, Volume 1»,[1] Addison Wesley. В этом повествовании представлены только особенности, объясняющие работу TCP. Авторы настоятельно рекомендуют с ними ознакомиться.

Инициирование соединения и затор SYN

В TCP существует специальный способ создания нового соединения, называемый «тройным рукопожатием» (three-way handshake). В ходе этого процесса задействуется много очередей и настраиваемых параметров. Для того чтобы настройка была эффективной, необходимо не только понимать, что происходит в проводах, но и то, что происходит в системе. А именно как `listen()` и `accept()` взаимодействуют с очередями.

Когда сервер вызывает `listen()`, ядро переводит сокет из состояния `CLOSED` в состояние `LISTEN`. Одновременно ядро создает и инициализирует различные структуры данных, включая буферы сокетов и две очереди незавершенных и завершенных соединений.

[1] У. Ричард Стивенс «Протоколы TCP/IP. В подлиннике», BHV-СПб, 2003.

Когда в TCP-соединении с сервера[1] отправляется первый пакет (в котором установлен только бит SYN), то в очередь незавершенных соединений добавляется еще один элемент. Сервер посылает клиенту пакет с установленными битом ACK (подтверждение SYN клиента) и битом SYN. Теперь сокет находится в состоянии SYN_RCVD. Когда сторона-клиент получает ACK в ответ на пакет сервера с SYN, то соединение остается в этой очереди на один цикл передачи данных «туда-обратно» (round-trip time), а далее перемещается в очередь завершенных соединений.

Очередь завершенных соединений содержит элементы для каждого активного соединения (для тех, в которых тройное рукопожатие было успешно завершено, однако пользовательское приложение еще «не признало» соединение). В очереди завершенных соединений сокеты находятся в состоянии ESTABLISHED. Каждый вызов accept() убирает передний элемент из очереди. Если элементов нет, то вызов accept() обычно блокируется. Параметр tcp_conn_req_min определяет минимальное количество доступных соединений в очереди завершенных соединений для того, чтобы select() и poll() возвращали значение «читаемый» дескриптору прослушивающего сокета. Этот параметр никогда не следует изменять.

Обе описанные очереди ограничены в размере. Когда вызывается listen(), сервер может указать размер второй очереди. Исторически это значение определяло суммарное количество элементов в обеих очередях. В системы Solaris начиная с версии 2.6 включен «настроечный параметр» (умножение аргумента на три вторых). В очереди незавершенных соединений обычно нужно больше элементов, чем в очереди завершенных соединений. Большой размер очереди незавершенных соединений нужен лишь тогда, когда необходимо разрешить ее увеличение по мере прихода от клиентов пакетов с SYN. В Solaris есть два параметра ядра, устанавливающие размер этих очередей. Параметр tcp_conn_req_max_q0 – это максимальное количество соединений с незавершенным рукопожатием (очередь незавершенных соединений). Параметр tcp_conn_req_max_q – это максимальное количество завершенных соединений, которые ожидают «принятия», осуществляемого вызовом accept(). Это очередь завершенных соединений.

По умолчанию параметры tcp_conn_req_max_q0 и tcp_conn_req_max_q равны 1024 и 128 соответственно. Вывод команды *netstat -sP tcp* позволяет проанализировать значения tcpListenDrop и tcpListenDropQ0. Если сбоев много, то значение этих параметров следует увеличить. Вероятно, значение параметра tcp_conn_req_max_q0 не должно превышать

[1] Независимо от того, что называть сервером, а что – клиентом, один компьютер отправляет пакет с установленным SYN на открытый порт (или, что то же самое, в сокет), а другой, приняв этот пакет, отправляет обратно пакет с установленными SYN и ACK. См., например, *http://www.crime-research.org/library/xenon2.htm*. – *Примеч. науч. ред.*

10 000. Разумный верхний предел для параметра tcp_conn_req_max_q равен текущему значению tcp_conn_req_max_q0.

В Linux эквивалентом параметра tcp_conn_req_max_q0 является параметр tcp_max_syn_backlog, который можно найти *в /proc/sys/net/ipv4/*.

В системах Solaris до 2.5.1 (с установленными «заплатами» 103630-09 и 103582-12 и выше) есть единая очередь соединений, которой управляет параметр tcp_conn_req_max. Хотя этот параметр можно настраивать, авторы советуют провести обновление системы или установить две «заплаты» TCP (текущие версии).

Атака из серии «отказ от обслуживания» (denial-of-service), называемая *затором SYN* (*SYN flooding*), подразумевает посылку большого количества пакетов SYN с несуществующими адресами источника. Поскольку второй SYN никогда не подтверждается, то очередь listen полностью заполняется. Это означает, что новые соединения обрабатываются лишь тогда, когда «старые» соединения удаляются по тайм-ауту. Всякий раз, когда сомнительное соединение удаляется из очереди, значение счетчика tcpHalfOpenDrop инкрементируется. Высокое значение свидетельствует о возможной попытке организации затора SYN. В этом случае защиту можно усилить с помощью увеличения значения tcp_conn_req_max_q0.

Определение путевого MTU и максимальный размер сегмента

Во время тройного рукопожатия для соединения устанавливается *размер максимального сегмента* (*maximum segmetn size*, MSS). Это значение определяет наибольший объем данных, которые могут быть посланы в одном TCP-пакете. MSS устанавливается равным наименьшему MTU выходного интерфейса или значению MSS, объявленному другой стороной соединения (MTU, *единица передачи сообщения*, *message transfer unit*, – это наибольший объем данных, которые можно послать в одном пакете через заданный интерфейс). Если удаленный узел не объявляет свое значение MSS, то MSS обычно назначается значение 536. Если режим определения путевого MTU (path MTU discovery) активен, то во всех исходящих IP-пакетах устанавливается флаг DF («не фрагментировать»).

Во всех версиях Solaris значение MSS по умолчанию равно 536 байт. Это значение может отличаться от желаемого. Изменить его можно с помощью параметра tcp_mss_def_ipv4:[1]

```
# ndd -set /dev/tcp tcp_mss_def_ipv4 1460
```

Значение MSS, определяется исходя из значения MTU самого загруженного выходного интерфейса минус 40 байт для заголовков TCP/IP.

[1] В версиях Solaris, предшествующих версии 8, этот параметр называется tcp_mss_def.

Поскольку в Ethernet значение MTU равно 1500 байт, то параметру `tcp_mss_def_ipv4` следует присвоить значение 1460. Заметим, что если другая сторона соединения требует меньшее значение MSS, то такой запрос будет исполнен. В Linux значение MSS настраивать не следует.

Если стек TCP получает ошибку ICMP «необходима фрагментация сообщения», то это означает, что на пути сообщения к пункту назначения маршрутизатор должен был его фрагментировать, но сделать это ему не было позволено (вероятно, из-за установленного бита DF). В итоге маршрутизатор был вынужден отбросить пакет и послать отправителю сообщение-ошибку ICMP. На самом деле большинство новых маршрутизаторов сообщают хосту необходимое значение MSS, однако если такого не происходит, то правильное значение MSS должно быть определено на основе проб и ошибок.

RFC 1191 рекомендует повторно определять путевой MTU для контура TCP каждые 10 минут. К сожалению, в версиях Solaris, предшествующих версии 2.5, путевой MTU определялся заново каждые 30 секунд. Такая частота переопределения допустима в локальных сетях, однако для глобальных сетей это слишком высокое значение, поскольку оно приводит к излишней загрузке сети. Параметру `ip_ire_pathmtu_interval` следует присвоить значение 600 000. В Solaris управлять определением путевого MTU можно с помощью параметра `ip_path_mtu_discovery`, которому следует присвоить значение 1. В Linux значение параметра `ip_no_pmtu_disc` в */proc/sys/net/ipv4* следует оставить равным 0.

Буферы, отметки уровня и окна

Для того чтобы описать различные буферы и отметки уровня в современном стеке TCP, необходимо кратко обсудить, что происходит при прохождении данных по различным уровням стека. В этом разделе ради более легкого восприятия предложено упрощенное объяснение. Приложение может отправить в *буферы сокетов* (*socket buffers*) данные почти любого размера. Буферы сокетов расположены на транспортном уровне, в данном случае TCP. Все данные, отправленные приложением, будут скопированы в один из них. Если данных недостаточно, то приложение «засыпает». Далее модуль TCP сегментирует содержимое буфера сокета, причем размер сегмента не превышает значение MSS. Из буфера сокета данные изымаются лишь после того, как от удаленного узла пришло уведомление об их приеме. Далее эти сегменты «опускаются» на уровень IP. Если для базового физического канала размер сегментов слишком велик (больше чем значение MTU), то сегменты фрагментируются.

Параметр ядра, задающий размер приемного буфера конкретного протокола, представляет собой отметку верхнего уровня приема. В Solaris она задается параметрами `tcp_recv_hiwat` и `udp_recv_hiwat` или вызовом `setsockopt()` с параметром `SO_RCVBUF`. В версиях Solaris, предшествующих версии 8, параметры `tcp_recv_hiwat` и `udp_recv_hiwat` равны 8192,

а в Solaris 8 – 24 576. Для большинства приложений авторы рекомендуют диапазон от 16 до 32 Кбайт, однако в каждой конкретной среде необходимо экспериментировать. Также следует помнить, что этот параметр оказывает сильное влияние на потребление памяти. Если в системе присутствуют 512 одновременных соединений, а отметки приема и передачи равны 64 Кбайт, то только для буферов сокетов TCP необходимо 64 Мбайт памяти. Параметр `tcp_recv_hiwat_minmss` – это еще один параметр, относящийся к буферам сокетов. Он определяет количество сегментов максимального размера, которые должен вмещать приемный буфер. Размер приемного буфера невозможно установить меньше того предельного значения, при котором это количество сегментов еще вмещается в буфер. По умолчанию значение параметра `tcp_recv_hiwat_minmss` равно 4. Такое значение следует оставить.

Верхняя отметка уровня передачи, `tcp_xmit_hiwat`, задает размер буфера отправки.[1] Для каждого сокета ее можно установить с помощью параметра `SO_SNDBUF`. Когда объем данных в буфере отправки больше значения `tcp_xmit_lowat`, но меньше значения `tcp_xmit_hiwat`, то в буфер нельзя записывать данные. По умолчанию параметр `tcp_xmit_hiwat` равен 8192, а значение для него необходимо выбрать из диапазона от 32 до 48 Кбайт. Поскольку буфер передачи должен хранить все неподтвержденные сегменты, то разумно сделать его больше приемного буфера.

Несколько слов в качестве предостережения. Если значение верхней отметки уровня передачи будет близким к 65 535, то может появиться побочный эффект включения опции масштабирования окна TCP, поскольку это значение округляется до кратного MTU для каждого соединения. В некоторых случаях масштабирование окна не нужно. Для того чтобы предотвратить его непреднамеренное использование, значения параметров `tcp_recv_hiwat` и `tcp_xmit_hiwat` должны быть выбраны по крайней мере на 1 MTU меньше 65 535. Для интерфейсов Ethernet значение, равное 64 000, будет хорошим выбором.

Значения нижних отметок уровня, `tcp_xmit_lowat` и `udp_xmit_lowat`, изменять не следует. При необходимости их изменения приложения могут использовать `SO_SNDLOWAT` и `SO_RCVLOWAT`.

Параметры `tcp_max_buf` и `udp_max_buf` определяют максимальный размер буфера, который может быть запрошен вызовом `setsockopt()`. По умолчанию эти значения равны 1 048 576 и 262 144 соответственно. Попытка указать больший размер буфера не достигнет успеха – будет выдан код возврата `EINVAL`.

В версиях Linux, предшествующих версии 2.4, существуют эквивалентные параметры, расположенные в */proc/sys/net/core*. Это параметры `rmem_max`, `rmem_default`, `wmem_max` и `wmem_default`, значения которых

[1] В UDP фактически нет буферизации передачи.

можно изменять. По умолчанию все они равны 65 536. Начиная с версии 2.4 в Linux существует новый алгоритм автоматического регулирования размеров буферов сокетов. Этим алгоритмом неточно управляют параметры, имеющие отношение к TCP, а именно параметры `tcp_rmem` и `tcp_wmem`, расположенные в */proc/sys/net/ipv4*. Каждый из этих файлов содержит три значения размера, допустимые для каждого буфера: минимальный, по умолчанию и максимальный.

- Буфер минимального размера гарантирован каждому сокету TCP, даже когда в системе не хватает памяти. По умолчанию минимальный размер равен 8 Кбайт.
- Размер буфера по умолчанию подменяет значение `rmem_default` в */proc/sys/net/core/*, которое еще используется в протоколах, отличных от TCP. Значение по умолчанию равно 87 380 байт, что приводит к размеру окна TCP 65 535 байт.
- Максимальный размер приемных буферов не подменяет значение `rmem_max` в */proc/sys/net/core*. Однако такая подмена происходит при вызове `setsockopt()` с `SO_RCVBUF`. Значение по умолчанию равно 174 760 байт.

Окно TCP имеет прямое отношение к буферу сокета. Окно TCP – это объем данных, ожидающих отправки (данные, неподтвержденные адресатом), который может быть послан перед тем, как отправляющий модуль получит подтверждение от получателя. Например, если два хоста взаимодействуют через соединение TCP, в котором размер окна равен 64 Кбайт, то отправитель может послать только 64 Кбайт данных. Затем он должен остановиться и ждать подтверждения части или всех данных. Если получатель уведомляет, что все данные были получены, то отправитель может передавать следующие 64 Кбайт. Если получатель подтверждает только первые 32 Кбайт (например, если последние 32 Кбайт еще не пришли или были утеряны при передаче), то отправитель может послать только 32 Кбайт. Главное назначение окна – контроль переполнения. Само сквозное сетевое соединение, объединяющее оба хоста, все промежуточные маршрутизаторы, а также все вспомогательные соединения (будь то оптоволокно, медные провода, спутниковая передача и т. д.) могут оперировать данными только с некоторой максимальной скоростью. Если передача происходит слишком быстро, то в соединении появится узкое место, а часть данных будет потеряна. Окно TCP служит для «перекрывания» скорости передачи до того уровня, при котором не происходит переполнения и соответствующей потери данных. В заголовке TCP-пакета есть 16-битное поле, задающее размер окна. Поэтому максимальный размер окна составляет 2^{16}, или 65 536 байт.

Для того чтобы вычислить наивысшую скорость передачи большого объема данных, нужно размер окна, сообщенный узлом на другой стороне, разделить на время, требуемое для передачи и подтверждения приема (round-trip time). Оптимальное значение этого параметра на-

зывается *базой пропускания* (*bandwidth-delay product*). Другими словами, это произведение скорости соединения на время, необходимое для отправки пакета данных и получения уведомления о доставке.[1] Здесь есть интуитивный смысл: чем меньше период цикла «туда-обратно» или чем больше окно, тем быстрее передача данных. Пусть в соединении по каналу Ethernet 10 Мбит/с (~1,0 Мбайт/с) среднее время задержки при обращении к наиболее часто посещаемым хостам составляет 6 мс. Тогда значение базы пропускания равно 6 Кбайт, то есть значение по умолчанию, равное 8 Кбайт, будет достаточным. Однако если задержка превышает 8 мс, то увеличение размера буфера может быть полезным, особенно если речь идет о медленных каналах, таких как ISDN. Если задержка составляет 40 мс, а скорость передачи равна 12 Кбайт/с, то размер буфера должен быть 48 Кбайт! Таким образом, настройка TCP-окна на стороне клиента действительно необходима. Она будет особенно эффективной в нагруженных системах с длительным временем получения ответа. Вероятно, в системах, подключенных к высокоскоростным сетям, такая настройка также будет полезна.

Для того чтобы преодолеть ограничения производительности в каналах с очень высокой задержкой, налагаемые максимальным размером окна 64 Кбайт, в документе RFC 1323 был представлен метод, известный как *масштабирование окна* (*window scaling*). Благодаря масштабированию поле окна TCP расширяется до 32 бит, а для увязки 32-битного значения с первоначальным 16-битным полем используется масштабный коэффициент. Масштабный коэффициент переносится в новой опции TCP-пакета – масштабе окна (window scale). Эта опция посылается лишь в том случае, если в сегменте также отправляется бит `SYN`. При открытии соединения такая трехбайтная опция задает масштабирование окна в каждом направлении. Она выполняет две функции:

- Свидетельствует о том, что TCP готов к масштабированию окна при приеме и передаче.
- Определяет масштабный коэффициент, который должен быть применен к окну приема. Стек TCP, готовый к масштабированию, должен послать эту опцию, даже если его масштабный коэффициент равен 1. Масштабный коэффициент кодируется путем логарифмирования по основанию 2. Для того чтобы включить масштабирование окна в обоих направлениях, обе стороны должны послать опции масштаба окна в своих сегментах `SYN`.

В Solaris масштабированием окна управляет параметр `tcp_wscale_always`, который по умолчанию равен нулю. Если значение `tcp_wscale_always` не равно нулю, то опция масштаба окна всегда будет согласовываться во время инициации TCP-соединения. Иначе масштабирование окна будет применяться только в тех случаях, когда размер буфера

[1] Приближенное значение задержки можно получить с помощью *ping -s hostname*.

больше 64 Кбайт. Эквивалент в Linux – параметр `tcp_window_scaling` в */proc/sys/net/ipv4/*.

Повторные передачи

Иногда данные в сети теряются. Тому может быть много причин. Возможно, экскаватор повредил подземный сетевой кабель или работник, обслуживающий здание, случайно выдернул вилку маршрутизатора из розетки и т. д. Точные причины потери данных здесь не так важны. Вместо их рассмотрения обратимся к механизмам TCP для управления повторными передачами данных.

Основным параметром, применяемым при настройке повторной передачи, является параметр `tcp_rexmit_interval_initial`. Он определяет, сколько времени (в миллисекундах) должно пройти до того, как отправленные, но не подтвержденные данные будут переданы повторно. Начиная с Solaris 2.5.1 значение этого параметра по умолчанию равно 3000, что является разумным значением для большинства случаев. В лабораторных средах, не подключенных к Интернету, более предпочтительным может стать меньшее значение (скажем, 400 мс).

После того как была проведена первоначальная повторная передача данных, дальнейшие повторные передачи начнутся после истечения `tcp_rexmit_interval_min` миллисекунд. Начиная с Solairs 8 значение этого параметра по умолчанию равно 400 мс, а в предшествующих версиях – 200 мс. Вероятно, это значение должно быть примерно равно половине значения `tcp_rexmit_interval_initial`.

Дальнейшие повторные передачи происходят по алгоритму экспоненциального убывания, однако задержка между повторными передачами не превосходит `tcp_rexmit_interval_max` миллисекунд. Начиная с 2.6 значение этого параметра по умолчанию равно 240 000 (что соответствует RFC 1122). В Solaris 8 это время уменьшено до 60 000 мс.

Параметры `tcp_ip_abort_interval` и `tcp_ip_abort_cinterval` также относятся к повторным передачам. Они задают время в миллисекундах, в течение которого должны предприниматься попытки повторной передачи, – до того как будет послан сегмент `RST`, а соединение TCP будет разорвано. Параметр `tcp_ip_abort_cinterval` применяется для соединений, которые еще не достигли состояния `ESTABLISHED` (то есть они попрежнему находятся в процессе рукопожатия). Значения этих параметров по умолчанию равны 480 000 и 180 000 соответственно. Параметр `tcp_ip_abort_interval` следует увеличить примерно до 600 000.

Отложенные уведомления

В системах Solaris очень полезными являются параметры, управляющие отложенными уведомлениями. Эти связанные друг с другом параметры определяют порядок отсылки уведомлений, которыми получатель подтверждает передачу данных по локальной сети. Первый пара-

метр, tcp_deferred_ack_interval, измеряется в миллисекундах. Второй параметр, tcp_deferred_acks_max, задает максимальное количество сегментов, которые могут быть получены, прежде чем ACK будет немедленно отправлен.

Для понимания взаимосвязи этих параметров обратимся к примеру. Когда получен первый входящий пакет, происходят два события: запускается таймер отложеных уведомлений и увеличивается значение счетчика пакетов. Когда приходят следующие пакеты, могут возникнуть также два события. В первом случае значение счетчика пакетов становится равным tcp_deferred_acks_max. Тогда посылается уведомление, перезапускается таймер и обнуляется счетчик пакетов. Во втором случае срабатывает таймер отложенных уведомлений (когда его значение становится равным tcp_deferred_ack_interval). Тогда, даже если количество накопленных пакетов меньше tcp_deferred_acks_max, отправителю посылается уведомление. Далее отложенные уведомления будут *отключены*. Теперь уведомления будут посылаться на каждый второй пакет (именно так работает стек TCP, если два взаимодействующих хоста находятся на большом расстоянии друг от друга – не сравнимом с расстояниями в локальной сети).[1] В Solaris 8 значение параметра tcp_deferred_ack_interval по умолчанию равно 100 мс. В предшествующих версиях он был равен 50 мс. Параметр tcp_deferred_ack_max по умолчанию равен 8 пакетам. Такая схема соответствует документу RFC 1122, описывающему алгоритм медленного старта.

Окно переполнения и алгоритм медленного старта

При установлении соединения время задержки и доступная пропускная способность неизвестны. Для того чтобы избежать затора в медленной сети, в протоколе TCP реализован *алгоритм медленного старта* (*slow start algoritm*). Этот алгоритм ограничивает количество пакетов, которые могут быть посланы до получения уведомления. Такой предел называется *окном переполнения* (*congestion window*). Согласно стандарту, первоначальное окно переполнения устанавливается равным одному пакету, а далее его размер будет удваиваться при получении каждого уведомления.[2] В большинстве реализаций TCP есть два изъяна, относящиеся к алгоритму медленного старта:

- В большинстве реализаций TCP первоначальное значение окна переполнения по умолчанию равно двум пакетам. Поскольку такое значение встречается очень часто, то в версиях Solaris, предшествующих версии 7, значение параметра tcp_slow_start_initial следует

[1] Это сделано в целях избежания проблем в системах, работающих по алгоритму медленного старта.

[2] Такой алгоритм не слишком хорош в случае непродолжительных соединений, поскольку количество отправляемых пакетов недостаточно для того, чтобы размер окна значительно увеличился.

изменить с 1 на 2. За счет этого в незначительно нагруженных сетях скорость передачи увеличивается (слегка). В системах Solaris начиная с версии 7 это сделано по умолчанию.

