для профессионалов

# Внутреннее устройство LINUX

Брайан Уорд

Brian Ward

# How Linux Works:
## What Every Superuser Should Know

no starch
press

Брайан Уорд

# Внутреннее устройство LINUX

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2016

ББК 32.973.2-018.2
УДК 004.451
У64

**Уорд Б.**

У64 Внутреннее устройство Linux. — СПб.: Питер, 2016. — 384 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-496-01952-1

Книга, которую вы держите в руках, уже стала бестселлером на Западе. Она описывает все тонкости работы с операционной системой Linux, системное администрирование, глубокие механизмы, обеспечивающие низкоуровневый функционал Linux. На страницах этого издания вы приобретете базовые знания о работе с ядром Linux и о принципах правильной эксплуатации компьютерных сетей. В книге также затрагиваются вопросы программирования сценариев оболочки и обращения с языком C, освещаются темы защиты информации, виртуализации и прочие незаменимые вещи.

**12+** (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.2
УДК 004.451



ISBN 978-1593275679 англ.

ISBN 978-5-496-01952-1

# Краткое содержание

# Оглавление

# Предисловие

Я написал эту книгу, будучи уверенным в том, что для успешной, эффективной работы вы должны понимать, как устроено и функционирует программное обеспечение вашего компьютера.

Операционная система Linux прекрасно подходит для изучения, поскольку конфигурация системы хранится большей частью в простых файлах, которые достаточно легко прочитать. Следует только выяснить, за что отвечает каждая из частей, а затем собрать все воедино. Именно этому и посвящена данная книга.

## Кому следует читать книгу

Интерес к устройству операционной системы Linux может быть вызван разными причинами. Профессионалы в сфере информационно-технологического обслуживания, а также разработчики программного обеспечения для Linux найдут в этой книге практически все, что необходимо знать, чтобы использовать операционную систему наилучшим образом. Исследователи и студенты, которым зачастую приходится подстраивать систему под себя, найдут здесь практичные объяснения того, почему все устроено именно так, а не иначе. Есть еще «затейники» — пользователи, которым нравится проводить время за компьютером ради развлечения, выгоды или и того и другого сразу.

Хотите узнать, почему некоторые вещи работают, а другие — нет? Вам интересно, что произойдет, если что-либо изменить? Тогда вы относитесь к числу «затейников».

## Необходимые условия

Вам не обязательно быть программистом, чтобы читать эту книгу. Понадобятся лишь основные навыки пользователя компьютера: вы должны ориентироваться в графическом интерфейсе (при установке и настройке интерфейса системы), а также иметь представление о файлах и каталогах (папках). Следует также быть готовыми к поиску дополнительной документации в вашей системе и онлайн. Как отмечалось выше, самым важным является ваша готовность и желание исследовать свой компьютер.

## Как читать книгу

Когда речь идет о технических предметах, донести все необходимые знания — непростая задача. С одной стороны, читатель увязает в излишних подробностях и с тру-

дом усваивает суть, поскольку человеческий разум просто не может одновременно обработать большое количество новых понятий. С другой — отсутствие подробностей приводит к тому, что читатель получает лишь смутное представление о предмете и не готов к усвоению дальнейшего материала.

В этой книге я упростил изложение и структурировал материал. В большинстве глав важная информация, которая необходима для дальнейшей работы, предлагается в первую очередь. По мере чтения главы вы встретите в ней и дополнительный материал. Надо ли вам сразу усваивать эти частности? В большинстве случаев полагаю, что нет. Если ваши глаза начинают тускнеть при виде большого количества подробностей, относящихся к только что изученному материалу, не раздумывая переходите к следующей главе или сделайте перерыв. Вас ожидают другие важные вещи.

# Практический подход

Для работы вам понадобится компьютер с операционной системой Linux. Возможно, вы предпочтете виртуальную установку — для проверки большей части материала данной книги я пользовался приложением VirtualBox. Вы должны обладать правами доступа superuser (root), хотя в основную часть времени следует использовать учетную запись обычного пользователя. Вы будете работать главным образом в командной строке, окне терминала или в удаленном сеансе. Если вы нечасто работали в этой среде, ничего страшного, из главы 2 вы узнаете об этом подробнее.

Как правило, команды будут выглядеть следующим образом:

```
$ ls /
[some output]
```

Вводите текст, который выделен жирным шрифтом; обычным шрифтом показан ответный текст, который выдаст машина. Символ `$` является приглашением для пользователя с обычной учетной записью. Если вы увидите в качестве приглашения символ #, следует работать в учетной записи superuser (подробнее об этом — в главе 2).

# Как устроена эта книга

В начале книги дается обзор системы Linux, а затем предлагается ряд практических заданий с инструментами, которые понадобятся вам для дальнейшей работы в системе. Далее вы детально изучите каждую часть системы, начиная с управления оборудованием и заканчивая конфигурацией сети, следуя обычному порядку, в котором происходит запуск системы. И наконец, вы получите представление о некоторых деталях работающей системы, освоите несколько важных навыков, а также познакомитесь с инструментами, используемыми программистами.

В большинстве первых глав (кроме главы 2) активно задействовано ядро системы Linux, но по мере продвижения по книге вы будете работать и в своем пространстве пользователя. Если вы не понимаете, о чем я сейчас говорю, не беспокойтесь, объяснения будут даны в главе 1.

Материал излагается по возможности без привязки к какому-либо дистрибутиву системы. Было бы скучно описывать все варианты системы, поэтому я попытался рассказать о двух основных семействах дистрибутивов: Debian (включая Ubuntu) и RHEL/Fedora/CentOS. Упор сделан на серверные версии и версии для рабочих станций. Представлены также внедренные системы, например Android и OpenWRT, но изучение отличий этих платформ предоставляется вам.

## Что нового во втором издании

Первое издание этой книги касалось главным образом пользовательской стороны работы в системе Linux. Основное внимание было уделено устройству ее частей и тому, как заставить их функционировать. В то время многие элементы системы было трудно установить и корректно настроить.

Благодаря упорному труду разработчиков ПО и создателей дистрибутивов Linux ситуация изменилась. Я пересмотрел материал первого издания в поисках обновлений: особое внимание уделил процессу загрузки системы и тому, как она управляет оборудованием, а также удалил устаревший материал (например, подробное объяснение процесса печати), чтобы расширить рассмотрение роли ядра системы Linux в каждом дистрибутиве. Вы, вероятно, взаимодействуете с ядром гораздо чаще, чем сами об этом догадываетесь, и я специально отметил моменты, когда это бывает.

Я также изменил последовательность подачи материала в книге, чтобы он соответствовал интересам и потребностям современных читателей. Единственное, что не изменилось, — это объем книги.

Мне хотелось снабдить вас сведениями, которые понадобятся для быстрого начала работы. Их усвоение потребует некоторых усилий, однако я не намереваюсь делать из вас «тяжелоатлетов», чтобы вы смогли одолеть эту книгу. Когда вы будете понимать важнейшие моменты, изложенные здесь, для вас не составит труда отыскать подробности и разобраться в них.

Я изъял некоторые исторические детали, которые были в первом издании, главным образом чтобы сконцентрировать ваше внимание. Если вы интересуетесь системой Linux и ее отношением к истории системы Unix, обратитесь к книге Питера Салуса (Peter H. Salus) *The Daemon, the Gnu, and the Penguin* (Reed Media Services, 2008) — в ней рассказано о том, как развивалось используемое нами программное обеспечение.

## Примечание о терминологии

В настоящее время ведутся споры о наименованиях некоторых элементов операционных систем. В них вовлечено даже само название системы Linux — как она должна называться: Linux или же GNU/Linux (чтобы отразить применение некоторых элементов проекта GNU)? В книге я старался использовать самые употребительные и по возможности наименее громоздкие названия.

# Благодарности

Благодарю всех, кто помогал мне в работе над первым изданием. Это Джеймс Дункан (James Duncan), Дуглас Н. Арнольд (Douglas N. Arnold), Билл Феннер (Bill Fenner), Кен Хорнстайн (Ken Hornstein), Скотт Диксон (Scott Dickson), Дэн Эрлих (Dan Ehrlich), Феликс Ли (Felix Lee), Скотт Шварц (Scott Schwartz), Грегори П. Смит (Gregory P. Smith), Дэн Салли (Dan Sully), Кэрол Джурадо (Karol Jurado) и Джина Стил (Gina Steele). В этом издании я особо признателен Жорди Гутьеррес Хермозо (Jordi Gutiérrez Hermoso) за превосходное техническое рецензирование; его предложения и уточнения неоценимы. Спасибо также Доминику Пулэну (Dominique Poulain) и Дональду Кэрону (Donald Karon) за быстрый отклик на ранних стадиях работы, а также Синьчжу Шей (Hsinju Hsieh), который терпеливо работал со мной над переделкой этой книги.

Я хотел бы также поблагодарить своего редактора по развитию Билла Поллока (Bill Pollock) и выпускающего редактора Лорел Чан (Laurel Chun). Серена Янг (Serena Yang), Элисон Ло (Alison Law) и все сотрудники издательства No Starch Press, как обычно, замечательно выполнили свою работу при подготовке нового издания книги.

# От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты sivchenko@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

# 1 Общая картина

На первый взгляд современная операционная система, например Linux, является достаточно сложной и состоит из большого количества частей, которые одновременно функционируют и взаимодействуют друг с другом. Так, веб-сервер может обмениваться данными с сервером базы данных, который, в свою очередь, использует совместную библиотеку, применяемую многими другими программами. Как же все это работает?

Наиболее эффективно понять устройство операционной системы можно с помощью *абстракции* — изящного способа сказать о том, что вы игнорируете большинство деталей. Например, когда вы едете в автомобиле, вам, как правило, не приходится задумываться о таких деталях, как крепежные болты, которые удерживают двигатель внутри машины, или же о людях, проложивших дорогу и поддерживающих ее в хорошем состоянии. Если вы едете в машине как пассажир, вам нужно знать лишь то, для чего предназначен автомобиль (он перемещает вас куда-либо), а также некоторые элементарные правила его использования (как обращаться с дверью и ремнем безопасности).

Если же вы ведете машину, вам необходимо знать больше. Вам потребуется изучить элементы управления (например, рулевое колесо и педаль газа), а также усвоить, что следует делать в случае неисправности.

Предположим, автомобиль движется рывками. Можно разбить абстракцию «автомобиль, который едет по дороге» на три части: автомобиль, дорога и ваш стиль вождения. Это поможет установить причину. Если дорога ухабиста, вам не придется винить машину или себя. Вместо этого вы можете попытаться выяснить, почему дорога испортилась, или же, если дорога новая, почему ее строители так отвратительно выполнили работу.

Разработчики программного обеспечения пользуются абстракцией как инструментом при создании операционных систем и приложений. Имеется множество терминов для абстрагированных разделов компьютерного ПО, в их число входят *подсистема*, *модуль* и *пакет*. Однако мы будем применять в данной главе термин *компонент*, поскольку он прост. При создании программного компонента, как правило, разработчиков не сильно заботит внутренняя структура других компонентов, однако им все же приходится думать о том, какие компоненты и как они могут использовать.

В этой главе приводится общий обзор компонентов, составляющих систему Linux. Хотя каждый из них обладает немыслимым количеством технических

деталей, относящихся к внутреннему устройству, мы не будем обращать на них внимания и сосредоточимся на том, что эти компоненты делают по отношению к системе в целом.

## 1.1. Уровни и слои абстракции в операционной системе Linux

Использование абстракций для разделения компьютерных систем на компоненты упрощает их понимание, но не приносит пользы, если отсутствует структура. Мы упорядочим компоненты в виде слоев, или уровней. *Слой*, или *уровень*, — это способ классификации (или группирования) компонентов в соответствии с их расположением между пользователем и аппаратными средствами. Браузеры, игры и т. п. расположены на верхнем слое; на нижнем слое мы видим память компьютера: нули и единицы. Операционная система занимает наибольшее число слоев между этими двумя.

В операционной системе Linux три главных уровня. На рис. 1.1 показаны уровни, а также некоторые компоненты внутри каждого из них. В основе расположены *аппаратные средства*. Они включают память, а также один или несколько центральных процессоров (CPU), выполняющих вычисления и запросы на чтение из памяти и запись в нее. Такие устройства, как жесткие диски и сетевые интерфейсы, также относятся к аппаратным средствам.

Уровнем выше располагается *ядро*, которое является сердцевиной операционной системы. Ядро — это программа, расположенная в памяти компьютера и отдающая распоряжения центральному процессору. Ядро управляет аппаратными средствами и выступает главным образом в качестве интерфейса между аппаратными средствами и любой запущенной программой.

Процессы — запущенные программы, которыми управляет ядро, — в совокупности составляют верхний уровень системы, именующийся *пространством пользователя*.

ПРИМЕЧАНИЕ

Более точным термином, чем «процесс», является термин «пользовательский процесс», вне зависимости от того, взаимодействует ли пользователь с этим процессом напрямую. Например, все вебсерверы работают как пользовательские процессы.

Существует важное различие между тем, как запускаются процессы ядра и процессы пользователя: ядро запускается в *режиме ядра*, а пользовательские процессы — в *режиме пользователя*. Код, работающий в режиме ядра, обладает неограниченным доступом к процессору и оперативной памяти. Это сильное преимущество, но оно может быть опасным, поскольку позволяет процессам ядра с легкостью нарушить работу всей системы. Область, которая доступна только ядру, называется *пространством ядра*.

В режиме пользователя, для сравнения, доступен лишь ограниченный (как правило, небольшой) объем памяти и разрешены лишь безопасные инструкции для процессора. *Пространством пользователя* называют участки оперативной памяти, которые могут быть доступны пользовательским процессам. Если какой-либо процесс завершается с ошибкой, ее последствия будут ограниченными и ядро сможет

их очистить. Это означает, что, если, например, произойдет сбой в работе браузера, выполнение научных расчетов, которые вы запустили на несколько дней в фоновом режиме, не будет нарушено.

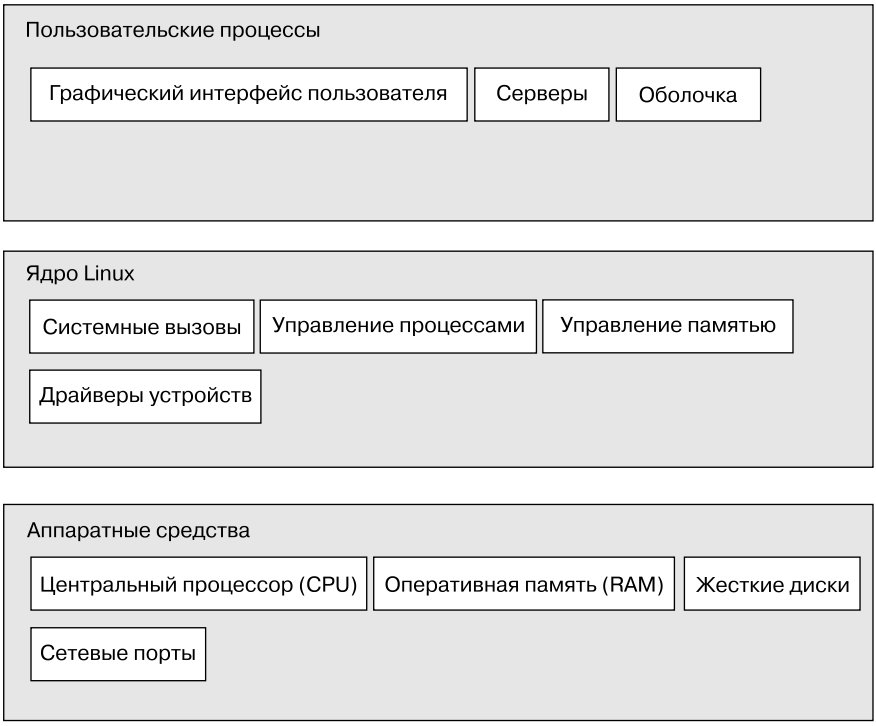


**Рис. 1.1.** Общая структура операционной системы Linux

Теоретически неконтролируемый пользовательский процесс не способен причинить существенный вред системе. В действительности же все зависит от того, что именно вы считаете «существенным вредом», а также от особых привилегий данного процесса, поскольку некоторым процессам разрешено делать больше, чем другим. Например, может ли пользовательский процесс полностью уничтожить данные на жестком диске? Если должным образом настроить разрешения, то сможет, и для вас это окажется крайне опасным. Для предотвращения этого существуют защитные меры, и большинству процессов просто не будет позволено сеять смуту подобным образом.

## 1.2. Аппаратные средства: оперативная память

Из всех аппаратных средств компьютера *оперативная память* является, пожалуй, наиболее важным. В своей самой «сырой» форме оперативная память — это всего лишь огромное хранилище для последовательности нулей и единиц. Каждый ноль или единица называется *битом*. Именно здесь располагаются запущенное ядро

и процессы — они являются лишь большими наборами битов. Все входные и выходные данные от периферийных устройств проходят через оперативную память также в виде наборов битов. Центральный процессор просто оперирует с памятью: он считывает из нее инструкции и данные, а затем записывает данные обратно в память.

Вам часто будет встречаться термин *«состояние»*, который будет относиться к памяти, процессам, ядру и другим частям компьютерной системы. Строго говоря, состояние — это какое-либо упорядоченное расположение битов. Например, если в памяти находятся четыре бита, то последовательности 0110, 0001 и 1011 представляют три различных состояния.

Если принять во внимание то, что процесс может с легкостью состоять из миллионов бит в памяти, зачастую проще использовать абстрактные термины, говоря о состояниях. Вместо описания состояния с применением битов вы говорите о том, что произошло или происходит в данный момент. Например, вы можете сказать «данный процесс ожидает входных данных» или «процесс выполняет второй этап процедуры запуска».

ПРИМЕЧАНИЕ

Поскольку при описании состояния обычно используются абстрактные понятия, а не реальные биты, для обозначения какого-либо физического размещения битов применяется термин «образ».

# 1.3. Ядро

Практически все, что выполняет ядро, касается оперативной памяти. Одной из задач ядра является распределение памяти на несколько подразделов, после чего ядро должно постоянно содержать в порядке информацию о состоянии этих подразделов. Каждый процесс использует выделенную для него область памяти, и ядро должно гарантировать то, что процессы придерживаются своих областей.

Ядро отвечает за управление задачами в четырех основных областях системы.

- **Процессы.** Ядро отвечает за то, каким процессам разрешен доступ к центральному процессору.
- **Память.** Ядру необходимо отслеживать состояние всей памяти: какая часть в данный момент отведена под определенные процессы, что можно выделить для совместного использования процессами и какая часть свободна.
- **Драйверы устройств.** Ядро выступает в качестве интерфейса между аппаратными средствами (например, жестким диском) и процессами. Как правило, управление аппаратными средствами выполняется ядром.
- **Системные вызовы и поддержка.** Обычно процессы используют системные вызовы для взаимодействия с ядром.

Теперь мы вкратце рассмотрим каждую из этих областей.

ПРИМЕЧАНИЕ

Подробности о работе ядра вы можете узнать из книг Operating System Concepts («Основные принципы операционных систем»), 9-е издание, авторы: Авраам Зильбершатц (Abraham Silberschatz), Питер Б. Гелвин (Peter B. Galvin) и Грег Гэнн (Greg Gagne) (Wiley, 2012) и Modern Operating Systems («Современные операционные системы»), 4-е издание, авторы: Эндрю С. Таненбаум (Andrew S. Tanenbaum) и Герберт Бос (Herbert Bos) (Prentice Hall, 2014).

## 1.3.1. Управление процессами

*Управление процессами* описывает запуск, остановку, возобновление и прекращение работы процессов. Понятия, которые стоят за процессами запуска и прекращения процессов, достаточно просты. Немного сложнее описать то, каким образом процесс использует центральный процессор в нормальном режиме работы.

В любой современной операционной системе несколько процессов функционируют «одновременно». Например, в одно и то же время вы можете запустить на компьютере браузер и открыть электронную таблицу. Тем не менее на самом деле все обстоит не так, как выглядит: процессы, которые отвечают за эти приложения, как правило, не запускаются *в точности* в один момент времени.

Рассмотрим систему с одним центральным процессором. Его могут использовать несколько процессов, но в каждый конкретный момент времени только один процесс может в действительности применять процессор. На практике каждый процесс использует процессор в течение малой доли секунды, а затем приостанавливается; после этого другой процесс применяет процессор в течение малой доли секунды; далее наступает черед третьего процесса и т. д. Действие, при котором какой-либо процесс передает другому процессу управление процессором, называется *переключением контекста*.

Каждый отрезок времени — *квант времени* — предоставляет процессу достаточно времени для выполнения существенных вычислений (и, конечно же, процесс часто завершает свою текущую задачу в течение одного кванта). Поскольку кванты времени настолько малы, человек их не воспринимает и ему кажется, что в системе одновременно выполняется несколько процессов (такая возможность известна под названием *«многозадачность»*).

Ядро отвечает за переключение контекста. Чтобы понять, как это работает, представим ситуацию, в которой процесс запущен в режиме пользователя, но его квант времени заканчивается. Вот что при этом происходит.

1. Процессор (реальное аппаратное средство) прерывает текущий процесс, опираясь на внутренний таймер, переключается в режим ядра и возвращает ему управление.
2. Ядро записывает текущее состояние процессора и памяти, которые будут необходимы для возобновления только что прерванного процесса.
3. Ядро выполняет любые задачи, которые могли появиться в течение предыдущего кванта времени (например, сбор данных или операции ввода/вывода).
4. Теперь ядро готово к запуску другого процесса. Оно анализирует список процессов, готовых к запуску, и выбирает какой-либо из них.
5. Ядро готовит память для нового процесса, а затем подготавливает процессор.
6. Ядро сообщает процессору, сколько будет длиться квант времени для нового процесса.
7. Ядро переводит процессор в режим пользователя и передает процессору управление.

Переключение контекста дает ответ на важный вопрос: *когда* работает ядро? Ответ следующий: ядро работает *между* отведенными для процессов квантами времени, когда происходит переключение контекста.

В системе с несколькими процессорами дело обстоит немного сложнее, поскольку ядру нет необходимости прекращать управление текущим процессором, чтобы позволить запуск какого-либо процесса на другом процессоре. И тем не менее, чтобы извлечь максимальную пользу из всех доступных процессоров, ядро все же так поступает (и может применить определенные хитрости, чтобы получить дополнительное процессорное время).

## 1.3.2. Управление памятью

Поскольку ядро должно управлять памятью во время переключения контекста, оно наделено этой сложной функцией. Работа ядра сложна, поскольку необходимо учитывать следующие условия:

- ядро должно располагать собственной областью памяти, к которой не могут получить доступ пользовательские процессы;
- каждому пользовательскому процессу необходима своя область памяти;
- какой-либо пользовательский процесс не должен иметь доступ к области памяти, предназначенной для другого процесса;
- пользовательские процессы могут совместно использовать память;
- некоторые участки памяти для пользовательских процессов могут быть предназначены только для чтения;
- система может применять больше памяти, чем ее есть в наличии, задействовав в качестве вспомогательного устройства дисковое пространство.

У ядра есть помощник. Современные процессоры содержат *модуль управления памятью* (MMU), который активизирует схему доступа к памяти под названием *«виртуальная память»*. При использовании виртуальной памяти процесс не обращается к памяти напрямую по ее физическому расположению в аппаратных средствах. Вместо этого ядро настраивает каждый процесс таким образом, словно в его распоряжении находится вся машина. Когда процесс получает доступ к памяти, модуль MMU перехватывает такой запрос и применяет карту адресов памяти, чтобы перевести местоположение памяти, полученное от процесса, в физическое положение памяти на компьютере. Однако ядро все же должно инициализировать, постоянно поддерживать и изменять эту карту адресов. Например, во время переключения контекста ядро должно изменить карту после отработавшего процесса и подготовить его для наступающего.

