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, 2003
Authorized translation of the English edition © 2002 O'Reilly & Associates Inc. This translation is published and sold by permission of O'Reilly & Associates Inc., the owner of all rights to publish and sell the same.

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

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

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.