- Сходные затруднения характерны для реализации TCP Microsoft. Здесь не происходит немедленной отправки уведомления о получении одного пакета. Вместо этого модуль TCP ожидает прихода двух пакетов либо истечения короткого периода задержки. После получения двух пакетов сразу же отсылается уведомление. Такая схема приводит к более высокому времени ответа по сравнению с высокоскоростными каналами. Увеличение времени отклика незаметно из-за задержки в глобальных сетях, а также в случае низкоскоростных каналов связи. Однако такое затруднение можно устранить, если изменить значение параметра `tcp_slow_start_initial` так, как описано выше. В Linux такую настройку провести нельзя.

Таймеры и интервалы TCP

В TCP есть три других интересных таймера. Это keepalive-таймер, таймер `TIME_WAIT` и таймер `FIN_WAIT_2`.

Keepalive-таймер – это одна из самых спорных тем в настройке стека TCP. Назначение системы keepalive – высвобождение ресурсов хоста за счет очистки соединений TCP, в которых удаленный хост не отвечает на запросы. Такое высвобождение ресурсов повышает эффективность использования сети. Наиболее часто keepalive-таймер применяется в веб-серверах. Напомним, что в большинстве транзакций TCP присутствует простой четырехшаговый процесс согласования:

1. Клиент HTTP открывает соединение с сервером HTTP.
2. Клиент посылает свой запрос (запрашивает файл).
3. Сервер отвечает выдачей файла (отклик на запрос).
4. Сервер закрывает соединение.

Если после того, как соединение установлено, в системе клиента или броузере происходит сбой, подвисание либо что-то иное, препятствующее отправке запроса, то сервер вынужден поддерживать открытое соединение неопределенное время. Если веб-сервер сильно нагружен, то продолжительность этого периода должна быть небольшой, от 5 до 10 минут. В Solaris этим значением управляет параметр `tcp_keepalive_interval`. В Linux для этого предназначены три параметра, находящиеся в */proc/sys/net/ipv4*. Параметр `tcp_keepalive_time` определяет, как часто TCP посылает keepalive-сообщения, если поддержка keepalive включена (по умолчанию каждые два часа). Параметр `tcp_keepalive_probes` задает количество пробных сегментов, которые пошлет модуль TCP, прежде чем он решит, что соединение разорвано (по умолчанию девять). Наконец, параметр `tcp_keepalive_interval` определяет, как часто отправляются пробные сегменты. Если значение этого параметра умножить на значение `tcp_keepalive_probes`, то можно получить

продолжительность существования соединения, в котором одна из сторон не отвечает на запросы. По истечении этого времени соединение будет разорвано. Значение параметра `tcp_keepalive_interval` по умолчанию равно 75 секундам.

Состояние TCP-соединения `TIME_WAIT` имеет негативную репутацию. Зачастую администраторы тщательно стараются его избегать. Однако на самом деле его наличие весьма полезно. Соединение TCP переходит в состояние `TIME_WAIT` после завершения связи, но до повторного использования сокета. *Максимальное время жизни сегмента* (*maximum segment lifetime*, MSL) – это максимальное время, в течение которого сегмент TCP может существовать в сети. В два раза больший период ожидания обеспечивает отсутствие сегментов, не дошедших до получателя, а также безопасность повторного использования ресурса сокета. Таким образом, становится ясно, почему состояние `TIME_WAIT` так необходимо. Время пребывания сокета в состоянии `TIME_WAIT` можно уменьшить. В качестве хорошего начального значения можно выбрать двойное значение времени передачи данных и получения подтверждения при соединении хостов, значительно удаленных друг от друга (round-trip time). Значение 60 000 или около того будет вполне разумным. В Solaris время пребывания сокета в состоянии `TIME_WAIT` можно настраивать напрямую с помощью параметра `tcp_time_wait_interval`, значение которого задается в миллисекундах (заметим, что в версиях Solaris, предшествующих версии 7, параметр `tcp_time_wait_interval` назывался `tcp_close_wait_interval`). В системах Linux этот параметр напрямую не настраивается, поскольку в Linux уже реализован быстрый алгоритм повторения `TIME_WAIT` (этим алгоритмом управляет параметр `tcp_tw_recycle`, значение которого не следует изменять). Однако в Linux есть параметр `tcp_max_tw_buckets`, определяющий максимальное количество сокетов, находящихся в состоянии `TIME_WAIT`, которые ядро может обслуживать одновременно. Если это количество превышено, то для того, чтобы опуститься ниже такого предела, значительная часть этих сокетов будет уничтожена. Кроме того, будет выдано предупреждение. Основное назначение такой схемы – предотвращение простых атак вида «отказ от обслуживания». Значение этого предела никогда не следует уменьшать.

Таймер `FIN_WAIT_2` применяется при закрытии соединения. Состояние `FIN_WAIT_2` означает, что соединение, по сути, закрыто, но `FIN` с другой стороны еще не пришел, а может и вообще не прийти. Длительное пребывание соединения TCP в состоянии `FIN_WAIT_2` может быть вызвано неполадками на стороне клиента. В таком состоянии соединение может пребывать до истечения времени, задаваемого таймером `FIN_WAIT_2`. Если вывод команды *netstat -f inet* (*netstat inet* в Linux) показывает, что в состоянии `FIN_WAIT_2` находится много соединений, то значение этого таймера следует уменьшить. В Solaris таймером `FIN_WAIT_2` управляет параметр `tcp_fin_wait_2_flush_interval`, а в Linux – параметр `tcp_fin_timeout`, находящийся в */proc/sys/net/ipv4*. В Solaris зна-

чение параметра tcp_fin_wait_2_flush_interval по умолчанию равно 675 000 мс. В Linux (ядро 2.4) значение параметра tcp_fin_timeout по умолчанию равно 60 с (180 с в системах с ядром 2.2). Вероятно, в системах Solaris значение параметра tcp_fin_wait_2_flush_interval следует снизить до 67 500 (на порядок).

Алгоритм Нагла

Алгоритм Нагла (*Nagl*) – это метод, позволяющий регулировать задержку отправки данных с помощью объединения маленьких пакетов. Известно, что при определенных обстоятельствах время ожидания перед отправкой данных может достигать 200 мс. Алгоритм Нагла, опубликованный в RFC 896, предназначен для решения простой задачи, относящейся к интерактивным соединениям, таким как *telnet*. В соединении *telnet* при каждом интерактивном нажатии клавиши обычно генерируется один пакет данных. Более того, удаленный сервер *telnet* отвечает за отображение знака на экране. Вследствие этого по сети передается до четырех сегментов (передача нажатия клавиши, передача отображенного знака и два пакета подтверждения). Эти маленькие пакеты, называемые *маленькими дейтаграммами* (*tinygrams*), обычно не вызывают затруднений в локальных сетях. Однако в глобальных сетях такие пакеты могут привести к серьезному переполнению. По существу, алгоритм Нагла предлагает накапливать маленькие порции данных и отправлять их в одном сегменте. Еще одним преимуществом этого алгоритма является автосинхронизация: чем быстрее возвращаются уведомления, тем быстрее посылаются данные.

Для того чтобы определить время отправки данных при отстутствии подтверждения о получении предыдущих пакетов, применяются следующие правила:

- Если нет неподтвержденных маленьких пакетов, то текущий пакет немедленно отправляется.
- Если окно TCP не заполнено, а размер пакета равен или больше размера максимального сегмента, то содержимое буфера немедленно отсылается (объем таких данных равен значению MTU).
- Если окно TCP не заполнено, а также либо бездействует интерфейс, либо установлен флаг TCP_NODELAY, то данные буфера немедленно отсылаются.
- Если объем данных, подготовленных к отправке, меньше половины окна TCP, то данные немедленно отсылаются.
- Иначе, прежде чем отсылать содержимое буфера, следует накапливать больший объем данных для отправки – до истечения 200 мс.

С приходом уведомления буфер Нагла немедленно очищается (заметим, что к этому времени были отправлены пакеты размером MTU и, возможно, один маленький пакет).

Если алгоритм Нагла плохо взаимодействует с таймером уведомлений (для надлежащей работы необходимо, чтобы за маленьким пакетом следовал полный пакет), то посылать большие пакеты данных не удастся. Это приводит к серьезному снижению производительности.

В большинстве случаев (но, конечно, не во всех) алгоритм Нагла следует отключить. Для этого параметру `tcp_naglim_def` нужно присвоить 1. Если в приложениях этот алгоритм необходимо отключить для конкретного сокета, то при создании сокета следует использовать флаг `NODELAY`.

Настройка параметров TCP может ненароком натворить бед в сети. Пожалуйста, будьте внимательны.

UDP

Протокол передачи дейтаграмм пользователя (*User Datagram Protocol*), или UDP, – это «самый эффективный» протокол без установления прямого соединения. В свете высокой надежности TCP трудно вообразить, что кто-то захочет применять UDP. Однако преимущество UDP состоит в следующем: вместо того чтобы устанавливать и разрывать соединение, необходимое для TCP, данные просто посылаются по сети. Благодаря этому задержки при передаче данных очень малы, а значит, приложения, посылающие данные небольшими пакетами по надежным сетям, ощутят преимущество в быстродействии.

Еще одна причина более высокой производительности UDP – маленький заголовок пакета (только 64 бит). См. табл. 7.16.

Таблица 7.16. Заголовок протокола передачи дейтаграмм пользователя

Биты	Содержимое
0–15	Порт источника
16–31	Порт пункта назначения
32–47	Длина UDP
58–64	Контрольная сумма

Порты источника и пункта назначения идентичны полям заголовка TCP. Длина UDP – это длина всего сегмента в байтах, а контрольная сумма – это необязательное поле, содержащее контрольную сумму для всего сегмента.

Поскольку UDP является намного более простым протоколом, то в нем нет такого количества параметров, как в TCP. Самые важные параметры UDP – это размеры буферов передачи и приема (они были обсуждены наряду с их TCP-аналогами; см. раздел «Буферы, отметки уровня и окна» ранее в этой главе).

Сетевой транспорт: TCP или UDP

Если приложение (например, NFS) предоставляет возможность выбора между TCP и UDP, то возникает вопрос, какой же сетевой транспорт использовать. По существу, во внимание следует принять три соображения:

- Теоретически UDP дает более высокую производительность.[1]
- TCP имеет большие издержки, однако, в отличие от UDP, гарантирует надежность.
- В средах, подверженных ошибкам, например в глобальных сетях, способность модулей TCP управлять издержками при передаче данных очень полезна.

По существу, в быстрых, хорошо обслуживаемых, коммутируемых локальных сетях протокол UDP может дать значительное увеличение производительности (например, в машинном зале). Однако производительность сети в большой степени зависит от конкретной реализации. Поэтому для принятия решения необходимо тестирование.

NFS

Сетевая файловая система (Network Filesystem, NFS), первоначально разработанная в Sun Microsystms, стала неотъемлемой частью компьютерного ландшафта – в значительной степени благодаря своей простоте и доступности на большинстве платформ. Существуют две разновидности NFS: версия 2, применяемая с 1985 года, и версия 3, представленная в 1993 году. В большинстве современных реализаций NFS поддерживаются обе версии, причем более новой версии отдается предпочтение при условии, если сервер и клиент могут ее использовать. Наиболее важная концепция NFS состоит в том, что NFS основана на протоколе без поддержки состояния. Схема работы такова, что серверу и клиенту не нужно поддерживать непрерывное соединение. Клиент просто выдает запрос, а сервер его обрабатывает и отвечает. В период между запросом и ответом в сервере может произойти сбой либо он может быть выключен, однако это не потревожит ни одну из сторон.

NFS работает совместно с другими службами. К ним относятся диспетчер блокировки файлов, механизм для запуска службы (протокол монтирования) и автомонтировщик, перехватывающий обращения к файловым системам. Для протокола разделения файлов без поддержки состояния блокировка файлов представляет собой интересную задачу, поскольку она предполагает наличие состояния. Поэтому блокировка

[1] На самом деле большинство производителей приложили немало усилий в настройке своих реализаций TCP, поэтому в каждой операционной среде следует проводить эксперименты, которые помогут оценить действительное различие.

осуществляется с помощью протокола диспетчера сетевой блокировки (Network Lock Manager, NLM), который, в свою очередь, основан на протоколе, реализованном в `rpc.statd`. Сервер ведет список всех блокированных файлов. Кроме того, он отвечает за выдачу информации о блокировке своим клиентам. Поскольку в NFS состояние не поддерживается, то серверу и клиенту нужен механизм для определения состояния друг друга – для того чтобы знать, когда нужно вновь затребовать блокировку (например, при перезагрузке сервера), а когда ее следует отменить (например, когда клиент выполняет демонтирование файловой системы). Этим занимается *statd*.

Сам по себе протокол монтирования не особенно интересен, за исключением того, что согласование всех параметров происходит во время монтирования. Другими словами, размер блока, управление локальными свойствами, версия используемого протокола и транспортный протокол определяются при монтировании удаленной файловой системы.

Автомонтировщик (*automounter*) – это приложение на стороне клиента, осуществляющее монтирование файловых систем. Удаленным файловым системам назначается местоположение внутри файловой системы на стороне клиента, а все обращения к этим каталогам перехватываются автомонтировщиком. Если удаленная система не монтирована, то автомонтировщик находит удаленную систему и монтирует ее в соответствующее местоположение. Если к монтированной файловой системе долгое время не происходит обращение, то автомонтировщик осуществляет ее демонтирование. Совместно с универсальными службами имен, например NIS и NIS+, автомонтировщик помогает поддерживать согласованность файловой системы на уровне предприятия. Автомонтировщик не накладывает существенной нагрузки на компоненты системы.

Протокол версии 2 чрезвычайно прост: в нем реализованы только 18 операций. В 1993 году, в ответ на давление потребителей, требовавших новых технических возможностей, протокол NFS подвергся переработке. Результатом явилась версия 3, в которой есть улучшения в некоторых ключевых областях:

- Операции записи стали намного быстрее. Используется двухэтапный протокол фиксации.
- Количество пакетов, передаваемых по сети, снижено благодаря возможности возврата файловых атрибутов при каждой операции. Каждая операция, в которой изменяется состояние файла, возвращает модифицированные атрибуты.
- Максимальный размер файла увеличен с 2^{32} до 2^{64} байт. Максимальный размер блока данных увеличен с 8 Кбайт до 4 Гбайт.[1]

[1] Максимальный размер блока данных обычно определяется базовыми атрибутами сети. Например, Solaris поддерживает размер блока до 64 Кбайт. Это самый большой блок, допустимый для реализаций TCP и UDP.

- Значительно улучшено управление доступом к файлам (например, поддержка списков управления доступом).

Несмотря на то что версии NFS достаточно похожи, системы, поддерживающие разные версии, не могут взаимодействовать друг с другом. Во время монтирования файловой системы клиент и сервер согласовывают, какой именно протокол будет использоваться. Заметим, что не все клиенты по умолчанию способны работать с версией 3, поэтому на некоторых платформах могут потребоваться дополнительные настройки.

Сервер сообщает размер экспортируемой файловой системы только для того, чтобы позволить клиенту оценить размер свободного пространства. Эта величина не имеет отношения к обработке файлов. Несмотря на то что сообщаемый размер свободного пространства на стороне клиента может быть очень странным (отрицательным), тем не менее доступ к файлам будет работать должным образом. Однако в серверах версии 2 есть сложности с большими файлами (более 2 Гбайт). Это объясняется ограничениями адресуемости, присущими NFS версии 2.

Версию протокола NFS можно задать с помощью ключа *vers=n* команды *mount*, где *n* равно либо 2, либо 3. Кроме того, версии 2 и 3 могут работать как по TCP, так и по UDP. Протокол задается с помощью ключа *proto=tcp* или *proto=udp* команды *mount* соответственно. По умолчанию версия 2 монтируется по UDP, а версия 3 – по TCP.

Описание операций NFS

В NFS версии 2 операции представлены шестью функциями: `lookup`, `getattr`, `setattr`, `readlink`, `read` и `write`. Эти функции можно разбить на две категории. Первые четыре манипулируют с файловыми атрибутами (поиск имени файла, получение и изменение атрибутов и разрешение символических ссылок). В последних двух операциях происходит чтение и запись содержимого файла. Все эти операции оказывают весьма разную нагрузку на сервер.

Операции с атрибутами легковесны. Вследствие своего маленького размера большинство атрибутов файловой системы кэшируется в памяти, а если даже и нет, то они легко извлекаются с диска. Несмотря на то что издержки обработки таких запросов скорее велики (так как доля полезных данных в передаваемой информации мала), общий маленький объем пакетов не приводит к сильной загрузке сети.

Другое дело – операции с данными. В NFS версии 2 максимальный размер данных почти всегда равен 8 Кбайт, а в версии 3 размер может быть намного больше. Кроме того, хотя каждый файл имеет только один набор атрибутов, он может содержать значительное количество блоков данных, большинство из которых обычно не кэшируется сервером. Вследствие большего размера такие операции занимают намного бульшую часть полосы пропускания.

Основную часть времени серверы NFS обычно тратят на обслуживание атрибутов, а не на операции с данными. Когда клиентской системе необходимо задействовать файл на удаленном сервере, то для его нахождения она применяет серию операций `lookup`. Далее для получения маски прав файла выполняется операция `getattr`, а за ней – операция `read` для получения первого блока данных. Для примера рассмотрим чтение пользовательского файла *.forward*. В системах с большим количеством пользователей каталоги обычно отсортированы по алфавиту, поэтому ради иллюстрации предположим, что файл находится в */shared/home/u/us/user/.forward*. Для того чтобы прочитать файл, у клиента должен быть доступ ко всем четырем каталогам, ведущим к нему. Для того чтобы убедиться в этом, необходимо найти записи в каждом каталоге и получить маски доступа. С помощью этих масок можно определить, достаточны ли права доступа. Таким образом, необходимо выполнить четыре операции `lookup`.[1] Далее должен быть найден сам файл, проверены права доступа и прочитан один блок. Таким образом, из шести операций лишь одна связана с данными!

По причинам, описанным ранее, сценарий доступа ко множеству маленьких файлов насыщен операциями с атрибутами. Классический пример – среда разработки программного обеспечения, в которой присутствуют много маленьких файлов, будь то файлы с исходным кодом, файлы заголовков, объектные файлы или файлы управления версиями. Кроме того, такая модель характерна для большинства серверов с домашними каталогами. Поскольку транзакции NFS блокируются на стороне клиента, то производительность системы в основном определяется скоростью, с которой сервер может обработать запросы атрибутов. В коммутируемых средах, для которых характерна такая рабочая нагрузка, сеть Ethernet 10 Мбит/с будет приемлемым выбором.

В средах с интенсивной обработкой данных, где средний размер файла очень велик (например, при обработке изображений), издержки при передаче данных превосходят время обработки атрибутов. Когда в сети с повторителями присутствуют много активных клиентов, а потребление ресурсов сети достигает 40%, то пользователи ощущают невысокую производительность. Коммутируемые сети от такого явления защищены, хотя канал от коммутатора до сервера может переполняться. К счастью, трафик NFS не является постоянным, а характеризуется всплесками активности. Несмотря на то что клиенты могут выдавать огромное количество запросов к серверам и сетям, это происходит лишь эпизодически, а в оставшееся время потребление сети несущественно.

Настройка клиентов

Важно осознавать, что в системах UNIX управление удаленной монтированной файловой системой осуществляется точно так же, как и ло-

[1] При операции `lookup` атрибуты каталогов возвращаются автоматически.

кальной дисковой подсистемой. Это означает, что подсистема виртуальной памяти делится между локальными приложениями и клиентским программным обеспечением NFS. Файлы монтированной системы кэшируются в памяти точно так же, как и локальные файлы. Такой механизм кэширования приводит к задержкам в работе NFS, а иногда к ее приостанову.

Одним из способов минимизации сетевой активности является применение системы CacheFS, которая подробно описана в разделе «Кэширующие файловые системы (CacheFS)» главы 5. Одно из преимуществ CacheFS состоит в том, что она полностью работает на стороне клиента. Сервер не осведомлен о клиентском внутреннем кэшировании. Однако это не «волшебная пуля». Поскольку CacheFS копирует блоки данных, она должна периодически сканировать кэшированные файлы. Каждый кэшированный файл, который соответствует файлу, измененному на сервере, должен быть удален, а измененный файл должен быть повторно загружен при следующем обращении к серверу. Поскольку большинство программ работает с целыми файлами (через `read(2)` и `write(2)`), а не с отдельными блоками данных (например, через `mmap(2)`), то в CacheFS кэширование целых файлов обычно прекращается. Поэтому если предполагается, что файловые системы будут часто модифицироваться, то применять CacheFS не стоит. Файлы будут непрерывно кэшироваться, а затем удаляться, поэтому сетевой трафик будет больше, чем в случае простой конфигурации NFS. Один замечательный пример – каталог */var/mail*. Почтовые спул-файлы больших размеров будут очищаться и перечитываться с приходом каждого нового сообщения. Если новая почта приходит часто, то такая работа будет медленной.

Производительность файловой системы CacheFS можно оценить с помощью команд *cachefsstat(1m)*, *cachefswssize(1m)* и *cachefslog(1m)*. Общее правило таково: коэффициент эффективности поиска для заданной файловой системы должен быть выше 35%. Меньший коэффициент может означать следующее: либо слишком мал кэш, что можно проверить, если сравнить размер рабочего комплекта файловой системы (working set), который сообщается командой *cachefswssize*, с размером кэша, либо в кэшированной файловой системе преобладает произвольный тип доступа. Если значение коэффициента ошибок соответствия превышает 20%, то это ясно свидетельствует о том, что файловая система обновляется слишком быстро, чтобы CacheFS приносила реальную пользу.

В средах NFS версии 3, для которых характерна интенсивная обработка данных, клиенты Solaris могут присвоить параметру ядра `nfs:nfsv3_nra` значение 6. Тогда клиент будет запрашивать, чтобы сервер читал шесть блоков данных вперед. Такое значение показало хорошие результаты при тестировании.

Получение статистики для монтированных файловых систем NFS

Одним из инструментов получения статистики монтирования NFS является команда *iostat* (команда *iostat* описана в разделе «Применение iostat» главы 5).