ПРИМЕЧАНИЕ

Реализация карты адресов памяти называется таблицей страниц.

О том, как отслеживать производительность памяти, вы узнаете из главы 8.

## 1.3.3. Драйверы устройств и управление ими

Задача ядра по отношению к устройствам довольно проста. Как правило, устройства доступны только в режиме ядра, поскольку некорректный доступ (например, когда пользовательский процесс пытается выключить питание) может вызвать от-

каз в работе компьютера. Еще одна проблема заключается в том, что различные устройства редко обладают одинаковым программным интерфейсом, даже если они выполняют одинаковую задачу: например, две различные сетевые карты. По этой причине драйверы устройств традиционно являются частью ядра и стремятся предоставить унифицированный интерфейс для пользовательских процессов, чтобы облегчить труд разработчиков программного обеспечения.

## 1.3.4. Системные вызовы и поддержка

Существуют и другие типы функций ядра, доступные для пользовательских процессов. Например, *системные вызовы* выполняют специальные задачи, которые пользовательский процесс не может выполнить хорошо в одиночку или вообще не может справиться с ними. Так, все действия, связанные с открытием, чтением и записью файлов, вовлекают системные вызовы.

Два системных вызова — `fork()` и `exec()` — важны для понимания того, как происходит запуск процессов:

- `fork()`. Когда процесс осуществляет вызов `fork()`, ядро создает практически идентичную копию данного процесса;
- `exec()`. Когда процесс осуществляет вызов `exec(`*`program`*`)`, ядро запускает программу *program*, которая замещает текущий процесс.

За исключением процесса `init` (глава 6), *все* пользовательские процессы в системе Linux начинаются как результат вызова `fork()`, и в большинстве случаев осуществляется вызов `exec()`, чтобы запустить новую программу, а не копию существующего процесса. Простым примером является любая программа, которую вы запускаете из командной строки, например команда `ls`, показывающая содержимое каталога. Когда вы вводите команду `ls` в окне терминала, запущенная внутри окна терминала оболочка осуществляет вызов `fork()`, чтобы создать копию оболочки, а затем эта новая копия оболочки выполняет вызов `exec(ls)`, чтобы запустить команду `ls`. На рис. 1.2 показана последовательность процессов и системных вызовов для запуска таких программ, как `ls`.


**Рис. 1.2.** Запуск нового процесса

ПРИМЕЧАНИЕ

Системные вызовы обычно обозначаются с помощью круглых скобок. В примере, показанном на рис. 1.2, процесс, который запрашивает ядро о создании другого процесса, должен осуществить системный вызов fork(). Такое обозначение происходит от способа написания вызовов в языке программирования C. Чтобы понять эту книгу, вам не обязательно знать язык C. Помните лишь о том, что системный вызов — это взаимодействие между процессом и ядром. Более того, в этой книге упрощены некоторые группы системных вызовов. Например, вызов exec() обозначает целое семейство системных вызовов, выполняющих сходную задачу, но отличающихся программной реализацией.

Ядро также поддерживает пользовательские процессы, функции которых отличаются от традиционных системных вызовов. Самыми известными из них являются *псевдоустройства*. С точки зрения пользовательских процессов, псевдоустройства выглядят как обычные устройства, но реализованы они исключительно программным образом. По сути, формально они не должны находиться в ядре, но они все же присутствуют в нем из практических соображений. Например, устройство, которое генерирует случайные числа (`/dev/random`), было бы сложно реализовать с необходимой степенью безопасности с помощью пользовательского процесса.

ПРИМЕЧАНИЕ

Технически пользовательский процесс, который получает доступ к псевдоустройству, все же вынужден осуществлять системный вызов для открытия этого устройства. Таким образом, процессы не могут полностью обойтись без системных вызовов.

## 1.4. Пространство пользователя

Область оперативной памяти, которую ядро отводит для пользовательских процессов, называется *пространством пользователя*. Поскольку процесс является лишь состоянием (или образом) в памяти, пространство пользователя обращается также к памяти за всей совокупностью запущенных процессов. Вам также может встретиться термин *«участок пользователя» (userland)*, который применяется вместо пространства пользователя.

Большинство реальных действий системы Linux происходит в пространстве пользователя. Несмотря на то что все процессы с точки зрения ядра являются одинаковыми, они выполняют различные задачи для пользователей. Системные компоненты, которые представляют пользовательские процессы, организованы в виде элементарной структуры — сервисного уровня (или слоя). На рис. 1.3 показан примерный набор компонентов, связанных между собой и взаимодействующих с системой Linux. Простые службы расположены на нижнем уровне (ближе всего к ядру), сервисные программы находятся в середине, а приложения, с которыми работает пользователь, расположены вверху. Рисунок 1.3 является крайне упрощенной схемой, поскольку показаны только шесть компонентов, но вы можете заметить, что верхние компоненты находятся ближе всего к пользователю (пользовательский интерфейс и браузер); компоненты среднего уровня располагают почтовым сервером, который использует браузер; в нижней части присутствует несколько малых компонентов.

Нижний уровень состоит, как правило, из малых компонентов, выполняющих простые задачи. Средний уровень содержит более крупные компоненты, такие как почтовая служба, сервер печати и база данных. Компоненты верхнего уровня выполняют сложные задачи, которые зачастую непосредственно контролирует пользователь. Если один компонент желает воспользоваться другим, то этот второй компонент находится либо на том же сервисном уровне, либо ниже.

Рисунок 1.3 только приблизительно отображает устройство пространства пользователя. В действительности в пространстве пользователя нет правил. Например, большинство приложений и служб записывают диагностические сообщения, ко-

торые называются *журналами*. Большинство программ использует стандартную службу `syslog` для записи сообщений в журнал, но некоторые предпочитают вести журнал самостоятельно.

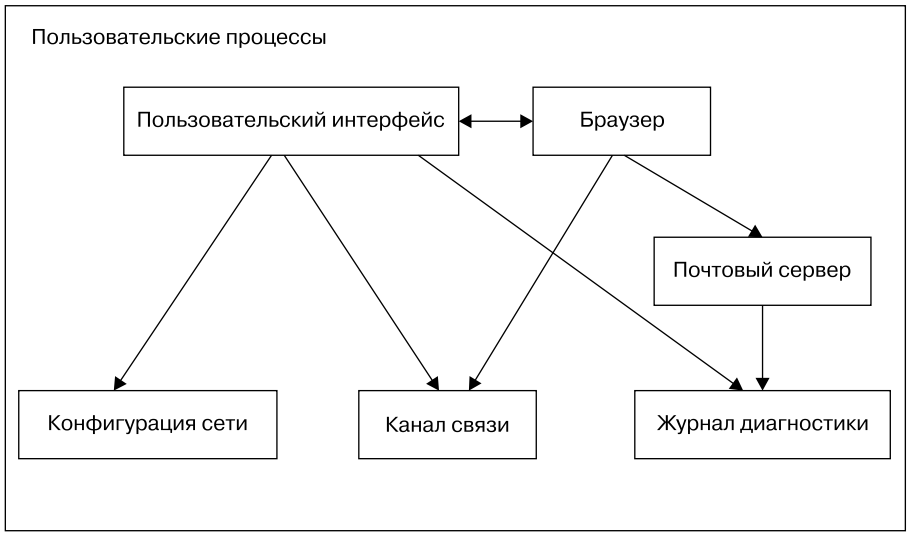


**Рис. 1.3.** Типы процессов и взаимодействий

Кроме того, некоторые компоненты пространства пользователя бывает трудно отнести к какой-либо категории. Серверные компоненты, например веб-сервер или сервер базы данных, можно рассматривать как приложения очень высокого уровня, поскольку они выполняют довольно сложные задачи. Такие приложения можно поместить в верхней части рис. 1.3. В то же время пользовательские приложения могут зависеть от серверных, когда необходимо выполнять задачи, с которыми они не могут справиться самостоятельно. В таком случае серверные компоненты следовало бы поместить на средний уровень.

## 1.5. Пользователи

Ядро системы Linux поддерживает традиционную концепцию пользователя системы Unix. *Пользователь* — это сущность, которая может запускать процессы и обладать файлами. С пользователем связано *имя пользователя*. Например, в системе может быть пользователь `billyjoe`. Однако ядро не работает с именами пользователей, вместо этого оно идентифицирует пользователя с помощью простого числового *идентификатора пользователя* (в главе 7 рассказывается о том, как идентификаторы сопоставляются с именами пользователей).

Пользователи существуют главным образом для того, чтобы соблюдались права доступа и ограничения. У каждого процесса из пространства пользователя существует пользователь-*владелец*, а о процессах говорят, что они запущены

*в качестве* владельцев. Пользователь может прервать или изменить ход принадлежащих ему процессов (в определенных пределах), но не может вмешаться в процессы других пользователей. Кроме того, пользователи могут обладать файлами и предоставлять совместный доступ к ним для других пользователей.

В системе Linux обычно присутствуют дополнительные пользователи помимо тех, которые соответствуют реальным людям, работающим в системе. Более подробно об этом рассказывается в главе 3, но самым важным пользователем является `root`. Этот пользователь — исключение из приведенных выше правил, поскольку он может прерывать и изменять ход процессов другого пользователя, а также выполнять чтение любого локального файла. По этой причине пользователь `root` известен как `superuser`. О человеке, который может работать как пользователь `root`, говорят, что у него есть *root-доступ*. В традиционной системе Unix это администратор.

ПРИМЕЧАНИЕ

Работать с правами root может оказаться опасно. Будет сложно выявить и исправить ошибки, поскольку система позволит вам выполнить что угодно, даже если вы пытаетесь причинить ей вред. Системщики постоянно стараются сделать так, чтобы root-доступ не был необходим, насколько это возможно. Например, при переключении между сетями беспроводного доступа на ноутбуке. В то же время, каким бы могущественным ни был пользователь root, он все-таки работает в режиме пользователя системы, а не в режиме ядра.

*Группы* состоят из пользователей. Основная цель групп заключается в том, чтобы пользователь мог предоставлять файлы для совместного доступа другим пользователям группы.

## 1.6. Заглядывая вперед

Итак, вы увидели, из чего состоит работающая система Linux. Пользовательские процессы создают среду, с которой вы непосредственно взаимодействуете. Ядро управляет процессами и аппаратными средствами. Обе эти составляющие — ядро и процессы — располагаются в памяти.

Теоретическая информация — это замечательно, но вы не сможете изучить детали системы Linux, только читая о ней. В следующей главе вы начнете свое путешествие, освоив некоторые основы пространства пользователя. Попутно вы узнаете о более обширных частях системы Linux, о которых не говорилось в этой главе: о долговременных запоминающих устройствах (жестких дисках, файлах и т. п.). Вам ведь необходимо где-то хранить свои программы и данные.

# 2 Основные команды и структура каталогов

Эта глава является справочником по командам и утилитам операционной системы Unix. Вероятно, вы уже знаете большинство этих команд, но для подкрепления уверенности, в особенности в том, что касается структуры каталогов (см. раздел 2.19), прочитайте данную главу до конца. Здесь представлен вводный материал, на который я буду ссылаться на протяжении всей книги.

Почему речь пойдет о командах Unix? Разве эта книга не о том, как работает Linux? Да, конечно же, об этом, но в самой сердцевине Linux заложена система Unix.

В этой главе вы встретите слово Unix чаще, чем слово Linux, поскольку полученные сведения можно тут же применить к Solaris, BSD и к другим системам, связанным с Unix. Я попытался уйти от рассмотрения излишнего числа расширений пользовательского интерфейса, специфичных для Linux, не только ради того, чтобы у вас появился прочный фундамент для использования других систем, но и потому, что такие расширения довольно нестабильны. Вы сможете адаптировать новые выпуски системы Linux быстрее, если будете знать основные команды.

ПРИМЕЧАНИЕ

Дополнительные подробности для начинающих изучать Linux можно найти в книгах The Linux Command Line («Командная строка Linux») (No Starch Press, 2012), UNIX for the Impatient («UNIX для нетерпеливых») (Addison-Wesley Professional, 1995) и Learning the UNIX Operating System («Осваиваем операционную систему UNIX»), 5-е издание (O'Reilly, 2001).

## 2.1. Оболочка Bourne shell: /bin/sh

Оболочка является одной из важнейших частей системы Unix. *Оболочка* — это программа, запускающая команды (например, те, которые вводит пользователь). Оболочка выступает также в роли небольшой среды программирования. Программисты, работающие в Unix, часто разбивают обычные задачи на несколько небольших компонентов, а затем используют оболочку, чтобы управлять задачами и собирать части воедино.

Многие важные части системы в действительности являются *сценариями оболочки* — текстовыми файлами, которые содержат последовательность команд оболочки. Если вам до этого приходилось работать в системе MS-DOS, то вы можете представлять сценарии оболочки как очень мощные файлы `.BAT`. Поскольку эти файлы очень важны, их рассмотрению полностью посвящена глава 11.

По мере чтения этой книги и приобретения практических навыков вы пополните свои знания командами, использующими оболочку. Одно из свойств оболочки позволяет, если вы совершили ошибку, сразу же увидеть, что именно набрано неверно, и попробовать заново.

Существуют различные варианты оболочки Unix, но все они заимствуют некоторые функции от оболочки Bourne shell (`/bin/sh`) — стандартной оболочки, разработанной в компании Bell Labs для ранних версий системы Unix.

Каждой системе Unix для правильной работы необходимо наличие оболочки Bourne shell. Система Linux использует улучшенную версию оболочки Bourne shell — `bash`, или «заново рожденную»[1] оболочку. Оболочка `bash` является оболочкой по умолчанию в большинстве дистрибутивов Linux, а путь `/bin/sh`, как правило, — ссылка на эту оболочку. При запуске примеров из книги следует применять оболочку `bash`.

ПРИМЕЧАНИЕ

Возможно, оболочка bash не является установленной у вас по умолчанию, если вы используете эту главу как справочник для учетной записи Unix в организации, где вы не являетесь системным администратором. Можно изменить оболочку с помощью команды chsh или попросить помощи у вашего системного администратора.

# 2.2. Использование оболочки

При установке системы Linux вы должны создать по крайней мере одну учетную запись обычного пользователя (в дополнение к корневому пользователю), которая будет вашей личной учетной записью. В этой главе необходимо входить в систему с учетной записью обычного пользователя.

## 2.2.1. Окно оболочки

После входа в систему откройте окно оболочки, которое часто называют *терминалом*. Проще всего это выполнить с помощью запуска терминального приложения из графического интерфейса пользователя Gnome или Unity в Ubuntu. При этом оболочка открывается в новом окне. После запуска оболочки в верхней части окна должно отобразиться приглашение, которое обычно заканчивается символом доллара (`$`). В Ubuntu это приглашение будет выглядеть примерно так: *`name@host:path$`*, а в Fedora оно такое: *`[name@host path]$`*. Если вы хорошо знакомы с Windows, окно оболочки будет выглядеть подобно окну командной строки DOS; приложение Terminal в OS X, по сути, такое же, как окно оболочки Linux.

В этой книге много команд следует набирать в строке приглашения оболочки. Чтобы обозначить приглашение, все они начинаются с символа `$`. Наберите приведенную ниже команду (только выделенный жирным шрифтом текст, без символа `$`) и нажмите клавишу Enter:

```
$ echo Hello there.
```

[1] В оригинале игра слов, основанная на сходстве произношения фамилии Bourne и слова born («рожденный»). — *Примеч. пер.*

ПРИМЕЧАНИЕ

Многие команды в этой книге начинаются символом #. Такие команды следует запускать с правами пользователя superuser (root). Как правило, с этими командами следует обращаться осторожно.

Теперь введите такую команду:

```
$ cat /etc/passwd
```

Эта команда отображает содержимое файла с системной информацией /etc/passwd, а затем снова выдает приглашение оболочки. Для чего нужен этот файл, вы узнаете из главы 7.

## 2.2.2. Команда cat

Команда cat системы Unix является одной из простейших для понимания. Она просто выводит содержимое одного или нескольких файлов. Общий синтаксис команды cat выглядит следующим образом:

```
$ cat file1 file2 ...
```

После запуска команда cat выполняет вывод содержимого файлов *file1*, *file2* и каких-либо еще (они обозначены символом ...), а затем завершает работу. Эта команда названа так, поскольку она выполняет конкатенацию (сцепление) содержимого файлов, если выводится более одного файла.

## 2.2.3. Стандартный ввод и стандартный вывод

Мы воспользуемся командой cat, чтобы кратко изучить ввод и вывод (I/O) в системе Unix. Процессы системы Unix используют *потоки* ввода/вывода для чтения и записи данных. Процессы считывают данные из потоков ввода и записывают данные в потоки вывода.

Потоки являются очень гибкими. Например, источником потока ввода может быть файл, устройство, терминал и даже поток вывода от другого процесса.

Чтобы увидеть работу потока ввода, введите команду cat (без названий файлов) и нажмите клавишу Enter. На этот раз вы не получите повторного приглашения, поскольку команда cat все еще работает. Начните набирать что-либо, нажимая клавишу Enter в конце каждой строки. Команда cat будет возвращать каждую набранную вами строку. Чтобы завершить работу команды cat и вернуться к строке приглашения, нажмите сочетание клавиш Ctrl+D в пустой строке.

Команда cat имеет интерактивный характер, поскольку работает потоками. Вы не указали имя входного файла, поэтому команда cat выполнила считывание из *стандартного потока ввода*, предусмотренного ядром системы Linux, а не из потока, связанного с файлом. В таком случае стандартный поток ввода был присоединен к терминалу, в котором вы запустили команду cat.

ПРИМЕЧАНИЕ

Нажатие сочетания клавиш Ctrl+D в пустой строке останавливает текущий стандартный поток ввода из терминала (и зачастую завершает работу программы). Не смешивайте эту команду с сочетанием клавиш Ctrl+C, которое завершает работу программы вне зависимости от ее ввода или вывода.

*Стандартный вывод* прост. Ядро предоставляет каждому процессу стандартный поток вывода, в который можно записывать выходные данные. Команда `cat` всегда записывает свои выходные данные в стандартный вывод. Когда вы запускаете команду `cat` в терминале, стандартный вывод уже подключен к терминалу, поэтому вы видите результат в нем.

Для стандартных ввода и вывода часто используют сокращения `stdin` и `stdout`. Многие команды работают подобно команде `cat`: если вы не укажете входной файл, команда будет производить считывание из `stdin`. Вывод немного отличается. Некоторые команды (подобно `cat`) отправляют выходные данные только в `stdout`, другие же обладают возможностью отправки данных напрямую в файлы.

Существует также и третий стандартный поток ввода/вывода, который называется *стандартной ошибкой*. Этот поток будет рассмотрен в подразделе 2.14.1.

Одним из самых важных свойств стандартных потоков является легкость манипулирования ими с целью записи и чтения, причем не только в терминале. В частности, из раздела 2.14 вы узнаете о том, как подключать потоки к файлам или другим процессам.

# 2.3. Основные команды

Бóльшая часть приведенных ниже команд Unix использует множество аргументов, причем у некоторых настолько много параметров и форматов, что полный их перечень нецелесообразен. Ниже приведен упрощенный список основных команд.

## 2.3.1. Команда ls

Команда `ls` выводит перечень содержимого какого-либо каталога. По умолчанию это текущий каталог. Используйте вариант `ls -l`, чтобы получить детализированный (длинный) список, или `ls -F`, чтобы отобразить информацию о типах файлов. Дополнительные сведения о типах файлов и правах доступа, отображающиеся в левом столбце, рассмотрены в разделе 2.17.

Ниже приведен пример длинного перечня, он содержит информацию о владельце файла (столбец 3), группе (столбец 4), размере файла (столбец 5), а также о дате и времени его изменения (между столбцом 5 и названием файла):

```
$ ls -l
total 3616
-rw-r--r--   1 juser   users    3804 Apr 30  2011 abusive.c
-rw-r--r--   1 juser   users    4165 May 26  2010 battery.zip
-rw-r--r--   1 juser   users  131219 Oct 26  2012 beav_1.40-13.tar.gz
-rw-r--r--   1 juser   users    6255 May 30  2010 country.c
drwxr-xr-x   2 juser   users    4096 Jul 17 20:00 cs335
-rwxr-xr-x   1 juser   users    7108 Feb  2  2011 dhry
-rw-r--r--   1 juser   users   11309 Oct 20  2010 dhry.c
-rw-r--r--   1 juser   users      56 Oct  6  2012 doit
drwxr-xr-x   6 juser   users    4096 Feb 20 13:51 dw
drwxr-xr-x   3 juser   users    4096 May  2  2011 hough-stuff
```

В разделе 2.17 вы больше узнаете о символе d, который встречается в столбце 1 этого перечня.

## 2.3.2. Команда cp

В своей простейшей форме команда cp копирует файлы. Например, чтобы скопировать *file1* в файл *file2*, введите следующее:

```
$ cp file1 file2
```

Чтобы скопировать несколько файлов в какой-либо каталог (папку) с названием *dir*, попробуйте такой вариант:

```
$ cp file1 ... fileN dir
```

## 2.3.3. Команда mv

Команда mv (от англ. move — «переместить») подобна команде cp. В своей простейшей форме она переименовывает файл. Например, чтобы переименовать файл *file1* в *file2*, введите следующее:

```
$ mv file1 file2
```

Можно также использовать команду mv, чтобы переместить несколько файлов в другой каталог:

```
$ mv file1 ... fileN dir
```

## 2.3.4. Команда touch

Команда touch создает файл. Если такой файл уже существует, команда touch не изменяет его, но обновляет информацию о времени изменения файла, выводимую с помощью команды ls -l. Например, чтобы создать пустой файл, введите следующее:

```
$ touch file
```

Теперь примените к этому файлу команду ls -l. Вы должны увидеть результат, подобный приведенному ниже. Символом ❶ отмечены дата и время запуска команды touch:

```
$ ls -l file
-rw-r--r-- 1 juser users 0 May 21 18:32❶ file
```

## 2.3.5. Команда rm

Чтобы удалить файл, воспользуйтесь командой rm (от англ. remove — «удалить»). После удаления файла он исчезает из системы и, как правило, не может быть восстановлен.

```
$ rm file
```

### 2.3.6. Команда echo

Команда `echo` выводит свои аргументы в стандартный вывод:

```
$ echo Hello again.
Hello again.
```

Команда `echo` весьма полезна для раскрытия значений паттернов оболочки, использующих джокерные символы вроде *, и переменных, таких как `$HOME`, с которыми вы познакомитесь чуть позже в этой главе.

## 2.4. Перемещение по каталогам

Иерархия каталогов в системе Unix начинается с каталога `/`, который иногда называют *корневым каталогом*. Каталоги разделяются с помощью символа «слеш» (`/`), но *не* с помощью обратного слеша (`\`). В корневом каталоге присутствует несколько стандартных подкаталогов, например `/usr`, как вы узнаете из раздела 2.19.

Когда вы ссылаетесь на файл или каталог, вы указываете *путь* или *имя пути*. Когда путь начинается с символа `/` (например, `/usr/lib`), такой путь называется *полным* или *абсолютным*.

Часть пути, которая представлена двумя точками (`..`), указывает на родительский каталог по отношению к данному. Если, например, вы работаете в каталоге `/usr/lib`, то в этом случае путь `..` будет означать `/usr`. Подобным же образом `../bin` означает `/usr/bin`.

Одна точка (`.`) ссылается на текущий каталог. Если, например, вы сейчас в каталоге `/usr/lib`, то путь `.` по-прежнему означает `/usr/lib`, а путь `./X11` будет значить `/usr/lib/X11`. Вам не придется слишком часто применять точку, поскольку для большинства команд по умолчанию указан текущий каталог, если путь не начинается с символа `/` (в предыдущем примере вы могли бы использовать `X11` вместо `./X11`).