Другое мощное средство – это команда *nfsstat -c*, которая сообщает статистику NFS на стороне клиента. Вот пример, полученный на незначительно нагруженной рабочей станции, в которой */home* смонтирован как NFS:

```
# nfsstat -c
...
Client rpc:
Connection oriented:
calls       badcalls    badxids     timeouts    newcreds    badverfs
164250      0           0           0           0           0
timers      cantconn    nomem       interrupts
0           0           0           0
...
Client nfs:
calls       badcalls    clgets      cltoomany
159786      0           159786      0
Version 3: (158872 calls)
null        getattr     setattr     lookup      access      readlink
0 0%        58283 36%   1863 1%     10003 6%    19134 12%   47 0%
read        write       create      mkdir       symlink     mknod
39926 25%   23395 14%   2209 1%     2 0%        12 0%       0 0%
remove      rmdir       rename      link        readdir     readdirplus
2276 1%     2 0%        487 0%      93 0%       480 0%      461 0%
fsstat      fsinfo      pathconf    commit
66 0%       1 0%        27 0%       105 0%
...
Version 3: (914 calls)
null        getacl      setacl
0 0%        914 100%    0 0%
```

Заметим, что не относящиеся к делу строки усечены. Вот на что следует обратить внимание:

- Если значение поля `badxids` примерно равно значению поля `timeouts`, и оба этих значения больше 5% от количества вызовов, то следует искать узкие места в сервере NFS.
- Если значение поля `badxids` относительно мало, а значение поля `timeouts` больше, чем 5% от количества вызовов, то следует искать неполадки в сети, которые приводят к пропуску пакетов.

Поле `badxids` извещает, сколько раз от сервера был получен ответ, который не соответствовал любому ожидающему отклика запросу. Поле `timeouts` уведомляет, сколько раз вызовы NFS находились в тайм-ауте, ожидая ответа от сервера.

Если монтирование осуществляется по UDP, то *nfsstat -m* выдаст статистику среднего времени выполнения определенных команд:

```
% nfsstat -m
/home from cassandra:/workspaces
 Flags: vers=3,proto=tcp,sec=sys,hard,intr,link,symlink,acl,rsize=32768,
wsize=32768,retrans=5

/mnt from london:/services/patches
 Flags: vers=3,proto=udp,sec=sys,hard,intr,link,symlink,acl,rsize=32768,
wsize=32768,retrans=5
 Lookups:     srtt=7 (17ms), dev=3 (15ms), cur=2 (40ms)
 Reads:     srtt=18 (45ms), dev=11 (55ms), cur=7 (140ms)
```

Видно, что первая точка монтирования (*/home*) монтирована как NFS версии 3 по TCP, а вторая (*/mnt*) монтирована как NFS версии 3 по UDP. Поле `srtt` (сглаженный период времени на передачу данных и подтверждение приема, smoothed round-trip time) является ключевым. Если его значение больше 60 мс или около того, то работа устройства будет «медленной». Поле `dev` представляет ожидаемое отклонение значения в поле `srtt`, а поле `cur` описывает текущий уровень тайм-аута для повторных передач.

В Solaris получить сведения о файловых системах, монтированных как NFS, можно с помощью *iostat -xn*, как описано в разделе «Применение iostat» главы 5:

```
# iostat -xn 30
...
                            extended device statistics
  r/s  w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
  8.2 20.4   68.2 1042.1  0.0  1.3    0.0   46.7   0  47 c0t0d0
  0.0  0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 cassandra:/workspaces
  2.8  0.0   78.1    0.0  0.2  0.3   56.9   89.9   1   8 london:/services/
patches
...
```

Это иллюстрация того, что при настройке производительности есть более одного способа сделать что-либо.

Кэш удаленных дескрипторов

Кэш удаленных дескрипторов соответствует кэшу индексных дескрипторов UFS, только кэширует информацию о файлах, доступ к которым идет через NFS. Эти данные являются результатом запросов `getatrr` к серверу NFS, который хранит такую информацию в своем виртуальном индексном дескрипторе. По умолчанию размер кэша удаленных дескрипторов в два раза больше размера DNLC. Размер этого кэша не следует настраивать.

Настройка клиентов NFS для пульсирующих передач

В основном этот раздел относится к работе приложений, которые переключаются между двумя режимами активности. Отличие этих двух режимов состоит в том, что в одном из них происходит большой объем ввода-вывода. Именно такая модель характерна для большинства высокопроизводительных вычислительных приложений.

Все операции NFS с отдельным файлом, даже состоящие из одного процесса, разделяются ядром на потоки, количество которых равно параметру `nfs3_max_threads` (по умолчанию 8). Перед блокированием процесса клиент кэширует до 64 Кбайт записанных данных на каждый поток. Поэтому увеличение количества NFS3-потоков ядра может снизить или устранить то время, которое приложение затрачивает на сетевой ввод-вывод, – вместо того чтобы выполнять вычисления. Значение параметра `nfs3_max_threads` должно быть выбрано из расчета 6–8 потоков на каждый процессор. Благодаря такой схеме возможно параллельное выполнение передачи данных через буферы NFS-потоков ядра и обработки данных в приложении. Однако такая схема не увеличит максимальное значение установившейся производительности NFS.

Для того чтобы эта схема работала, приложение должно хранить открытый файловый дескриптор. Если файловый дескриптор закрыт, то необходимо монтировать удаленную файловую систему с помощью недокументированного ключа *-o nocto*. По умолчанию при закрытии файла клиенты NFS ожидают завершения своих дисковых операций записи. За счет этого обеспечивается согласованность всех систем, которые могут читать этот файл. Однако при использовании ключа *nocto* такая согласованность не гарантируется. Это может увеличить производительность, однако побочным эффектом будет то, что изменения в этом файле будут на некоторое (непредсказуемое) время скрыты от других клентов.

Настройка клиентов NFS для последовательных передач

Параметр ядра `nfs3_nra` определяет количество блоков размером 32 Кбайт, которые клиент NFS пытается «заранее» прочесть в заданном файле. Это может существенно увеличить производительность операций чтения в NFS, поскольку издержки вызова RPC «распределяются» по данным большего объема. Верхняя граница для `nfs3_nra` – это количество NFS3-потоков ядра (см. раздел «Настройка клиентов NFS для пульсирующих передач» ранее в этой главе).

Еще один полезный параметр ядра – это параметр `clnt_max_conns`. Все запросы NFS заданного клиента, использующие TCP, будут разделять пул соединений. Таким пулом управляет параметр `clnt_max_conns`, значение по умолчанию которого равно 1. Побочным эффектом является распределение обработки среди нескольких процессоров. Вероятно, на быстрых многопроцессорных машинах, соединенных через Gigabit Ethernet, настройка параметра `clnt_max_conns` принесет увеличение производительности.

Настройка серверов

По существу, настройка серверов NFS достаточно сложна. Есть много вопросов, на которые надо ответить:

- Характеризуется ли рабочая нагрузка интенсивной обработкой данных или атрибутов?
- Могут ли клиенты кэшировать бульшую часть NFS-монтированных данных?
- Сколько активных клиентов необходимо поддерживать?
- Насколько велики разделяемые файловые системы?
- Обращаются ли запросы NFS повторно к одним и тем же файлам или преобладает произвольный тип доступа?
- Что представляет собой базовая сетевая конфигурация?
- Достаточно ли быстра базовая дисковая подсистема?
- Распределяется ли рабочая нагрузка соразмерно среди доступных дисков?

При настройке сервера NFS очень важно обеспечить достаточную полосу пропускания для клиента. А определить, какая ширина полосы достаточна, можно с помощью анализа операций, которые доминируют в рабочей нагрузке. Системы NFS с интенсивной обработкой атрибутов легко обслуживаются на недорогих сетях,[1] однако для систем NFS с интенсивной обработкой данных необходима инфраструктура с высокой полосой пропускания. Несмотря на то что в средах с интенсивной обработкой атрибутов коммутируемые сети Ethernet не подвержены переполнению, такие сети не выручают в случае NFS с интенсивной обработкой данных, поскольку их пропускная способность недостаточна. Дуплексные операции здесь менее важны, так как операции NFS не сбалансированы (соотношение операций чтения и записи не приближается к 1:1). В большинстве реализаций NFS дуплексный канал Ethernet неотличим от полудуплексного.

В средах с интенсивной обработкой данных важно применять NFS версии 3 просто потому, что в NFS версии 2 максимальный размер блока равен 8 Кбайт. При таком размере блока и соединении 100 Мбит/с пропускная способность сети снизится до 4 Мбайт/с. В случае NFS версии 2 преимущество высокоскоростных сетей заключается в том, что множественные полноскоростные согласования могут происходить без снижения сетевой производительности. Что касается NFS версии 3, то вследствие более крупного размера блока данных возможная пропускная способность сети существенно увеличивается, приближаясь к пределам базовой сетевой инфраструктуры. Это означает, что стратегия

[1] Безусловно, оборудование Ethernet 100 Мбит/с полностью приемлемо для такого приложения. Однако основное преимущество состоит в том, что в сети может быть сконфигурировано больше клиентов без возникновения перегрузки.

организации сети должна быть существенно иной – необходимо задействовать выделенный канал Ethernet 100 Мбит/с или очень разреженную сеть ATM.

Кроме того, важно учитывать, сколько клиентов могут быть активны одновременно. Непрерывные запросы NFS характерны для очень малого количества приложений. Таблица 7.17 может быть полезна для примерной оценки количества активных клиентов, поддержку которых можно обеспечить.

Таблица 7.17. Оценка количества поддерживаемых клиентов

Сетевой носитель	Клиенты с интенсивной обработкой атрибутов/сеть	Клиенты с интенсивной обработкой данных/сеть
10BASE-T	15–20	Не лучший выбор
100BASE-T (с повторителями)	150–200	10–15
100BASE-T (коммутируемая)	200–280	15–18
ATM/FDDI	200–300	15–20

Обычно производительность процессора не является ограничителем для производительности сервера NFS. В основном процессор задействован в обработке самих сетевых пакетов и выполнении работы, требуемой протоколом (обычно именно в таком порядке), а также в управлении интерфейсами дисковых и сетевых устройств. Несмотря на то что системы на платформе IA-32 обычно очень привлекательны с точки зрения цены–производительности, зачастую их работу стесняют относительно медленные подсистемы памяти. По существу, любая быстрая, современная однопроцессорная или многопроцессорная система среднего класса должна справляться с обслуживанием большого объема сетевого трафика, возникающего при интенсивной обработке атрибутов. В случае интенсивного трафика данных нужно придерживаться следующего правила: один процессор UltraSPARC-Iii 440 МГц может обслуживать около 170 Мбит/с. В Solaris или Linux, использующих встроенный в ядро демон NFS, масштабируемость находится в пределах 75–90%, то есть второй процессор увеличит производительность на 75–90%. Если говорить о масштабируемости SMP, то это почти наихудший случай, поскольку операции NFS *полностью* обслуживаются в пространстве ядра, где блокировка оказывает сильное влияние на работу системы. Кэши второго уровня увеличивают производительность примерно на 10%.

Производительность дисковой подсистемы обычно очень важна для производительности NFS. Пожалуйста, помните, что обсужденного уровня производительности можно достичь лишь в том случае, когда обслуживаемая файловая система располагается на дисковой подсистеме, способной работать на таких скоростях.

Проектирование дисковых подсистем для серверов NFS

Для NFS с интенсивной обработкой атрибутов почти всегда характерен произвольный тип доступа. В результате, дисковые головки считывания/записи тратят значительно больше времени на поиск, чем на действительное извлечение данных. Когда в рабочей нагрузке преобладают операции с произвольным доступом, пропускная способность диска относительно мала. Следовательно, можно поместить много дисков на одну шину SCSI. Цель конфигурирования – максимальное количество шпинделей, поскольку диски являются узким местом, если говорить о производительности. Одно хорошее правило гласит: для того чтобы задержка была минимальной, на каждые три клиента необходимо компоновать один диск 7 200 rpm. В средах с интенсивной обработкой атрибутов, где происходит большой объем операций записи, память NVRAM, задействованная в дисковом массиве, может существенно снизить коэффициент использования диска (utilization).

В средах с интенсивной обработкой данных все намного проще. Основной принцип таков: если сеть не является узким местом, то один активный клиент NFS версии 2 потребляет 5,5 Мбайт/с, а клиент версии 3 – 11 Мбайт/с. Это означает, что дисковую подсистему необходимо компоновать как блок (stripe) или другой массив RAID, поскольку даже самый быстрый диск вряд ли сможет передавать данные со скоростью 14 Мбайт/с, а любые параллельные обращения к диску могут существенно снизить пропускную способность.

Для сервера NFS применение файловых систем с протоколированием может быть очень полезным (см. раздел «Файловые системы с протоколированием» главы 5), поскольку такие файловые системы предлагают надежный и быстрый способ выполнять обновления и ускоряют проверку целостности данных при загрузке. Однако в большинстве случаев протоколирование «ущемляет» производительность: сначала операция должна быть отображена в протоколе, далее применена к файловой системе, а затем вычищена из протокола. Когда в системе преобладают небольшие операции записи (до 16 Кбайт), то операция записи в файловой системе с протоколированием выполняется примерно так же быстро, как в файловой системе без него. Если одновременно проводится много операций записи, то протоколирование потенциально будет осуществляться быстрее, поскольку расстояние поиска в протоколе намного меньше расстояния поиска в реальной файловой системе. Однако в случае больших операций записи протоколирование может привести к снижению производительности (до 20%).

Кэширование в модулях NVRAM

В средах NFS версии 2 ускорение записи, осуществляемое памятью NVRAM, значительно увеличивает производительность операций записи, поскольку в версии 2 все операции записи синхронные. Перед тем как такие операции смогут выдать код возврата, их результаты

должны быть зафиксированы в энергонезависимой памяти.[1] В случае типичного файла это приводит по меньшей мере к 3 операциям дисковой записи: одна для обновления реальных данных, одна для обновления информации в каталоге файла и одна для косвенного блока (и, вполне возможно, для двойного косвенного блока). Это означает, что операция записи в NFS версии 2 может занять 120 мс (3 синхронных записи по 20 мс каждая) или в 6 раз больше времени, чем обычные 20 мс, или около того в случае записи на локальный диск. Если фиксировать записи в NVRAM, а не на диске, то производительность операций записи сильно возрастает. Увеличение производительности может составить 100%, поэтому конфигурирование NVRAM почти всегда будет кстати. Единственное исключение – серверы NFS «только для чтения», поскольку из-за издержек управления кэшем NVRAM максимальная пропускная способность системы снижается примерно на 5%. Такой компромисс почти всегда хорош для улучшения обработки запросов, поскольку в большинстве систем максимальная пропускная способность намного больше типичной нагрузки. Это классический компромисс между производительностью и задержкой.

Одной из целей при разработке NFS версии 3 было устранение синхронных операций записи. Хотя в этом случае ускорение NVRAM по-прежнему приносит пользу, увеличение производительности намного меньше (порядка 5%). Поэтому в серверах, предлагающих *только* NFS версии 3, для компоновки кэшей NVRAM нет оснований.

В случае массива RAID 5 проводить кэширование в энергонезависимой памяти по-прежнему очень важно, поскольку дисковый массив не имеет соединения с файловой системой. Это означает, что кэширование может дать больше, чем просто ускорение операций синхронной записи (см. раздел «RAID 5: разбивка на блоки с участками четности, распределенными по ширине блока» в главе 6).

Требования к памяти

Инстинктивное желание многих системных администраторов – компоновать в серверах NFS очень большой объем оперативной памяти. Предполагается, что в этом случае выгода от кэширования, реализованного в подсистеме виртуальной памяти, будет максимальной. Однако, как бы это ни противоречило интуиции, в большинстве случаев это бессмысленно по двум причинам. Первая – размер экспортируемой файловой системы намного больше размера оперативной памяти. Вторая – большинство клиентов обычно не разделяют основную часть своего рабочего дискового комплекта (working set) с другими клиентами.

[1] Заметим: это отнюдь не то же самое, что и «запись на диск». Метод определяется разработчиками протокола.

Однако есть одно исключение – наличие временных файлов, которые создаются, когда клиентским системам не хватает памяти.

Для выбора размера оперативной памяти при компоновке сервера NFS существует простое правило: памяти должно быть достаточно для кэширования данных, к которым за каждые пять минут возможно обращение более одного раза.[1] *Приблизительная* оценка – 128 Мбайт оперативной памяти на каждый микропроцессор класса UltraSPARC. Следует иметь в виду, что выигрыш от большей оперативной памяти в средах с интенсивной обработкой атрибутов значительнее, чем в средах с интенсивной обработкой данных. Если в сервере NFS пространство временных файлов используется очень активно, то размер оперативной памяти должен составлять около 75% размера активных временных файлов.[2] Для серверов приложений размер оперативной памяти должен примерно соответствовать размеру всех активно используемых бинарных файлов и библиотек. Такое применение не типично для NFS, и, по существу, кэширование активных данных будет весьма кстати. Поскольку служба NFS обычно полностью работает в пространстве ядра, то системе, по сути, не требуется пространство свопинга, за исключением случая, когда нужно сохранить аварийный дамп при возникновении неполадок в системе. Фактически сервер NFS можно конфигурировать вообще без пространства свопинга.

Компоновка большого объема памяти в серверах NFS обычно является пустой тратой времени.

Два основных типа серверов NFS

В целом, существуют два класса серверов NFS. Часть из них работают как пользовательские процессы, подобно *sendmail*. Такие серверы называются *серверами NFS пользовательского пространства* (*user-space NFS servers*). Однако поскольку улучшение производительности, достигаемое при наличии сервера NFS, тесно связано с операционной системой, то большинство современных серверов NFS полностью работают внутри ядра, подобно *fsflush* или *bdflush* (см. разделы «Solaris: fsflush» или «Linux: bdflush» в главе 4). Такие серверы называются *серверами NFS внутри ядра* (*in-kernel NFS servers*). Если производительность NFS важна, то в первую очередь необходимо выбрать именно сервер NFS внутри ядра. Стандартный сервер NFS Solaris, а также сервер *knfsd* в Linux как раз такого типа.

---

1 На самом деле «правило пяти минут» возникло при вычислении необходимого объема оперативной памяти для серверов баз данных. Оно основано на оценке показателя стоимость/время в случае кэширования данных.

2 Однако следует помнить, что размещение таких временных файлов в локальной файловой системе клиента даст более значительный прирост производительности, а также снизит сетевой трафик.

Побочный эффект запуска в пространстве ядра состоит в том, что все операции NFS обслуживаются с бульшим приоритетом, чем пользовательские процессы. Поэтому на загруженном сервере NFS не следует запускать другие приложения, так как остальные процессы будут работать медленно.

Другой побочный эффект таков: в выводе, полученном при запуске утилит мониторинга процессора, время процессора, затраченное на обслуживание NFS, сообщается в поле `sys` (см. раздел «Разбивка времени процессора» в главе 3). Вообще, если системное время постоянно больше 50% всего времени работы процессора, то мощность процессорного блока на сервере NFS следует увеличить.

Настройка количества потоков NFS

Каждый поток NFS может за один раз обслуживать один запрос NFS. Отсюда следует, что большой пул потоков позволит серверу обрабатывать больше одновременных запросов NFS. Значение по умолчанию, равное 16, почти всегда слишком мало. Однако добавление потоков NFS не представляет сложности. Для того чтобы найти подходящее количество потоков, следует применить следующие правила и выбрать наибольшее значение:

- Для каждого активного клиента, обращающегося к ресурсам NFS, следует задействовать по меньшей мере 2 потока.
- На один процессор следует предусматривать от 16 до 384 потоков NFS. Медленные системы, подобные SPARCstation 5, должны применять 16 потоков, а системы с процессорами 450 МГц Ultra-SPARC-II – 384.
- Для каждых 10 Мбайт/с сетевого трафика необходимо 16 потоков NFS.

Этот параметр можно регулировать с помощью изменения строки в файле */etc/init.d/nfs.server*, которая отвечает за запуск *nfsd*. Например, для того чтобы были запущены 128 потоков, строка должна выглядеть так:

```
/usr/lib/nfs/nfsd -a 128
```

Если количество активных потоков приближается к количеству, заданному при загрузке, то количество потоков сервера NFS следует увеличить. Получить примерное количество активных потоков сервера NFS можно с помощью такой последовательности команд:[1]

```
csh# echo '$<threadlist' | adb -k |& grep svc_run | grep -v grep | wc -l
       4
```

[1] На самом деле такая последовательность команд подсчитывает количество потоков, запущенных через `svc_run`. Существуют и другие потоки, не являющиеся потоками NFS, которые запущены таким же образом, однако обычно потоков NFS большинство.

Регулирование буферного кэша

Буферный кэш применяется для кэширования дискового ввода-вывода, относящегося только к индексным дескрипторам и косвенным блокам. Размером такого кэша управляет параметр `buhwm`, значение которого указывается в килобайтах. По умолчанию он равен нулю, что позволяет использовать до 2% системной памяти. Это значение можно увеличить до 20%. Возможно, в больших системах значение параметра `buhwm` следует уменьшить, чтобы в сервере не произошло исчерпания всего адресного пространства ядра.

За работой буферного кэша можно наблюдать с помощью *sar -b*:

```
% sar -b 5 10
...
01:22:48 bread/s lread/s %rcache bwrit/s lwrit/s %wcache pread/s pwrit/s
01:23:03       0      71     100      32     103      69       0       0
```

Похоже, в системе затишье. Однако если количество операций чтения и записи велико (более 50 в секунду), а значения коэффициентов попаданий в кэш при чтении (`%rcache`) и записи (`%wcache`) опускаются ниже 90 и 65% соответственно, то размер буферного кэша следует увеличить. Более подробные сведения о буферном кэше можно найти в разделе «Буферный кэш» главы 5.

Параметр maxusers

Параметр `maxusers` управляет размером различных таблиц ядра, таких как таблица процессов. Его размер, определяемый динамически, соответствует объему скомпонованной физической памяти:[1] в любой системе с объемом RAM более 1 Гбайт параметр `maxusers` будет равен 1024. Если параметр `maxusers` устанавливается вручную, то минимум равен 8, а максимум – 2048. Самое важное в этом параметре то, что он применяется для вычисления размеров по умолчанию кэша поиска имен каталогов и кэша индексных дескрипторов. Этими размерами управляют параметры `ufs_ninode` и `ncsize`. По умолчанию размер таких кэшей вычисляется по формуле `(17 x maxusers) + 90`.

Кэш поиска имен каталогов (DNLC)

Кэш поиска имен каталогов (*directory name lookup cache*), или DNLC, кэширует операции поиска в каталогах. Промах при обращении к этому кэшу означает, что для чтения каталога, лежащего на пути к извлекаемому файлу, может потребоваться дисковый ввод-вывод. Взаимодействие с кэшем DNLC происходит при выполнении операций `getattr`, `setattr` и `lookup`, которые составляют более 50% всего количества вызовов NFS. Коэффициент попаданий в такой кэш можно определить с помощью *vmstat -s*.

[1] За исключением памяти, забираемой ядром при загрузке.

```
% vmstat -s | grep name
  4300527 total name lookups (cache hits 92%)
```

Как правило, если коэффициент попаданий ниже 90%, то параметр `ncsize` следует настроить. Этот параметр управляет размером DNLC, задавая количество преобразований «имя – место на диске», которые могут быть кэшированы. Для каждого элемента кэша необходимо около 50 байт. Единственным ограничением размера DNLC является объем доступной памяти ядра. Поскольку серверам NFS обычно не нужно много физической памяти, то здесь может потребоваться настройка. На выделенном сервере NFS удвоение значения по умолчанию будет разумным шагом (см. раздел «Параметр maxusers» ранее в этой главе).

Кэш индексных дескрипторов

Во время каждой операции с файловой системой индексный дескриптор, считываемый с диска, кэшируется на случай повторной в нем необходимости. Параметр `ufs_ninode` управляет количеством индексных дескрипторов, хранящихся в кэше. Для каждого индексного дескриптора необходимо около 320 байт памяти ядра. Поскольку каждый элемент в DNLC указывает на элемент в кэше индексных дескрипторов, то размеры этих двух кэшей должны выбираться совместно.

Так как `ufs_ninode` – это просто предел, то его значение на работающей системе можно изменить с помощью *adb*. Протестированный верхний предел соответствует значению 2048 параметра `maxusers`, что эквивалентно значению 34 906 для `ncsize`. Однако вместо того чтобы напрямую настраивать размер кэша индексных дескрипторов, следует настроить `ncsize` и предоставить системе возможность самой выбрать значение параметра `ufs_ninode`.

Наблюдение за производительностью сервера NFS с помощью nfsstat

Одной из самых полезных утилит для получения сведений о производительности сервера NFS является команда *nfsstat -s*, отображающая статистику службы NFS:

```
# nfsstat -s
Server rpc:
Connection oriented:
calls       badcalls    nullrecv    badlen      xdrcall     dupchecks
9035483     0           0           0           0           165117
dupreqs
0
...
Version 3: (8616387 calls)
null        getattr     setattr     lookup      access      readlink
3651 0%     3446452 39% 25 0%       1509014 17% 2317518 26% 1828 0%
read        write       create      mkdir       symlink     mknod
1037342 12% 0 0%        12 0%       1 0%        1 0%        0 0%
```

```
remove      rmdir       rename      link        readdir     readdirplus
69 0%       0 0%        8 0%        0 0%        102506 1%   165003 1%
fsstat      fsinfo      pathconf    commit
1263 0%     1248 0%     30446 0%    0 0%
...
```

Здесь много данных. Вот самое важное, на что следует обратить внимание:

- Большое число в поле `badcalls` свидетельствует о том, что существует пользователь, являющийся членом слишком многих групп, либо есть много попыток доступа к неэкспортированной файловой системе.
- Если `readlink` больше 10%, то это означает, что используется слишком много символических ссылок.
- Если `getattr` выше 60%, то, скорее всего, на стороне клиента недостаточно хорошо кэшируются атрибуты. При монтировании это можно настроить с помощью параметра `actimeo`.
- Если `null` больше 1%, то тайм-аут автомонтировщика, вероятно, слишком мал.
- Если количество записей превышает 5%, то хорошей подмогой для сервера будет энергонезависимое кэширование или файловая система с протоколированием.

Глобальные сети и NFS

Зачастую клиенты и серверы NFS расположены в несмежных сетях, объединенных через один или несколько маршрутизаторов. Это привносит сложность, которая пока не обсуждалась: задержку в сетевых каналах. В локальных сетях, вследствие коротких расстояний, задержка среды не является существенной. Величина задержки очень важна, поскольку для большинства операций с интенсивной обработкой атрибутов характерен режим «запрос–ответ» – каждый запрос должен получить ответ до выдачи следующего запроса. В глобальных сетях задержка может быть высокой в силу трех причин:

- Глобальные сети гораздо более восприимчивы к ошибкам при передаче, которые вызывают существенные повторные пересылки данных. Время передачи одного пакета может быть намного больше предполагаемого.
- Значительную задержку могут привнести физические носители, применяемые для передачи данных на длинные расстояния (особенно спутниковые каналы).
- Перенаправление пакетов из одной сети в другую, осуществляемое маршрутизаторами, занимает некоторое конечное время. Поскольку между сетями обычно есть несколько маршрутизаторов, то это время может складываться.

Безусловно, работа NFS в глобальных сетях допустима, особенно в средах с интенсивной обработкой данных, где простота использования NFS компенсирует сниженную пропускную способность. В средах с интенсивной обработкой атрибутов повышенная задержка может существенно снизить производительность, но это не обязательно сводит на нет применение NFS. CacheFS, а также TCP как транспортный механизм, могут быть намного более полезны именно в таких средах.

CIFS и UNIX

Samba – это популярный бесплатный программный продукт, который позволяет системам UNIX предоставлять файловые и печатные сервисы системам Windows по протоколам SMB и CIFS. Большинство настроек системного уровня, которые можно применить для NFS, будут так же действенны для Samba. Поэтому внимание здесь будет сосредоточено на немногих выделяющихся особенностях настройки Samba. К сожалению, Samba не находит широкого распространения в тех областях, где важна производительность. Систему на базе Samba следует масштабировать горизонтально за счет добавления дополнительных серверов, а не выжимать последние несколько процентов из ресурсов сервера. В итоге, вследствие ограниченной потребности в Samba, трудно найти какие-то сведения о настройке этого продукта, которая могла бы обеспечить высокую производительность. Однако вот несколько советов.

Для того чтобы добиться максимальной производительности, при настройке Samba совершенно необходимо учитывать следующее. Во-первых, не следует оставлять высокий уровень протоколирования, если не ведется работа по отладке системы.[1] Во-вторых, важно понимать, что `strict sync`, `strict locking` и `strict always` оказывают огромное воздействие на производительность (`strict locking` – в меньшей степени).

Кроме того, можно экспериментировать с установкой флагов `TCP_NODELAY` и `IPTOS_LOWDELAY`. Установка этих флагов уменьшает задержку за счет снижения производительности, однако их применение может быть ограничено. Настройка буферов отправки и приема с помощью `SO_SNDBUF` и `SO_RCVBUF` обычно не приносит пользы, поэтому их лучше настраивать на системном уровне (см. раздел «Буферы, отметки уровня и окна» ранее в этой главе). В конфигурационный файл следует добавить опции `read raw` и `write raw`, поскольку они позволяют Samba получать из сети большие порции данных (до 64 Кбайт). Если не задана опция `wide links`, позволяющая следовать по символическим ссылкам за пределы файловой системы, то следует добавить опцию `getwd cache` для того, чтобы кэшировать пути к каталогам. Это предотвратит «обход» дерева каталогов и может существенно повысить производительность.

[1] По сути, его следует оставить равным нулю.

Заключение

С исторической точки зрения сетевая пропускная способность отставала от системной производительности. Обычно сеть была узким местом. Всегда можно было установить достаточно быстрые процессоры, чтобы заполнить сетевые каналы, а также создать дисковые массивы, более быстрые, чем сети. В последние пять лет наиболее интересные усовершенствования сводились к значительному повышению сетевой производительности. С 1991 по 2001 год пропускная способность локальных сетей Ethernet возросла с 10 Мбит/с до 1 Гбит/с. Ведется работа над стандартом 10 Гбайт. Поскольку доступная пропускная способность постепенно повышается, нагрузка на системы возрастает, а значит, для того чтобы осилить поток данных, эти данные необходимо обрабатывать достаточно быстро.

В этой главе рассмотрен большой объем материала, который системные администраторы зачастую не принимают во внимание: основы сетей, физические носители для передачи данных на расстояния, базовые интерфейсы (Ethernet, FDDI, ATM) и протоколы (особенно настройка TCP). В конце главы были обсуждены приложения, позволяющие разделять файлы как в пределах офиса, так и по всему миру. Теоретические основы здесь сложные, поэтому глава может потребовать дополнительного прочтения. Кроме того, при наличии интереса к этой теме следует обратиться к книгам, названия которых упомянуты в тексте. Эти книги помогут лучше понять взаимодействие между различными компонентами систем и сетевой инфраструктурой.

8

- *Два важнейших принципа*
- *Методы анализа кода*
- *Примеры оптимизации*
- *Взаимодействие с компиляторами*
- *Заключение*

Оптимизация кода

La perfection est atteinte non quand il ne reste rien a ajouter, mais quand il ne reste rien a enlever.
Совершенство достигается не тогда, когда уже нечего прибавить, но когда уже ничего нельзя отнять.

Антуан де Сент-Экзюпери[1]

Искусство написания программ, работающих с максимальной производительностью, изучить трудно, но еще труднее им овладеть. Бесспорно, это может быть темой целой книги. Однако настройка операционной системы только подтверждает, что неоптимизированное программное обеспечение может стать узким местом так же легко, как и нехватка физической памяти. В таких случаях аккуратная доработка исходного кода приложения может способствовать увеличению производительности. В целом, невысокое быстродействие программного обеспечения может объясняться двумя причинами: выбором неверного алгоритма и неэффективной реализацией.

В этой главе речь пойдет о двух важнейших принципах оптимизации, связанных с повышением производительности системы. Кроме того, будут рассмотрены средства, позволяющие анализировать быстродействие приложений (такие как система проб трассировочной нормальной формы (trace normal form, TNF) в Solaris и сервисная программа профилирования GNU *gprof*), а также несколько простых примеров оптимизации.

---

[1] Антуан де Сент-Экзюпери «Планета людей», перевод Норы Галь. – *Примеч. перев.*

Два важнейших принципа

В целом, есть два исключительно важных принципа, следование которым ведет к улучшению производительности приложений. Первый принцип – при написании кода следует учитывать методы *рефакторинга*. Рефакторинг – это технология, пришедшая из Smalltalk и академических программистских сообществ. Особое значение в ней придается ясному, лаконичному программированию, а также автоматическому тестированию. Когда код подвергнут тщательному рефакторингу, его легко понять, а также намного проще модифицировать в будущем. К сожалению, такой стиль программирования предполагает дисциплинированность разработчика. Отличным справочником по этой методике программирования является книга Мартина Фаулера (Martin Fowler) и др. «Refactoring: Improving the Design of Existing Code»,[1] Addison Wesley.

Второй принцип – правильный выбор алгоритма. Хотя полное обсуждение алгоритмического анализа далеко выходит за рамки этой книги, отметим следующее. Важно понимать, что к решению одной и той же задачи ведут много разных путей, одни из которых намного короче других. Далее в качестве примера будут рассмотрены различные алгоритмы для поиска в строке по заданному шаблону (pattern).

Обычно замена алгоритма ведет к значительному повышению быстродействия приложений. Ясный код, подвергнутый рефакторингу, сделает проще нахождение узких мест и внесение поправок в алгоритм.

Читатель может заметить, что эти основные принципы сводятся к «написанию хорошего кода на базе быстрых алгоритмов». Это правда, однако опыт авторов свидетельствует, что зачастую программисты пишут плохой код, применяя медленные алгоритмы. Тому есть различные причины:

- Написать плохой код намного быстрее, чем хороший. Разработчики хороших программ зачастую идут на увеличение времени первоначального написания кода в обмен на снижение времени, необходимого для его поддержки. Это достигается за счет исключения ошибок на этапе проектирования. Кроме того, анализ производительности на этапе разработки окупается снижением времени прогона приложения. Однако в преддверии крайнего срока сдачи программ и перед лицом разгневанного начальства легче просто сдать работу, чем выполнить ее хорошо.
- Техническая реализация хороших алгоритмов – дело непростое. Как будет показано в примере с поиском в строке, реализация безыс-

[1] Мартин Фаулер и др. «Рефакторинг: улучшение существующего кода», Символ-Плюс, 2002.

кусного алгоритма поиска намного проще реализации более сложной системы. Когда продукт должен быть сдан вчера, а времени на применение добротного алгоритма просто нет (или отсутствует время на его разработку, если такого алгоритма нет в наличии), то качественные алгоритмы остаются в стороне.

- Найти хороших разработчиков программ непросто. Их отличительной чертой является способность создавать хороший исходный код, но не только это. Разработка программного обеспечения подразумевает проектирование архитектуры системы, взаимодействие с другими организациями-разработчиками, анализ формального кода, ведение документации, управление версиями и обычно несколько хорошо продуманных технологий совместной работы. Концепции, подобные программированию в паре, только начинают выходить из учебных стен в реальный мир и пока не применяются повсеместно. По-настоящему отменные разработчики программ требуют очень высоких окладов, поэтому команду из них собрать трудно.
- Довольно редко группа разработчиков приступает к разработке программ с нуля. Более вероятно, что они подключаются к уже ведущимся проектам или разрабатывают следующую версию текущего продукта. Уже были допущены ошибки, а все отбросить и начать заново будет нерентабельно.

По существу, при модификации исходного кода приложений есть два пути к улучшению производительности: эволюционный и революционный. Эволюционный путь не предполагает значительных переделок алгоритмов, лежащих в основе приложения. Например, если в многопотоковых приложениях возникают конфликты при блокировках по нажатию клавиш, то незначительная модификация методов блокировки есть эволюционное изменение. Революционные изменения опираются на переработку базового алгоритма. Различие состоит в том, что эволюционные изменения улучшат производительность в несколько раз. Это сильно зависит от того, что собой представлял исходный код приложения. Революционные изменения могут повысить производительность на несколько порядков, однако воплотить их труднее.

Несомненно, алгоритмические изменения таят огромный потенциал для улучшения производительности. Но как при разработке программного обеспечения применять методы рефакторинга? Основная идея такова: при написании нового кода следует учитывать методы рефакторинга, а для оптимизации существующего кода рефакторинг просто необходим. Рефакторинг необходимо понять, и поначалу он отнимает немало времени. Однако его применение быстро входит в привычку. Когда код оптимизирован за счет применения методов рефакторинга, то улучшать его производительность намного легче. Если код ясен, компактен и его легко понять, то интерпретация результатов профилирования становится простой. При этом изменять алгоритм будет намного проще, а сама модификация отнимет меньше времени.

Важность рефакторинга при разработке и настройке приложений трудно переоценить. Помня об этих принципах, перейдем к обсуждению поиска в строке. Это явится наглядной иллюстрацией того, как проявляет себя выбор алгоритма.

Алгоритмы поиска в строке

Пусть есть строка текста «Here is a string of text which we wish to search». Необходимо найти слово «wish» в этой строке. Впредь строку, в которой проводится поиск, будем называть *S*, или строкой поиска. Строку, которую необходимо найти, назовем *T*, или целевой строкой. Как можно решить эту задачу? Рассмотрим несколько различных подходов.

Алгоритм 1: безыскусный поиск

Первый способ поиска строки *T* в строке *S* такой. Сравниваем первый символ в *T* и первый символ в *S*, а затем придерживаемся следующих правил:

1. Найдено несоответствие. Это означает, что при текущем размещении строк друг относительно друга совпадения между ними нет (там, где начато сравнение). Сдвигаем *T* на одну позицию вдоль строки *S* и снова сравниваем.[1]
2. Найдено соответствие. Это потенциальное совпадение, поэтому необходимо продолжать поиск. Сравниваем следующие символы в *T* и *S*.
3. Пройдены все символы в *T*. Совпадение найдено!
4. Пройдены все символы в *S*. Целевой строки в строке поиска нет.

Вычислительная сложность и система обозначений «О большое»

Один из методов оценки сложности алгоритма основан на *критерии асимптотической сложности* (*asymptotic complexity measure*). Вместо точного подсчета количества шагов, требуемых для реализации алгоритма, в этом методе рассматривается, как возрастает их количество с увеличением размера входных данных. Эта асимптотическая сложность записывается с помощью системы обозначений *«О большое»* (*big O*). «О» происходит от слова «order» (порядок, степень).

[1] На самом деле это слегка улучшенный вариант «по-настоящему безыскусного» алгоритма поиска. В по-настоящему безыскусном алгоритме сравнение будет продолжаться, пока не иссякнут символы в *T*, вне зависимости от того, найдено ли несоответствие во время поиска. Процесс прерывания поиска сразу же после нахождения несоответствия называется *коротким замыканием* (*short-circuiting*). В реальных безыскусных алгоритмах короткое замыкание применяется почти всегда.

Для примера рассмотрим очень простой фрагмент кода:

```
a = 3 b * 5;
c = c + 1;
```

Выполнение этих строк занимает фиксированное время, которое запишем как O(1). Точное значение, вычисляемое исходя из времени выполнения команд, определить бывает трудно. Однако, каким бы это значение ни было, оно должно быть одним и тем же, когда бы код ни выполнялся.

Рассмотрим простой цикл:

```
for (i = 0; i <= n; i++) {
     j[i] = j[i]++;
}
```

Этот цикл будет пройден ровно n раз.[a] Поскольку время выполнения команд внутри цикла постоянно, то общее время его прохождения линейно пропорционально n. Запишем это как O(n). Точное время прохождения будет примерно $9n$ нс. Возможно, даже $9n + 6$ нс, поскольку для запуска цикла необходимо немного времени. Система обозначений «О большое» позволяет применять как коэффициент умножения (подобно 9), так и слагаемое (подобно 6). Поскольку время прогона линейно пропорционально n, то правильная запись – O(n).

Обратимся к еще одному, более сложному циклу:

```
for (i = 0; i <= n; i++) {
     for (j = 0; j <= n; j++) {
          k[i][j] = k[i][j]++;
     }
}
```

Внешний цикл выполняется n раз, а внутренний – n раз для каждого прохода внешнего цикла. Код внутри цикла выполняется за фиксированное время. Так как внутренний цикл, по существу, выполняется n^2 раз, то сложность этого цикла – O(n^2). В алгоритмическом анализе такая система обозначений широко распространена, поэтому она будет применяться в дальнейших рассуждениях.

---

[a] Любой, кто знает C, заметит, что это не так. Первый раз цикл выполнится при значении i=0, а n-й раз – при значении i=n-1. Автор был бы прав, написав в условии цикла строгое неравенство i<n. Впрочем, для понимания рассматриваемого вопроса это неважно. – *Примеч. науч. ред.*

Повторять, пока не выполнится условие 2 или условие 3. Пусть необходимо найти строку «cad» (*T*) в строке «cabcadchad» (*S*). В табл. 8.1 показано продвижение по строке согласно этому алгоритму.

Таблица 8.1. Безыскусный алгоритм поиска

Шаг	Символ в S	Символ в T	Правило	Состояние поиска
1	c	c	2	S: *c*abcadchad T: *c*ad
2	a	a	2	S: *ca*bcadchad T: *ca*d
3	b	d	1	S: *cab*cadchad T: *cad*
4	a	c	2	S: cabcadchad T: cad
5	b	a	1	S: c*ab*cadchad T: *ca*d
6	c	b	1	S: c*ab*cadchad T: *c*ad
7	c	c	2	S: cabcadchad T: *c*ad
8	a	a	2	S: cabcadchad T: *ca*d
9	b	b	2	S: cabcadchad T: *cad*
10			3	

В худшем случае для безыскусного алгоритма поиска необходимо следующее количество сравнений:

$$\text{длина шаблона} - \text{количество возможных начальных точек} = T \times (S - T)$$

Например, в худшем случае количество необходимых сравнений 48-символьной строки S и 4-символьной строки T составит $4 \times (48 - 4) = 176$. Сложность безыскусного алгоритма поиска выражается как O(st), где s – длина S, а t – длина T. Ясно, что для повышения производительности необходим более эффективный алгоритм.

Алгоритм 2: поиск Кнута–Морриса–Пратта

Если в безыскусном алгоритме обнаруживается несоответствие символов, то начальная точка сдвигается вперед на один символ. К сожалению, таким образом отбрасывается вся информация о сочетаемости символов, полученная при продвижении по строке. Пусть есть строка T «abababcd» и строка поиска «abababab c...» (неважно, что стоит после c, – пока это вызывает несоответствие). Когда после позиции c обнаружи-

вается несоответствие, то безыскусный алгоритм рекомендует сдвинуться на один символ в S и снова начать проверку. Поскольку известно, что первые шесть символов в T совпали с символами в S, то сдвиг на единицу не будет работать. Однако сдвиг шаблона на две позиции даст другое возможное соответствие. Известно, что первые четыре символа уже совпадают, поэтому можно продолжать поиск с того же места в S. Это означает, что проверять символы исходного текста более одного раза никогда не придется. Сложность алгоритма Кнута-Морриса-Пратта (Knuth-Morris-Pratt, KMP) выражается как O(s+t).

Как правило, KMP быстр именно там, где безыскусный алгоритм работает медленно – когда S и T содержат повторяющиеся последовательности символов. KMP особенно эффективен, когда алфавит (набор символов) мал (например, генетический код, использующий только четыре символа: G, C, A и T, или битовые шаблоны).

Алгоритм 3: поиск Бойера–Мура

Последний из алгоритмов, рассматриваемых на предмет быстрого и простого нахождения соответствия строк, – это алгоритм Бойера-Мура (Boyer-Moore). Впервые он был опубликован в 1977 году. Этот алгоритм применяет иной подход и опирается на два метода, позволяющих в случае несоответствия сдвигать шаблон на большее количество символов. За счет этого достигается более высокая производительность.

Изменение в алгоритме связано с тем, что соответствие строки T строке S проверяется справа налево, а не слева направо, в то время как T по-прежнему движется вдоль S слева направо. Например, если T – это «wish», а S – это «dish», то совпадения «h», «s» и «i» будут найдены до обнаружения несоответствия «d» и «w».

Первый метод, применяемый для более эффективного сдвига шаблона в случае несовпадения, – сдвигать шаблон таким образом, чтобы стало возможно последующее совпадение с символом в строке S. Рассмотрим пример. Пусть в строке «Here is a string of text which we wish to search.» необходимо найти слово «wish»:

```
S:   here is a piece of text which we wish to search
        |
T:   wish
```

«H» не совпадает с «e», и «e» нет в «T», поэтому «e» не может быть в подстроке совпадения. Значит, шаблон можно сдвинуть за «e» – на четыре позиции. По сути, на длину T. Текущее состояние таково:

```
S:   here is a piece of text which we wish to search
               |
T:       wish
```

Пробел ничему не соответствует в T, поэтому можно снова сместиться на четыре символа:

```
S:   here is a piece of text which we wish to search
                 |
T:           wish
```

Снова несовпадение, но теперь с «i», которая есть в шаблоне. Поэтому смещаем T так, чтобы «i» в S и «i» в T были выровнены:

```
S:   here is a piece of text which we wish to search
                 |
T:             wish
```

В T нет «c», поэтому сдвигаем T на четыре позиции. Если и далее следовать этому алгоритму, то в конечном итоге местонахождение T в S будет найдено. Совпадение встречается на 34-й позиции в S, однако потребовалось только 15 сравнений. Безыскусный метод и алгоритм KMP потребовали бы по меньшей мере 37 сравнений. Таким образом, описываемый подход демонстрирует огромный выигрыш в производительности. Вообще, чем длиннее шаблон, тем меньше требуется сравнений.