Путь, который не начинается с символа `/`, называется *относительным путем*. Чаще всего вам придется работать с относительными путями, поскольку вы уже будете находиться в необходимом каталоге или где-либо неподалеку от него.

Теперь, когда у вас есть основное представление об устройстве каталогов, рассмотрим несколько важных команд для работы с ними.

### 2.4.1. Команда cd

*Текущий рабочий каталог* — это каталог, в котором в данный момент находится процесс (например, оболочка). Команда `cd` изменяет текущий рабочий каталог оболочки:

```
$ cd dir
```

Если вы опустите параметр *dir*, оболочка вернет вас в *домашний каталог*, с которого вы начали работу при входе в систему.

## 2.4.2. Команда mkdir

Команда `mkdir` создает новый каталог с именем *dir*:

```
$ mkdir dir
```

## 2.4.3. Команда rmdir

Команда `rmdir` удаляет каталог с именем *dir*:

```
$ rmdir dir
```

Если каталог *dir* не пуст, эта команда не сработает. Чтобы удалить каталог со всем его содержимым, используйте команду `rm -rf dir`. Однако будьте осторожны! Это одна из немногих команд, которая может причинить существенный вред, особенно если вы работаете как superuser. Параметр `-r` задает *рекурсивное удаление*, которое последовательно удаляет все, что находится внутри каталога *dir*, а параметр `-f` делает эту операцию принудительной. Не используйте флаги `-rf` с такими джокерными символами, как звездочка (*). Перепроверяйте команды перед их запуском.

## 2.4.4. Универсализация файловых имен (джокерные символы)

Оболочка способна сопоставлять простые шаблоны с именами файлов и каталогов. Этот процесс называется *универсализацией файловых имен*. Он напоминает применение джокерных символов в других системах. Простейшим из таких символов является звездочка (*), которая указывает оболочке на то, что вместо него можно подставить любое количество произвольных символов. Например, следующая команда выдаст список файлов в текущем каталоге:

```
$ echo *
```

Оболочка сопоставляет аргументы, которые содержат джокерные символы, с именами файлов, заменяет аргументы подходящими именами, а затем запускает исправленную командную строку. Такая подстановка называется *развертыванием*, поскольку оболочка подставляет все подходящие имена файлов. Вот несколько примеров того, как можно использовать символ * для развертывания имен файлов:

- `at*` — развертывается во все имена файлов, которые начинаются с символов `at`;
- `*at` — развертывается во все имена файлов, которые заканчиваются символами `at`;
- `*at*` — развертывается во все имена файлов, которые содержат символы `at`.

Если ни один из файлов не удовлетворяет шаблону, оболочка не выполняет развертывание и вместо него буквально использует указанные символы, в том числе и *. Попробуйте, например, набрать такую команду, как `echo *dfkdsafh`.

ПРИМЕЧАНИЕ

Если вы привыкли к работе в системе MS-DOS, то вы могли бы инстинктивно набрать символы *.*, которые подходят для всех файлов. Расстаньтесь теперь с этой привычкой. В системе Linux и других версиях системы Unix вы должны использовать символ *. В оболочке Unix комбинация *.* соответствует только тем файлам и каталогам, названия которых содержат точку. Для названий файлов в системе Unix расширения не являются обязательными, и файлы часто обходятся без них.

Еще один джокерный символ, вопросительный знак (`?`), указывает оболочке на то, что необходимо подставить только один произвольный символ. Например, комбинации `b?at` будут соответствовать имена `boat` и `brat`.

Если вы желаете, чтобы оболочка не развертывала джокерный символ в команде, заключите его в одиночные кавычки (`' '`). Так, например, команда `'*'` выдаст символ звездочки. Вы увидите, что это удобно применять в некоторых командах, например `grep` и `find` (они рассматриваются в следующем разделе, подробности об использовании кавычек вы узнаете из раздела 11.2).

ПРИМЕЧАНИЕ

Важно помнить о том, что оболочка выполняет развертывание перед запуском команд и только в это время. Следовательно, если символ * приводит к отсутствию развертывания, оболочка ничего не будет с ним делать; и уже от команды зависит решение о дальнейших действиях.

В современных версиях оболочки существуют дополнительные возможности настройки шаблонов, однако сейчас вам необходимо знать лишь джокерные символы * и ?.

# 2.5. Вспомогательные команды

В приведенных ниже разделах рассмотрены наиболее важные вспомогательные команды системы Unix.

## 2.5.1. Команда grep

Команда `grep` выдает строки из файла или входного потока, которые соответствуют какому-либо выражению. Например, чтобы напечатать строки из файла `/etc/passwd`, которые содержат текст `root`, введите следующее:

```
$ grep root /etc/passwd
```

Команда `grep` чрезвычайно удобна, когда приходится работать одновременно с множеством файлов, поскольку она выдает название файла в дополнение к найденной строке. Например, если вы желаете отыскать все файлы в каталоге `/etc`, которые содержат слово `root`, можно применить данную команду так:

```
$ grep root /etc/*
```

Двумя наиболее важными параметрами команды `grep` являются `-i` (для соответствий, нечувствительных к регистру символов) и `-v` (который инвертирует условие поиска, то есть выдает все строки, не отвечающие условию). Существует

также более мощный вариант команды под названием egrep (это всего лишь синоним команды grep -E).

Команда grep понимает шаблоны, которые известны как *регулярные выражения*. Они укоренились в теории вычислительной техники и являются весьма обычными для утилит системы Unix. Регулярные выражения более мощные, чем шаблоны с джокерными символами, и имеют другой синтаксис. Следует помнить два важных момента, относящихся к регулярным выражениям:

- сочетание .* соответствует любому количеству символов (подобно джокерному символу *);
- символ . соответствует одному произвольному символу.

ПРИМЕЧАНИЕ

Страница руководства grep(1) содержит подробное описание регулярных выражений, но оно может оказаться трудным для восприятия. Чтобы узнать больше, можете прочитать книгу Mastering Regular Expressions («Осваиваем регулярные выражения»), 3-е издание (O'Reilly, 2006) или главу о регулярных выражениях в книге Programming Perl («Программирование на языке Perl»), 4-е издание (O'Reilly, 2012). Если вы любите математику и вам интересно, откуда возникли регулярные выражения, загляните в книгу Automata Theory, Languages and Computation («Теория автоматов, языки и вычисления»), 3-е издание (Prentice Hall, 2006).

## 2.5.2. Команда less

Команда less становится удобной тогда, когда файл довольно большой или когда выводимый результат длинен и простирается за пределы экрана.

Чтобы постранично просмотреть такой большой файл, как /usr/share/dict/words, воспользуйтесь командой less /usr/share/dict/words. После запуска команды less вы увидите содержимое файла, разбитое на части и умещающееся в пределах одного экрана. Нажмите клавишу Пробел, чтобы переместиться далее по содержимому, или клавишу B, чтобы вернуться назад на один экран. Чтобы выйти, нажмите клавишу Q.

ПРИМЕЧАНИЕ

Команда less является улучшенной версией устаревшей команды more. Большинство рабочих станций и серверов на основе системы Linux содержат команду less, однако она не является стандартной для многих встроенных и других систем на основе Unix. Если вы когда-либо столкнетесь с невозможностью использовать команду less, попробуйте применить команду more.

Можно также воспользоваться поиском текста внутри команды less. Например, чтобы отыскать слово *word*, наберите /word. Для поиска в обратном направлении применяйте вариант ?word. Когда результат будет найден, нажмите клавишу N для продолжения поиска.

Как вы узнаете из раздела 2.14, можно отправить стандартный вывод практически из любой команды непосредственно на стандартный вход другой команды. Это исключительно полезно, если команда выводит большое количество результатов, которые вам пришлось бы просеивать с использованием какой-либо команды вроде less. Вот пример отправки результатов команды grep в команду less:

```
$ grep ie /usr/share/dict/words | less
```

Попробуйте самостоятельно поработать с этой командой. Возможно, вы станете часто применять команду `less` подобным образом.

## 2.5.3. Команда pwd

Команда `pwd` (отобразить рабочий каталог) выводит название текущего рабочего каталога. Возникает вопрос: зачем нужна эта команда, если в большинстве версий системы Linux учетные записи настроены так, чтобы в строке приглашения отображался рабочий каталог? На это есть две причины.

Во-первых, не все приглашения включают текущий рабочий каталог, и вам может даже потребоваться избавиться от его отображения, поскольку путь занимает слишком много места. Если это так, вам понадобится команда `pwd`.

Во-вторых, символические ссылки, о которых вы узнаете из подраздела 2.17.2, иногда могут делать неясным подлинный полный путь к текущему рабочему каталогу. Чтобы избежать путаницы, вам потребуется применить команду `pwd -P`.

## 2.5.4. Команда diff

Чтобы увидеть различия между двумя текстовыми файлами, воспользуйтесь командой `diff`:

```
$ diff file1 file2
```

Формат вывода можно контролировать с помощью нескольких параметров, однако для человеческого восприятия наиболее понятен формат, принятый по умолчанию. Тем не менее большинство программистов предпочитает использовать формат `diff -u`, когда приходится отправлять выходные данные еще куда-либо, поскольку автоматизированные устройства могут лучше справиться с таким форматом.

## 2.5.5. Команда file

Если вы видите файл и не уверены в том, какой у него формат, попробуйте использовать команду `file`, чтобы система попыталась выяснить это за вас:

```
$ file file
```

## 2.5.6. Команды find и locate

Бывает досадно, если вы знаете, что какой-либо файл точно расположен где-то в данном дереве каталогов, но вы не помните точно, где именно. Запустите команду `find`, чтобы отыскать файл *file* в каталоге *dir*:

```
$ find dir -name file -print
```

Подобно большинству команд в этом разделе, команда `find` обладает некоторыми интересными особенностями. Однако не пробуйте применять параметры, описанные здесь как `-exec`, пока не выучите наизусть их форму и станете понимать,

зачем нужны параметры `-name` и `-print`. Команда `find` допускает применение специальных шаблонных символов вроде *, но вы должны заключать такие символы в одиночные кавычки (`'*'`), чтобы оградить их от функции универсализации, которая действует в оболочке. Вспомните из подраздела 2.4.4 о том, что оболочка выполняет развертывание джокерных символов *перед* выполнением команд.

В большинстве систем есть также команда `locate` для поиска файлов. Вместо отыскивания файла в реальном времени эта команда осуществляет поиск в индексе файлов, который система периодически создает. Поиск с помощью команды `locate` происходит гораздо быстрее, чем с помощью `find`, но если искомый файл появился после создания индекса, команда `locate` не сможет его найти.

### 2.5.7. Команды head и tail

Чтобы быстро просмотреть фрагмент файла или потока данных, используйте команды `head` и `tail`. Например, команда `head /etc/passwd` отобразит первые десять строк файла с паролем, а команда `tail /etc/passwd` покажет заключительные десять строк.

Чтобы изменить количество отображаемых строк, применяйте параметр `-n`, в котором число *n* равно количеству строк, которые необходимо увидеть (например, `head -5 /etc/passwd`). Чтобы вывести строки, начиная со строки под номером *n*, используйте команду `tail +n`.

### 2.5.8. Команда sort

Команда `sort` быстро выстраивает строки текста в алфавитно-числовом порядке. Если строки файла начинаются с чисел и вам необходимо выстроить их в порядке следования чисел, применяйте параметр `-n`. Параметр `-r` изменяет порядок следования на обратный.

## 2.6. Изменение вашего пароля и оболочки

Для изменения своего пароля воспользуйтесь командой `passwd`. Система попросит вас указать старый пароль, а затем дважды пригласит ввести новый. Выбирайте пароль, который не содержит реальных слов какого-либо языка, а также не старайтесь комбинировать слова.

Один из простейших способов придумать хороший пароль состоит в следующем. Выберите какую-либо фразу, составьте из нее акроним (оставив только первые буквы входящих в нее слов), а затем измените этот акроним с помощью цифр и знаков пунктуации. После этого вам понадобится только запомнить исходную фразу.

Вы можете сменить оболочку (на альтернативную, например `ksh` или `tcsh`) с помощью команды `chsh`, но помните о том, что в данной книге предполагается, что вы работаете в оболочке `bash`.

## 2.7. Файлы с точкой

Перейдите в домашний каталог, посмотрите его содержимое с помощью команды `ls`, а затем запустите команду `ls -a`. Видите различия в результатах вывода? После запуска команды `ls` без параметра `-a` вы не увидите конфигурационные файлы, которые называются *файлами с точкой*. Имена таких файлов и каталогов начинаются с точки (`.`). Обычными файлами с точкой являются файлы `.bashrc` и `.login`. Существуют также и каталоги с точкой, например `.ssh`.

У файлов или каталогов с точкой нет ничего особенного. Некоторые команды по умолчанию не показывают их, чтобы при отображении содержимого домашнего каталога вы не увидели полнейшую неразбериху. Так, например, команда `ls` не показывает файлы с точкой, если не указан параметр `-a`. Кроме того, паттерны оболочки не рассматривают файлы с точкой, если вы намеренно не укажете это с помощью шаблона `.*`.

ПРИМЕЧАНИЕ

У вас могут возникнуть трудности с шаблонами, поскольку комбинации .* соответствуют варианты . и .. (то есть текущий и родительский каталоги). Следует использовать шаблоны вроде .[^.]* или .??*, чтобы получить список всех файлов с точкой, кроме текущего и родительского каталогов.

## 2.8. Переменные окружения и оболочки

Оболочка может хранить временные переменные, которые называются *переменными оболочки* и содержат значения текстовых строк. Переменные оболочки весьма полезны для отслеживания значений в сценариях. Некоторые переменные оболочки контролируют режим работы оболочки (например, оболочка `bash` считывает значение переменной `PS1` перед отображением приглашения).

Чтобы присвоить значение переменной оболочки, используйте знак равенства (`=`). Вот простой пример:

```
$ STUFF=blah
```

В этом примере переменной с именем `STUFF` присваивается значение `blah`. Чтобы обратиться к этой переменной, применяйте синтаксис `$STUFF` (попробуйте, например, запустить команду `echo $STUFF`). Множество вариантов использования переменных оболочки приведено в главе 11.

*Переменная окружения* подобна переменной оболочки, но она не является привязанной к оболочке. Все процессы в системах Unix пользуются хранилищем переменных окружения. Основное отличие переменных окружения от переменных оболочки заключается в том, что операционная система передает все переменные окружения вашей оболочки командам, запускаемым оболочкой, в то время как переменные оболочки не могут быть доступны командам, которые вы запускаете.

Назначение переменной окружения производится с помощью команды `export`. Если, например, вы желаете сделать переменную оболочки `$STUFF` переменной окружения, используйте следующий синтаксис:

```
$ STUFF=blah
$ export STUFF
```

Переменные окружения практичны, поскольку многие команды считывают из них значения своих настроек и параметров. Например, вы можете поместить ваши излюбленные параметры для команды `less` в переменную окружения `LESS`, и тогда команда `less` будет использовать их при запуске. Многие страницы руководства содержат раздел с названием Environment («Окружение»), в котором описаны такие переменные.

# 2.9. Командный путь

`PATH` является специальной переменной окружения, которая содержит *командный путь*, или просто *путь*. Командный путь — это перечень системных каталогов, которые просматривает оболочка, пытаясь найти какую-либо команду. Например, когда вы запускаете команду `ls`, оболочка просматривает каталоги, перечисленные в переменной `PATH`, в поисках команды `ls`. Если команда встречается сразу в нескольких каталогах, оболочка запустит первый найденный экземпляр.

Если вы запустите команду `echo $PATH`, вы увидите, что компоненты пути отделены с помощью двоеточия (`:`). Например, так:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin
```

Чтобы указать оболочке дополнительные места для поиска команд, измените переменную окружения `PATH`. Например, с помощью приведенной ниже команды можно добавить в начало пути каталог *`dir`*, чтобы оболочка начала просмотр каталогов с него, до других каталогов из переменной `PATH`.

```
$ PATH=dir:$PATH
```

Можно также добавить название каталога в конец переменной `PATH`, тогда оболочка обратится к нему в последнюю очередь:

```
$ PATH=$PATH:dir
```

ПРИМЕЧАНИЕ

Будьте осторожны при изменении пути, поскольку вы можете случайно полностью стереть его, если неправильно наберете $PATH. Если это случилось, не следует паниковать. Ущерб не является необратимым, вы можете просто запустить новую оболочку. Чтобы сильнее нарушить работу, необходимо сделать ошибку при редактировании некоторого файла конфигурации, но даже и в этом случае ее нетрудно исправить. Один из простейших способов вернуться в нормальное состояние — закрыть текущее окно терминала и открыть другое.

# 2.10. Специальные символы

Вам следует знать названия некоторых специальных символов системы Linux. Если вам нравятся вещи подобного рода, загляните на страницу *Jargon File* (http://www.catb.org/jargon/html/) или посмотрите ее печатную версию в книге *The New Hacker's Dictionary* («Новый словарь хакера») (MIT Press, 1996).

В табл. 2.1 описан набор специальных символов, многие из них вы уже встречали в этой главе. Некоторые утилиты, например язык программирования Perl, используют почти все эти символы! Помните о том, что здесь приведены американские варианты названий символов.

**Таблица 2.1.** Специальные символы

| Символ | Название | Применение |
|---|---|---|
| * | Звездочка | Регулярные выражения, джокерные символы |
| . | Точка | Текущий каталог, разделитель имени файла и хоста |
| ! | Восклицательный знак | Отрицание, история команд |
| \| | Вертикальная черта | Командный конвейер |
| / | Прямой слеш | Разделитель каталогов, команда поиска |
| \ | Обратный слеш | Литералы, макросы (но не каталоги) |
| $ | Доллар | Обозначение переменной, конец строки |
| ' | Одиночная кавычка | Буквенные строки |
| ` | «Обратная галочка» | Замена команды |
| " | Двойная кавычка | Частично буквенные строки |
| ^ | Знак вставки | Отрицание, начало строки |
| ~ | Тильда | Отрицание, ярлык каталога |
| # | Диез, знак фунта | Комментарии, препроцессор, подстановки |
| [ ] | Квадратные скобки | Массивы |
| { } | Фигурные скобки | Блоки инструкций, массивы |
| _ | Подчеркивание | Просто замена символа пробела |

ПРИМЕЧАНИЕ

Вам часто будут встречаться символы, снабженные знаком вставки; например, ^C вместо Ctrl+C.

# 2.11. Редактирование командной строки

Во время работы с оболочкой обратите внимание, что вы можете редактировать командную строку, используя клавиши ← и →, а также перемещаться к предыдущим командам с помощью клавиш ↑ и ↓. Такой способ работы является стандартным в большинстве систем Linux.

Кроме того, вместо них можно применять управляющие сочетания клавиш. Если вы запомните сочетания, перечисленные в табл. 2.2, то обнаружите, что у вас появились дополнительные возможности ввода текста во многих командах Unix по сравнению с использованием стандартных клавиш.

**Таблица 2.2.** Сочетания клавиш для командной строки

| Сочетание клавиш | Действие |
|---|---|
| Ctrl+B | Перемещение курсора влево |
| Ctrl+F | Перемещение курсора вправо |
| Ctrl+P | Просмотр предыдущей команды (или перемещение курсора вверх) |
| Ctrl+N | Просмотр следующей команды (или перемещение курсора вниз) |

| Сочетание клавиш | Действие |
|---|---|
| Ctrl+A | Перемещение курсора в начало строки |
| Ctrl+E | Перемещение курсора в конец строки |
| Ctrl+W | Удаление предыдущего слова |
| Ctrl+U | Удаление текста от курсора до начала строки |
| Ctrl+K | Удаление текста от курсора до конца строки |
| Ctrl+Y | Вставка удаленного текста (например, после команды CTRL+U) |

# 2.12. Текстовые редакторы

Для работы в системе Unix вы должны уметь редактировать текстовые файлы, не повреждая их. Большинство частей системы использует для конфигурации простые текстовые файлы (подобные тем, которые расположены в каталоге `/etc`). Редактировать их несложно, но вам предстоит делать это часто, поэтому понадобится мощный инструмент.

Попробуйте освоить один из двух редакторов, которые де-факто являются стандартом для системы Unix: vi и Emacs. Большинство «кудесников» Unix относятся к выбору редактора с большим трепетом. Не прислушивайтесь к ним — выбирайте редактор сами, вам будет проще его освоить.

При выборе редактора принимайте во внимание следующие моменты.

- Если вам необходим редактор, который может практически все, а также обладает обширной справочной системой и при этом вы не возражаете против набора дополнительных символов для использования различных функций, попробуйте редактор Emacs.
- Если скорость работы превыше всего, выберите редактор vi. Работа в нем чем-то напоминает видеоигру.

ПРИМЕЧАНИЕ

Книга Learning the vi and Vim Editors: Unix Text Processing («Изучаем редакторы vi и Vim: работа с текстом в системе Unix»), 7-е издание (O'Reilly, 2008), расскажет вам все, что необходимо знать о редакторе vi. В редакторе Emacs можно воспользоваться справочной системой, запустив редактор, а затем нажав сочетание клавиш Ctrl+H и клавишу T. Можно также почитать книгу GNU Emacs Manual («Руководство по редактору GNU Emacs») (Free Software Foundation, 2011).

У вас может появиться искушение поэкспериментировать на начальных этапах с более дружественным редактором, таким как Pico или одним из множества имеющихся GUI-редакторов. Однако, если вы сразу привыкаете к программе, лучше не следовать такой практике.

ПРИМЕЧАНИЕ

Именно при редактировании текста вы впервые увидите отличие терминала от графического интерфейса пользователя. Редакторы вроде vi запускаются внутри окна терминала с помощью стандартного интерфейса ввода/вывода. Редакторы с графическим интерфейсом запускаются в собственном окне, которое не зависит от терминала и снабжено собственным интерфейсом. Редактор Emacs по умолчанию запускается в GUI-режиме, но может быть запущен и в окне терминала.

# 2.13. Получение интерактивной справки

Операционные системы Linux снабжены документацией. *Страницы руководства* расскажут вам обо всем необходимом для основных команд. Например, чтобы увидеть страницу руководства по команде `ls`, введите следующий запрос:

```
$ man ls
```

Большинство страниц руководства содержат главным образом справочную информацию, возможно, с некоторыми примерами и перекрестными ссылками. Не ожидайте увидеть учебное пособие и не рассчитывайте на воодушевляющий литературный стиль.

Когда у команды много параметров, они перечисляются на странице руководства в каком-либо систематизированном виде (например, в алфавитном порядке). Степень их важности при этом определить невозможно. Если вы достаточно терпеливы, то в большинстве случаев сможете найти все необходимые сведения в руководстве.

Чтобы выполнить поиск на странице руководства по ключевому слову, используйте параметр `-k`:

```
$ man -k keyword
```

Это полезно, если вы не вполне уверены в названии необходимой команды. Например, если вы ищете команду для сортировки чего-либо, введите следующее:

```
$ man -k sort
--snip--
comm (1)         - сравнивает два отсортированных файла строка за строкой
qsort (3)        - сортирует массив
sort (1)         - сортирует строки текстового файла
sortm (1)        - сортирует сообщения
tsort (1)        - выполняет топологическую сортировку
--snip--
```

Результат включает название страницы руководства, раздел руководства (см. ниже), а также краткое описание того, что содержит данная страница руководства.

ПРИМЕЧАНИЕ

Если у вас есть какие-либо вопросы насчет команд, рассмотренных в предыдущих разделах, можете попробовать найти ответы с помощью команды man.

На страницы руководства ссылаются с помощью пронумерованных разделов. Когда кто-либо приводит ссылку на страницу руководства, то в скобках после названия указывается номер раздела, например, так: `ping(8)`. В табл. 2.3 перечислены все разделы с их нумерацией.

**Таблица 2.3.** Разделы интерактивного руководства

| Раздел | Описание |
|---|---|
| 1 | Команды пользователя |
| 2 | Системные вызовы |
| 3 | Документация к высокоуровневой программной библиотеке Unix |

| Раздел | Описание |
|---|---|
| 4 | Интерфейс устройств и информация о драйверах |
| 5 | Описание файлов (файлы конфигурации системы) |
| 6 | Игры |
| 7 | Форматы файлов, условные обозначения и кодировки (ASCII, суффиксы и т. д.) |
| 8 | Системные команды и серверы |

Разделы 1, 5, 7 и 8 служат хорошим дополнением к данной книге. Раздел 4 может пригодиться в редких случаях, а раздел 6 был бы замечателен, если бы его сделали немного больше. Вероятно, вы не сможете воспользоваться разделом 3, если вы не программист, однако вам удастся понять раздел 2, когда вы узнаете из этой книги немного больше о системных вызовах.

Вы можете выбрать страницу руководства по разделу. Это важно, поскольку система отображает первую из страниц, на которой ей удается обнаружить нужное понятие. Например, чтобы прочитать описание файла `/etc/passwd` (а не команды `passwd`), можно добавить номер раздела перед названием страницы:

```
$ man 5 passwd
```

Страницы руководства содержат основные сведения, но есть и другие способы получить интерактивную справку. Если вам необходимо узнать что-либо о параметре команды, введите название команды, а затем добавьте параметр `--help` или `-h` (варианты различны для разных команд). В результате вы можете получить множество ответов (как, например, в случае с командой `ls --help`) или, возможно, сразу то, что нужно.

Когда-то участники проекта GNU Project решили, что им не очень нравятся страницы руководства, поэтому появился другой формат справки — `info` (или `texinfo`). Зачастую объем этой документации больше, чем у обычных страниц руководства, и иногда она более сложная. Чтобы получить доступ к этой информации, введите команду `info` и укажите название команды:

```
$ info command
```

Некоторые версии системы сваливают всю доступную документацию в каталог `/usr/share/doc`, не заботясь о предоставлении какой-либо справочной системы вроде `man` или `info`. Загляните в этот каталог, если вам необходима документация, и, конечно же, поищите в Интернете.

## 2.14. Ввод и вывод с помощью оболочки

Теперь, когда вы знакомы с основными командами системы Unix, файлами и каталогами, вы готовы к изучению перенаправления стандартных ввода и вывода. Начнем со стандартного вывода.

Чтобы направить результат команды *command* в файл, а не в терминал, используйте символ перенаправления >:

```
$ command > file
```

Оболочка создаст файл *file*, если его еще нет. Если такой файл существует, то оболочка сначала *сотрет* его *(затрет данные)*. Некоторые оболочки снабжены параметрами, предотвращающими затирание данных. В оболочке bash, например, наберите для этого set -C.

Вы можете добавить выводимые данные к файлу вместо его перезаписи с помощью такого синтаксиса перенаправления >>:

```
$ command >> file
```

Это удобный способ собрать все выходные данные в одном месте, когда выполняется последовательность связанных команд.

Чтобы отправить стандартный вывод какой-либо команды на стандартный вход другой команды, используйте символ вертикальной черты (|). Чтобы понять, как он работает, попробуйте набрать следующие команды:

```
$ head /proc/cpuinfo
$ head /proc/cpuinfo | tr a-z A-Z
```

Можно пропустить выходные данные через любое количество команд. Для этого добавляйте вертикальную черту перед каждой дополнительной командой.

## 2.14.1. Стандартная ошибка

Иногда при перенаправлении стандартного вывода вы можете обнаружить, что команда выводит в терминал что-то еще. Это называется *стандартной ошибкой* (stderr). Она является дополнительным выходным потоком для диагностики и отладки.

Например, такая команда вызовет ошибку:

```
$ ls /fffffffff > f
```

После ее выполнения файл f должен быть пустым, однако вы увидите в терминале следующее сообщение стандартной ошибки:

```
ls: cannot access /fffffffff: No such file or directory
```

Если желаете, можете перенаправить стандартную ошибку. Чтобы, например, отправить стандартный вывод в файл f, а стандартную ошибку в файл e, используйте синтаксис 2> следующим образом:

```
$ ls /fffffffff > f 2> e
```

Число 2 определяет *идентификатор потока*, который изменяет оболочка. Значение 1 соответствует стандартному выводу (по умолчанию), а значение 2 — стандартной ошибке.

Вы можете также направить стандартную ошибку туда же, куда осуществляется стандартный вывод, с помощью нотации >&.

Например, чтобы отправить стандартный вывод и стандартную ошибку в файл f, попробуйте такую команду:

```
$ ls /fffffffff > f 2>&1
```

### 2.14.2. Перенаправление стандартного ввода

Чтобы отправить файл на стандартный ввод команды, используйте оператор <:

```
$ head < /proc/cpuinfo
```

Вам может встретиться команда, которой потребуется указанный выше тип перенаправления, но поскольку большинство команд системы Unix принимает имена файлов в качестве аргументов, такое происходит нечасто. Эту команду можно было бы переписать в виде `head /proc/cpuinfo`.

## 2.15. Объяснение сообщений об ошибках

Когда у вас возникают проблемы при работе в системе, похожей на Unix (например, в Linux), вы *должны* читать сообщения об ошибках. В отличие от сообщений в других операционных системах, в системе Unix ошибки, как правило, точно указывают вам на то, что именно вышло из строя.

### 2.15.1. Структура сообщений об ошибке в Unix

Большинство команд системы Unix выдает одинаковые основные сообщения об ошибках, однако окончательный вид может немного различаться для разных команд. Вот пример сообщения, которое в том или ином виде обязательно вам встретится:

```
$ ls /dsafsda
ls: cannot access /dsafsda: No such file or directory
```

Это сообщение состоит из трех частей.

- Название команды: `ls`. Некоторые команды опускают такую идентифицирующую информацию, и это может раздражать при написании сценариев оболочки, хотя, по сути, это не так уж и важно.
- Имя файла, `/dsafsda`, которое является более конкретной информацией. Указанный путь содержит ошибку.
- Сообщение об ошибке `No such file or directory` указывает на ошибку в имени файла.

Если собрать эти части воедино, получится нечто вроде «команда `ls` пыталась открыть файл `/dsafsda`, но не смогла, поскольку такого файла нет». Это может казаться очевидным, однако подобные сообщения сбивают с толку, если вы запустите сценарий оболочки, который содержит ошибку в другой команде.

При устранении ошибок всегда начинайте разбираться с первой ошибки. Некоторые команды сообщают о том, что они ничего не смогут сделать до выяснения причин других ошибок. Представьте, например, что вы запускаете выдуманную команду под названием `scumd`, которая выдает такое сообщение об ошибке:

```
scumd: cannot access /etc/scumd/config: No such file or directory
```

За ним следует огромный перечень других сообщений об ошибках, который выглядит катастрофически. Не отвлекайтесь на остальные ошибки. Скорее всего, вам всего лишь надо создать файл `/etc/scumd/config`.

ПРИМЕЧАНИЕ

Не смешивайте сообщения об ошибках с предупреждениями. Предупреждения часто выглядят как ошибки, но они содержат слово warning. Они говорят о наличии какой-либо неисправности, однако команда будет пытаться продолжить работу. Чтобы устранить проблему, указанную в предупреждении, вам потребуется отыскать ошибочный процесс и завершить его, прежде чем делать что-либо еще (о списке процессов и об их завершении вы узнаете из раздела 2.16).

## 2.15.2. Общие ошибки

Многие ошибки, которые вы встретите при выполнении команд системы Unix, являются результатом неправильных действий с файлами или процессами. Приведем «хит-парад» сообщений об ошибках.

### No such file or directory

Вероятно, вы пытались получить доступ к несуществующему файлу. Поскольку ввод/вывод в системе Unix не делает различий между файлами и каталогами, это сообщение об ошибке появляется везде. Вы получите его, если попытаетесь выполнить чтение несуществующего файла, или решите перейти в отсутствующий каталог, или попробуете записать файл в несуществующий каталог и т. д.

### File exists

В данном случае вы, возможно, пытались создать файл, который уже существует. Это часто бывает, когда вы создаете каталог, имя которого уже занято каким-либо файлом.

### Not a directory, Is a directory

Эти сообщения возникают, когда вы пытаетесь использовать файл в качестве каталога или каталог в качестве файла. Например, так:

```
$ touch a
$ touch a/b
touch: a/b: Not a directory
```

Обратите внимание на то, что сообщение об ошибке относится только к части `a` пути `a/b`. Когда вы столкнетесь с такой проблемой, вам потребуется время, чтобы отыскать компонент пути, с которым обращаются как с каталогом.

### No space left on device

На вашем жестком диске закончилось свободное пространство.

### Permission denied

Эта ошибка возникает, когда вы пытаетесь выполнить чтение или запись, указав файл или каталог, к которым вам не разрешен доступ (вы обладаете недостаточными правами). Эта ошибка говорит также о том, что вы пытаетесь запустить файл,

для которого не установлен бит выполнения (даже если вы можете читать этот файл). Из раздела 2.17 вы больше узнаете о правах доступа.

### Operation not permitted

Обычно такая ошибка возникает, когда вы пытаетесь завершить процесс, владельцем которого не являетесь.

### Segmentation fault, Bus error

Суть *ошибки сегментации* состоит в том, что разработчик программы, которую вы только что запустили, где-то ошибся. Программа пыталась получить доступ к области памяти, к которой ей не разрешено обращаться, в результате операционная система завершила работу программы. Подобно ей, *ошибка шины* означает, что программа пыталась получить доступ к памяти недолжным образом. Если вы получаете одну из этих ошибок, то, вероятно, вы передали на ввод программы какие-либо неожиданные для нее данные.

## 2.16. Получение списка процессов и управление ими

*Процесс* — это работающая программа. Каждому процессу в системе присвоен числовой *идентификатор процесса* (PID). Чтобы быстро получить перечень работающих процессов, запустите команду `ps`. Вы получите результат вроде этого:

```
$ ps
  PID TTY STAT TIME COMMAND
  520  p0 S    0:00 -bash
  545   ? S    3:59 /usr/X11R6/bin/ctwm -W
  548   ? S    0:10 xclock -geometry -0-0
 2159  pd SW   0:00 /usr/bin/vi lib/addresses
31956  p3 R    0:00 ps
```

Эти поля означают следующее.

- `PID` — идентификатор процесса.
- `TTY` — оконечное устройство, в котором запущен процесс (об этом подробнее чуть позже).
- `STAT` — статус процесса, а именно: что выполняет данный процесс и где расположена отведенная для него память. Например, символ `S` обозначает ждущий процесс, а символ `R` — работающий (описание всех символов можно найти на странице ps(1) в руководстве).
- `TIME` — количество времени центрального процессора в минутах и секундах, которое использовал данный процесс к настоящему моменту. Другими словами, это общее количество времени, потраченное процессом на выполнение инструкций в процессоре.
- `COMMAND` — поле может показаться очевидным, однако имейте в виду, что процесс может изменить исходное значение этого поля.

## 2.16.1. Параметры команды ps

Команда ps обладает множеством параметров. Чтобы запутать дело еще больше, можно указывать параметры в трех разных стилях: Unix, BSD и GNU. Многие пользователи считают, что стиль BSD наиболее удобен (вероятно, в силу большей краткости набора), поэтому в данной книге мы будем применять именно его. В табл. 2.4 приведено несколько наиболее полезных сочетаний параметров.

**Таблица 2.4.** Параметры команды ps

| Команда с параметром | Описание |
|---|---|
| ps x | Показать все процессы, запущенные вами |
| ps ax | Показать все процессы системы, а не только те, владельцем которых являетесь вы |
| ps u | Включить детализированную информацию о процессах |
| ps w | Показать полные названия команд, а не только те, что помещаются в одной строке |

Как и в других командах, эти параметры можно комбинировать, например, так: ps aux или ps auxw. Чтобы проверить какой-либо конкретный процесс, добавьте значение PID к списку аргументов команды ps. Например, чтобы просмотреть информацию о текущем процессе оболочки, можно использовать команду ps u $$, поскольку параметр $$ является переменной оболочки, которая содержит значение PID текущего процесса. В главе 8 вы найдете информацию о командах администрирования top и lsof. Они могут пригодиться при локализации процессов, причем не обязательно тогда, когда необходимо заниматься обслуживанием системы.

## 2.16.2. Завершение процессов

Чтобы завершить процесс, отправьте ему *сигнал* с помощью команды kill. Сигнал — это сообщение процессу от ядра. Когда вы запускаете команду kill, вы просите ядро отправить сигнал другому процессу. В большинстве случаев для этого необходимо набрать следующее:

```
$ kill pid
```

Существует много типов сигналов. По умолчанию используется сигнал TERM («прервать»). Вы можете отправлять другие сигналы, добавляя параметр в команду kill. Например, чтобы приостановить процесс, не завершая его, применяйте сигнал STOP:

```
$ kill -STOP pid
```

Остановленный процесс остается в памяти и ожидает повторного вызова. Используйте сигнал CONT, чтобы продолжить выполнение этого процесса:

```
$ kill -CONT pid
```

ПРИМЕЧАНИЕ

Применение сочетания клавиш Ctrl+C для прерывания процесса, работающего в терминале, равносильно вызову команды kill для завершения процесса по сигналу INT («прервать»).

Самый грубый способ завершения процесса — с помощью сигнала KILL. Другие сигналы дают процессу шанс прибрать за собой, а сигнал KILL — нет. Операционная система прерывает процесс и принудительно выгружает его из памяти. Используйте это в качестве последнего средства. Не следует завершать процессы без разбора, особенно если вы не знаете, что они делают.

Вы можете увидеть, что некоторые пользователи вводят числа вместо названий параметров команды kill. Например, kill -9 вместо kill -KILL. Это объясняется тем, что ядро использует числа для обозначения различных сигналов. Можете применять команду kill подобным образом, если вам известно число, которое соответствует необходимому сигналу.

## 2.16.3. Управление заданиями

Оболочка поддерживает также *управление заданиями* — один из способов отправки командам сигналов TSTP (подобен сигналу STOP) и CONT с помощью различных сочетаний клавиш и команд. Например, вы можете отправить сигнал TSTP с помощью сочетания клавиш Ctrl+Z, а затем возобновить процесс командой fg (вывести из фона) или bg (перевести в фон; см. следующий раздел). Несмотря на практичность управления заданиями и его привычное использование многими опытными пользователями, оно не является необходимым и может запутать начинающих. Очень часто пользователи нажимают сочетание Ctrl+Z вместо Ctrl+C, забывают о запущенных процессах и в итоге получают множество «подвешенных» процессов.

СОВЕТ

Чтобы узнать, не висят ли в текущем терминале приостановленные процессы, выполните команду jobs.

Если вы намерены использовать несколько оболочек, запустите каждую из них в отдельном окне терминала, переведите в фон неинтерактивные процессы (см. следующий раздел) или научитесь использовать команду screen.

## 2.16.4. Фоновые процессы

Обычно при запуске из оболочки команды в системе Unix вы не увидите строки приглашения, пока команда не завершит работу. Тем не менее можно отделить процесс от оболочки и поместить его в «фон» с помощью символа амперсанда (&); после этого строка приглашения вернется. Если вам необходимо распаковать большой архив с помощью команды gunzip (о ней вы узнаете из раздела 2.18), а тем временем вы намерены заняться чем-либо другим, запустите команду такого вида:

```
$ gunzip file.gz &
```

Оболочка должна в ответ выдать номер PID нового фонового процесса, а строка приглашения появится немедленно, чтобы вы смогли работать далее. Процесс продолжит свое выполнение и после того, как вы выйдете из системы. Это чрезвычайно удобно, если приходится запускать программу, которая производит довольно много вычислений. В зависимости от настроек системы оболочка может уведомить вас о завершении процесса.

Обратной стороной фоновых процессов является то, что они могут ожидать начала работы со стандартным вводом (и, хуже того, выполнять чтение прямо из терминала). Если фоновая программа пытается считывать что-либо из стандартного ввода, она может зависнуть (попробуйте команду `fg`, чтобы вызвать ее из фона) или прекратить работу. К тому же, если программа выполняет запись в стандартный вывод или стандартную ошибку, все это может отобразиться в терминале вне всякой связи с какими-либо выполняемыми командами: вы можете увидеть неожиданный результат, пока работаете над чем-то другим.

Лучший способ добиться того, чтобы фоновый процесс не беспокоил вас, — перенаправить его вывод (и, возможно, ввод), как описано в разделе 2.14.

Узнайте о том, как обновить окно терминала. Оболочка `bash` и большинство полноэкранных интерактивных программ поддерживают сочетание клавиш Ctrl+L для повторной отрисовки всего экрана. Если программа производит считывание из стандартного ввода, сочетание клавиш Ctrl+R обычно обновляет текущую строку, однако нажатие неверного сочетания в неподходящий момент может привести вас к более печальной ситуации, чем была раньше. Нажав, например, Ctrl+R в строке приглашения оболочки `bash`, вы окажетесь в режиме реверсивного поиска `isearch mode` (для выхода из него нажмите клавишу Esc).

# 2.17. Режимы файлов и права доступа

Каждый файл системы Unix обладает набором *прав доступа*, которые определяют, можете ли вы читать, записывать или запускать данный файл. Команда `ls -l` отображает эти права доступа. Вот пример такой информации:

```
-rw-r--r--❶ 1 juser somegroup  7041 Mar 26 19:34 endnotes.html
```

*Режим* файла ❶ представляет права доступа и некоторые дополнительные сведения. Режим состоит из четырех частей, как показано на рис. 2.1. Первый символ в режиме обозначает тип файла. Дефис (-) на этом месте, как в примере, указывает на то, что файл является *обычным* и не содержит никаких особенностей. Безусловно, это самый распространенный тип файлов. Каталоги также весьма обычны и обозначаются символом `d` в позиции, указывающей тип файла (в разделе 3.1).

Оставшаяся часть режима файла содержит сведения о правах доступа, которые разделены на три группы: *права пользователя*, *права группы* и *другие права* — в указанном порядке. Символы `rw-` в приведенном примере относятся к правам доступа пользователя, за ними следуют символы `r--`, определяющие права доступа для группы, и, наконец, символы `r–` определяют другие права доступа.

Каждый из наборов прав доступа может содержать четыре основных обозначения (табл. 2.5).

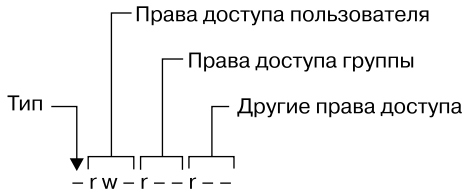


**Рис. 2.1.** Составляющие режима файла

**Таблица 2.5.** Обозначение прав доступа

| Обозначение | Описание |
|---|---|
| r | Файл доступен для чтения |
| w | Файл доступен для записи |
| x | Файл является исполнимым (можно запустить его как программу) |
| – | Ничего не обозначает |

Права доступа пользователя (первый набор символов) относятся к пользователю, который является владельцем файла. В приведенном выше примере это `juser`. Второй набор символов относится к группе (в данном случае `somegroup`). Любой пользователь из этой группы обладает указанными правами доступа. Воспользуйтесь командой `groups`, чтобы узнать, в какую группу вы входите. Дополнительную информацию можно получить в подразделе 7.3.5.

В третьем наборе (другие права доступа) указано, кто еще будет обладать в системе правами доступа. Их часто называют правами доступа *для всех*.

ПРИМЕЧАНИЕ

Каждая позиция, содержащая обозначение права доступа на чтение, записи или исполнение, иногда называется битом прав доступа. По этой причине вы можете слышать, как пользователи упоминают про «биты доступа для чтения».

У некоторые исполняемых файлов в правах доступа пользователя вместо символа `x` указан символ `s`. Это говорит о том, что исполняемый файл является файлом `setuid` — при его выполнении он запускается так, словно владельцем файла является пользователь с указанным идентификатором, а не вы. Многие команды используют бит `setuid`, чтобы получить корневые права доступа, необходимые для изменения системных файлов. В качестве примера можно привести команду `passwd`, которой необходимо изменять файл `/etc/passwd`.

## 2.17.1. Изменение прав доступа

Чтобы изменить права доступа, используйте команду `chmod`. Сначала укажите набор прав, который вы желаете изменить, а затем укажите бит, подлежащий изменению. Например, чтобы добавить права доступа на чтение файла для группы (`g`) и всех пользователей (`o`, по первой букве слова other — «остальные»), нужно запустить следующие две команды:

```
$ chmod g+r file
$ chmod o+r file
```

Можно также объединить их таким образом:

```
$ chmod go+r file
```

Чтобы удалить эти права доступа, используйте `go-r` вместо `go+r`.

ПРИМЕЧАНИЕ

Довольно очевидно, что не следует предоставлять всем пользователям права доступа на запись файла, поскольку каждый получит возможность изменить системные файлы. Но сможет ли при этом кто-либо, подключившись через Интернет, изменить ваши файлы? Вероятно, не сможет, если только в вашей системе нет уязвимости в сетевой защите. Если же она уязвима, то права доступа к файлам ничем вам не помогут.

Иногда пользователи меняют права доступа, указывая числа, например, так:

```
$ chmod 644 file
```

Такой способ называется *абсолютным* изменением, поскольку при нем сразу же устанавливаются все биты прав доступа. Чтобы разобраться, как это устроено, вам необходимо понять, как представлять биты прав доступа в восьмеричной форме (каждое значение является числом в системе счисления по основанию 8 и соответствует набору прав доступа). Дополнительную информацию об этом можно прочитать в руководстве на странице chmod(1).

Вам не обязательно знать, как составить абсолютные значения режимов, запомните те из них, которыми вы будете пользоваться чаще всего. В табл. 2.6 перечислены наиболее распространенные варианты.

**Таблица 2.6.** Абсолютные режимы прав доступа

| Режим | Значение | Применение |
|---|---|---|
| 644 | пользователь: чтение/запись;<br>группа, другие: чтение | Файлы |
| 600 | пользователь: чтение/запись;<br>группа, другие: нет | Файлы |
| 755 | пользователь: чтение/запись/исполнение;<br>группа, другие: чтение/исполнение | Каталоги, команды |
| 700 | пользователь: чтение/запись/исполнение;<br>группа, другие: нет | Каталоги, команды |
| 711 | пользователь: чтение/запись/исполнение;<br>группа, другие: исполнение | Каталоги |

Каталогам также можно назначить права доступа. Вы можете вывести список содержимого каталога, если он доступен для чтения, но доступ к файлу в этом каталоге можно получить лишь в том случае, если каталогу назначено право доступа на исполнение. Часто при указании прав доступа к каталогам пользователи совершают ошибку, ненароком удаляя разрешение на исполнение при применении абсолютных значений режимов.

Наконец, вы можете указать набор прав доступа по умолчанию с помощью команды оболочки `umask`, которая применяет заранее определенный набор прав доступа к любому создаваемому новому файлу. В общем, применяйте команду `umask 022`, если вы желаете, чтобы каждый мог видеть все создаваемые вами фай-

лы и каталоги, или команду `umask 077` в противном случае. Вам необходимо поместить команду `umask` с указанием желаемого режима в один из файлов запуска системы, чтобы новые права доступа по умолчанию применялись в следующих сеансах работы (см. главу 13).

## 2.17.2. Символические ссылки

*Символическая ссылка* — это файл, который указывает на другой файл или каталог, создавая, по сути, псевдоним (подобно ярлыку в Windows). Символические ссылки обеспечивают быстрый доступ к малопонятным путям каталогов.

В длинном списке каталогов символические ссылки будут выглядеть примерно так (обратите внимание на символ `l`, указанный в режиме файла в качестве типа файла):