Второй механизм, используемый в алгоритме Бойера-Мура, по сути является адаптацией метода «размещения несовпадений», применяемого в KMP для определения того, насколько нужно сдвинуть T вдоль S. Однако здесь этот метод приспособлен для поиска справа налево. В целом, по сравнению с безыскусным подходом и алгоритмом KMP, поиск по методу Бойера-Мура намного быстрее, особенно для длинных шаблонов и текста с большим алфавитом. Его сложность – $O(s/t)$.

Ловушки оптимизации

Есть три ловушки, которых следует избегать при оптимизации кода: убывание эффекта, соотношение времени разработки и времени прогона, а также функциональная эквивалентность. В табл. 8.2 приведены характеристики программы, иллюстрирующие ловушку убывания эффекта.

Таблица 8.2. Временные характеристики простой программы

Функция	Доля времени	Функция	Доля времени
`main`	8%	`bar`	33%
`foo`	14%	`baz`	45%

Если производительность `baz` увеличить на 20%, то время прогона программы снизится на 9%, поскольку `baz` занимает 45% всего времени работы. Если увеличить производительность `foo` на те же 20%, то время работы программы улучшится только на 2,8%. Поэтому усилия по настройке следует прикладывать к тем участкам, где улучшения принесут наибольшую отдачу. Обычно это означает устранение узких мест в критических участках кода. Латание узких мест на некритических участках вряд ли приведет к значительному приросту производительности.

Другая ловушка – соотношение времени разработки и времени прогона. Во многих случаях программа может быть настолько специализированной, что ей суждено быть запущенной только несколько раз. В таких случаях время, затраченное разработчиками на профилирование, настройку и тестирование кода, может перекрыть улучшения, достигнутые благодаря оптимизации.

Последнее важное предупреждение, связанное с оптимизацией кода, – это функциональная эквивалентность. Не все методы оптимизации гарантируют одинаковость результатов с точностью до бита. Например, вместо деления двух чисел можно одно число умножить на инверсное (обратное) значение второго числа. Такая операция может быть намного быстрее, однако в некоторых случаях результаты не будут одинаковыми. Так и происходит, когда делитель очень мал.[1] При вычислении обратного значения этого маленького числа произойдет переполнение, то есть результатом будет бесконечность. Таких ошибок следует избегать. Для этого после применения каждого метода оптимизации следует внимательно проверять результаты программ с помощью репрезентативного набора входных данных.

Методы анализа кода

Для анализа производительности приложения существует немало способов. В целом, их можно разбить на две группы: анализ *с помощью проб* (*probe-based*) и *с помощью профилирования* (*profiler-based*). Анализ с помощью проб основан на кусках кода, которые программист вставляет в приложение. Эти фрагменты помогают замерять время выполнения программы, а также предоставляют другие данные, значимые для последующего анализа. Для того чтобы проводить анализ кода с помощью профилирования, приложение должно быть скомпилировано с его поддержкой. В этом случае программа сможет выдавать данные о своей производительности. Впоследствии эти данные можно проанализировать с помощью внешней программы профилирования.

Перед тем как приступить к анализу того, как распределяется время выполнения приложения, необходимо точно выяснить, сколько же всего времени затрачивается. Для этого существуют инструменты с относительно грубой шкалой измерения: *time*, *timex*, а также *ptime* (в Solaris).

---

[1] В частности, делитель должен быть *денормализован*. Число может быть настолько мало, что длина поля степени числа в представлении с плавающей точкой является недостаточной для отображения всей степени. Поэтому старшие биты дробной части устанавливаются в нуль, чтобы уменьшить величину числа. Например, в числе $6{,}0\times10^{-324}$ первые пятьдесят один бит дробной части будут установлены в нуль, то есть в дробной части будет оставлена только одна значимая цифра. Обычно денормализация происходит в том случае, когда число меньше $2{,}3\times10^{-308}$.

Измерение времени работы приложения: time, timex и ptime

Самый простой вопрос о быстродействии программы может звучать так: «Как долго она выполняется?» Вместо того чтобы сидеть перед компьютером с секундомером, для измерения времени выполнения программы можно воспользоваться простыми инструментами.

time

Первый из этих инструментов – команда *time*, присутствующая в командных интерпретаторах *csh* и *ksh* (также это может быть отдельная утилита в */bin*). В конце работы программы *time* представляет раскладку того, как распределилось время прогона приложения:

- *Истекшее время* (*elapsed time*). Это время по «настенным часам», затраченное на работу приложения.
- Время, проведенное в *пользовательском режиме* (*user mode*). Другое название – *пользовательское время* (то есть время выполнения команд, которые компилятор генерирует на основе исходного кода, плюс время, затраченное на библиотечные подпрограммы; *user time*).
- Время, проведенное в *режиме ядра* (*kernel mode*). Другое название – *системное время* (то есть время, затраченное на системные вызовы, выдачу запросов ввода-вывода и т. д.; *system time*).

Кроме того, команда *time*, встроенная в *csh*, выдает другую полезную информацию. Рассмотрим пример:

```
% time sieve > /dev/null
6.490u 0.530s 0:16.74 41.9%     0+0k 0+0io 111pf+0w
```

Первое, второе и третье поля соответствуют секундам пользовательского времени, системного времени и истекшего времени процесса. Вообще, пользовательское время должно значительно превышать системное время. Если системное время несоразмерно велико, то это свидетельствует о том, что программа делает что-либо неэффективно. Возможно, программа генерирует много исключений либо неэффективно использует системные вызовы. Однако заметим, что «время ожидания ввода-вывода» (время, затраченное на перемотку ленты, движение дисковых головок считывания-записи и т. д.) не считается временем процессора.

Четвертое поле относится к проценту использования (percentage utilization). Это отношение истекшего времени ко времени процессора (сумме пользовательского и системного времени). Есть много причин, из-за которых время процессора может существенно отличаться от истекшего времени. В конце концов, в системе может быть значительное количество других процессов. Кроме того, данное приложение может инициировать большой ввод-вывод (либо напрямую, либо вследствие

исчерпания физической памяти и неизбежного пейджинга). Также возможно, что система недостаточно быстро извлекает данные из памяти и загружает их в нее.

Пятое поле представляет статистику *среднего потребления памяти* (*average memory utilization*). Первое значение – это *средний объем пространства разделяемой памяти* (*average shared-memory space*). Оно определяет, сколько памяти использует приложение (эта память может разделяться между несколькими вызовами одного и того же процесса – вот откуда название «разделяемая память»). Второе значение – *средний объем пространства неразделяемой памяти* (*average unshared-memory space*), в котором хранятся структуры данных. Эти значения сообщают только о потреблении реальной физической памяти. Кроме того, такие измерения поддерживаются не на всех платформах, поэтому иногда значения сообщаются как `0+0k`.

Шестое поле представляет объем операций блочного ввода и вывода соответственно. Это поле также поддерживается не во всех реализациях команды *time*, поэтому иногда оно сообщается как `0+0io`.

Седьмое, и последнее, поле описывает количество страничных ошибок и свопов, вызванных приложением. Если количество страничных ошибок слишком велико, то это свидетельствует о том, что процессу, вероятно, не хватало памяти. Однако страничные ошибки вызывает каждая программа (просто в силу механизма выделения страниц; см. раздел «Свободный список» в главе 4).

Заметим, что существует другая версия команды *time*, которая не встроена в командный интерпретатор:

```
# /bin/time catman -w

real       36.3
user       11.9
sys         3.5
```

В Linux утилита *time* расположена в */usr/bin*. Она представляет иной вывод:

```
% /usr/bin/time sieve > /dev/null
5.55user 0.25system 0:05.95elapsed 97%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (112major+9778minor)pagefaults 0swaps
```

Этот вывод сопоставим с выводом функции *time*, встроенной в интерпретатор.

timex

Команда *timex* предлагает чуть более точный способ измерения времени, которое затрачивается на выполнение программы. Синтаксис команды *timex* похож на синтаксис уже обсужденных команд. Она лишь предоставляет более высокую точность:

```
# timex catman -w

real       12.78
user       11.71
sys         2.67
```

(Заметим, что время работы процесса *catman -w* существенно снизилось по сравнению с последним запуском. Это произошло вследствие кэширования файловой системы; сама команда *timex* не является тому причиной.)

В команде *timex* следует отметить три ключа. Если инсталлированы средства учета процессов, то с помощью ключа *-o* можно получить количество прочитанных и записанных блоков. Ключ *-s* сообщает общую активность системы во время обслуживания процесса. Ключ *-p* предоставляет данные учета процессов, относящиеся к запущенной программе.

ptime

В качестве точного метода измерения времени, затрачиваемого при работе приложения, Solaris предлагает утилиту *ptime*. Преимущество *ptime* – в простоте ее запуска:

```
# ptime catman -w

real       14.455
user        0.005
sys         0.009
```

Этот пример также хорошо иллюстрирует, что сумма системного и пользовательского времени необязательно соответствует реальному (истекшему) времени. Утилита *catman -w*, индексирующая базу данных windex, необходимую для команд *man -f*, *man -k* и *apropas*, довольно интенсивно обращается к диску. А огромная разница объясняется тем, что время выполнения дисковых операций не входит ни в пользовательское, ни в системное время.

В системах с параллельной обработкой пользовательское время может быть больше истекшего, так как включает в себя пользовательское время всех процессоров.

Механизмы измерения времени

Пожалуй, есть смысл рассмотреть, как работают эти команды. Утилиты */bin/time* и *timex* квантуют состояние процессора с помощью прерываний по таймеру. Такой подход дает точность около 1/100 с (это соответствует частоте прерываний по таймеру, заданной по умолчанию).

Утилита *ptime* применяет совершенно иной механизм. Когда ядро обслуживает изменение состояния, системный вызов, страничную ошибку или изменение таблицы планировщика задач, то в протокол заносятся высокоточные временны*е* отметки. Такой процесс называется *учетом микросостояний* (*microstate accounting*). Он значительно

улучшает точность, однако вызывает дополнительные издержки. Кроме того, учет микросостояний можно включить программно.

Участки кода, относящиеся к измерению времени

Для получения данных о продолжительности выполнения отдельных участков кода существует немало способов. Здесь будут рассмотрены два из них, а именно `gethrtime()` – вызов функции, возвращающий текущее время в наносекундах, и прямой доступ к регистру `TICK`.

Измерение времени с помощью gethrtime

Перед тем как применять системный вызов `gethrtime()`, рекомендуется включить учет микросостояний. Сделать это можно двумя способами: включить учет микросостояний в самом коде либо задействовать *ptime* для оценки истекшего времени программы. В примерах 8.1 и 8.2 представлены оба способа.

Пример 8.1. gethrtime_timing.c (без включения учета микросостояний)

```
/* gethrtime_timing.c */

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int
main(int argc, char **argv)
{
     hrtime_t timeStart, timeEnd;
     int i, j = 250000;
     timeStart = gethrtime();
     for (i = 0; i < j; i++) {
     /* Здесь размещается функция, время выполнения  которой необходимо измерить */
     }
     timeEnd = gethrtime();
     printf ("Среднее время: %lld ns\n", (timeEnd - timeStart) / j);
     return 0;
}
```

Пример 8.2. gethrtime_ustate_timing.c (включение учета микросостояний)

```
/* gethrtime_ustate_timing.c */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/time.h>
#define _STRUCTURED_PROC 1
#include <sys/procfs.h>
/* Функция init_micro_acct() взята из файла timing.c, который включен в состав
```

```
   Forte Developer 6: см./opt/SUNWspro/WS6/examples/analyzer/omptest/timing.c */
void
init_micro_acct()
{ /* SunOS 5.5 и более новые версии */
     int    ctfld;
     long   ctl[2];
     char   procname[1024];

     sprintf(procname, "/proc/%d/ctl", getpid());
     ctlfd = open(procname, O_WRONLY);
     if(ctlfd < 0) {
          fprintf(stderr, "open %s failed, errno = %d\n", procname, errno);
     }
     ctl[0] = PCSET;
     ctl[1] = PR_MSACCT;
     if (write(ctlfd, ctl, 2*sizeof(long)) < 0) {
          fprintf(stderr, "write failed, errno = %d\n", errno);
     }
     close(ctlfd);
     printf("Включение учета микросостояний.\n");
     return;
}

int
main(int argc, char **argv)
{
     hrtime_t timeStart, timeEnd;
     int i, j = 250000;
     /* Запуск учета микросостояний */
     init_micro_acct();

     timeStart = gethrtime();
     for (i = 0; i < j; i++) {
     /* Здесь размещается функция, время выполнения которой необходимо измерить */
     }
     timeEnd = gethrtime();

     printf ("Среднее время: %lld ns\n", (timeEnd - timeStart) / j);
     return 0;
}
```

Эти программы были собраны на Sun Ultra 10 (процессор 300 МГц UltraSPARC-II) с системой Solaris 8. Применялся компилятор Forte Developer 6 update 1, а системный вызов `getpid()` служил функцией, время прогона которой измерялось. Кроме того, количество итераций (`j`) равнялось 5 миллионам. Пример 8.3 иллюстрирует различия между запуском `gethrtime_timing` без учета микросостояний, запуском `gethrtime_timing` с включенным учетом микросостояний (с помощью *ptime*) и запуском `gethrtime_ustate_timing`.

Пример 8.3. Различия в выполнении между тремя программами измерения времени

```
% /opt/SUNWspro/bin/cc gethrtime_timing.c -o gethrtime_timing
% /opt/SUNWspro/bin/cc gethrtime_ustate_timing.c -o gethrtime_ustate_timing
```

```
% gethrtime_timing
Average time: 1714 ns
% ptime gethrtime_timing
Average time: 2708 ns

real       13.550
user        9.785
sys         3.620
% gethrtime_ustate_timing
Enabling microstate accounting.
Average time: 2648 ns
```

Можно увидеть, что значения `gethrtime_ustate_timing` и `gethrtime_timing` сравнимы, тогда как значение `gethrtime_timing` без учета микросостояний значительно ниже. Именно этого и следовало ожидать: такие результаты являются наглядной иллюстрацией издержек, вызванных учетом микросостояний. Преимущество учета микросостояний – повышение точности измерений.

Измерение времени с помощью регистра TICK

В архитектуре SPARC существует регистр `TICK`, предоставляющий точный, необременительный способ измерения истекшего времени в показателях циклов процессора. Здесь будет представлен метод прямого доступа к регистру `TICK` из пользовательских программ. Такой метод привносит намного меньше издержек по сравнению с уже обсужденными механизмами учета времени. Издержки `gethrtime()` составляют около 75 циклов процессора, а обращение к регистру `TICK` занимает шесть циклов. Доступ приложений к регистру `TICK` возможен в Solaris 8, но не в предшествующих версиях.[1]

К регистру `TICK` удобнее всего обращаться с помощью встраиваемых участков ассемблерного кода. Это проиллюстрировано в примерах 8.4 и 8.5.

Пример 8.4. readtickreg.il

```
.inline readtickreg,1
rd      %tick, %o1
stx     %o1, [%o0]
.end
```

Пример 8.5. gettick_timing.c

```
/* gettick_timing.c */
#include <math.h>
#include <stdio.h>
```

[1] Если в процессоре установлен бит непривилегированного захвата (nonprivileged trap, NPT), обращение к регистру `TICK` приводит к захвату, а выполнение кода обычно прекращается. В Solaris 8 этот бит неактивен, поэтому непривилегированный пользовательский код может читать регистр `TICK`.

```
#include <stdlib.h>
#include <sys/types.h>

extern void readtickreg(void *);
#pragma no_side_effect(readtickreg);

int
main(int argc, char **argv)
{
     uint64_t c1=0, c2=0, overhead=0, total=0, cost;
     int i, j=1, *x;
     x = &j;

     for (i = 0; i < 10; i++) {
          readtickreg ((void *)&c1);
          readtickreg ((void *)&c2);
          overhead += c2 - c1;
     }
     overhead = floor (overhead * 1.0 / 10);

     for (i = 0; i < 134217728; i++) {
          readtickreg ((void *)&c1);
          /* Здесь размещается функция, время выполнения которой необходимо
             измерить: в качестве примера используется *x += i */
          *x += i;
          readtickreg ((void *)&c2);
          total += c2 - c1;
     }
     cost = floor (total * 1.0 / 134217728 - overhead);
     printf ("Издержки обращения к регистру TICK: %lld\n", overhead);
     printf ("Затраты в циклах процессора: %lld\n", cost);
     return 0;
  }
```

134 217 728 – это произвольное число.

Далее этот код можно собрать и протестировать:

```
% /opt/SUNWspro/bin/cc -xarch=v8plusa gettick_timing.c -o gettick_timing
readtickreg.il -lm
% gettick_timing
Overhead of readtickreg: 6
Cost in processor cycles: 7
```

Таким образом, издержки при обращении к регистру TICK составляют шесть циклов, тогда как количество затраченных циклов процессора равно семи.

При замене тестовой функции на `getpid()` среднее количество циклов составило 495. Поскольку эта тестовая программа, как и предыдущие, была запущена на 300 МГц Ultra 10, то можно вычислить, что один цикл равен 3,33 нс (1 с/300 000 000 циклов равно 0,000000000333 с, или 3,33 нс). Это означает, что обработка вызова `getpid()` составляет около 1650 нс, что довольно близко к результату, полученному с помощью `gethrtime()`.

Анализ с помощью проб: Solaris TNF

Трассировочная нормальная форма (*trace normal form*, TNF) позволяет получать подробные данные о выполнении кода. Система проб TNF подразумевает вставку специальных макровызовов в код приложения и последующий анализ вывода, сгенерированного ими. Анализ проб TNF проводится с помощью инструментов *prex*, *tnfdump* и *tnfxtract*, которые уже были описаны при обсуждении мониторинга с помощью проб, присутствующих в самом ядре Solaris. Для беглого обзора этих инструментов следует обратиться к разделу «Мониторинг ввода-вывода» главы 5.

Вставка проб

Сами пробы объявляются с помощью серии макросов. Расположить эти пробы можно в любых программах, написанных на C и C++: в многопоточном коде, драйверах, загружаемых модулях ядра, разделяемых объектах и др.

Общий формат пробы таков:

```
TNF_PROBE_2 (name, keys, detail,
             argument_type_1, argument_name_1, argument_value_1,
             argument_type_2, argument_type_2, argument_value_2);
```

Рассмотрим этот формат по частям.

- Обычно имя макроса выглядит так: `TNF_PROBE_`*N*, где *N* – это число от 1 до 5. Оно определяет количество «информационных точек» (триплетов тип–имя–значение аргумента), которые будут записаны макросом. Если для `N` указан нуль, то записи «информационных точек» производиться не будут.[1] Кроме того, в конце имени макроса может стоять `_DEBUG`. Тогда проба будет вставлена только в том случае, если при компилировании задан ключ препроцессора `-DTNF_DEBUG`.
- Первый аргумент – имя пробы. Задание имени объявляет пробу, поэтому других объявлений больше не нужно. Кроме того, это объявление действует только в пределах блока, поэтому во всем остальном оно не влияет на пространство имен программы.
- Существует соглашение именовать пробы по именам их функций, а также добавлять окончание `_start` к первой пробе функции и `_end` к последней. Например, функция с именем `foo` может иметь пробы `foo_start` и `foo_end`.

[1] Однако проба и в этом случае будет определяться по имени. Зачастую это бывает полезным для анализа простого участка кода и точек выхода из функции (exit path), а также для подтверждения того, что команды выполняются в нужной последовательности (code path).

- Второй аргумент – список ключей. Это список ключевых слов, разделенных пробелами. Они определяют группы, к которым принадлежит проба. Точка с запятой, одинарные кавычки и знак равенства в этом списке не допускаются. Ключевые слова наиболее полезны в сочетании с *prex* для выбора определенного набора функций. Например, все пробы в функции `bar` могут иметь слово «bar», указанное в их ключевых полях, поэтому в такой функции очень легко включить все пробы.
- Третий аргумент – список атрибутов (list of details). Эта строка образуется из пар «атрибут-значение», где атрибут и значение разделены пробелом, а каждая пара отделена точкой с запятой. У списка атрибутов два назначения. Первое – задавать атрибут, который может быть выбран через *prex*. Например, если один атрибут – это *shape*, то *prex* может выбрать пробы на основе их значения *shape*. Второе применение – снабжать пробу строкой-комментарием, которая однократно записывается в файл трассировки. Атрибуты следует именовать согласно аббревиатуре производителя или ставить первыми какие-нибудь уникальные символы, а за ними знак процента (например, `ORA%coverart`). Некоторые атрибуты предопределены. Это `name` (название пробы), `keys` (ключи, ассоциированные с пробой), `file` (в каком файле находится проба), `line` (в какой строке находится проба), `slots` (сколько слотов/точек данных проба записывает в трассировочный файл), `object` (исполняемый или разделяемый объект, содержащий пробу) и `debug` (применяется ли проба только в процессе отладки или при работе).

Следующие три взаимосвязанных аргумента определяют сведения, которые будут записываться в протокол.

- `argument_type_`*n* определяет тип *n*-го аргумента. Предопределенные типы приведены в табл. 8.3.

Таблица 8.3. Типы аргументов TNF

Тип	Соответствующий тип в C
`tnf_int`	Int
`tnf_uint`	Unsigned int
`tnf_long`	Long
`tnf_ulong`	Unsigned long
`tnf_longlong`	Long long (только в 64-разрядных системах)
`tnf_ulonglong`	Unsigned long long (только в 64-разрядных системах)
`tnf_float`	Float
`tnf_double`	Double

Таблица 8.3 (продолжение)

Тип	Соответствующий тип в C
tnf_string	Char *
tnf_opaque	Void *

Для объявления нового типа TNF существует документированный интерфейс. Подробные сведения о нем можно получить, обратившись к странице документации (manpage) TNF_DECLARE_RECORD.

- argument_name_*n* определяет имя *n*-го аргумента. Кавычки необязательны.
- argument_value_*n* определяет значение *n*-го аргумента. Оно оценивается для получения значения, которое включается в трассировочный файл. Если в многопоточной программе в argument_value_*n* содержится переменная, которую следует защитить от чтения, то в пробе TNF нужно разместить блокировки.

В любой исходный код, применяющий макросы проб TNF, необходимо включать файл заголовка TNF <tfn/probe.h>. Кроме того, код должен быть компилирован с указанием библиотеки TNF (*-ltnfprobe*; заметим: если библиотека *libtnfprobe* явно задается в программе, то она должна находиться перед *libthread*). Отключить все пробы можно с помощью ключа препроцессора -DNFPROBE.

Предостережения

Одно существенное предупреждение, касаемое этого метода трассировки, сводится к тому, что пробы TNF могут оказывать значительное влияние на производительность приложения. Необходимо тщательно экспериментировать, чтобы точно определить влияние проб на выполнение кода. Однако этот метод во многих случаях очень полезен, поэтому ошибок при экспериментах не стоит бояться.

Анализ на основе профилирования: gprof

Профилирование предоставляет сведения о том, как распределялось время выполнения программы и какие функции вызывались из каких. Эти данные соответственно называются *профилем* (*profile*) и *графом вызовов* (*call graph*). Метод профилирования очень хорош, когда нужно выяснить, что же делает программа. Особенно если речь идет о больших или сложных приложениях, где использование непосредственных методов было бы нерационально (например, применение проб TNF). Одно предостережение – при профилировании программа должна работать в реальном рабочем режиме, поскольку ее операции влияют на результаты профилирования.

Вообще, профилирование приложения включает в себя три этапа:

- Компиляция и компоновка программы с включенным профилированием.
- Выполнение приложения.
- Запуск анализатора профиля,[1] такого как *gprof* GNU, для получения удобных для восприятия результатов.

Перед тем как обратиться к примеру профилирования, рассмотрим, как оно реализовано.

Реализация профилирования

Во время компиляции с включенным профилированием вызов каждой функции модифицируется. При этом сохраняется информация о том, откуда вызывались функции.[2] На основе таких данных анализатор профиля строит граф вызовов. Кроме того, при профилировании периодически отслеживается, какая функция исполняется в настоящее время. Обычно это происходит с частотой около 100 раз в секунду, однако частота варьируется от системы к системе. Такую периодическую выборку можно реализовать двумя способами: с помощью системного вызова либо путем указания ядру «пробудиться» и доставить процессу сигнал профилирования.

В некоторых операционных системах реализован системный вызов `profil()`, создающий массив в пространстве памяти ядра. В этом массиве каждый элемент отвечает за несколько байт (обычно от 2 до 8) адресного пространства программы. Во время выполнения программы по каждому такту системных часов проверяется счетчик команд, а значение соответствующего слота в структуре памяти ядра инкрементируется. Обычно это привносит небольшие издержки, поскольку так или иначе ядро должно «пробудиться» и обработать прерывание по таймеру.

Однако в других операционных системах вызов `profil()` или что-либо подобное не реализованы. В таких случаях ядро (с помощью какого-либо механизма) периодически доставляет сигнал процессу, который затем выполняет ту же описанную процедуру проверки-инкрементирования. Поскольку в этом методе ядро должно «пробуждаться» и передавать сигнал в пользовательское пространство, это привносит намного больше издержек.

---

[1] «Анализатор профиля» – это сокращенная форма более точного выражения «анализатор результатов профилирования». Профилирование – это процесс, дающий сведения о частоте вызова элементов программного кода. Соответственно промежуточный результат профилирования – это файл трассировки программы, который анализатор этого файла превращает в удобочитаемый листинг. – *Примеч. науч. ред.*

[2] Обычно это реализуется так. Исследуется стековый фрейм с целью нахождения в родительской функции адреса потомка и адреса возврата. Поскольку такие действия в значительной степени зависят от архитектуры, это обычно выполняется средствами ассемблера.

Компиляция с поддержкой профилирования

В следующих примерах применяется фрагмент кода на C, представляющий собой очень простую реализацию решета Эратосфена:[1]

```
/* sieve.c */

#include <stdio.h>
#include <math.h>

#define  UPPERBOUND    10000000

int SievedIntegers[UPPERBOUND];

void
init()
{
     int i;

     for (i = 0; i <= UPPERBOUND; i++) {
          SievedIntegers[i] = 1;
     }
}

void
printPrimes()
{
     int i;

     for (i = 2; i <= UPPERBOUND; i++) {
          if (SievedIntegers[i] == 1) {
               printf ("%d\n\r", i);
          }
     }
}

void
cancel (int p)
{
     int q = 0;

     for (q = 2 * p; q <= UPPERBOUND; q = q + p) {
          SievedIntegers[q] = 0;
     }
}
```

[1] Точное предназначение этого (неэффективного) фрагмента кода не особенно важно. Однако суть решета Эратосфена – найти все простые числа до некоторой верхней границы с помощью *вычеркивания* кратных для маленьких простых чисел. Например, кратное числа 2 не может быть простым числом, так как 2 – это простое число. Поэтому в списке целых чисел вычеркиваются все кратные числа 2. Знаменитая, давно доказанная теорема из теории чисел утверждает, что если число a – простое, то нужно пробовать делить его на числа только до квадратного корня от a. Таким образом, объем работы значительно снижается. Существуют *еще более* быстрые методы нахождения простых чисел.

```
int
isPrime(int p)
{
     int q;
     int r = 1;

     for (q = 2; q <= sqrt(p); q++) {
          if ((p % q) == 0) {
               r = 0;
               break;
          }
     }

     return r;
}

int
main(int argc, char **argv)
{
     int i;

     init();
     for (i = 2; i <= sqrt(UPPERBOUND); i++) {
          if (isPrime(i)) {
               cancel(i);
          }
     }
     printPrimes();

     return 0;
}
```

Первый этап – компилировать этот код. Для этого применим *gcc*:

```
% gcc -g -Wall -o sieve sieve.c -pg -lm
```

Некоторые ключи заслуживают пояснений. Ключ *-g* задает включение отладочной информации в бинарный файл. За счет этого *gprof* будет выдавать построчный список, сообщающий количество исполнений каждой строки кода. Ключ *-pg* указывает на необходимость сбора данных профилирования.

Выполнение с профилированием

Исполняемый файл запускается точно так же, как и любой другой. Если выполнение программы завершается нормально,[1] то в момент выхода в текущий рабочий каталог будет записан файл *gmon.out*. Это означает, что если в программе вызывается `chdir()`, то файл *gmon.out* будет записан в тот каталог, в который программа перешла по последнему вызову `chdir()`. В этом файле содержатся все данные профилирования, относящиеся к проведенному запуску программы.

[1] То есть посредством вызова `return()` или `exit()` из `main()`.

Анализ профиля

Теперь, когда данные собраны, их можно проанализировать с помощью *gprof executable*. Здесь *gprof* вызывается с ключом *-b*, чтобы для краткости подавить вывод подробных пояснений. Рассмотрим вывод по частям:

```
% gprof -b sieve
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 72.78      4.01     4.01      446  8991.03  8991.03  cancel
 15.06      4.84     0.83        1 830000.00 830000.00  init
 12.16      5.51     0.67        1 670000.00 670000.00  printPrimes
  0.00      5.51     0.00     3161     0.00     0.00  isPrime
...
```

Здесь представлены все вызываемые функции. В поле `time` сообщается процентное содержание полного времени исполнения программы, затраченного в этом процессе. Сумма значений по этому полю должна составлять 100%. Поле `cumulative seconds` показывает время, затраченное на исполнение функции плюс всех функций «над ней». Поле `self_seconds` – время, ассоциированное исключительно с этой функцией. Функции, у которых `self_seconds` равно 0.00, будут представлены в грбфе вызовов, но не появятся в выборках гистограммы. Следовательно, они были выполнены, но ни разу не попались при случайной выборке.

Поле `calls` сообщает, сколько раз вызывались функции. Значение `self us/call` представляет среднее время в микросекундах, затраченное на один вызов этой функции, а поле `total ms/call` показывает среднее время, затраченное этой функцией и всеми функциями «над ней» (тоже в микросекундах). Поле `total_us/call` сообщает общее количество микросекунд, затраченных на вызов этой функции и ее потомков.

Вторая часть вывода *gprof* – это граф вызовов:

```
...
                     Call graph

granularity: each sample hit covers 4 byte(s) for 0.18% of 5.51 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    5.51                 main [1]
                4.01    0.00     446/446         cancel [2]
                0.83    0.00       1/1           init [3]
                0.67    0.00       1/1           printPrimes [4]
                0.00    0.00    3161/3161        isPrime [5]
-----------------------------------------------
                4.01    0.00     446/446         main [1]
[2]     72.8    4.01    0.00     446          cancel [2]
-----------------------------------------------
```

```
                0.83    0.00       1/1           main [1]
[3]     15.1    0.83    0.00       1         init [3]
-----------------------------------------------
                0.67    0.00       1/1           main [1]
[4]     12.2    0.67    0.00       1         printPrimes [4]
-----------------------------------------------
                0.00    0.00    3161/3161        main [1]
[5]      0.0    0.00    0.00    3161         isPrime [5]
-----------------------------------------------

Index by function name

   [2] cancel                  [5] isPrime
   [3] init                    [4] printPrimes
```

Граф вызовов схематически изображает, сколько времени было затрачено на исполнение каждой функции и ее потомков. Каждая штриховая линия делит таблицу на элементы – по одному для каждой функции. В каждом элементе первичная строка – это строка, начинающаяся с индекса в квадратных скобках. Первичная строка сообщает, какую функцию описывает элемент. В предшествующих строках приводятся функции, вызывающие текущую. Последующие строки описывают подпрограммы этой функции (другое название в этом контексте – *потомки*).

В каждой первичной строке есть четыре важных поля. В поле `% time` сообщается, какая часть общего времени была затрачена в этой функции и ее потомках (в процентах). В поле `self` приводится время, потраченное на выполнение только этой функции. В поле `children` сообщается суммарное время, затраченное на вызовы подпрограмм из этой функции, а в поле `called` – количество вызовов функции. Если функция вызывала сама себя (рекурсивно), то в поле `called` будут два числа, разделенных знаком `+`. Первое число сообщает нерекурсивные вызовы, а второе – рекурсивные.

Другие строки элемента таблицы сравнительно схожи с первичной строкой. Но есть одно исключение. Поле `called` для непервичных строк содержит два числа: сколько раз текущая функция вызывалась из функции, представленной в этой строке, и общее количество вызовов, осуществленных всеми модулями.

Если идентичность вызывающей функции не может быть установлена, то в таблице создается строка фиктивной (dummy) вызывающей функции, которая получает название `<spontaneous>`. Такое иногда происходит с обработчиками сигналов. Кроме того, фиктивная функция почти всегда присутствует в качестве функции, вызывающей функцию `main()`.

Здесь были охвачены только основные свойства *gprof*. Для получения более подробной информации следует обратиться к документации (manpage).

Предостережения

При профилировании приложений необходимо учитывать следующее.

Механизм измерений осуществляет замеры в фиксированные интервалы работы программы. Если программа не выполняется, а механизм измерений запущен, то данные не будут собраны. Представим процесс, генерирующий рабочий набор (working set), размер которого намного больше емкости физической памяти. Значительная часть системного времени будет затрачена на обслуживание перемещения страниц с диска в память и обратно. Такие действия не рассматриваются как работа программы и не будут отражены в профиле. С другой стороны, такое поведение может пойти на пользу: результаты профилирования будут достаточно схожими вне зависимости от нагрузки на систему во время сбора данных.

Сам процесс измерений характеризуется статистическими погрешностями. Если функция выполняется за очень короткий промежуток времени, то она может быть «пропущена». Если ожидалось, что функция появится только один раз, то возможно, что она не обнаружится вовсе или попадется несколько раз. Другое дело – граф вызовов и количество вызовов функций, которые получаются на основе точного подсчета и потому будут безошибочными.

Одно общее правило гласит: показатели времени работы будут точными, если время прогона значительно больше периода измерений (по меньшей мере на два порядка).

Некоторые данные графа вызовов подсчитываются приблизительно, поскольку непосредственной информации о них нет. Допущение состоит в том, что время, затраченное на выполнение функции, не ассоциируется с функциями, вызывающими ее. Например, если функция `WriteBook()` работала в общей сложности 60 секунд и вызывалась 10 раз, причем 3 вызова исходили из `CallFromEditor()`, то доля `WriteBook()` в периоде времени потомков `CallFromEditor()` якобы составляет 18 секунд. Однако такое допущение не всегда верно. Пусть другая функция, `OnTrain()`, вызывает `WriteBook()` во всех остальных случаях, причем `WriteBook()` выполняется очень быстро (как-никак, трудно успеть много написать в поезде[1]). В этом случае почти все время, затраченное в `WriteBook()`, следует приписать к `CallFromEditor()`, но *gprof* не знает об этом, и поэтому вслепую – и совершенно неверно – относит 42 секунды времени работы `WriteBook()` к `OnTrain()`.

Примеры оптимизации

Настало время обратиться к примерам оптимизации. Методы, представленные здесь, служат двум целям. Первая – объяснить, что пред-

[1] Авторы обыгрывают названия функций. `WriteBook()` – «написать книгу», `OnTrain()` – «на поезде». – *Примеч. перев.*

ставляет собой оптимизация. Тогда будет понятно, приведет ли особый ключ оптимизации, задаваемый при запуске компилятора, к повышению производительности. Вторая – описать, каким образом в некоторых случаях можно проводить оптимизацию вручную. Иногда это необходимо, если компилятор не способен сделать предположение о полезности оптимизации.[1] Это вовсе не значит, что такую оптимизацию необходимо проводить повсюду! Обычно компиляторы достаточно надежны. Методы измерения времени, примененные к критическим участкам кода, помогут определить случаи, когда компилятору нужно содействие.

В рамках книги описать все методы оптимизации невозможно, поэтому сосредоточимся на самых распространенных и важных из них. По существу, эти методы можно разбить на три категории: оптимизация арифметических действий, циклов и операций со строками.

Примеры кода будут представлены на C. Тем, кто интересуется настройкой приложений Java, следует обратиться к книге Джека Шираза (Jack Shiraz) «Java Performance Tuning» (Java: настройка производительности), O'Reilly. По этой теме есть замечательное руководство IBM – «Optimization and Tuning Guide for Fortran, C and C++» (Руководство по оптимизации и настройке для Фортрана, C и C++).

Арифметика

Один из основных методов оптимизации арифметических действий – замена последовательности операций на эквивалентные, но менее сложные операции. Например, компилятор может заменить строку кода

```
x = pow (y, 5);
```

на строку

```
x = y * y * y * y * y;
```

Дело в том, что во многих случаях повторяющиеся умножения требуют меньших усилий, чем возведение в степень. Такое преобразование называется *снижением мощности* (*strength reduction*). Безусловно, это самый распространенный метод оптимизации арифметических действий.

Другой типичный пример снижения мощности – замена деления на умножение на обратную величину. Например, строку кода

```
x = y / z;
```

можно заменить на строку

```
x = y * (1 / z);
```

[1] По умолчанию компиляторы пессимистичны в отношении допустимости оптимизации. Зачастую программист может взглянуть на фрагмент кода и за счет визуального анализа и глубокого понимания программы определить, будет ли оптимизация полезной.

На многих платформах это приведет к увеличению производительности просто потому, что инверсия и умножение выполняются намного быстрее деления.

В коде с интенсивными математическими операциями можно применить еще один хороший прием: компоновать программу с высокопроизводительной математической библиотекой производителя, если таковая есть. Такой подход может дать фантастический прирост производительности за счет вызова оптимизированных ассемблерных процедур для базовых математических функций.

Циклы

Заметим, что современные компиляторы применяют все эти методы оптимизации цикла автоматически. Однако компиляторам положено быть слегка консервативными (стремиться скорее к «правильному» коду, чем к быстрому). Ручная реализация этих методов может облегчить применение других, более сложных методов оптимизации, которые возьмет на себя сам компилятор.

Слияние циклов (*loop fusion*) – это сведение операторов из нескольких циклов в один.

Неоптимизированный код	**Оптимизированный код**

`for (i = 0; i <= 1024; i++) {`
`  x = x * a[i] * b[i];`
`}`
`for (j = 0; j <= 1024; j++) {`
`  y = y * a[j] * b[j];`
`}`
	`for (i = 0; i <= 1024; i++) {`
	`  x = x * a[i] * b[i];`
	`  y = y * a[j] * b[j];`
	`}`

Такой метод не только наполовину сокращает издержки цикла, но и дает компилятору больше возможностей для перекрытия команд, что позволяет воспользоваться способностью базового микропроцессора выполнять параллельные вычисления. Однако необходимо быть внимательным, поскольку при такой оптимизации производительность может снизиться из-за увеличения промахов кэша (например, когда в каждом цикле происходит обращение к большому участку кэша данных). Кроме того, дело может усложниться из-за наличия взаимосвязей данных. Необходимо удостовериться, что выполнение второго цикла не зависит от завершения выполнения первого.

Перемещение инвариантных управляющих операторов (*floating invariant flow control*) предполагает перенос за пределы цикла тех операторов управления последовательностью команд, которые не изменяются между итерациями цикла.

Неоптимизированный код	Оптимизированный код

`for (i = 0; i <= 1024; i++) {`
`  for (j = 0; j <= 1024; j++) {`
`  if (a[i] >= 64) {`
`  b[i] = -a[i];`
`  }`
`  x = x + a[j] - b[j];`
`  }`
`}`
	`for (i = 0; i <= 1024; i++) {`
	`  if (a[i] >= 64) {`
	`  b[i] = -a[i];`
	`  }`
	`  for (j = 0; j <= 1024; j++) {`
	`  x = x + a[j] - b[j];`
	`  }`
	`}`

Большинство компиляторов делают это автоматически. Однако если цикл содержит вызовы других процедур либо инвариантность скрыта в сложных циклах, то компилятор может не справиться.

При *развертке цикла* (*loop unrolling*) функциональная часть цикла трансформируется в повторяющиеся операторы. При этом пропорционально снижается количество итераций.

Неоптимизированный код	Оптимизированный код

`for (i = 0; i <= 1024; i++) {`
`  for (j = 0; j <= 4; j++) {`
`  a[i] = b[i, j];`
`  }`
`}`
	`for (i = 0; i <= 1024; i++) {`
	`  a[i] = b[i, 1];`
	`  a[i] = b[i, 2];`
	`  a[i] = b[i, 3];`
	`  a[i] = b[i, 4];`
	`}`

В основном такой прием способствует организации параллельной обработки как в компиляторе, так и в микропроцессоре.

Движение по индексу (*stride*) основано на связи между расположением элементов массива в памяти и порядком, в котором они обрабатываются. Например, к массиву из 32-битных целых чисел доступ идет с шагом индекса 1, если обращение к элементам происходит за счет монотонного увеличения значения правого индекса.[1]

Неоптимизированный код	Оптимизированный код

`for (i = 0; i <= 1024; i++) {`
`  for (j = 0; j <= 16; j++) {`
`  a[j] = b[j, i];`
`  }`
`}`
	`for (j = 0; j <= 16; i++) {`
	`  for (i = 0; i <= 1024; i++) {`
	`  a[j] = b[j, i];`
	`  }`
	`}`

Чем меньше шаг индекса, тем выше производительность программы вследствие оптимального использования иерархии кэша.

[1] Индекс *j*. – *Примеч. перев.*

Строки

Время выполнения стандартных библиотечных функций C пропорционально длине их операндов. Довольно часто такие функции вызываются из циклов. На обработку таких вызовов уходит немало времени процессора. Вот что можно предпринять для повышения быстродействия:

- Вместо того чтобы применять `strlen()` к самой строке при каждой итерации цикла, нужно перед циклом создать временную переменную и присвоить ей результат вызова `strlen()`, а затем двигаться по циклу, добавляя и удаляя символы.
- Вероятно, `strlen()` – не самый быстрый способ проверки пустоты строки. Если размеры строк велики, то `strlen()` будет проверять каждый символ, пока не дойдет до символа конца строки. Поэтому такую проверку лучше заменить на `*s=='\0'`. Тогда будет проверяться только первый символ и сократится количество вызовов функций. Точно так же, если необходимо очистить строку, то `*s ='\0'` будет намного быстрее, чем `strcpy(s,"")`.[1]
- `strcat()` проходит по всей длине строки при каждом вызове. Поэтому если при продвижении вперед (как в случае с `stlen()`) сохранять длину строки, то к концу строки будет готов индекс и можно будет воспользоваться функциями `strcpy()` и `memcpy()`.
- Если с помощью `strcmp()` сравниваются строки, содержащие символы естественного языка, то время выполнения такого сравнения можно сократить, если сверять первые символы строк перед вызовом этой функции. Вследствие статистически неоднородного распределения букв в естественных языках, шансы на эффективность такого метода выше, чем в других случаях. Однако такая оптимизация может быть неэффективной, поэтому следует учитывать контекст операций (например, при сравнении соседних строк из отсортированного списка это не принесет никакой пользы).

Взаимодействие с компиляторами

Хорошо известно, что в компиляторах есть большое количество ключей, назначение которых может сбивать с толку. По существу, необходимо применять лишь малую их часть. Однако правильное задание ключей при вызове компилятора может значительно повысить произво-

[1] Оба эти способа плохи, если требуется именно очистить строку, а не придать ей нулевую длину. Дело в том, что в C строка – это последовательность байтов и указатель на нее. Вышеуказанный способ запишет признак конца строки в первый байт строки. Однако если после этого напрямую обратиться к третьему символу строки s вот так: `c=s[2]`, то результатом будет прежнее значение третьего символа строки. Чтобы очистить строку, надо либо освободить память, которую она занимает с помощью `free()`, либо заполнить всю эту строку байтами с кодом 0. – *Примеч. науч. ред.*

дительность приложения. В то же время для надлежащей оптимизации приложений вовсе не нужно полностью понимать работу компилятора.[1]

В средах Sun и Linux наиболее распространены два компилятора. Первый компилятор – это Forte Developer, продаваемый Sun в качестве обновления для ныне устаревших компиляторов Sun WorkShop. Дополнительная информация, а также пробная версия доступны по адресу *http://www.sun.com/forte/*. Второй компилятор – это бесплатная программа GNU *gcc*. Главные преимущества компилятора *gcc* состоят в том, что он бесплатен и перенесен на множество различных платформ. Версии *gcc* существуют почти для всех систем, даже для малоизвестных. По мнению авторов, компиляторы Forte Developer генерируют значительно лучший код. Кроме того, они намного лучше документированы и с ними легче работать. Большей частью внимание здесь будет сосредоточено именно на ключах компиляторов Forte. Вне зависимости от выбора компилятора важно помнить, что компиляторы постоянно оптимизируются и обновляются, поэтому следует применять самые последние версии. Нет ничего необычного в том, что с выходом каждой новой версии компилятора быстродействие увеличивается на 10–15%.

Вот два предостережения, относящиеся к ключам компиляторов. Первое предостережение – порядок задания ключей важен. Последующие ключи перекрывают более ранние. Это особенно важно учитывать при использовании ключа *-fast*. Задание ключей *-fast -xO4* для компиляторов Forte Developer 6 приведет к меньшему уровню оптимизации, нежели один ключ *-fast*. Второе предостережение призывает к внимательности – не перед каждым флагом компилятора нужны символы *-x*.

Типичная оптимизация: -fast

В системах Sun самым важным ключом компилятора является ключ *-fast*. В зависимости от языка программы и используемой версии компилятора этот макрос расширяется по-разному. В табл. 8.4 представлена сводка по *-fast*.

Таблица 8.4. Расширения макроса -fast

Компилятор	C	Фортран 77	Фортран 90
Sun WorkShop 5.0	`-xO4 -single -fns -fsimple=1 -ftrap=%none -libmil -native`	`-xO4 -dalign -depend -fns -fsimple=1 -ftrap=%none -libmil -native`	`-xO3 -dalign -fns -ftrap=common -f -native -xlibmopt`

[1] При желании подробно изучить работу компилятора следует обратиться к книге A. Axo, P. Сэти и Дж. Ульмана (A. Aho, R. Sethi, J. Ullman) «Compilers: Principles, Techniques, and Tools» (Компиляторы: принципы, методы и инструменты), Addison Wesley. Книга известна под названием «Книга дракона» («dragon book»). Это одно из лучших руководств по теоретическим основам компилятора.

Таблица 8.4 (продолжение)

Компилятор	C	Фортран 77	Фортран 90
Forte Developer 6	-O5 -single -xmemalign=8s -fns -fsimple=2 -ftrap=%none -xlibmil -native -xprefetch=no -xvector=no	-O5 -dalign -depend -xpad=local -fns -fsimple=2 -ftrap=%none -xlibmil -native -xlibmopt -xvector=yes	-O5 -dalign -depend -xpad=local -fns -fsimple=2 -ftrap=common -f -xlibmil -native -xlibmopt -xvector=yes
Forte Developer 6 Update 1	-O5 -single -xmemalign=8s -fns -fsimple=2 -ftrap=%none -xalias_level=basic -xbuiltin=%all -xlibmil -native -xprefetch=no -xvector=no	-O5 -dalign -depend -xprefetch -xpad=local -fns -fsimple=2 -ftrap=%none -xlibmil -native -xlibmopt -xvector=yes	-O5 -dalign -depend -xprefetch -xpad=local -fns -fsimple=2 -ftrap=common -f -xlibmil -native -xlibmopt -xvector=yes

Большинство этих ключей здесь не будут обсуждаться. Достаточно сказать, что они представляют неплохую смесь базовых оптимизационных методов.

Уровень оптимизации: -xO

Наиболее часто настраиваемый ключ оптимизации – это ключ глобального управления оптимизацией *-xOn*, где *n* определяет уровень оптимизации (от 1 до 5 включительно). В компиляторах Forte Developer 6 для программ на C уровнем оптимизации по умолчанию является *-xO2, а* для программ на Фортране – уровень *-xO3*. В табл. 8.5 обобщены сведения по каждому уровню оптимизации.

Таблица 8.5. Сводка по уровням оптимизации для компиляторов Forte

Уровень	Описание
-xO1	Только основные методы локальной оптимизации
-xO2	Значение по умолчанию для C. Уровень 1 плюс методы глобальной оптимизации: алгебраическое упрощение, устранение подвыражений, оптимизированное выделение регистров, устранение невыполняемого кода, распространение констант, устранение хвостовых вызовов (tail-call)[a]
-xO3	Значение по умолчанию для Фортрана. Уровень 2 плюс методы оптимизации циклов (развертка и слияние). Программная конвейерная обработка

[a] Если указан ключ *-g*, то для облегчения отладки хвостовые вызовы не устраняются, если уровень оптимизации ниже *-xO4*.

Уровень	Описание
-xO4	Уровень 3 плюс встраивание функций и более активные методы глобальной оптимизации
-xO5	Наивысший уровень оптимизации. Лучше всего сочетать этот уровень с применением профилирования для получения обратной связи (см. раздел «Применение профилирования для получения обратной связи» далее в этой главе)

Компилятор GNU *gcc* по умолчанию не оптимизирует код. В *gcc* есть три уровня оптимизации, которые представлены в табл. 8.6.

Таблица 8.6. Сводка уровней оптимизации gcc

Уровень	Описание
-O1	Только базовые методы локальной оптимизации
-O2	Любые методы оптимизации, не задействующие улучшение скорости за счет увеличения размера кода. В первую очередь, исключены развертка и встраивание функций. (Включение развертки цикла выполняется с помощью *-funroll_loops*.)
-O3	Уровень оптимизации 2 плюс встраивание функций

Задание структуры системы команд: -xarch

Ключ *-xarch* указывает, на основе какой структуры системы команд должна быть собрана программа. Для платформ SPARC существуют девять вариантов выбора, которые представлены в табл. 8.7.

Таблица 8.7. Применимые значения -xarch

Значение	Система команд	Разрядность	Ограничения
v7	SPARC V7 (без команды `fsmuld` и целочисленных команд `mul` и `div`)	32	Любая машина SPARC
v8a	SPARC V8 (без команды `fsmuld`)	32	Любая машина microSPARC-I или последующая
v8	SPARC V8	32	Любая машина SuperSPARC или последующая
v8plus	SPARC V9 (без VIS)	32	Любая машина UltraSPARC
v8plusa	SPARC V9	32	Любая машина UltraSPARC
v8plusb	SPARC V9 (с расширениями UltraSPARC-III)	32	Машины UltraSPARC-III

Таблица 8.7 (продолжение)

Значение	Система команд	Разрядность	Ограничения
v9	SPARC V9 (без VIS)	64	Любая машина UltraSPARC, запущенная в 64-разрядном режиме
v9a	SPARC V9	64	Любая машина UltraSPARC, запущенная в 64-разрядном режиме
v9b	SPARC V9 (с расширениями UltraSPARC-III)	64	Любая машина UltraSPARC, запущенная в 64-разрядном режиме

Вообще, наилучшая производительность 32-разрядных приложений в системах с процессорами класса UltraSPARC достигается с помощью ключа *-xarch=v8plusa* (или *v8plusb* в системах UltraSPARC-III). Если указать какую-либо предшествующую систему команд, то производительность приложений может существенно снизиться. Из тех систем команд, которые поддерживаются на всех платформах, где предполагается запускать приложение, следует выбрать новейшую. Возможно, будет полезно скомпилировать несколько версий программы. Дело в том, что некоторые методы оптимизации доступны только на определенных платформах.

Если приложения компилируются с ключом *-fast* на платформе UltraSPARC, причем значение *-xarch* либо не указывается, либо указывается *-xarch=native*, то будет использовано значение *v8plusa* или *v8plusb*. При этом будет сгенерирован код, который не будет работать на машинах с процессорами, предшествующими UltraSPARC. Вместо этого будет выдано сообщение об ошибке:

```
cc: Warning: -xarch=native has been explicitly specified, or implicitly
specified by a macro option, -xarch=native on this architecture implies -
xarch=v8plusa which generates code that does not run on pre-UltraSPARC
processors
(cc: Предупреждение. -xarch=native был указан явно или с помощью макроса. В
этой архитектуре -xarch=native подразумевает -xarch=v8plusa. При этом
генерируется код, который не работает на процессорах-предшественниках
UltraSPARC.)
```

Это просто предупреждение, которое ни на что не повлияет. Для того чтобы избежать выдачи этого сообщения, нужно явно указать *-xarch=v8plusa* (или любое другое значение, кроме *native*).

Задание архитектуры процессора: -xchip

Явное указание типа процессора, который будет применяться для запуска приложения, дает компилятору много информации. Большей частью эти данные нужны для оптимального планирования команд и

обслуживания переходов. Например, от типа процессора сильно зависит количество циклов задержки между загрузкой значений данных и использованием их в последующих вычислениях (так называемая *задержка загрузки–использования*, *load-use delay*). Правильный выбор значения ключа *-xchip* может принести большую пользу. Для систем SPARC есть тринадцать значений, которые представлены в табл. 8.8.[1]

Таблица 8.8. Применимые значения -xchip

Значение	Архитектура
old	Очень старые процессоры (процессоры, предшествующие SuperSPARC)
super	Микросхемы SuperSPARC (любая микросхема SuperSPARC с частотой менее 60 МГц: SM61 или медленнее)
super2	Микросхемы SuperSPARC-II (любая микросхема SuperSPARC с частотой более 75 МГц: SM71 или лучше)
micro	Микросхемы MicroSPARC-I (SPARCclassic, LX)
micro2	Микросхемы MicroSPARC-II (Voyager, SPARCstation 4, SPARCstation 5)
hyper	Микросхемы HyperSPARC-I
hyper2	Микросхемы HyperSPARC-II
ultra	Микросхемы UltraSPARC-I (все процессоры UltraSPARC-I с частотой менее 200 МГц)
ultra2	Микросхемы UltraSPARC-II
ultra2i	Микросхемы UltraSPARC-IIi (Ultra 5, Ultra 10 и т. д.)
ultra3	Микросхемы UltraSPARC-III
native	Текущая архитектура (будет применяться при компилировании) – подразумевается 32-разрядная среда
native64	Текущая архитектура (будет применяться при компилировании) – подразумевается 64-разрядная среда

Встраивание функций: –xinlining и –xcrossfile

Встраивание функций – это процесс вставки одной функции в тело другой функции, которая ее вызывала. За счет этого устраняются издержки при переходе от одного участка памяти к другому. Кроме того, у компилятора появляется больше возможностей организовать параллельную обработку.

[1] Кроме того, существуют значения для Solaris, работающих на платформе Intel. Наиболее полезное значение: *pentium_pro*.

В *gcc* оптимизация, связанная со встраиванием функций, включается с помощью *-finline-functions*.

Не стоит включать встраивание, если указываемый уровень оптимизации выше *-xO4*, поскольку на этом уровне уже организовано встраивание функций. В противном случае двойное указание выполнять встраивание может вызвать спады производительности. Однако указание *-xcrossfile* будет весьма кстати: процесс встраивания охватит функции, находящиеся в отдельных файлах исходного кода. Для наилучшего результата этот параметр следует применять вместе с *-xO4* или более высокими уровнями оптимизации.

Анализ взаимосвязи данных: -xdepend

Одна из задач компилятора – анализ взаимосвязи данных внутри циклов и последующее реструктурирование циклов при необходимости. Такое реструктурирование способствует развертке и другим методам оптимизации цикла, применяемым компилятором. За счет этого можно улучшить производительность. Кроме того, компилятор выполняет действия, способствующие *кэшированию блоками*.[1]

Вместе с параметром *-xdepend* следует задавать уровень оптимизации *-xO3* или выше. При указании *-fast* параметр *-xdepend* задается по умолчанию.

Векторные операции: -xvector

При указании ключа *-xvector* компилятор применяет оптимизированную векторную математическую библиотеку, быстродействие которой может быть существенно выше, чем у ее скалярного эквивалента. Такой ключ особенно эффективен, когда приложение неоднократно вызывает встроенные средства математической библиотеки, например `log`, `sin` и `exp`.

При задании *-xvector* параметр *-xdepend* так же включается автоматически.

Размер по умолчанию констант с плавающей точкой: -xsfpconst

По умолчанию компилятор Forte C рассматривает константы с плавающей точкой как `double`, если они явно не указаны как `float`. В результате во многих командах преобразования, использующих константы с плавающей точкой, зачастую необходимо переходить от двойной точ-

[1] Кэширование блоками – это метод, подразумевающий разбиение вычислений на части. При этом данные из каждой части, к которым происходит обращение, умещаются в кэше процессора. Таким образом повышается эффективность кэша.

ности к обычной. Это особенно актуально в программах, где выполняется большое количество делений или извлечений квадратного корня, которые в случае двойной точности требуют почти в два раза больше циклов. При задании ключа *-xcfpconst* такие константы будут рассматриваться как константы с обычной точностью.

Упреждающая выборка данных: –xprefetch

Упреждающая выборка данных (*prefetching*) – это метод, позволяющий процессору совмещать выполнение команд с выборкой данных из памяти. Такая схема особенно полезна для приложений, характеризующихся ограниченной задержкой, которые выполняются в системах с большой величиной задержки. Примером могут служить приложения, в которых повторяется один и тот же шаблон доступа (например, много больших циклов) при обращении к оборудованию с большой задержкой. При задании ключа *-xprefetch* компилятор вставляет в программу специальные команды упреждающей выборки. Поскольку этот метод во многом зависит от архитектуры, то этот ключ лучше всего применять вместе с ключами *-xchip* и *-xtarget*.

Заметим, что процессоры UltraSPARC-I допускают команды упреждающей выборки, но на самом деле ничего с ними не делают. Поэтому программа, компилированная с ключом *-xprefetch*, будет работать на всех системах с UltraSPARC. При этом в системах UltraSPARC-I от такого ключа не будет никакой отдачи, в системах UltraSPARC-II возможны существенные улучшения, а в системах UltraSPARC-III – значительные улучшения вследствие встроенных кэшей, предназначенных для упреждающей выборки.

Поверхностный обзор ключей компилятора

Здесь представлен поверхностный обзор ключей компилятора для особых видов приложений. Как обычно, влияние ключей необходимо тестировать, поскольку они могут не подходить для конкретной платформы. При рассмотрении всех ключей подразумевается система на базе UltraSPARC:

- Для приложений, в которых необходимо строго придерживаться операций с плавающей точкой (IEEE 754), следует попробовать *-fast -xarch=v8plus -fsimple=0*.
- Для приложений на C, в которых аргументы указателей на функции не являются псевдонимами (aliases) друг друга, можно попробовать *-fast -xarch=v8plus -xrestrict*. Если приложения следуют правилам разыменования указателей ISO C 1999, то также следует указать *-xalias_level=std*.
- Для приложений на Фортране подойдет *-stackvar*. Этот ключ заставляет компилятор размещать локальные переменные в стеке программы. Можно попробовать *-fast -xarch-v8plus -stackvar*.

- Для приложений, запускаемых в системах UltraSPARC-III, следует включить упреждающую выборку с помощью *-xprefetch*. Например, так: *-fast -xdepend -xchip=ultra3 -xprefetch*.

Применение профилирования для получения обратной связи

В компиляторе есть встроенный механизм для целевой активной оптимизации наиболее часто выполняемых кусков программы. Такая схема называется *профилированием для получения обратной связи* (*profile feedback*). Она опирается на данные о частоте выполнения кода, которые применяются для дальнейшей оптимизации при следующем компилировании.

Для применения оптимизации на основе такой обратной связи приложение необходимо собрать с ключом *-xprofile=collect:name*, где *name* – название исполняемого файла. При запуске приложения будет создан каталог *name.profile*, который будет содержать данные запуска. Такой «обучающий» запуск займет больше времени, чем обычный запуск приложения. Наконец, приложение следует пересобрать с ключом *-xprofile=use:name*. Данные, собранные при первом запуске приложения, будут учтены компилятором при дальнейшей оптимизации.

В примере 8.6 представлен исходный код, который демонстрирует полезность такого профилирования.

Пример 8.6. profiling.c

```
/* profiling.c */
#include <stdio.h>
int
main(int argc, char **argv)
{
        int i, n = 512, sum = 0;
        for (i = 0; i < 100000000; i++) {
                if (i > n) {
                        sum++;
                } else {
                        sum--;
                }
        }
        printf ("sum: %d\n", sum);
}
```

Соберем *profiling.c*, а затем для улучшения его производительности применим профилирование, как показано в примере 8.7.

Пример 8.7. Профилирование для улучшения производительности приложения

```
% /opt/SUNWspro/bin/cc -fast -xarch=v8plusa profiling.c -o profiling \
-xprofile=collect:profiling
% ./profiling
sum: 99998974
% timex ./profiling
sum: 99998974
real        3.39
user        3.38
sys         0.01
% /opt/SUNWspro/bin/cc -fast -xarch=v8plusa profiling.c -o profiling \
-xprofile=use:profiling
% timex ./profiling
sum: 99998974
real        0.69
user        0.67
sys         0.01
```

Замечательное достижение! Конечно, не все приложения можно улучшить до такой степени.

Заключение

В этой главе обсуждались методы написания быстрого кода, анализа алгоритмов, измерения времени выполнения приложений, а также профилирование для нахождения «горячих участков». Кроме того, здесь было рассказано о том, как добиться эффективности работы компиляторов. По сути, это лишь краткое введение в обширную область оптимизации кода. Желающим углубиться в эту тему можно порекомендовать замечательное руководство по оптимизации приложений – книгу Райэта Гарга (Rajat Garg) и Ильи Шарапова (Ilya Sharapov) «Techniques for Optimizing Applications: High Performance Computing» (Методы оптимизации приложений: высокопроизводительные вычисления), изданную Prentice Hall.

Главное, что следует вынести из всей этой главы: при написании кода необходимо учитывать методы рефакторинга, а также выделять время для реализации быстрых алгоритмов. Это сердцевина настройки приложений.

9

- *Горячая пятерка советов по настройке*
- *Рецепты первоочередной настройки*

Первоочередная настройка

Пойман во времени... окружен злом... вперед.

Слоган из фильма «Армия тьмы»

Порой у администратора нет времени на постижение всех тонкостей. Его осаждают пользователи и руководство. Он просто обязан незамедлительно справиться с возникшей задачей. Возможно, он замещает коллегу, который уехал за город накануне, а сегодня десяток пользователей осаждает их общий офис и требует улучшения производительности. Возможно, это системный администратор, работающий по контракту. Ему нужен удобный перечень событий, которые могут происходить в системе. Он прочел эту книгу от корки до корки и ищет краткую шпаргалку для того, чтобы отложить в сторону блокнот, полный заметок.

В этой главе будут обобщены методы, которые помогают ответить на два обычных, но непростых вопроса: как выявлять узкие места и повышать эффективность решения задач, а также какую вообще настройку необходимо провести на новой системе для того, чтобы она работала с максимальной производительностью?

Однако прежде чем обратиться к этой теме, призовем к бдительности.

Внимание! Эта глава содержит сведения повышенной опасности![1] Решения и подходы, представленные здесь, излагаются без объяснений. А значит, системный администратор должен быть очень внимателен, чтобы не превратиться в эксперта по производительности систем, которые построили приверженцы культа Карго (см. введение к главе 2)!

---

[1] Klatuu, Barata, Nicto! (В фильме «Армия тьмы» эти слова возвращают к жизни главного героя. В фильме «День, когда Земля остановилась» их произносит гигантский робот: «Я пришел к вам с миром». – *Примеч. перев.*)

Горячая пятерка советов по настройке

Если система работает плохо, то следует задать себе пять вопросов. Отметим, что во всех представленных инструкциях подразумевается, что данные накапливаются за 30-секундные интервалы (если иное не оговорено).

Где узкое место дисковой подсистемы?

Почти каждая система, работающая медленно, характеризуется значительной нагрузкой ввода-вывода. Особого внимания заслуживают диски, у которых «время обслуживания» больше 50 мс, *и* диск, нагруженный больше, чем на несколько процентов. На самом деле показатель времени обслуживания следует называть «временем ответа». Значение этого показателя соответствует паузе между выдачей запроса на чтение и завершением его обработки. Очень часто это критический участок в работе пользовательских приложений. Если диск сильно перегружен, то вполне возможно, что время ответа будет измеряться в тысячах миллисекунд (это не опечатка). В Solaris такие данные можно получить с помощью *iostat -xnP 30*:

```
# iostat -xnP 30
...
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 c0t0d0s0
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 c0t0d0s2
    0.7    2.7    5.3    9.5  0.0  0.1    0.0   16.1   0   2 c0t0d0s3
...
```

В Linux следует инсталлировать дополнительный пакет, содержащий утилиту *iostat*, а затем проанализировать колонку `await`:

```
# iostat -x -d 30
Linux 2.4.2-2 (aiua)      07/29/2001
...
Device: rrqm/s wrqm/s  r/s  w/s  rsec/s  wsec/s avgrq-sz avgqu-sz  await svctm  %util
hde       4.79  1.40 86.95 3.26 158.87   37.43     2.18     1.01  11.22  8.14  7.34
hde1      0.00  0.00 82.15 0.00  82.15    0.00     1.00     0.46   5.64  5.64  4.63
hde2      4.79  1.40  4.80 3.26  76.70   37.43    14.16     0.55  68.05 34.40  2.77
hde5      0.00  0.00  0.00 0.00   0.02    0.00     8.00     0.00 100.00 100.00 0.00
hdg       9.85  2.36 36.69 5.20 371.85   60.86    10.33     8.60 205.22 45.61 19.11
hdg1      0.00  0.00  0.06 0.00   0.07    0.00     1.03     0.00   2.94  2.94  0.00
hdg2      9.85  2.36 36.63 5.20 371.78   60.86    10.34     8.60 205.54 45.68 19.10
...
```

В файловых системах UFS, нагрузка на которые обычно невелика, время обслуживания иногда может быть высоким вследствие работы *fsflush*. Поскольку файловая система, по сути, «бездействует», то это

не доставляет хлопот. В Linux с таким явлением авторы никогда не сталкивались. Что касается систем Solaris с емкостью памяти более 512 Мбайт, то в них кэш индексных дескрипторов достаточно большой. Однако в системах с меньшей емкостью памяти увеличение размера этого кэша может способствовать тому, что количество операций дискового ввода-вывода, необходимых для управления файловой системой, будет снижено.

По существу, для того чтобы снизить издержки дискового ввода-вывода, нужно по возможности распределить нагрузку. «Горячие» каталоги или файлы лучше всего переместить на другой, менее нагруженный диск. Кроме того, можно задействовать дисковый массив с кэшем на базе энергонезависимой памяти (NVRAM), что позволит ускорить операции записи.

Достаточно ли памяти в системе?

В Solaris есть один признак, по которому можно судить о нехватке памяти. Это поле `sr` в выводе команды *vmstat*. Следует помнить, что первая строка вывода *vmstat* не несет никакого смысла! Что касается других полей, то для ответа на этот вопрос они не представляют ценности, особенно в Solaris 7 и предшествующих версиях. По этой теме существует много неверных суждений. Вот два наиболее распространенных:

- Размер свободного списка не является индикатором нехватки памяти, поскольку Solaris будет расходовать всю незадействованную память для кэширования последних используемых файлов.
- Количество подкачек (page-ins) и откачек (page-outs) страниц в секунду – это плохой показатель, так как весь ввод-вывод файловых систем в Solaris обслуживается с помощью пейджинга. Тысячи загруженных и откачанных килобайт как раз и означают, что система работает.

К сожалению, по сравнению с Solaris 7 вывод команды *vmstat* в Solaris 8 изменился. Теперь информация выводится «правильно»: страницы, задействованные для кэширования файловой системы, представлены как свободные (поскольку так оно и есть). Поэтому вопрос в Solaris 7: «Куда делась вся свободная память?» после перехода на Solaris 8 звучит так: «Откуда возникла вся эта свободная память?»

Индикатором нехватки памяти по-прежнему является интенсивность, с которой страничный сканер просматривает страницы для того, чтобы выяснить, можно ли их высвободить. При нехватке памяти в Solaris 7 и предшествующих версиях предел, по достижении которого необходимо установить больше памяти, составляет 250 страниц в секунду. В системе Solaris 8 *любая* активность сканера страниц означает нехватку памяти.

Не перегружены ли процессоры?

Лучший показатель перегруженности процессора – длина очереди запуска. Как в Solaris, так и в Linux получить такие сведения можно с помощью команды *vmstat*. Для этого в ее выводе следует проанализировать значения колонки `procs r`. Если это значение примерно в четыре раза больше количества процессоров в системе, то процессы, судя по всему, слишком долго ожидают выделения для них времени.

В Solaris количество взаимных исключений можно определить по полю `smtx` вывода команды *mpstat*. Если это значение велико (более чем в 250 раз превышает количество процессоров), то процессоры следует заменить на более быстрые, а не добавлять другие.

Не блокируются ли процессы при дисковом вводе-выводе?

Если процессы блокируются, то это свидетельствует об узком месте в дисковой подсистеме. Количество блокированных процессов можно увидеть в поле `procs b` команды *vmstat*. Всякий раз, когда в системе есть блокированные процессы, все время простоя процессора представлено как время ожидания ввода-вывода. Если количество блокированных процессов приближается к количеству процессов в очереди запуска и даже превышает его, то в дисковой подсистеме есть серьезное узкое место. Отправной точкой в его нахождении будет команда *iostat*.

В среде с преобладающей пакетной обработкой можно увидеть изрядное количество блокированных процессов. Это не повод для беспокойства. В то же время улучшение производительности дисковой подсистемы улучшит эффективность обработки пакетных заданий.

Системное время намного больше пользовательского?

Если утилиты, оценивающие потребление процессора (*vmstat*, *mpstat* и любые другие), сообщают, что бульшую часть времени процессор обслуживает запросы ядра (системное время), а не пользовательские процессы (пользовательское время), то это означает, что в системе не все ладно. Поскольку в Solaris сервер NFS полностью работает внутри ядра, то в этом случае системное время будет намного больше пользовательского. То же самое относится к дополнительному пакету *knfsd* в Linux.

Однако если система не является сервером NFS, то выяснить, в чем же дело, может быть трудно. Наилучший подход – определить горячую десятку процессов исходя из показателя потребления времени процессора (с помощью *prstat*, *top* или *ps*). После этого следует применить утилиту трассировки процессов, подобную *truss*. С ее помощью можно увидеть, какие системные вызовы обслуживались при выполнении

процессов. Если дисбаланс система/пользователь сопровождается большим количеством взаимных исключений (колонка `smtx` вывода *mpstat*), то необходимо задуматься о замене процессоров на более быстрые (но не добавлять новые).

Рецепты первоочередной настройки

В этом разделе представлены рецепты по настройке производительности. Конечно, это рекомендации общего назначения, поэтому их можно видоизменять для каждой конкретной среды.

Наилучший рецепт для любой машины, будь то сервер или рабочая станция, – инсталлировать новейшую версию операционной системы и своевременно устанавливать обновления. Обновления, нацеленные на улучшение производительности, довольно типичны и порой могут дать впечатляющие результаты. Кроме того, они могут позволить применять новые алгоритмы в предшествующих версиях операционных систем (например, приоритетный пейджинг в Solaris).

Однопользовательская рабочая станция разработчика

Рассмотрим типичный сценарий настройки компьютерной системы, находящейся в распоряжении одного разработчика. Зачастую таким пользователям необходима высокая производительность задач, для которых характерно интенсивное потребление процессора и файловой системы. Примером может служить компилирование. Однако здесь может не быть таких быстрых дисков, как на большом сервере.

Файловые системы

Авторы предпочитают разбивать системный диск на два раздела: один для пространства свопинга, а другой для всего остального. Раздел свопинга должен иметь наименьший номер на диске. Для объяснения причины следует обратиться к разделу «Минимизация времени поиска на уровне файловой системы» главы 5. Однако многие администраторы с такой разбивкой не согласны – это дело вкуса. Если в системе присутствуют несколько дисков, то для распределения нагрузки необходимо предусмотреть пространство свопинга на каждом из них. В Solaris все файловые системы следует монтировать с параметром *logging*. Это снизит время восстановления в случае неполадок с питанием (см. раздел «Файловые системы с протоколированием» в главе 5).

Если предполагается, что монтируемые файловые системы NFS будут в основном содержать неизменяемые файлы, например прикладные программы, то их нужно монтировать только для чтения. Это позволит избежать операций записи времени доступа к файлам NFS. Кроме того, необходимо применять файловую систему CacheFS, если она до-

ступна для данной платформы. С ее помощью можно кэшировать файлы, которые используются наиболее часто. Это особенно полезно для каталогов приложений. Файловую систему CacheFS не следует применять для почтовых каталогов. Однако она может быть полезна для домашних каталогов – в зависимости от соотношения файловых операций чтения и записи. Если приложение постоянно выполняет большие записи в файловой системе NFS, то применять CacheFS в этом случае не следует. Более подробные сведения о реализации CacheFS можно найти в разделе «Кэширующие файловые системы (CacheFS)» главы 5.

Пространство свопинга

Выбор размера пространства свопинга достаточно сложен. Если нет рекомендаций от производителя программного обеспечения, то следует предусмотреть не менее 128 Мбайт виртуальной памяти. Разделы свопинга более предпочтительны, чем своп-файлы. Диски дешевы, а значит, этим стоит воспользоваться – лучше больше, чем меньше. В основном для настольных машин никогда не нужно более 512 Мбайт пространства свопинга.

Настройка ядра

В большинстве операционных систем работа ядра хорошо отлажена, поэтому настраивать предстоит не так уж и много. Одно исключение: при запуске Solaris 7 или предшествующих версий следует включить приоритетный пейджинг. Для этого в */etc/system* нужно добавить следующие строки:

```
*
* enable priority paging on pre-Solaris 8 systems
set priority_paging = 1
```

Подробное объяснение приоритетного пейджинга можно найти в разделе «Приоритетный пейджинг» главы 4.

Серверы рабочих групп

В данном контексте *сервер рабочих групп* – это машина, выполняющая различные задания по обслуживанию небольшого отдела. Это могут быть почтовые сервисы, хранение файлов, простой веб-сервер и т. д. В среде UNIX каталоги экспортированы как NFS. В сетях Windows необходимо поддерживать SMB/CIFS с помощью Samba. Кроме того, сервер может быть прокси-кэшем первой линии для веб-броузеров отдела. К нему могут подключаться любые тонкие клиенты, внешние терминалы (X terminals) и другие сетевые компьютеры. Вероятно, система подключена к сети через несколько соединений Fast Ethernet.

Рабочая нагрузка большей частью сводится к NFS и Samba. Базовый пакет sendmail, поставляемый с Solaris, способен поддерживать по

меньшей мере несколько сотен пользователей. Для его работы не требуется много настройки – если она вообще необходима. Если нужно поддерживать больше пользователей, то речь идет уже не о сервере рабочих групп. В то же время пакеты, подобные iPlanet Messaging Server, могут поддерживать миллионы пользователей. Кроме того, они основаны на более эффективных методах хранения сообщений. Скорее всего, веб-сервер также не будет сильно загружен. Однопроцессорная система на базе UltraSPARC-II или быстрого Pentium-3 легко справляется с такими задачами.

Память

Такая рабочая нагрузка не характеризуется интенсивным обращением к памяти. Вероятно, наиболее активной пользовательской программой является sendmail, а остаток оперативной памяти будет кэшем для файловых систем. Так как вследствие кэширования на стороне клиентов большинство файлов будут запрашиваться только один раз, то кэш файловой системы может быть небольшим. Все, к чему планируется частое повторное обращение для чтения (например, приложения), является замечательным кандидатом на монтирование как CacheFS на стороне клиентов.

Примерный порядок таков. Следует начать со 128 Мбайт памяти для самой системы, а затем добавлять по 64 Мбайт для каждого присоединенного интерфейса Fast Ethernet. Если система поддерживает внешние терминалы или SunRays, то для каждого из этих сервисов необходимо добавить около 128 Мбайт – в зависимости от заданий, которые будут запускать клиенты. Хорошее правило: дополнительно устанавливать на сервер столько памяти, сколько было бы установлено на настольной машине – до 32 Мбайт на каждого клиента.

Диски

При конфигурировании дисковой подсистемы следует обеспечить надежность данных и максимально ускорить операции записи. Если в системе присутствует аппаратный RAID (такой как Sun StorEdge A1000), то его лучше сконфигурировать как RAID 1+0 или RAID 5 исходя из того, насколько важна емкость дисковой памяти (см. раздел «Рецепты RAID» в главе 6). Если устанавливается программный RAID, то следует воспользоваться возможностями протоколирования, которые предоставляет DiskSuite. Параметр *logging* файловой системы UFS будет менее эффективен, поскольку приведет к снижению производительности. Файловые системы, в которых протоколирование ведется средствами DiskSuite, нужно размещать на выделенном диске со скоростью вращения не менее 7 200 rpm.[1] Аппаратные дисковые массивы можно на-

[1] Это может выглядеть как пустая трата пространства. На самом деле это не так. Здесь важна производительность диска, а не его емкость.

дежно монтировать, задействуя параметр *logging*, поскольку записи в протокол будут кэшироваться в памяти NVRAM контроллера массива.

В такой конфигурации необходимо зеркалировать системный диск.

Файловые системы

Авторы обычно создают один большой раздел root, как и в случае сервера рабочей группы. Однако каталог */var* должен находиться в отдельном разделе. Операции записи, относящиеся к нему, должны быть ускорены. Это не связано с файлами протоколирования. Речь идет об улучшении производительности таких операций, как доставка почты в почтовый ящик (перезапись пользовательского почтового файла после доставки почты – это существенная дисковая операция). Каталог */var* должен быть достаточно большим. Его точный размер зависит от видов рабочей нагрузки.

Для домашних каталогов пользователей следует создать отдельную файловую систему.

Пространство свопинга

Такая рабочая нагрузка не требует большого пространства для свопинга. Нескольких сотен мегабайт будет более чем достаточно.

Оптимизация NFS

При настройке серверов NFS следует принять во внимание три рекомендации: задействовать NFS версии 3, увеличить количество потоков сервера NFS, а в системах Solaris, кроме того, увеличить размеры буферов TCP.

Несколько алгоритмических изменений, проведенных в NFS версии 3, способствуют значительному улучшению производительности, особенно файловых операций записи. Для всех клиентов NFS файловые системы должны монтироваться с параметром *vers=3*. С помощью *nfsstat* можно выявить наличие на сервере вызовов NFS версии 2. Более подробные сведения о NFS версии 3 можно найти в разделе «NFS» главы 7.

Как уже было сказано, важно правильно выбрать количество потоков NFS ядра. Несмотря на то что значение по умолчанию является приемлемым, оно, безусловно, не способствует максимальной производительности. На самом деле этот параметр можно настраивать путем редактирования строки в */etc/init.d/nfs.server*, которая отвечает за запуск *nfsd*. Разумно начать со значения 128:

```
/usr/lib/nfs/nfsd -a 128
```

В системах Solaris размеры буферов приема и передачи TCP следует увеличить. Размерами этих буферов управляют параметры `tcp_recv_hiwat` и `tcp_xmit_hiwat`. С их значениями необходимо поэкспериментировать, однако хорошей отправной точкой будут значения 56 Кбайт (57 344)

для tcp_xmit_hiwat и 32 Кбайт (32 768) для tcp_recv_lowat. Для того чтобы избежать масштабирования окна TCP, эти значения не должны превышать 64 000. В веб-серверах Linux есть эквивалентные параметры, но их значения достаточно высоки и не требуют настройки. Для получения более подробных сведений можно обратиться к разделу «Буферы, отметки уровня и окна» главы 7.

Настройка ядра

Значение maxphys стоит увеличить, чтобы ядро более эффективно сбрасывало на диск смежные участки модифицированных файлов. Более подробно о maxphys можно прочесть в разделе «Взаимодействия между кэшем файловой системы и диском» главы 4.

Если в системе Solaris основная часть нагрузки приходится на NFS, а емкость памяти меньше 128 Мбайт, то нужно увеличить размеры кэша поиска имен каталогов и кэша индексных дескрипторов. Об этих кэшах можно прочесть в разделах «Кэш поиска имен каталогов (DNLC)» и «Кэш индексных дескрипторов» главы 5. Будет достаточно увеличить их до 8000:

```
*
* increase the dnlc and inode caches for low-memory servers
set ncsize = 8000
set ufs:ufs_ninode = 8000
```

В системах, предшествующих Solaris 8, необходимо включить приоритетный пейджинг. См. предыдущие рекомендации для однопользовательских рабочих станций или раздел «Приоритетный пейджинг» главы 4.

Веб-серверы

Похоже, есть определенная тенденция достраивать веб-серверы для предоставления статического контента. Производительность процессора UltraSPARC-II или быстрого Pentium-3 такова, что таким контентом можно легко заполнить 100-мегабитный сегмент Ethernet. С добавлением процессоров производительность системы линейно возрастает (это верно по крайней мере для четырех процессоров). Однако поскольку веб-контент становится все более динамическим, то производительность процессорного блока становится немаловажной. К сожалению, хороших правил для управления динамической нагрузкой HTTP нет – просто потому, что она может быть совершенно различной.

Память

Для быстродействия веб-сервера необходимо много памяти. Лучше всего веб-серверы работают при наличии больших буферов TCP (которые потребляют немало памяти, если есть много соединений) и организации эффективного кэширования файловой системы. Примерный

подход таков: следует начать с 256 Мбайт памяти и к этой величине добавить суммарный размер файлов, обращение к которым составляет 90% всех запросов (эти данные можно получить, исследовав файлы протоколирования). Тогда памяти для кэширования «самых горячих» страниц будет достаточно.

В системах, предшествующих Solaris 8, следует включить приоритетный пейджинг. См. раздел «Приоритетный пейджинг» в главе 4.

Диски

Для очень крупных веб-сайтов или для серверов, обслуживающих большое количество сайтов, вопросы хранения данных на диске становятся немаловажными. Один диск 7200 rpm способен обслуживать около 50 статических операций HTTP в секунду. Поэтому необходимо решить, какое максимальное количество операций в секунду необходимо поддерживать и исходя из этого разбивать диски на блоки (striping). Массивы RAID 5 будут отличным выбором. Для таких дисковых массивов характерна неплохая восстанавливаемость после отказов и хорошая производительность (в основном вследствие преобладания операций чтения).

На загруженных веб-сайтах может возникнуть необходимость создания блока и зеркала для каталога, содержащего файлы протоколирования HTTP. Такие файлы растут быстро, особенно во время пиковой нагрузки.

Файловые системы

Как и в случае сервера рабочих групп, авторы рекомендуют создание одного большого раздела root. Однако, как было замечено ранее, для протоколов HTTP необходимо создать выделенную файловую систему, а также приложить усилия к улучшению производительности операций записи.

Зеркалирование системного диска или отсутствие зеркала – это выбор системного администратора. В больших системах, где веб-серверы – это несколько компьютеров Netra T1,[1] или в случае других маломощных машин за коммутатором, уравновешивающим нагрузку, или для службы DNS, работающей по круговой системе, неполадки в одной машине не должны доставлять хлопот. Инфраструктура должна быть организована так, чтобы было достаточно заменить машину, переустановить программное обеспечение с помощью автоматизированных средств и включить ее в работу. Легче всего это реализовать, если предоставление контента веб-серверов организовано с помощью NFS.

[1] Более подробно о семействе Netra T1 можно узнать по адресу *http://sunsolve.sun.com/handbook_pub/Systems/Netra_t1_105/spec.html*. – *Примеч. науч. ред.*

Пространство свопинга

И вновь веб-серверам не нужно большое пространство свопинга. 512 Мбайт будет достаточно – для обработки запросов анонимной памяти, которая не была использована. Подробные сведения можно найти в разделе «Анонимная память» главы 4.

Сети

Настройка сети более сложна. Здесь внимание будет сосредоточено на статических операциях. Если веб-сервер обслуживает значительный объем динамического контента, то системному администратору необходимо разработать свои правила на основе конкретной рабочей нагрузки. Назначение табл. 9.1 – в общих чертах описать возможную пропускную способность сети. Например, при наличии тысячи пользователей, подсоединяющихся по коммутируемым линиям 56 Кбит, не стоит пытаться поддерживать более 400 операций HTTP в секунду. И наоборот, попытка обслуживать 1000 операций в секунду по дуплексному каналу Fast Ethernet, вероятно, обречена на неудачу.

Таблица 9.1. Использование сети клиентами HTTP

Сетевое соединение	Полоса пропускания (в секунду)	Максимальное количество статических операций HTTP в секунду
Коммутируемое соединение 56 Кбит	56 Кбит/с	0,4
ISDN	128 Кбит/с	1
768K SDSL (дуплекс)	768 Кбит/с	5
T1	1,5 Мбит/с	10
Кабельный модем	6	35
Ethernet	10	60
DS3	45 Мбит/с	300
Fast Ethernet (дуплекс)	200 Мбит/с	600
Gigabit Ethernet (дуплекс)	2000 Мбит/с	5000

При настройке производительности веб-сервера следует помнить еще о двух фактах. Во-первых, протокол SSL (также известный как Secure HTTP) – это тиран номер один для производительности. В случае его применения количество операций HTTP, которые можно поддерживать, следует разделить на десять. Сейчас на рынок выходит новое поколение плат-ускорителей SSL, однако авторы не имеют опыта работы с ними и не могут что-либо посоветовать. Во-вторых, программные па-

кеты, подобные ускорителю сетевого кэша (Network Cache Accelerator, NCA) в Solaris, работают в пространстве ядра и кэшируют входящие запросы HTTP (обычно статические). При этом по-прежнему необходимо копировать запросы из ядра в пользовательское пространство, «пробуждать» веб-сервер, извлекать данные из какого бы то ни было кэша, в котором располагается страница (или, еще хуже, с диска), а затем доставлять их ядру. Программа NCA просто кэширует статические веб-страницы в памяти ядра и сама отвечает на запросы, правда, намного быстрее. Вероятно, в программах NCA нового поколения будут реализованы методы для извлечения динамического контента. Другой продукт, решающий те же задачи, – Tux. Это высокопроизводительный веб-сервер для Linux, запускающий значительное количество своих копий в пространстве ядра. Отметим, что иногда эти пакеты могут работать неустойчиво. Поэтому их необходимо тщательно тестировать, прежде чем внедрять.

Настройка ядра

Что касается настройки ядра, то необходимо увеличить размеры очередей незавершенных и завершенных соединений. В Solaris этими размерами управляют параметры `tcp_conn_req_max_q0` и `tcp_conn_req_maxq` соответственно. По умолчанию их значения равны 1024 и 128. Признаком того, что эти значения следует увеличить, являются большие значения `tcpListenDrop` и `tcpListenDropQ0`, которые можно найти в выводе *netstat -sP tcp*. Значение параметра `tcp_conn_req_max_q0` не должно быть больше 10 000, а значение параметра `tcp_conn_req_max_q` не должно превышать значение `tcp_conn_req_max_q0`. В Linux эквивалентом параметра `tcp_conn_req_max_q0` является параметр `tcp_max_sy_backlog`, который можно найти в */proc/sys/net/ipv4*. Дополнительную информацию об очередях соединений можно почерпнуть в разделе «Инициирование соединения и затор SYN» главы 7.

Кроме того, необходимо увеличить размеры буферов приема и передачи TCP. В Solaris их размеры задаются параметрами `tcp_recv_hiwat` и `tcp_xmit_hiwat`. С этими значениями следует экспериментировать. В качестве начальных значений авторы рекомендуют 48 Кбайт (49 152) для `tcp_xmit_hiwat` и 32 Кбайт (32 768) для `tcp_recv_lowat`. Для того чтобы избежать масштабирования окна TCP, эти значения не должны превышать 64000. В веб-серверах Linux эквивалентные параметры имеют значения по умолчанию, которые пригодны для достижения высокой производительности. Однако можно настраивать и их. Для получения дополнительных сведений следует обратиться к разделу «Буферы, отметки уровня и окна» главы 7.

Особый случай: прокси-серверы

Важное различие между веб-серверами и прокси-серверами состоит в том, что для прокси-серверов характерно намного большее количество

операций записи (так как кэшируемые страницы записываются на диск). Поэтому идеальная конфигурация прокси-сервера подразумевает дисковый массив с энергонезависимой памятью для кэширования и ускорения операций записи. Подобного эффекта можно достичь с помощью выделенного диска, предназначенного для протоколирования транзакций в кэширующей файловой системе.

Если речь идет о конфигурировании быстрого однопроцессорного прокси-сервера, например с восемью дисками по 9 Гбайт для данных, то авторы посоветовали бы следующее. Необходимо образовать блок из шести дисков с размером чередования 64 Кбайт с помощью DiskSuite или другого программного обеспечения RAID. На оставшихся двух дисках нужно создать по одному разделу 128 Мбайт, зеркалировать их и применять для хранения файлов протоколирования. Размещать что-то еще на этих двух дисках не следует. С точки зрения времени обслуживания такие диски полностью загружены, даже если они имеют резерв емкости и пропускной способности.

Алфавитный указатель

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

A

Б

В

Г

Д

Е

Ж

З

И

К

Л

М

H

O

П

P

C

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я