```
lrwxrwxrwx 1 ruser users 11 Feb 27 13:52 somedir -> /home/origdir
```

Если вы попытаетесь получить доступ к каталогу `somedir` в данном каталоге, система выдаст вам вместо этого каталог `/home/origdir`. Символические ссылки являются лишь именами, указывающими на другие имена. Их названия, а также пути, на которые они указывают, не обязаны что-либо значить. Так, например, каталог `/home/origdir` может и вовсе отсутствовать.

В действительности, если каталог `/home/origdir` не существует, любая команда, которая обратится к каталогу `somedir`, сообщит о том, что его нет (кроме команды `ls somedir`, которая проинформирует вас лишь о том, что каталог `somedir` является каталогом `somedir`). Это может сбивать с толку, поскольку вы прямо перед собой видите нечто с именем `somedir`.

Это не единственный случай, когда символические ссылки могут вводить в заблуждение. Еще одна проблема состоит в том, что вы не можете установить характеристики объекта, на который указывает ссылка, просто посмотрев на название ссылки; вы должны перейти по ней, чтобы понять, ведет ли она к файлу или в каталог. В системе могут быть также ссылки, указывающие на другие ссылки — *цепные символические ссылки*.

## 2.17.3. Создание символических ссылок

Чтобы создать символическую ссылку от цели *target* к имени ссылки *linkname*, воспользуйтесь командой `ln -s`:

```
$ ln -s target linkname
```

Аргумент *linkname* является именем символической ссылки, аргумент *target* задает путь к файлу или каталогу, к которому ведет ссылка, а флаг `-s` определяет символическую ссылку (см. следующее за этим предупреждение).

При создании символической ссылки перепроверьте команду перед ее запуском. Например, если вы поменяете аргументы местами (`ln -s linkname target`), то возникнет забавная ситуация, если каталог *linkname* уже существует. Когда такое происходит (а это случается часто), команда `ln` создает ссылку с именем *target* внутри каталога *linkname*, и эта ссылка указывает на себя, если только

*linkname* не является полным путем. Когда вы создаете символическую ссылку на каталог, проверьте его на наличие ошибочных символических ссылок и удалите их.

Символические ссылки могут также создать множество неприятностей, когда вы не знаете об их существовании. Например, вы легко можете отредактировать, как вам кажется, копию файла, хотя в действительности это символическая ссылка на оригинал.

ВНИМАНИЕ

Не забывайте о параметре -s при создании символической ссылки. Без него команда ln создает жесткую ссылку, присваивая дополнительное реальное имя тому же файлу. У нового имени файла тот же статус, что и у старого: оно непосредственно связывает с данными, а не указывает на другое имя файла, как это делает символическая ссылка. Жесткие ссылки могут запутать сильнее, чем символические. Пока вы не усвоите материал раздела 4.5, избегайте применять их.

Если символические ссылки снабжены таким количеством предупреждений, зачем их используют? Во-первых, они являются удобным способом упорядочения файлов и предоставления совместного доступа к ним, а во-вторых, позволяют справиться с некоторыми мелкими проблемами.

# 2.18. Архивирование и сжатие файлов

После того как вы узнали о файлах, правах доступа и возможных ошибках, вам необходимо освоить команды `gzip` и `tar`.

## 2.18.1. Команда gzip

Команда `gzip` (GNU Zip) — одна из стандартных команд сжатия файлов в системе Unix. Файл, имя которого оканчивается на `.gz`, является архивом в формате GNU Zip. Используйте команду `gunzip` *`file.gz`* для декомпрессии файла <*file*>`.gz` и удаления суффикса из названия; для сжатия применяйте команду `gzip file`.

## 2.18.2. Команда tar

В отличие от программ сжатия в других операционных системах, команда `gzip` не создает архивы файлов, то есть она не упаковывает несколько файлов и каталогов в один файл. Для создания архива используйте команду `tar`:

```
$ tar cvf archive.tar file1 file2 ...
```

Архивы, созданные с помощью команды `tar`, обычно снабжены суффиксом `.tar` (по договоренности; он не является необходимым). В приведенном выше примере команды параметры *`file1`*, *`file2`* и т. д. являются именами файлов и каталогов, которые вы желаете упаковать в архив <*archive*>`.tar`. Флаг `c` активизирует режим создания. У флагов `v` и `f` более специфичные роли.

Флаг `v` активизирует подробный диагностический вывод, при котором команда `tar` отображает имена файлов и каталогов в архиве по мере работы с ними. Если добавить еще один флаг `v`, команда `tar` отобразит такие подробности, как размер

файла и права доступа к нему. Если вам не нужны сообщения команды о том, что она сейчас делает, опустите флаг v.

Флаг f обозначает файл-параметр. Следующий после этого флага аргумент в командной строке должен быть именем файла-архива, который создаст команда tar (в приведенном выше примере это файл <archive>.tar). Вы всегда *должны* использовать этот параметр, указывая за ним имя файла (исключая вариант работы с накопителями на магнитной ленте). Для применения стандартных ввода и вывода введите дефис (-) вместо имени файла.

## Распаковка файлов tar

Чтобы распаковать файл .tar с помощью команды tar, используйте флаг x:

```
$ tar xvf archive.tar
```

Флаг x переводит команду tar *в режим извлечения (распаковки)*. Вы можете извлечь отдельные части архива, указав их имена в конце командной строки, но при этом вы должны знать их в точности. Чтобы быть уверенными, посмотрите краткое описание режима содержания.

ПРИМЕЧАНИЕ

При использовании режима извлечения помните о том, что команда tar не удаляет архивный файл .tar по окончании извлечения его содержимого.

## Режим содержания

Перед началом распаковки следует просмотреть файл .tar в *режиме содержания*, использовав в команде флаг t вместо флага x. Этот режим проверяет общую целостность архива и выводит имена всех находящихся в нем файлов. Если вы не просмотрите архив перед его распаковкой, то в результате можете получить в текущем каталоге хаос из файлов, который будет довольно трудно разобрать.

При проверке архива в режиме t убедитесь в том, что все расположено внутри разумной структуры каталогов, то есть все пути файлов в архиве должны начинаться с одного названия каталога. Если вы не уверены в этом, создайте временный каталог, перейдите в него, а затем выполните распаковку. Вы всегда сможете использовать команду mv * .., если архив не распаковался в виде хаоса.

При распаковке подумайте о возможности применить параметр p, чтобы сохранить права доступа. Используйте его в режиме распаковки, чтобы переопределить параметры команды umask и получить в точности те же права доступа, которые указаны в архиве. Параметр p применяется по умолчанию при работе в учетной записи superuser. Если у вас в качестве superuser-пользователя возникают сложности с правами доступа и владения при распаковке архива, дождитесь завершения работы команды и появления приглашения оболочки. Хотя вам может понадобиться извлечь лишь малую часть архива, команда tar должна обработать его полностью. Вам не следует прерывать этот процесс, поскольку команда выполняет назначение прав доступа только после проверки всего архива.

Выучите *все* параметры и режимы команды tar, которые упомянуты в этом разделе. Если для вас это сложно, выпишите их на карточки-подсказки — при работе с данной командой очень важно не допускать ошибок по небрежности.

## 2.18.3. Сжатые архивы (.tar.gz)

Начинающих пользователей часто смущает факт, что архивы обычно являются сжатыми, а их имена заканчиваются на .tar.gz. Чтобы распаковать сжатый архив, действуйте справа налево: сначала избавьтесь от суффикса .gz, а затем займитесь суффиксом .tar. Например, приведенные ниже команды производят декомпрессию и распаковку архива <*file*>.tar.gz:

```
$ gunzip file.tar.gz
$ tar xvf file.tar
```

Поначалу вы можете выполнять эту процедуру пошагово, запуская сначала команду gunzip для декомпрессии, а затем — команду tar для проверки и распаковки архива. Чтобы создать сжатый архив, выполните действия в обратном порядке: сначала запустите команду tar, а затем — gzip. Делайте это достаточно часто, и вскоре вы запомните, как работают процедуры архивации и сжатия.

## 2.18.4. Команда zcat

Описанный выше метод не является самым быстрым или наиболее эффективным при вызове команды tar для работы со сжатым архивом. Он расходует дисковое пространство и время ядра, затрачиваемое на ввод/вывод. Лучший способ заключается в комбинации функций архивирования и компрессии в виде конвейера. Например, данная команда-конвейер распаковывает файл <*file*>.tar.gz:

```
$ zcat file.tar.gz | tar xvf –
```

Команда zcat равносильна команде gunzip -dc. Параметр -d отвечает за декомпрессию, а параметр -c отправляет результат на стандартный вывод (в данном случае команде tar).

Поскольку использование команды zcat стало общепринятым, у версии команды tar, входящей в систему Linux, есть ярлык-сокращение. Можно применять в качестве параметра z, чтобы автоматически задействовать команду gzip. Это срабатывает как при извлечении архива (как в режимах x или t у команды tar), так и при создании (с параметром c). Используйте, например, следующую команду, чтобы проверить сжатый архив:

```
$ tar ztvf file.tar.gz
```

Вам следует хорошо освоить длинный вариант команды, прежде чем применять сокращение.

ПРИМЕЧАНИЕ

Файл .tgz — это то же самое, что и файл .tar.gz. Суффикс приспособлен для использования в файловых системах FAT (на основе MS-DOS).

## 2.18.5. Другие утилиты сжатия

Еще одной командой для компрессии в системе Unix является bzip2, которая создает файлы, оканчивающиеся на .bz2. Работая чуть медленнее по сравнению с gzip,

команда bzip2 зачастую сжимает текстовые файлы немного лучше и поэтому чрезвычайно популярна при распространении исходного программного кода. Для декомпрессии следует использовать команду bunzip2, а параметры для обоих компонентов довольно похожи на параметры команды gzip, так что вам не придется учить ничего нового. Для команды tar параметром компрессии/декомпрессии в режиме bzip2 является символ j.

Новая команда для сжатия под названием xz также приобретает популярность. Соответствующая ей команда декомпрессии называется unxz, а ее аргументы сходны с параметрами команды gzip.

В большинстве дистрибутивов системы Linux присутствуют команды zip и unzip, которые совместимы с ZIP-архивами Windows. Они работают как с обычными файлами .zip, так и с самораспаковывающимися архивами, файлы которых оканчиваются на .exe. Однако если вам встретится файл, который оканчивается на .Z, то вы обнаружили реликт, который был создан командой, бывшей когда-то стандартом для системы Unix. Команда gunzip способна распаковать такие файлы, однако создать их с помощью gzip невозможно.

## 2.19. Основные сведения об иерархии каталогов Linux

Теперь, когда вы знаете о том, как получать сведения о файлах, выполнять смену каталогов и читать страницы руководства, вы готовы начать исследование файлов вашей системы. Подробности о структуре каталогов изложены на веб-странице *Filesystem Hierarchy Standard, or FHS* (http://www.pathname.com/fhs/), но для начала будет достаточно следующего небольшого обзора.

На рис. 2.2 представлен упрощенный вариант иерархии каталогов, в котором показано несколько каталогов внутри /, /usr и /var. Обратите внимание на то, что внутри каталога /usr есть такие же названия, как и в каталоге /.

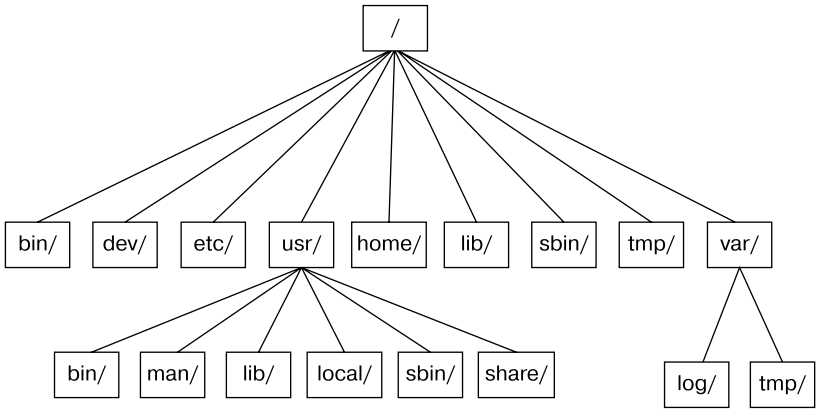


**Рис. 2.2.** Иерархия каталогов Linux

Наиболее важные подкаталоги в корневом каталоге таковы.

- `/bin`. Содержит готовые к запуску команды (известные также как исполняемые файлы), включая большинство основных команд системы Unix, таких как `ls` и `cp`. Большинство команд в каталоге `/bin` представлено в двоичном формате, поскольку они созданы компилятором языка C, однако в современных системах некоторые команды являются сценариями оболочки.
- `/dev`. Содержит файлы устройств (о них подробнее — в главе 3).
- `/etc`. Этот центральный каталог системной конфигурации содержит пароль пользователя, файлы загрузки, файлы устройств, сетевые настройки и другие параметры. Многие элементы каталога `/etc` зависят от аппаратного обеспечения. Например, каталог `/etc/X11` содержит конфигурацию видеокарты и «оконного» интерфейса.
- `/home`. Содержит личные каталоги обычных пользователей. В большинстве версий системы Unix соблюдается этот стандарт.
- `/lib`. Название является сокращением слова *library* («библиотека»). Этот каталог содержит файлы библиотек, хранящие программный код, который может быть применен исполняемыми файлами. Существуют два типа библиотек: статические и используемые совместно. Каталог `/lib` должен содержать только библиотеки для совместного пользования. Другие каталоги библиотек, например `/usr/lib`, содержат оба типа, а также другие вспомогательные файлы (более подробно мы рассмотрим совместно используемые библиотеки в главе 15).
- `/proc`. Этот каталог представляет системную статистику в виде интерфейса «каталог-файл». Основная часть структуры подкаталога `/proc` является уникальной для Linux, однако во многих других вариантах системы Unix присутствуют подобные функции. Каталог `/proc` содержит информацию о запущенных в данный момент процессах, а также о некоторых параметрах ядра.
- `/sys`. Данный каталог подобен каталогу `/proc` тем, что он предоставляет интерфейс устройствам и системе (подробнее о каталоге `/sys` вы узнаете из главы 3).
- `/sbin`. Здесь расположены системные исполняемые файлы. Команды из каталога `/sbin` относятся к управлению системой, поэтому у обычных пользователей, как правило, в командном пути не указаны компоненты каталога `/sbin`. Многие утилиты из этого каталога не будут работать, если вы запустите их не с правами корневого пользователя.
- `/tmp`. Хранилище для небольших временных файлов, о которых вам не стоит особо беспокоиться. Любой пользователь может выполнять чтение и запись в этом каталоге, однако доступ к файлам другого пользователя может быть запрещен. Многие команды применяют данный каталог в качестве рабочего пространства. Не помещайте что-либо важное в каталог `/tmp`, поскольку в большинстве систем происходит его очистка при запуске системы, а некоторые системы даже периодически удаляют из него старые файлы. Не позволяйте также наполнять каталог `/tmp` всяким хламом, поскольку выделенное на него место обычно совместно используется еще каким-либо важным компонентом (например, остальными частями корневого каталога).

- /usr. Хотя название этого каталога произносится как *user* («пользователь»), он не хранит никаких пользовательских файлов. Вместо этого содержит обширную иерархию каталогов, включающую основную часть системы Linux. Многие имена каталогов здесь такие же, как и в корневом каталоге (типа /usr/bin и /usr/lib), и содержат те же типы файлов. Причина, по которой корневой каталог не содержит систему полностью, главным образом историческая — в прошлом это было необходимо, чтобы корневой каталог не занимал много дискового пространства.
- /var. Изменяемый подкаталог, в котором команды хранят информацию во время исполнения. Системный журнал, отслеживание активности пользователей, кэш, а также другие файлы, которые создаются системными командами, расположены здесь. Вы увидите здесь каталог /var/tmp, но он не очищается при запуске системы.

## 2.19.1. Другие корневые подкаталоги

В корневом каталоге есть несколько интересных подкаталогов.

- /boot. Содержит файлы загрузчика ядра. Эти файлы имеют отношение только к самой первой стадии запуска системы Linux. В этом каталоге вы не найдете сведений о том, как система Linux запускает свои службы (загляните в главу 5 за подробностями).
- /media. Основная точка подключения для таких съемных устройств, как флеш-накопители. Присутствует во многих версиях системы.
- /opt. Здесь может находиться дополнительное ПО сторонних разработчиков. Многие версии системы не используют каталог /opt.

## 2.19.2. Каталог /usr

На первый взгляд каталог /usr может показаться довольно простым, однако быстрый просмотр подкаталогов /usr/bin и /usr/lib выявляет множество подробностей.

Каталог /usr предназначен для хранения большей части команд и данных пространства пользователя. Помимо подкаталогов /usr/bin, /usr/sbin и /usr/lib, каталог /usr содержит следующее.

- /include. Хранит заголовочные файлы, используемые компилятором C.
- /info. Содержит руководства GNU в формате info (см. раздел 2.13).
- /local. Здесь администраторы могут устанавливать собственное программное обеспечение. Структура этого каталога должна выглядеть подобно структуре каталогов / и /usr.
- /man. Содержит страницы руководства.
- /share. Содержит файлы, которые должны работать в других типах систем Unix без потери функциональности. В прошлом объединенные в сеть машины могли совместно использовать данный каталог, однако полноценный каталог /share становится редким, поскольку современные диски не содержат ошибок при распределении пространства. Поддержание каталога /share зачастую только

добавляет проблем. В любом случае каталоги /man, /info и некоторые другие подкаталоги часто присутствуют здесь.

### 2.19.3. Местоположение ядра

В системе Linux ядро обычно размещается в каталогах /vmlinuz или /boot/vmlinuz. *Загрузчик системы* загружает этот файл в память и приводит его в действие при запуске системы (подробности о загрузчике вы найдете в главе 5).

После того как загрузчик запустит ядро и приведет его в действие, основной файл ядра больше не используется работающей системой. Тем не менее вы обнаружите множество модулей, которые ядро может по запросу загружать и выгружать во время нормального функционирования системы. Такие *загружаемые модули ядра* расположены в каталоге /lib/modules.

## 2.20. Запуск команд с правами пользователя superuser

Прежде чем продвигаться дальше, вам следует узнать о том, как запускать команды в качестве пользователя superuser. Вероятно, вы уже знаете о том, что можно запустить команду su и указать пароль для входа в корневую оболочку. Такой вариант сработает, но у него есть некоторые неудобства:

- вы не отслеживаете команды, которые изменяют систему;
- вы не отслеживаете пользователей, которые выполнили команды, изменяющие систему;
- у вас нет доступа к обычной среде оболочки;
- вы должны вводить пароль.

### 2.20.1. Команда sudo

В большинстве дистрибутивов применяется пакет sudo, который позволяет администраторам запускать команды в корневом режиме, зайдя в систему со своей учетной записью. Например, из главы 7 вы узнаете об использовании команды vipw для редактирования файла /etc/passwd. Это можно выполнить так:

```
$ sudo vipw
```

При таком запуске команда sudo отмечает данное действие с помощью сервиса syslog в устройстве local2. Подробности о системных журналах изложены в главе 7.

### 2.20.2. Файл /etc/sudoers

Система не позволит *любому* пользователю запускать команды в качестве пользователя superuser. Вы должны настроить права таких привилегированных пользователей в файле /etc/sudoers. Пакет sudo обладает множеством параметров (которые, вероятно, вы никогда не станете применять), вследствие чего синтаксис файла /

etc/sudoers довольно сложен. Так, приведенный ниже фрагмент позволяет пользователям *user1* и *user2* запускать любую команду в качестве корневого пользователя, не вводя при этом пароль:

```
User_Alias ADMINS = user1, user2

ADMINS  ALL = NOPASSWD: ALL

root    ALL=(ALL) ALL
```

В первой строке для двух пользователей определен псевдоним ADMINS, а во второй строке им предоставляются права доступа. Фрагмент ALL = NOPASSWD: ALL означает, что пользователи с псевдонимом ADMINS могут использовать пакет sudo для выполнения команд в корневом режиме. Второе слово ALL значит «любая команда». Первое слово ALL обозначает «любой хост».

ПРИМЕЧАНИЕ

Если у вас несколько компьютеров, можно указать различные типы доступа для каждого из них или для группы машин, но мы не будем рассматривать такой вариант.

Фрагмент root ALL=(ALL) ALL означает, что пользователь superuser может также использовать пакет sudo для запуска любой команды на любом хосте. Дополнительный параметр (ALL) означает, что пользователь superuser может также запускать команды как и любой другой пользователь. Можно распространить это право на пользователей ADMINS, добавив параметр (ALL) в строку файла /etc/sudoers, как отмечено ниже символом ❶:

```
ADMINS ALL = (ALL)❶ NOPASSWD: ALL
```

ПРИМЕЧАНИЕ

Воспользуйтесь командой visudo для редактирования файла /etc/sudoers. Эта команда проверяет отсутствие синтаксических ошибок после сохранения файла.

Если вам необходимо использовать более продвинутые функции команды sudo, обратитесь к страницам sudoers(5) и sudo(8). Подробности механизма переключения между пользователями рассмотрены в главе 7.

## 2.21. Заглядывая вперед

Вы должны знать о том, как с помощью командной строки запускать команды, перенаправлять вывод, работать с файлами и каталогами, просматривать список процессов и обращаться к страницам руководства. Вы должны в общих чертах ориентироваться в пространстве пользователя системы Linux, а также уметь запускать команды в качестве пользователя superuser. Возможно, вы еще не так много узнали о внутреннем устройстве компонентов пространства пользователя или о том, что происходит в ядре, однако, получив основные представления о файлах и процессах, вы уже в пути.

В следующих главах вы будете работать как с системными компонентами ядра, так и с компонентами пространства пользователя, используя только что изученные инструменты командной строки.

# 3 Устройства

В этой главе представлены основные сведения об инфраструктуре устройств, которая обеспечивается ядром в работающей системе Linux. На протяжении истории этой системы многое изменилось в том, как ядро представляет устройства пользователю. Мы начнем с рассмотрения традиционной системы файлов устройств, чтобы понять, каким образом ядро обеспечивает конфигурацию устройства с помощью пути `sysfs`. Наша цель — научиться извлекать информацию об устройствах в системе, чтобы понимать некоторые элементарные операции. В следующих главах будет подробно рассмотрено взаимодействие с особыми типами устройств.

Важно понимать, как ядро взаимодействует с пространством пользователя, когда ему предъявляются новые устройства. Менеджер устройств `udev` позволяет программам из пространства пользователя автоматически конфигурировать и использовать новые устройства. Вы увидите основные моменты того, как ядро отправляет сообщение процессу в пространстве пользователя с помощью менеджера `udev` и какой процесс при этом задействован.

## 3.1. Файлы устройств

В системе Unix большинством устройств легко управлять, поскольку ядро представляет процессам в виде файлов многие интерфейсы ввода-вывода устройств. Такие файлы устройств иногда называют *узлами устройств*. Не только программист может применять обычные файловые операции для работы с каким-либо устройством: некоторые устройства доступны также стандартным командам вроде `cat`. Вам не обязательно быть программистом, чтобы использовать устройство, однако есть ограничения на то, что вы можете сделать с помощью файлового интерфейса, так что не все устройства или их возможности доступны в стандартном файловом вводе-выводе.

Система Linux применяет ту же схему файлов устройств, какая используется в других вариантах системы Unix. Файлы устройств расположены в каталоге `/dev`, и при запуске команды `ls /dev` обнаружится достаточное количество файлов в этом каталоге. При работе с устройствами для начала попробуйте такую команду:

```
$ echo blah blah > /dev/null
```

Как и положено любой команде с перенаправлением вывода, данная команда отправляет нечто из стандартного вывода в файл. Однако файл `/dev/null` является устройством, и ядро решает, что делать с данными, записываемыми в это устройство. В случае с `/dev/null` ядро просто игнорирует ввод и не использует данные.

Чтобы идентифицировать устройство и просмотреть его права доступа, применяйте команду `ls -l`:

**Пример 3.1.** Файлы устройств

```
$ ls -l
brw-rw----     1 root disk 8, 1 Sep  6 08:37 sda1
crw-rw-rw-     1 root root 1, 3 Sep  6 08:37 null
prw-r--r--     1 root root    0 Mar  3 19:17 fdata
srw-rw-rw-     1 root root    0 Dec 18 07:43 log
```

Обратите внимание на первый символ в каждой строке (первый символ режима файла) в примере 3.1. Если это символ `b`, `c`, `p` или `s`, такой файл является устройством. Эти буквы обозначают соответственно *блочное устройство*, *символьное устройство*, *канал* и *сокет*, что подробно объяснено ниже.

- **Блочное устройство.** Программы получают доступ к данным на *блочном устройстве* в виде фиксированных порций. В приведенном примере `sda1` является дисковым устройством — одним из типов блочных устройств. Диски можно легко разделить на блоки данных. Поскольку общий объем блочного устройства фиксирован и легко поддается индексации, процессы с помощью ядра получают случайный доступ к любому блоку устройства.
- **Символьное устройство.** Символьные устройства работают с потоками данных. Вы можете лишь считывать символы с таких устройств или записывать символы на них, как было показано в примере с `/dev/null`. Символьные устройства не обладают размером. Когда выполняется чтение или запись, ядро обычно осуществляет операцию чтения или записи на устройство. Принтеры, напрямую подключенные к компьютеру, представлены символьными устройствами. Важно отметить следующее: при взаимодействии с символьным устройством ядро не может выполнить откат данных и повторную их проверку после того, как данные переданы устройству или процессу.
- **Канал.** *Именованные каналы* подобны символьным устройствам, но у них на другом конце потока ввода-вывода располагается другой процесс, а не драйвер ядра.
- **Сокет.** *Сокеты* являются специализированными интерфейсами, которые часто используются для взаимодействия между процессами. Часто они располагаются вне каталога `/dev`. Файлы сокетов представляют сокеты домена Unix (о них вы узнаете из главы 10).

Числа перед датами в первых двух строках примера 3.1 являются *старшим* и *младшим* номерами устройств, помогая ядру при идентификации устройств. Сходные устройства, как правило, снабжены одинаковым старшим номером, например `sda3` и `sdb1` (оба являются разделами жесткого диска).

ПРИМЕЧАНИЕ

Не все устройства обладают файлами устройств, поскольку интерфейсы ввода-вывода для блочных и символьных устройств подходят не для всех случаев. Например, у сетевых интерфейсов нет файлов устройств. Теоретически возможно взаимодействие с сетевым интерфейсом с помощью одного символьного устройства, но, поскольку это оказалось бы исключительно сложным, ядро использует другие интерфейсы ввода-вывода.

# 3.2. Путь устройств sysfs

Традиционный каталог `/dev` в системе Unix является удобным способом, с помощью которого пользовательские процессы могут обращаться к устройствам, поддерживаемым ядром, а также предоставлять интерфейс для них. Однако такая схема также и слишком упрощена. Название устройства в каталоге `/dev` даст вам некоторую информацию об устройстве, но далеко не всю. Кроме того, ядро назначает устройства в порядке их обнаружения, поэтому они могут получать различные имена после перезагрузки.

Чтобы обеспечить унифицированный обзор присоединенных устройств, взяв за основу их действительные аппаратные характеристики, ядро системы Linux предлагает интерфейс `sysfs` для обозначения файлов и каталогов. Основным путем для устройств является `/sys/devices`. Например, жесткий диск SATA в файле `/dev/sda` мог бы получить следующий путь в интерфейсе `sysfs`:

```
/sys/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda
```

Как видите, такой путь достаточно длинный по сравнению с именем файла `/dev/sda`, который является также каталогом. Однако в действительности нельзя сравнивать эти пути, поскольку у них разное назначение. Файл `/dev` расположен здесь для того, чтобы пользовательский процесс мог применять устройство, в то время как путь `/sys/devices` использован для просмотра информации об устройстве и для управления им. Если вы составите список содержимого пути устройств, подобного приведенному выше, вы увидите что-либо подобное.

| alignment_offset | discard_alignment | holders | removable | size | uevent |
|---|---|---|---|---|---|
| bdi | events | inflight | ro | slaves | |
| capability | events_async | power | sda1 | stat | |
| dev | events_poll_msecs | queue | sda2 | subsystem | |
| device | ext_range | range | sda5 | trace | |

Названия файлов и подкаталогов предназначены в первую очередь для чтения программами, а не людьми, однако вы сможете понять, что они содержат и представляют, посмотрев какой-либо пример, скажем файл `/dev`. После запуска в этом каталоге команды `cat dev` отобразятся числа `8:0`, которые являются старшим и младшим номерами устройств в файле `/dev/sda`.

В каталоге `/sys` есть несколько ярлыков. Например, ярлык `/sys/block` должен содержать все доступные в системе блочные устройства. Однако это всего лишь символические ссылки. Запустите команду `ls -l /sys/block`, чтобы выяснить их настоящие `sysfs`-пути.

Могут возникнуть затруднения при отыскании sysfs-пути устройства в каталоге /dev. Используйте команду udevadm, чтобы показать путь и другие атрибуты:

```
$ udevadm info --query=all --name=/dev/sda
```

ПРИМЕЧАНИЕ

Команда udevadm расположена в каталоге /sbin; можно указать этот каталог в самом конце командного пути, если его там еще нет.

Подробности о команде udevadm и описание системы udev вы найдете в разделе 3.5.

# 3.3. Команда dd и устройства

Команда dd чрезвычайно полезна при работе с блочными и символьными устройствами. Единственная функция этой команды заключается в чтении из входного файла или потока и запись в выходной файл или поток, при этом попутно может происходить некоторое преобразование кодировки.

Команда dd копирует данные в блоках фиксированного размера. Пример использования команды dd с некоторыми распространенными параметрами для символического устройства:

```
$ dd if=/dev/zero of=new_file bs=1024 count=1
```

Как видите, формат параметров команды dd отличается от формата большинства других команд системы Unix. Он основан на старом стиле JCL (Job Control Language, язык управления заданиями), применявшемся в компании IBM. Вместо использования дефиса (-) перед параметром вы указываете название параметра, а затем после символа равенства (=) задаете его значение. В приведенном выше примере происходит копирование одного блока размером 1024 байта из потока /dev/zero (непрерывный поток нулевых байтов) в файл new_file.

Приведем важные параметры команды dd.

- if=*file*. Входной файл. По умолчанию применяется стандартный ввод.
- of=*file*. Выходной файл. По умолчанию применяется стандартный вывод.
- bs=*size*. Размер блока. Команда dd будет считывать и записывать указанное количество байтов за один прием. Чтобы сокращенно указать большие объемы данных, можно использовать символы b и k, которые соответствуют значениям 512 и 1024 байт. Так, в приведенном выше примере можно записать bs=1k вместо bs=1024.
- ibs=*size*, obs=*size*. Размеры блоков на вводе и выводе. Если вы можете использовать одинаковые размеры блоков для ввода и вывода, примените параметр bs. Если нет, используйте параметры ibs и obs для ввода и вывода соответственно.
- count=*num*. Общее количество блоков, подлежащих копированию. При работе с большим файлом (или с устройством, которое поддерживает бесконечный поток данных, такой как /dev/zero) следует остановить команду dd в определенной точке, чтобы не израсходовать впустую большое количество дискового

пространства, времени центрального процессора или того и другого сразу. Используйте команду `count` с параметром `skip`, чтобы скопировать небольшой фрагмент из большого файла или устройства.

- `skip=`*num*. Пропускает указанное количество блоков *num* во входном файле или потоке и не копирует их на вывод.

ВНИМАНИЕ

Команда dd является очень мощной, при ее запуске вы должны понимать, что делаете. Очень легко повредить файлы или данные устройств, совершив по небрежности ошибку. Часто может помочь запись вывода в новый файл, если вы не вполне уверены в результатах.

# 3.4. Сводка имен устройств

Иногда бывает сложно отыскать название устройства (например, при создании разделов диска). Вот несколько способов выяснить это.

- Выполните запрос `udevd` с помощью команды `udevadm` (см. раздел 3.5).
- Поищите устройство в каталоге `/sys`.
- Попробуйте угадать название на основе результатов вывода команды `dmesg` (которая выдает несколько последних сообщений ядра) или из файла системного журнала ядра (см. раздел 7.2). Эти результаты могут содержать описание устройств в вашей системе.
- Для дискового устройства, которое уже видно в системе, можно посмотреть результаты работы команды `mount`.
- Запустите команду `cat /proc/devices`, чтобы увидеть блочные и символьные устройства, для которых ваша система уже имеет драйверы. Каждая строка состоит из номера и имени. Номер является старшим номером устройства, как рассказано в разделе 3.1. Если вы сможете определить устройство по его имени, поищите в каталоге `/dev` символьное или блочное устройство с соответствующим старшим номером. Таким образом вы найдете файлы устройства.

Среди перечисленных методов только первый является надежным, но для него необходим менеджер устройств udev. Если вы окажетесь в ситуации, когда этот менеджер недоступен, попробуйте остальные методы, памятуя о том, что в ядре может не оказаться файла устройства для вашего аппаратного средства.

В следующих разделах перечислены наиболее распространенные устройства Linux и соглашения об их именовании.

## 3.4.1. Жесткие диски: /dev/sd*

Большинству жестких дисков в современных системах Linux соответствуют имена устройств с префиксом `sd`, например `/dev/sda`, `/dev/sdb` и т. д. Такие устройства представляют диски полностью; для разделов диска ядро создает отдельные файлы устройств, например `/dev/sda1` и `/dev/sda2`.

Соглашение об именовании потребует небольшого объяснения. Префикс `sd` в имени устройства соответствует *диску SCSI*. Интерфейс SCSI (Small Computer

System Interface, интерфейс малых вычислительных систем) изначально был разработан в качестве стандарта для аппаратных средств и протокола коммуникации между устройствами (например, дисками) и другой периферией. Хотя в большинстве современных компьютеров не используются традиционные аппаратные средства SCSI, сам этот протокол присутствует повсюду благодаря своей приспособляемости. Например, USB-накопители применяют его при подключении. В случае с дисками SATA дело немного усложняется, однако ядро системы Linux в определенный момент по-прежнему использует команды SCSI для обращения к таким дискам. Одним из наиболее элегантных инструментов является команда `lsscsi`. Вот пример того, что можно получить в результате ее работы:

```
$ lsscsi
[0:0:0:0]❶     disk❷    ATA      WDC WD3200AAJS-2 01.0  /dev/sda❸
[1:0:0:0]      cd/dvd   Slimtype DVD A  DS8A5SH   XA15  /dev/sr0
[2:0:0:0]      disk     FLASH    Drive UT_USB20   0.00  /dev/sdb
```

В столбце ❶ приводится адрес устройства в системе. В столбце ❷ представлено описание типа устройства, а в последнем столбце ❸ указано, где можно найти файл устройства. Остальная информация содержит сведения о производителе.

Система Linux назначает устройствам файлы устройств в том порядке, в каком их драйверы находят устройства. Так, в приведенном выше примере ядро сначала нашло жесткий диск, затем оптический привод и наконец флеш-накопитель.

К сожалению, такая схема назначения устройств обычно вызывала проблемы при перенастройке аппаратного обеспечения. Допустим, в вашей системе три диска: `/dev/sda`, `/dev/sdb` и `/dev/sdc`. Если диск `/dev/sdb` вышел из строя и вы должны заменить его, чтобы компьютер снова смог работать, то диск, который до этого был диском `/dev/sdc`, превратится в диск `/dev/sdb`, а диска `/dev/sdc` больше не будет. Если вы ссылались непосредственно на названия устройств в файле `fstab` (см. подраздел 4.2.8), то вам придется внести некоторые изменения в данный файл, чтобы вернуть (основную часть) устройства к нормальной работе. Чтобы устранить подобные проблемы, в большинстве современных систем Linux используется идентификатор UUID (Universally Unique Identifier, универсальный уникальный идентификатор, см. подраздел 4.2.4) для устойчивого доступа к дисковым устройствам.

Здесь лишь вкратце рассказано о том, как использовать диски и другие устройства для хранения данных в системе Linux. Дополнительную информацию о применении дисков можно найти в главе 4. Далее в текущей главе мы изучим то, каким образом интерфейс SCSI участвует в работе ядра системы Linux.

## 3.4.2. Приводы CD и DVD: /dev/sr*

Система Linux распознает большинство оптических приводов как устройства SCSI с именами `/dev/sr0`, `/dev/sr1` и т. д. Однако если такое устройство использует устаревший интерфейс, оно может отобразиться как устройство PATA (об этом пойдет речь ниже). Устройства `/dev/sr*` доступны только для чтения и применяются только для считывания оптических дисков. Для использования возможностей записи и перезаписи в оптических приводах следует применять «обобщенные» SCSI-устройства, такие как `/dev/sg0`.

## 3.4.3. Жесткие диски PATA: /dev/hd*

Блочные устройства системы Linux `/dev/hda`, `/dev/hdb`, `/dev/hdc` и `/dev/hdd` часто встречаются в старых версиях ядра системы и для устаревших аппаратных средств. Это фиксированные назначения, которые основаны на ведущем и ведомом устройствах с интерфейсами 0 и 1. Иногда вы можете обнаружить, что диск SATA распознан как один из таких дисков. Это означает, что диск SATA работает в режиме совместимости, который снижает производительность. Проверьте настройки системы BIOS, чтобы выяснить, можно ли переключить контроллер диска SATA в его штатный режим.

## 3.4.4. Терминалы: /dev/tty*, /dev/pts/* и /dev/tty

*Терминалы* — это устройства для перемещения символов между пользовательским процессом и устройством ввода-вывода, как правило, для вывода текста на экран терминала. Интерфейс терминальных устройств прошел довольно долгий путь с тех пор, когда терминалы были основаны на печатающих аппаратах.

*Псевдотерминальные устройства* — это эмулированные терминалы, которые понимают функции ввода-вывода реальных терминалов. Однако вместо того, чтобы обращаться к реальному аппаратному средству, ядро представляет интерфейс ввода-вывода посредством какой-либо программы, например окна терминала оболочки, в котором вы, вероятно, печатаете большинство команд.

Двумя общими терминальными устройствами являются `/dev/tty1` (первая виртуальная консоль) и `/dev/pts/0` (первое псевдотерминальное устройство). Каталог `/dev/pts` сам по себе является специализированной файловой системой.

Устройство `/dev/tty` — это управляющий терминал текущего процесса. Если какая-либо команда в данный момент производит чтение или запись в терминале, данное устройство выступает в качестве синонима такого терминала. Процесс не обязательно должен быть присоединен к терминалу.

### Режимы отображения и виртуальные консоли

В системе Linux есть два основных режима отображения: текстовый режим и оконная система *«Сервер X Window System»* (графический режим, как правило, с использованием менеджера отображения). Хотя система Linux традиционно загружается в текстовом режиме, сейчас в большинстве дистрибутивов используются параметры ядра и промежуточные механизмы графического отображения (экраны загрузки, такие как plymouth), чтобы полностью скрыть текстовый режим при загрузке системы. В подобных случаях система переключается в полноценный графический режим незадолго до окончания процесса загрузки.

Система Linux поддерживает *виртуальные консоли*, чтобы поддерживать несколько дисплеев. Каждая виртуальная консоль может работать в графическом или в текстовом режиме. В текстовом режиме можно переключаться между консолями с помощью сочетания клавиши Alt с какой-либо функциональной клавишей. Например, сочетание Alt+F1 переведет вас в консоль `/dev/tty1`, сочетание Alt+F2 — в консоль `/dev/tty2` и т. д. Многие из таких сочетаний могут быть

заняты процессом getty, который обслуживает вход в систему (подробнее — в разделе 7.4).

Виртуальная консоль, которая применяется X-сервером в графическом режиме, немного отличается. Вместо того чтобы получить назначение виртуальной консоли из начальной конфигурации, X-сервер занимает свободную виртуальную консоль, если он не был направлен на использование конкретной консоли. Например, если процесс getty запущен в консолях tty1 и tty2, новый X-сервер займет консоль tty3. Кроме того, после перевода виртуальной консоли в графический режим следует, как правило, нажимать сочетание клавиш Ctrl+Alt с функциональной клавишей для переключения к другой виртуальной консоли, а не просто сочетание клавиши Alt с какой-либо функциональной клавишей.

В довершение ко всему, если вы желаете увидеть текстовую консоль после загрузки системы, нажмите сочетание клавиш Ctrl+Alt+F1. Чтобы вернуться в сессию X11, нажимайте сочетания Alt+F2, Alt+F3 и т. д., пока не окажетесь в X-сессии.

Если у вас возникли сложности с переключением консолей в результате неверной работы механизма ввода или по каким-либо другим причинам, можно попытаться принудительно сменить консоли с помощью команды chvt. Например, чтобы переключиться в консоль tty1, запустите эту команду с правами доступа root:

```
# chvt 1
```

## 3.4.5. Последовательные порты: /dev/ttyS*

Старые последовательные порты RS-232 и подобные им являются специальными терминальными устройствами. Поскольку приходится заботиться о слишком большом количестве параметров, таких как скорость передачи и управление потоком, выполнить с последовательным портом с помощью командной строки можно немногое.

Порт, известный в Windows как COM1, называется /dev/ttyS0; COM2 — это /dev/ttyS1 и т. д. В именах подключаемых последовательных USB-адаптеров будет присутствовать сочетание USB или ACM: /dev/ttyUSB0, /dev/ttyACM0, /dev/ttyUSB1, /dev/ttyACM1 и т. п.

## 3.4.6. Параллельные порты: /dev/lp0 и /dev/lp1

Представляя тип интерфейса, который теперь широко заменен интерфейсом USB, устройства /dev/lp0 и /dev/lp1 с однонаправленным параллельным портом соответствуют портам LPT1 и LPT2 в Windows. Можно отправлять файлы (например, для распечатки) напрямую на параллельный порт с помощью команды cat, однако может потребоваться выполнить перед этим дополнительную подачу страницы или перезагрузку принтера. Сервер печати, например CUPS, гораздо лучше осуществляет взаимодействие с принтером.

Двунаправленными параллельными портами являются /dev/parport0 и /dev/parport1.

## 3.4.7. Аудиоустройства: /dev/snd/*, /dev/dsp, /dev/audio и другие

В системе Linux присутствуют два набора аудиоустройств. Это устройства для системного интерфейса *ALSA* (Advanced Linux Sound Architecture, продвинутая звуковая архитектура Linux), а также старый драйвер *OSS* (Open Sound System, открытая звуковая система). Устройства ALSA находятся в каталоге `/dev/snd`, однако работать с ними напрямую трудно. Системы Linux, которые используют интерфейс ALSA, поддерживают обратно совместимые со стандартом OSS устройства, если ядром загружена поддержка OSS.

Некоторые элементарные операции возможны с OSS-устройствами `dsp` и `audio`. Например, компьютер воспроизведет любой файл в формате WAV, если вы отправите его на устройство `/dev/dsp`. Тем не менее аппаратное средство может выполнить не совсем то, что вы ожидаете, вследствие несоответствия частоты. Кроме того, в большинстве систем устройство часто становится занято, как только вы войдете в систему.

ПРИМЕЧАНИЕ

Звук в системе Linux — довольно запутанная вещь вследствие большого количества вовлеченных слоев. Мы только что говорили об устройствах на уровне ядра, но обычно в пространстве пользователя присутствуют такие серверы, как pulse-audio, которые управляют звуком из различных источников и выступают в качестве посредника между звуковыми устройствами и другими процессами пространства пользователя.

## 3.4.8. Создание файлов устройств

В современных системах Linux вы не создаете файлы устройств сами; это выполняется с помощью файловой системы devtmpfs и менеджера устройств udev (см. раздел 3.5). Однако полезно увидеть, как это было сделано, поскольку в редких случаях вам может понадобиться создать именованный канал.

Команда `mknod` создает одно устройство. Вы должны знать имя устройства, а также его старший и младший номера. Например, чтобы создать устройство `/dev/sda1`, используйте следующую команду:

```
# mknod /dev/sda1 b 8 2
```

Параметры `b 8 2` определяют блочное устройство со старшим номером 8 и младшим номером 2. Для символьных устройств или именованных каналов используйте параметры `c` или `p` вместо `b` (для именованных каналов номера устройства опускаются).

Как отмечалось ранее, команда `mknod` полезна только для преднамеренного создания именованного канала. Одно время она была также полезна при создании отсутствующих устройств, когда производилось восстановление системы в режиме одного пользователя.

В старых версиях систем Unix и Linux обслуживание каталога `/dev` было весьма трудоемким. После каждого существенного обновления ядра или добавления драйвера появлялась возможность поддержки дополнительных типов

устройств, это означало, что именам файлов устройств можно назначать новый набор старшего и младшего номеров устройств. Поддерживать это было трудно, поэтому в каждой системе имелась команда MAKEDEV в каталоге /dev для создания групп устройств. После обновления системы следовало попробовать найти обновленную команду MAKEDEV, а затем запустить ее, чтобы создать новые устройства.

Такая статичная система становилась неуклюжей, поэтому ее пришлось заменить. Первой попыткой исправления ситуации была файловая система devfs, реализация каталога /dev в пространстве ядра, содержащая все устройства, которые поддерживает текущее ядро. Однако в ней присутствовали определенные ограничения, которые привели к разработке менеджера устройств udev и файловой системы devtmpfs.

## 3.5. Менеджер устройств udev

Излишняя усложненность ядра приводит к нестабильности в системе. Пример тому — управление файлами устройств: можно создавать файлы устройств в пространстве пользователя, так почему бы не выполнять это в ядре? Ядро системы Linux отправляет уведомления процессам в пространстве пользователя (они называются udevd) после обнаружения нового устройства в системе (например, после подключения USB-накопителя). С другой стороны, процесс в пространстве пользователя проверяет характеристики нового устройства, создает файл устройства, а затем выполняет необходимую инициализацию устройства.

Это была теория. К сожалению, на практике такой подход проблематичен: файлы устройств необходимы уже на ранней стадии загрузки системы, поэтому уведомления udevd должны начинать работу рано. Чтобы создать файлы устройств, менеджеру udevd не следует зависеть от устройств, для создания которых предназначен. Он должен выполнять начальный запуск очень быстро, чтобы остальная часть системы не оказалась в подвешенном состоянии, ожидая запуска менеджера udevd.

### 3.5.1. Файловая система devtmpfs

Файловая система devtmpfs была разработана для решения проблемы с доступностью устройств во время загрузки (см. подробности в разделе 4.2). Эта файловая система подобна старой devfs, но является упрощенной. Ядро создает файлы устройств по мере надобности, а также уведомляет менеджер udevd о том, что доступно новое устройство. После получения такого сигнала менеджер udevd не создает файлы устройств, а выполняет инициализацию устройства и отправляет уведомление процессу. Кроме того, он создает несколько символических ссылок в каталоге /dev для дальнейшей идентификации устройств. Примеры вы можете отыскать в каталоге /dev/disk/by-id, в котором каждому присоединенному диску соответствует одна или несколько записей.

Рассмотрим, например, такой типичный диск:

```
lrwxrwxrwx 1 root root 9 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671 ->
../../sda

lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671-
part1 ->

  ../../sda1

lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671-
part2 ->

  ../../sda2

lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671-
part5 ->

  ../../sda5
```

Менеджер udevd дает имена ссылкам по типу интерфейса, а затем по информации об изготовителе и модели, серийному номеру и разделу (если применяется).

В следующем разделе подробно рассказано о том, как менеджер udevd выполняет свою работу: как узнает о том, какие символические ссылки создать, и как создает их. Эта информация необязательна, чтобы продолжить чтение книги. Если вы впервые встречаетесь с устройствами Linux, можете смело переходить к следующей главе, чтобы начать изучение того, как использовать диски.

## 3.5.2. Работа и настройка менеджера udevd

Демон `udevd` работает следующим образом.

1. Ядро отправляет демону `udevd` событие-уведомление, называемое `uevent`, через внутреннюю сетевую ссылку.
2. Демон `udevd` загружает все атрибуты, содержащиеся в уведомлении `uevent`.
3. Демон `udevd` проводит разбор правил, а затем предпринимает действия или устанавливает дополнительные атрибуты на основе правил.

Входящее уведомление `uevent`, которое демон `udevd` получает от ядра, может выглядеть так:

```
ACTION=change
DEVNAME=sde
DEVPATH=/devices/pci0000:00/0000:00:1a.0/usb1/1-1/1-1.2/1-1.2:1.0/host4/
  target4:0:0/4:0:0:3/block/sde
DEVTYPE=disk
DISK_MEDIA_CHANGE=1
MAJOR=8
MINOR=64
SEQNUM=2752
SUBSYSTEM=block
UDEV_LOG=3
```

Можно заметить, что здесь присутствует смена устройства. После получения уведомления uevent демон udevd знает путь устройства в файловой системе sysfs, а также некоторые другие атрибуты, связанные с его свойствами, и готов начать обработку правил.

Файлы правил расположены в каталогах /lib/udev/rules.d и /etc/udev/rules.d. Правила из каталога /lib являются правилами по умолчанию, а правила из каталога /etc переопределяют их. Подробное объяснение правил вы сможете найти на странице руководства udev(7).

Взглянем на символические ссылки из примера /dev/sda в поразделе 3.5.1. Эти ссылки были определены с помощью правил из файла /lib/udev/rules.d/60-persistent-storage.rules. В нем вы увидите следующие строки:

```
# ATA devices using the "scsi" subsystem

KERNEL=="sd*[!0-9]|sr*", ENV{ID_SERIAL}!="?*", SUBSYSTEMS=="scsi",
ATTRS{vendor}=="ATA", IMPORT{program}="ata_id --export $tempnode"

# ATA/ATAPI devices (SPC-3 or later) using the "scsi" subsystem

KERNEL=="sd*[!0-9]|sr*", ENV{ID_SERIAL}!="?*", SUBSYSTEMS=="scsi",

  ATTRS{type}=="5", ATTRS{scsi_level}=="[6-9]*", IMPORT{program}="ata_id --export
$tempnode"
```

Эти правила сопоставляют диски ATA, представленные с помощью SCSI-подсистемы ядра (см. раздел 3.6). Можно заметить здесь несколько правил, предназначенных для отслеживания различных способов, с помощью которых могут быть представлены устройства, но суть заключается в том, что демон udevd будет пытаться подобрать устройство, начиная с символов sd или sr без номера (с использованием выражения KERNEL=="sd*[!0-9]|sr*"), а также подсистему (SUBSYSTEMS=="scsi") и, наконец, некоторые другие атрибуты. Если все эти условные выражения окажутся истинными, то демон udevd переходит к следующему выражению:

```
IMPORT{program}="ata_id --export $tempnode"
```

Это уже не условие, а скорее указание выполнить импорт переменных из команды /lib/udev/ata_id. Если у вас есть подобный диск, попробуйте самостоятельно ввести в командной строке следующее:

```
$ sudo /lib/udev/ata_id --export /dev/sda
ID_ATA=1
ID_TYPE=disk
ID_BUS=ata
ID_MODEL=WDC_WD3200AAJS-22L7A0
ID_MODEL_ENC=WDC\x20WD3200AAJS22L7A0\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20
  \x20\x20\x20\x20\x20\x20\x20\x20\x20\x20
ID_REVISION=01.03E10
ID_SERIAL=WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671
--snip--
```

Теперь импорт настраивает окружение так, чтобы все имена переменных в данном выводе приняли указанные значения. Например, любое правило, которое появится следом, будет распознавать ENV{ID_TYPE} в качестве диска.

Особо следует отметить переменную ID_SERIAL. В каждом из правил на втором месте следует такое условие:

```
ENV{ID_SERIAL}!="?*"
```

Это означает, что оно будет истинным, только если для переменной ID_SERIAL не указано значение. Следовательно, если она *уже* определена, условие будет ложным и все правило также станет ложным, в результате чего демон udevd перейдет к следующему правилу.

В чем же суть? Целью этих двух правил (и многих других в том же файле) является поиск серийного номера дискового устройства. Если значение переменной ENV{ID_SERIAL} установлено, демон udevd может проверить следующее правило:

```
KERNEL=="sd*|sr*|cciss*", ENV{DEVTYPE}=="disk", ENV{ID_SERIAL}=="?*",
  SYMLINK+="disk/by-id/$env{ID_BUS}-$env{ID_SERIAL}"
```

Для этого правила необходимо задать значение переменной ENV{ID_SERIAL}. В нем есть также одна директива:

```
SYMLINK+="disk/by-id/$env{ID_BUS}-$env{ID_SERIAL}"
```

После встречи с этой директивой демон udevd добавляет символическую ссылку на обнаруженное устройство. Итак, теперь вы знаете, откуда берутся символические ссылки на устройства.

Вероятно, вам любопытно узнать, как отличить условное выражение от директивы. Условные выражения записываются с помощью двух знаков равенства (==) или восклицательного знака и знака равенства (!=). Для директив используют либо один знак равенства (=), либо символ «плюс» и знак равенства (+=), либо двоеточие со знаком равенства (:=).

## 3.5.3. Команда udevadm

Команда udevadm является инструментом администрирования менеджера udevd. С ее помощью можно перезагрузить правила для udevd, а также события-триггеры. Однако, вероятно, самыми мощными функциями команды udevadm являются возможность поиска и обнаружения системных устройств, а также способность отслеживать уведомления uevents, когда демон udevd получает их от ядра. Единственную сложность может составить синтаксис этой команды, который становится немного запутанным.

Начнем с рассмотрения системного устройства. Вернувшись к примеру из подраздела 3.5.2, чтобы взглянуть на все атрибуты менеджера udev, которые использованы и сгенерированы в сочетании с правилами для устройства (такого как /dev/sda), запустите следующую команду:

```
$ udevadm info --query=all --name=/dev/sda
```

Результат будет выглядеть так:

```
P: /devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda
N: sda
S: disk/by-id/ata-WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671
S: disk/by-id/scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671
S: disk/by-id/wwn-0x50014ee057faef84 S: disk/by-path/pci-0000:00:1f.2-scsi-
0:0:0:0
E: DEVLINKS=/dev/disk/by-id/ata-WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671 /dev/disk/by-
id/scsi
  -SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671 /dev/disk/by-id/wwn-
0x50014ee057faef84 /dev/disk/by
  -path/pci-0000:00:1f.2-scsi-0:0:0:0
E: DEVNAME=/dev/sda
E: DEVPATH=/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda
E: DEVTYPE=disk
E: ID_ATA=1
E: ID_ATA_DOWNLOAD_MICROCODE=1
E: ID_ATA_FEATURE_SET_AAM=1
--snip--
```

Префикс в каждой строке указывает на атрибут или другую характеристику устройства. В данном случае `P:` в самом верху содержит путь устройства в файловой системе sysfs; `N:` является узлом устройства (то есть именем, которое присвоено файлу `/dev`), `S:` указывает символическую ссылку на узел устройства, которую демон `udevd` поместил в каталог `/dev` в соответствии со своими правилами; `E:` содержит дополнительную информацию об устройстве, извлеченную из правил `udevd`. В приведенном примере было гораздо больше строк вывода, не показанных здесь; попробуйте применить команду самостоятельно, чтобы получить представление о ее работе.

## 3.5.4. Отслеживание устройств

Чтобы отслеживать уведомления `uevents` с помощью инструмента `udevadm`, используйте команду `monitor`:

```
$ udevadm monitor
```

Результат (когда вы, например, вставите флеш-накопитель) будет выглядеть как этот сокращенный пример:

```
KERNEL[658299.569485]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2
(usb)
KERNEL[658299.569667]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-
1.2:1.0 (usb)
KERNEL[658299.570614]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-
1.2:1.0/host15
  (scsi)
KERNEL[658299.570645]   add    /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-
1.2:1.0/
  host15/scsi_host/host15 (scsi_host)
```

```
UDEV  [658299.622579]  add    /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2
(usb)
UDEV  [658299.623014]  add    /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-
1.2:1.0 (usb)
UDEV  [658299.623673]  add    /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-
1.2:1.0/host15
  (scsi)
UDEV  [658299.623690]  add    /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-
1.2:1.0/
  host15/scsi_host/host15 (scsi_host)
--snip--
```

Каждое сообщение здесь присутствует дважды, поскольку по умолчанию выводятся как входящие сообщения от ядра (помеченные словом `KERNEL`), так и те сообщения, которые демон `udevd` отправляет другим программам по окончании обработки и фильтрации событий. Чтобы увидеть только события ядра, добавьте параметр `--kernel`, а чтобы увидеть только исходящие события, используйте параметр `--udev`. Чтобы увидеть входящее уведомление `uevent` полностью, включая те атрибуты, которые показаны в подразделе 3.5.2, применяйте параметр `--property`.

Можно также отфильтровать события для какой-либо подсистемы. Например, чтобы увидеть только сообщения ядра, относящиеся только к изменениям в подсистеме SCSI, используйте следующую команду:

```
$ udevadm monitor --kernel --subsystem-match=scsi
```

Подробно о команде `udevadm` можно прочитать в руководстве на странице udevadm(8).

Помимо менеджера udev, есть и другие. Например, в шине D-Bus системы межпроцессного взаимодействия присутствует демон `udisks-daemon`, который отслеживает исходящие события менеджера `udevd`, чтобы автоматически подключать диски, а затем уведомлять программное обеспечение рабочей станции о доступности нового диска.

## 3.6. Подробнее: интерфейс SCSI и ядро Linux

В этом разделе мы рассмотрим поддержку интерфейса SCSI в ядре Linux, чтобы исследовать часть архитектуры ядра системы. Если вы торопитесь использовать какой-либо диск, переходите сразу к главе 4, изложенные здесь сведения вам будут не нужны. Кроме того, представленный в данном разделе материал более сложен и абстрактен, поэтому, если вы желаете остаться в практическом русле, вам определенно следует пропустить оставшуюся часть главы.

Начнем с небольшой предварительной информации. Традиционная конфигурация аппаратных средств SCSI состоит из хост-адаптера, который соединен с цепью устройств с помощью шины SCSI, как показано на рис. 3.1. Хост-адаптер подключен к компьютеру. У этого адаптера и у всех устройств есть идентифика-

торы SCSI ID, и в зависимости от версии интерфейса SCSI шина может поддерживать 8 или 16 идентификаторов. Наверное, вам приходилось слышать термин *исполнитель SCSI*, который используется по отношению к устройству и его идентификатору SCSI ID.

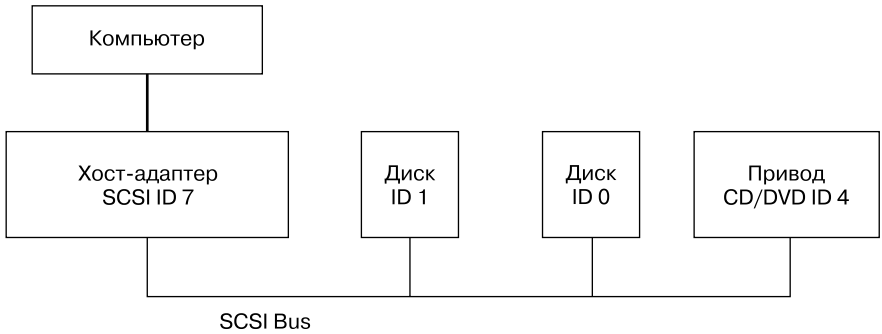


**Рис. 3.1.** Шина SCSI с хост-адаптером и устройствами

Хост-адаптер взаимодействует с устройствами с помощью набора команд SCSI в одноранговой связи; устройства высылают отклик хост-адаптеру. Компьютер не подключен к цепи устройств напрямую, он должен «пройти» через хост-адаптер, чтобы взаимодействовать с дисками и другими устройствами. Как правило, компьютер для связи с устройствами отправляет SCSI-команды хост-адаптеру, а устройства возвращают отклик также через него.

Новые версии интерфейса SCSI, например *SAS* (Serial Attached SCSI, интерфейс SCSI с последовательным подключением), обеспечивают исключительную производительность, однако вы, вероятно, не отыщете настоящие SCSI-устройства в большинстве компьютеров. Гораздо чаще вы встретите USB-устройства хранения, которые используют команды SCSI. В дополнение к ним устройства, поддерживающие интерфейс ATAPI (например, приводы CD/DVD-ROM), применяют вариант набора команд SCSI.

Диски SATA также присутствуют в системе в качестве устройств SCSI, что достигается с помощью слоя трансляции в библиотеке libata (см. подраздел 3.6.2). Некоторые контроллеры SATA (в особенности высокопроизводительные RAID-контроллеры) осуществляют такую трансляцию аппаратно.

Как же все это уживается вместе? Рассмотрите устройства, показанные в следующей системе:

```
$ lsscsi
[0:0:0:0]   disk    ATA      WDC WD3200AAJS-2 01.0  /dev/sda
[1:0:0:0]   cd/dvd  Slimtype DVD A  DS8A5SH   XA15  /dev/sr0
[2:0:0:0]   disk    USB2.0   CardReader CF    0100  /dev/sdb
[2:0:0:1]   disk    USB2.0   CardReader SM XD 0100  /dev/sdc
[2:0:0:2]   disk    USB2.0   CardReader MS    0100  /dev/sdd
[2:0:0:3]   disk    USB2.0   CardReader SD    0100  /dev/sde
[3:0:0:0]   disk    FLASH    Drive UT_USB20   0.00  /dev/sdf
```

Числа в скобках значат следующее (слева направо): номер хост-адаптера SCSI, номер шины SCSI, идентификатор устройства SCSI ID и номер логического устройства LUN (Logical Unit Number, дальнейшее подразделение устройства).

В данном примере в наличии четыре подключенных адаптера (`scsi0`, `scsi1`, `scsi2` и `scsi3`), каждый обладает единственной шиной (ее номер всюду 0) с одним устройством на ней (номер исполнителя также 0). Флеш-ридер USB с идентификатором 2:0:0 имеет четыре логических устройства — по одному на каждый тип флеш-карты, которая может быть вставлена. Ядро назначило отдельный файл устройства каждому логическому устройству.

На рис. 3.2 показана иерархия драйверов и интерфейсов внутри ядра для приведенной системной конфигурации, начиная от индивидуальных драйверов устройств и заканчивая драйверами блоков. Сюда не включены обобщенные драйверы SCSI (`sg`, SCSI generic).

Поначалу такая обширная структура может показаться необъятной, однако поток данных здесь весьма линейный. Начнем анализировать ее, рассмотрев подсистему SCSI и три ее слоя драйверов.

- Верхний слой отвечает за операции для класса устройств. Например, на этом слое имеется драйвер `sd` (для SCSI-диска); он знает, как переводить запросы от интерфейса блочных устройств в специальные команды протокола SCSI для дисков и наоборот.
- Средний слой анализирует и направляет сообщения SCSI между верхним и нижним слоями, а также отслеживает все шины SCSI и устройства, подключенные к системе.
- Нижний слой отвечает за действия, связанные с аппаратными средствами. Расположенные здесь драйверы отсылают исходящие сообщения протокола SCSI конкретным ведущим адаптерам или аппаратным средствам, а также принимают входящие сообщения от аппаратных средств. Причина для такого отделения от верхнего слоя заключается в том, что, хотя сообщения протокола SCSI и унифицированы для какого-либо класса устройств (например, для дисков), разные типы хост-адаптеров обладают отличающимися процедурами для отправки одинаковых сообщений.

Верхний и нижний слои содержат множество различных драйверов, однако важно помнить о том, что для любого файла устройства в вашей системе ядро применяет один драйвер верхнего слоя и один драйвер нижнего слоя. В нашем примере для диска `/dev/sda` ядро использует драйвер `sd` верхнего слоя и драйвер моста ATA на нижнем слое.

Иногда вам может потребоваться применить более одного драйвера верхнего слоя для одного аппаратного средства (см. подраздел 3.6.3).

Для настоящих аппаратных средств SCSI, таких как диск, подключенный к хост-адаптеру SCSI, или аппаратный RAID-контроллер, драйверы нижнего слоя напрямую «общаются» с расположенными ниже аппаратными средствами. Однако для большинства аппаратных средств, которые подключены к вашей подсистеме SCSI, история совсем другая.

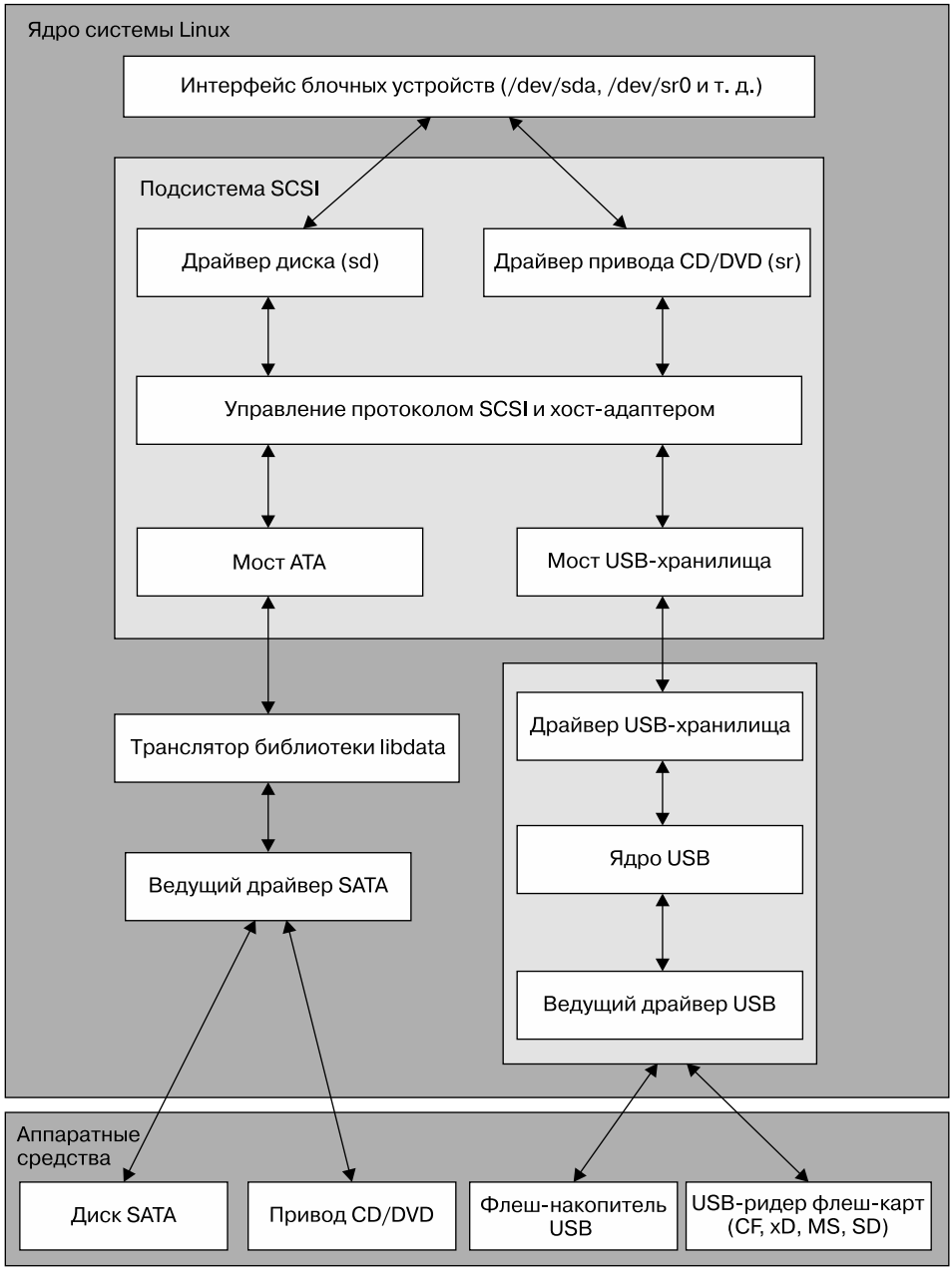


**Рис. 3.2.** Схема подсистемы SCSI в Linux

## 3.6.1. USB-хранилища и протокол SCSI

Чтобы подсистема SCSI могла взаимодействовать с обычными USB-накопителями, как показано на рис. 3.2, ядру необходим не только драйвер SCSI на нижнем

слое. Флеш-устройство USB, представленное файлом `/dev/sdf`, понимает команды SCSI, но для реальной связи с устройством ядру необходимо знать, каким образом «общаться» через систему USB.

В абстракции подсистема USB очень похожа на SCSI: у нее есть классы устройств, шины и хост-контроллеры. Следовательно, не должно вызывать удивления то, что ядро системы Linux содержит трехслойную подсистему USB, которая сильно напоминает подсистему SCSI: сверху расположены драйверы классов устройств, в середине находится ядро управления шиной, а внизу — драйверы хост-контроллера. Подобно тому как подсистема SCSI передает команды между своими компонентами, подсистема USB пересылает сообщения между своими компонентами. В ней есть даже команда `lsusb`, которая подобна команде `lsscsi`.

Часть, которая нам здесь особенно интересна, находится вверху. Это драйвер USB-хранилища. Данный драйвер играет роль переводчика. С одной стороны, он «говорит» на языке SCSI, а с другой — на языке USB. Поскольку аппаратные средства для хранения данных включают команды SCSI внутрь сообщений USB, у драйвера довольно простая работа: главным образом он занят переупаковкой данных.

Когда обе подсистемы (SCSI и USB) заняли свои места, у вас в наличии практически все, что необходимо для обращения к флеш-накопителю. Последнее недостающее звено — это драйвер нижнего слоя в подсистеме SCSI, так как драйвер USB-хранилища является частью подсистемы USB, а не подсистемы SCSI. (Из организационных соображений две подсистемы не должны совместно использовать один и тот же драйвер.) Чтобы две подсистемы смогли общаться друг с другом, на нижнем слое простой драйвер моста SCSI выполняет соединение с драйвером хранилища в подсистеме USB.

## 3.6.2. Интерфейсы SCSI и ATA

Жесткий диск SATA и оптический привод, показанные на рис. 3.2, используют один и тот же интерфейс SATA. Чтобы присоединить драйверы ядра, специфичные для интерфейса SATA, к подсистеме SCSI, ядро задействует драйвер-мост, подобный мосту для USB-накопителей, но с другим механизмом и дополнительными усложнениями. Оптический привод «говорит» на языке ATAPI (это версия команд SCSI, закодированных в протокол ATA). Однако жесткий диск не использует интерфейс ATAPI и не кодирует никаких команд SCSI!

Ядро Linux применяет часть библиотеки libata, чтобы «примирить» приводы SATA (и ATA) с подсистемой SCSI. Для оптических приводов с интерфейсом ATAPI это довольно простая задача, заключающаяся в упаковке и извлечении SCSI-команд, содержащихся в протоколе ATA. Для жесткого диска задача существенно усложняется, поскольку библиотека должна выполнять полную трансляцию команд.

Работа оптического привода подобна работе по набору на компьютере книги на английском языке: вам не обязательно понимать, о чем эта книга, чтобы выполнить работу. Не надо даже понимать английский язык. Задача для жесткого диска напоминает чтение немецкой книги и ее набор на компьютере в виде перевода на английский язык. В этом случае вам необходимо знать оба языка и понимать содержание книги.

Библиотека libata справляется с задачей и дает возможность подключить подсистему SCSI для устройств с интерфейсами ATA/SATA. Как правило, оказывается вовлеченным большее количество драйверов, а не всего лишь один ведущий драйвер SATA, как показано на рис. 3.2. Остальные драйверы не показаны в целях упрощения схемы.

## 3.6.3. Обобщенные устройства SCSI

Процесс из пространства пользователя взаимодействует с подсистемой SCSI с помощью слоя блочных устройств и/или другой службы ядра, расположенной над драйвером класса устройств SCSI (например, `sd` или `sr`). Другими словами, большинству пользовательских процессов нет нужды знать что-либо об устройствах SCSI или об их командах.

Тем не менее пользовательские процессы могут обходить драйверы классов устройств и отправлять команды протокола SCSI напрямую устройствам с помощью *обобщенных устройств*. Посмотрите, например, на систему, описанную ранее в разделе. Но сейчас взгляните на то, что произойдет, когда вы добавите параметр `-g` в команду `lsscsi`, чтобы отобразить обобщенные устройства:

```
$ lsscsi -g
[0:0:0:0]    disk    ATA      WDC WD3200AAJS-2 01.0  /dev/sda ❶/dev/sg0
[1:0:0:0]    cd/dvd  Slimtype DVD A  DS8A5SH   XA15  /dev/sr0 /dev/sg1
[2:0:0:0]    disk    USB2.0   CardReader CF    0100  /dev/sdb /dev/sg2
[2:0:0:1]    disk    USB2.0   CardReader SM XD 0100  /dev/sdc /dev/sg3
[2:0:0:2]    disk    USB2.0   CardReader MS    0100  /dev/sdd /dev/sg4
[2:0:0:3]    disk    USB2.0   CardReader SD    0100  /dev/sde /dev/sg5
[3:0:0:0]    disk    FLASH    Drive UT_USB20   0.00  /dev/sdf /dev/sg6
```

В дополнение к обычному файлу блочного устройства в каждой строке указан файл обобщенного SCSI-устройства (отмечен символом ❶). Так, обобщенным устройством для оптического привода `/dev/sr0` является `/dev/sg1`.

Зачем может понадобиться обобщенное SCSI-устройство? Ответ обусловлен сложностью кода ядра. Когда задачи становятся более тяжелыми, лучше их вывести за пределы ядра. Представьте запись и чтение CD/DVD. Чтение происходит существенно проще записи, при нем не затрагиваются важные службы ядра. Программа в пространстве пользователя выполнила бы запись чуть менее эффективно, чем служба ядра, однако такую программу гораздо проще создать и поддерживать, чем службу ядра, а ошибки в ней не затронут пространство ядра. Следовательно, чтобы записать оптический диск в системе Linux, мы запускаем программу, которая «разговаривает» с обобщенным SCSI-устройством, таким как `/dev/sg1`. Однако благодаря простоте чтения, по сравнению с записью, считывание с устройства происходит с помощью специального драйвера `sr` в ядре.

## 3.6.4. Методы коллективного доступа к одному устройству

На рис. 3.3 для SCSI-подсистемы Linux показаны две точки доступа (`sr` и `sg`) к оптическому приводу из пространства пользователя (опущены все драйверы, которые

расположены под самым нижним уровнем SCSI). Процесс А осуществляет чтение с помощью драйвера `sr`, а процесс Б производит запись с помощью драйвера `sg`. Однако такие процессы не могут одновременно получать доступ к одному устройству.

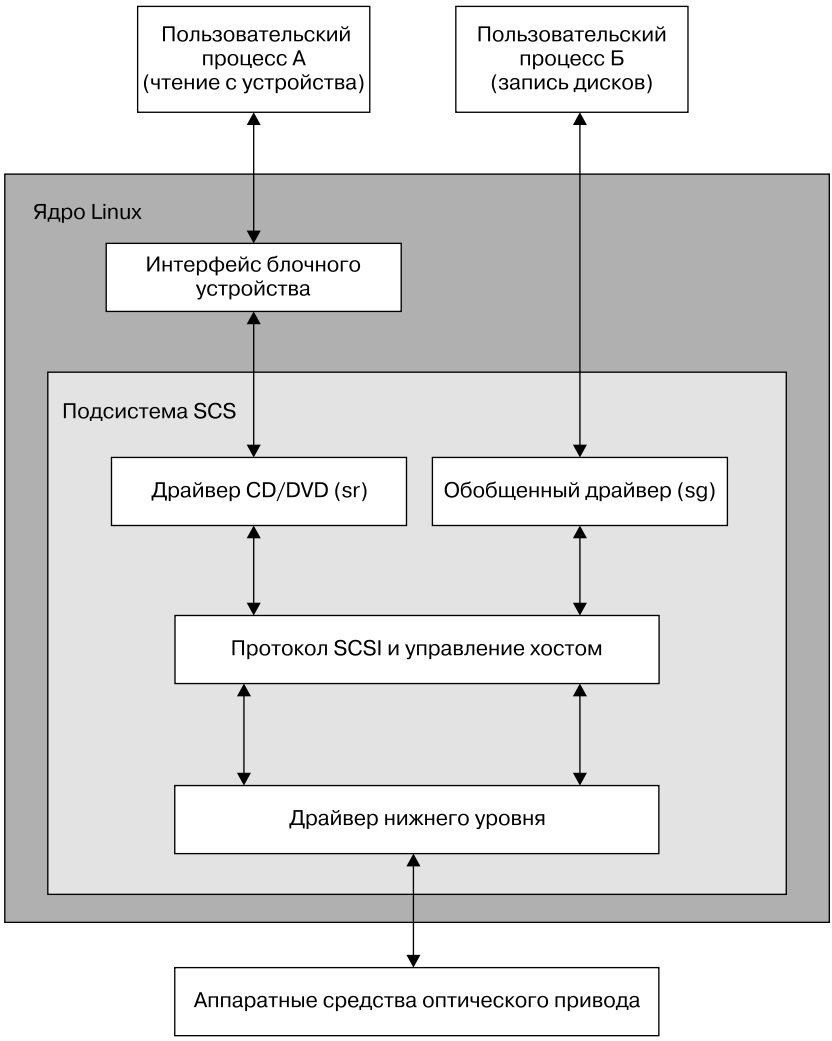


**Рис. 3.3.** Схема драйверов оптического привода

На рис. 3.3 процесс А осуществляет чтение с блочного устройства. Однако действительно ли пользовательские процессы считывают данные подобным образом? Ответ, как правило, отрицательный: нет, напрямую не считывают. Над блочными устройствами есть дополнительные слои, а для жестких дисков — также и дополнительные точки доступа, как вы узнаете из следующей главы.

# 4 Диски и файловые системы

В главе 3 мы рассмотрели дисковые устройства верхнего уровня, которые делают ядро доступным. В данной главе мы детально расскажем о работе с дисками в Linux. Вы узнаете о том, как создавать разделы дисков, настраивать и поддерживать файловые системы в этих разделах, а также работать с областью подкачки.

Вспомните о том, что у дисковых устройств есть имена вроде `/dev/sda`, первого диска подсистемы SCSI. Такой тип блочного устройства представляет диск целиком, однако внутри диска присутствуют различные компоненты и слои.

На рис. 4.1 приведена схема типичного диска в Linux (масштаб не соблюден). По мере изучения этой главы вы узнаете, где находится каждый его фрагмент.

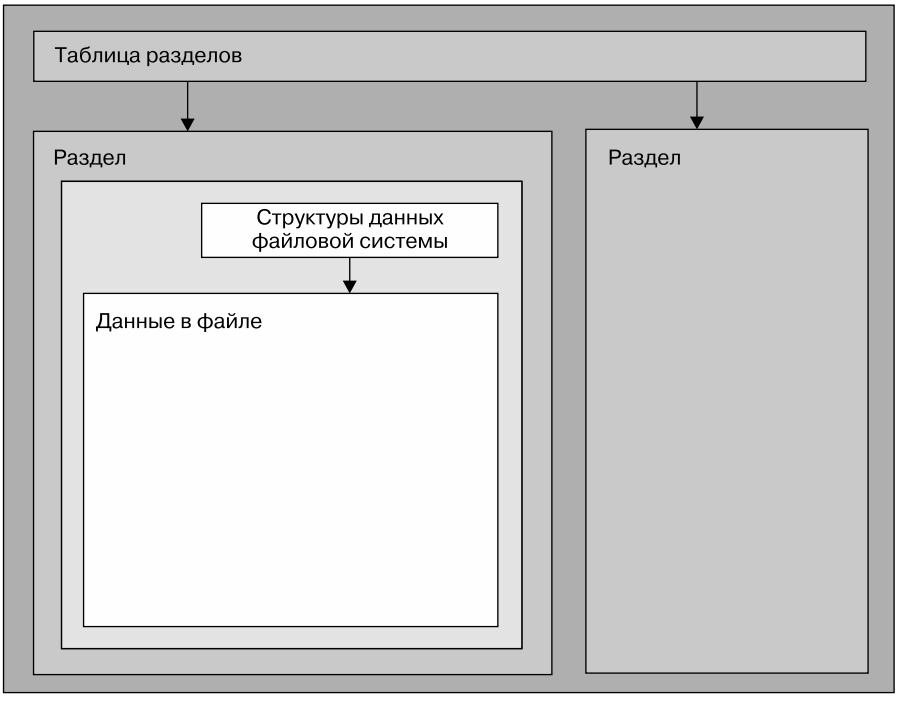


**Рис. 4.1.** Схема типичного диска Linux

Разделы являются более мелкими частями всего диска. В Linux они обозначаются с помощью цифры после названия блочного устройства и, следовательно, получают такие имена, как, например, `/dev/sda1` и `/dev/sdb3`. Ядро представляет каждый раздел в виде блочного устройства, как если бы это был целый диск. Разделы определяются в небольшой области диска, которая называется *таблицей разделов*.

ПРИМЕЧАНИЕ

Многочисленные разделы были когда-то распространены в системах с большими дисками, поскольку старые ПК могли загружаться только из определенных частей диска. К тому же администраторы использовали разделы, чтобы зарезервировать некоторое пространство для областей операционной системы. Например, они исключали возможность того, чтобы пользователи заполнили все свободное пространство системы и нарушили работу важных служб. Такая практика не является исключительной для Unix; вы по-прежнему сможете найти во многих новых системах Windows несколько разделов на одном диске. Кроме того, большинство систем располагает отдельным разделом подкачки.

Хотя ядро и позволяет вам иметь одновременный доступ ко всему диску и к одному из его разделов, вам не придется это делать, если только вы не копируете весь диск.

Следующий за разделом слой является *файловой системой*. Это база данных о файлах и каталогах, с которыми вы привыкли взаимодействовать в пространстве пользователя. Файловые системы будут рассмотрены в разделе 4.2.

Как можно заметить на рис. 4.1, если вам необходим доступ к данным в файле, вам потребуется выяснить из таблицы разделов расположение соответствующего раздела, а затем отыскать в базе данных файловой системы этого раздела желаемый файл с данными.

Чтобы обращаться к данным на диске, ядро Linux использует систему слоев, показанную на рис. 4.2. Подсистема SCSI и все остальное, описанное в разделе 3.6, представлены в виде одного контейнера. Обратите внимание на то, что с дисками можно работать как с помощью файловой системы, так и непосредственно через дисковые устройства. В этой главе вы попробуете оба способа.

Чтобы уяснить, как все устроено, начнем снизу, с разделов.

# 4.1. Разделы дисковых устройств

Существуют различные типы таблиц разделов. Традиционная таблица — та, которая расположена внутри *главной загрузочной записи MBR* (Master Boot Record). Новым, набирающим силу стандартом является *глобальная таблица разделов с уникальными идентификаторами* GPT (Globally Unique Identifier Partition Table).

Приведу перечень доступных в Linux инструментов для работы с разделами:

- `parted` — инструмент командной строки, который поддерживает как таблицу MBR, так и таблицу GPT;
- `gparted` — версия инструмента `parted` с графическим интерфейсом;
- `fdisk` — традиционный инструмент командной строки Linux для работы с разделами. Не поддерживает таблицу GPT;

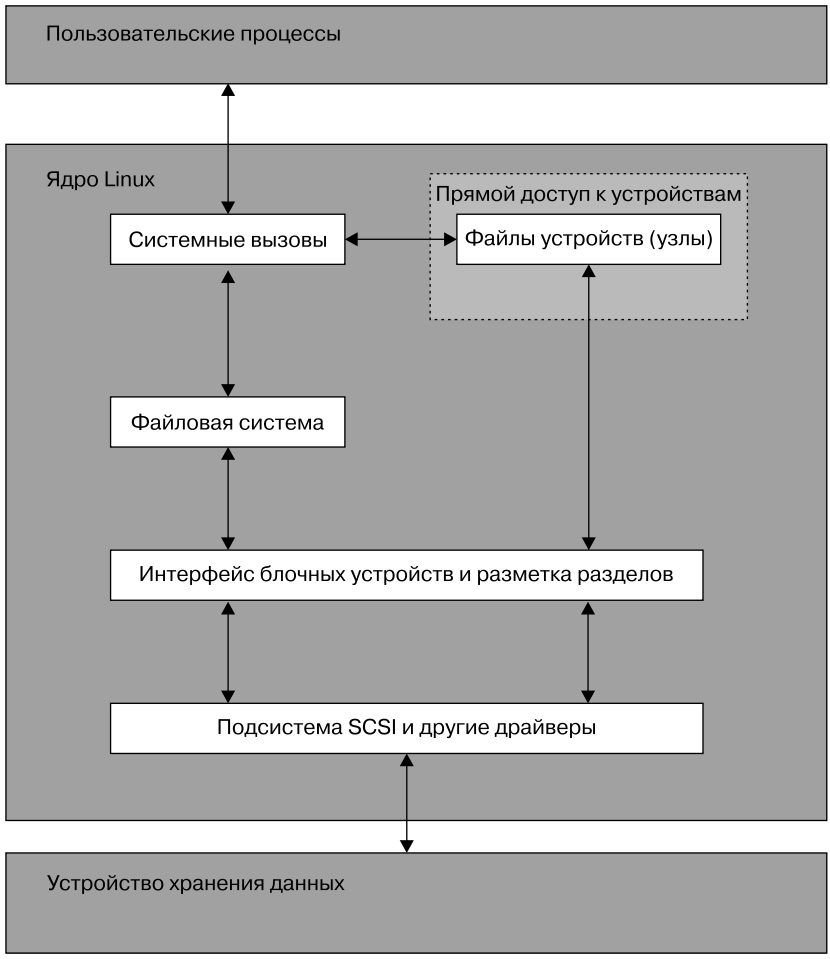


**Рис. 4.2.** Схема доступа ядра к диску

- `gdisk` — версия инструмента `fdisk`, которая поддерживает таблицу GPT, но не работает с MBR.

Поскольку инструмент `parted` поддерживает обе таблицы (MBR и GPT), в данной книге мы будем пользоваться им. Однако многие пользователи предпочитают интерфейс `fdisk`, и в этом нет ничего плохого.

ПРИМЕЧАНИЕ

Хотя команда parted способна создавать и изменять файловые системы, не следует использовать ее для манипуляций с файловой системой, поскольку вы можете легко запутаться. Имеется существенное отличие работы с разделами от работы с файловой системой. Таблица разделов устанавливает границы диска, в то время как файловая система гораздо сильнее вовлечена в структуру данных. Исходя из этого, мы будем применять команду parted для работы с разделами, а для создания файловых систем используем другие утилиты (см. подраздел 4.2.2). Даже документация к команде parted призывает вас создавать файловые системы отдельно.

## 4.1.1. Просмотр таблицы разделов

Можно просмотреть таблицу разделов вашей системы с помощью команды `parted -l`. Приведу пример результатов работы для двух дисковых устройств с различными типами таблиц разделов:

```
# parted -l

Model: ATA WDC WD3200AAJS-2 (scsi)
Disk /dev/sda: 320GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number   Start   End    Size   Type      File system    Flags
 1       1049kB  316GB  316GB  primary   ext4           boot
 2       316GB   320GB  4235MB extended
 5       316GB   320GB  4235MB logical   linux-swap(v1)

Model: FLASH Drive UT_USB20 (scsi)
Disk /dev/sdf: 4041MB
Sector size (logical/physical): 512B/512B
Partition Table: gpt

Number  Start   End     Size     File system  Name        Flags
 1      17.4kB  1000MB  1000MB                myfirst
 2      1000MB  4040MB  3040MB                mysecond
```

Первое устройство, `/dev/sda`, использует традиционную таблицу разделов MBR (которую команда `parted` назвала `msdos`), а второе устройство содержит таблицу GPT.

Обратите внимание на различающиеся параметры в этих таблицах разделов, поскольку сами таблицы различны. В частности, в таблице MBR нет столбца `Name` (Имя), поскольку в этой схеме имена отсутствуют. (Я произвольно указал имена `myfirst` и `mysecond` в таблице GPT.)

Таблица MBR в данном примере содержит основной, расширенный и логический разделы. *Основной раздел* является подразделом диска; раздел 1 — пример тому. В основной таблице MBR предельное количество основных разделов равно четырем. Если вам необходимо больше четырех разделов, вы обозначаете один из них как *расширенный раздел*. Затем вы делите расширенный раздел на *логические разделы*, которые операционная система может использовать подобно любому другому разделу.

В данном примере раздел 2 является расширенным разделом, который содержит логический раздел 5.

ПРИМЕЧАНИЕ

Файловая система, которую выводит команда parted, это не обязательно та система, что определена в поле идентификатора в большинстве записей таблицы MBR. Этот идентификатор является числом; например, 83 — это раздел Linux, а 82 — область подкачки Linux. Таким образом, команда parted пытается самостоятельно определить файловую систему. Если вам необходимо абсолютно точно узнать идентификатор системы для таблицы MBR, используйте команду fdisk -l.

### Первичное чтение ядром

При первичном чтении таблицы MBR ядро Linux выдает следующий отладочный результат (вспомните, что увидеть его можно с помощью команды `dmesg`):

```
sda: sda1 sda2 < sda5 >
```

Фрагмент `sda2 < sda5 >` означает, что устройство `/dev/sda2` является расширенным разделом, который содержит один логический раздел, `/dev/sda5`. Как правило, вы будете игнорировать расширенные разделы, поскольку вам будет нужен доступ только к внутренним логическим разделам.

## 4.1.2. Изменение таблиц разделов

Просмотр таблиц разделов — операция сравнительно простая и безвредная. Изменение таблиц разделов также осуществляется довольно просто, однако при таком типе изменений диска могут возникнуть опасности. Имейте в виду следующее.

- Изменение таблицы разделов сильно усложняет восстановление любых данных в удаляемых разделах, поскольку при этом меняется начальная точка привязки файловой системы. Обязательно создавайте резервную копию диска, на котором вы меняете разделы, если он содержит важную информацию.
- Убедитесь в том, что на целевом диске ни один из разделов в данный момент не используется. Это важно, поскольку в большинстве версий Linux автоматически монтируется любая обнаруженная файловая система (подробности о монтировании и демонтировании см. в подразделе 4.2.3).

Когда вы будете готовы, выберите для себя команду для работы с разделами. Если вы предпочитаете применять команду `parted`, то можете воспользоваться утилитой командной строки или таким графическим интерфейсом, как `gparted`. Для интерфейса в стиле команды `fdisk` воспользуйтесь командой `gdisk`, если вы работаете с разделами GPT. Все эти утилиты обладают интерактивной справкой и просты в освоении. Попробуйте применить их для флеш-накопителя или какого-либо подобного устройства, если у вас нет свободных дисков.

Существуют различия в том, как работают команды `fdisk` и `parted`. С помощью команды `fdisk` вы создаете новую таблицу разделов до выполнения реальных изменений на диске; команда `fdisk` только осуществляет их, когда вы выходите из нее. При использовании команды `parted` разделы создаются, изменяются и удаляются, *когда вы вводите команды*. У вас нет возможности просмотреть таблицу разделов до ее изменения.

Эти различия важны также для понимания того, как данные утилиты взаимодействуют с ядром. Команды `fdisk` и `parted` изменяют разделы полностью в пространстве пользователя; нет необходимости, чтобы ядро обеспечивало поддержку перезаписи таблицы разделов, поскольку пространство пользователя способно считывать и изменять все данные на блочном устройстве.

Однако в конечном счете ядро все же должно считывать таблицу разделов, чтобы представить разделы как блочные устройства. Утилита `fdisk` использует сравнительно простой метод: после изменения таблицы разделов эта команда осуществляет единичный системный вызов к диску, чтобы сообщить ядру о необходимости

повторного считывания таблицы разделов. После этого ядро генерирует отладочный вывод, который можно просмотреть с помощью команды `dmesg`. Например, если вы создаете два раздела на устройстве `/dev/sdf`, вы увидите следующее:

```
sdf: sdf1 sdf2
```

В сравнении с этой командой инструменты `parted` не используют системный вызов для всего диска. Вместо него они сигнализируют ядру об изменении отдельных разделов. После обработки изменения одного раздела ядро не производит приведенного выше отладочного вывода.

Есть несколько способов увидеть изменения разделов.

- Используйте команду `udevadm`, чтобы отследить изменения событий ядра. Например, команда `udevadm monitor --kernel` покажет удаленные устройства-разделы и добавленные новые.
- Посмотрите полную информацию о разделах в файле `/proc/partitions`.
- Поищите в каталоге `/sys/block/device/` измененные системные интерфейсы разделов или в каталоге `/dev` — измененные устройства-разделы.

Если вы хотите быть абсолютно уверенными в том, что таблица разделов изменена, можно выполнить «старомодный» системный вызов, который применяет команда `fdisk`, использовав команду `blockdev`. Например, чтобы ядро принудительно перезагрузило таблицу разделов на устройстве `/dev/sdf`, запустите следующую команду:

```
# blockdev --rereadpt /dev/sdf
```

На данный момент вы знаете все необходимое о работе с разделами дисков. Если вам интересно изучить некоторые дополнительные подробности о дисках, продолжайте чтение. В противном случае переходите к разделу 4.2, чтобы узнать о размещении файловой системы на диске.

## 4.1.3. Диск и геометрия раздела

Любое устройство с подвижными частями добавляет сложностей в систему программного обеспечения, поскольку физические элементы сопротивляются абстрагированию. Жесткие диски не являются исключением. Хоть и возможно представлять жесткий диск как блочное устройство с произвольным доступом к любому блоку, возникают серьезные последствия для производительности, если вы не позаботились о том, как располагаются данные на диске. Рассмотрим физические свойства простого диска с одной пластиной, изображенного на рис. 4.3.

Диск состоит из вращающейся на шпинделе пластины, а также головки, которая прикреплена к подвижному кронштейну, который может перемещаться вдоль радиуса диска. Когда диск вращается под головкой, последняя считывает данные. Когда кронштейн расположен в определенной позиции, головка может считывать данные только с одной окружности. Эта окружность называется *цилиндром*, поскольку у больших дисков несколько пластин, которые надеты на один шпиндель и вращаются вокруг него. Каждая пластина может иметь одну или две головки, для верхней и/или нижней части пластины, причем все головки крепятся на одном

кронштейне и перемещаются совместно. Поскольку кронштейн двигается, на диске есть много цилиндров, от самых малых около центра диска до самых больших по его краям. Наконец, цилиндр можно разделить на доли, называемые *секторами*. Такой способ представления геометрии диска называется *CHS* (cylinder-head-sector, цилиндр-головка-сектор).

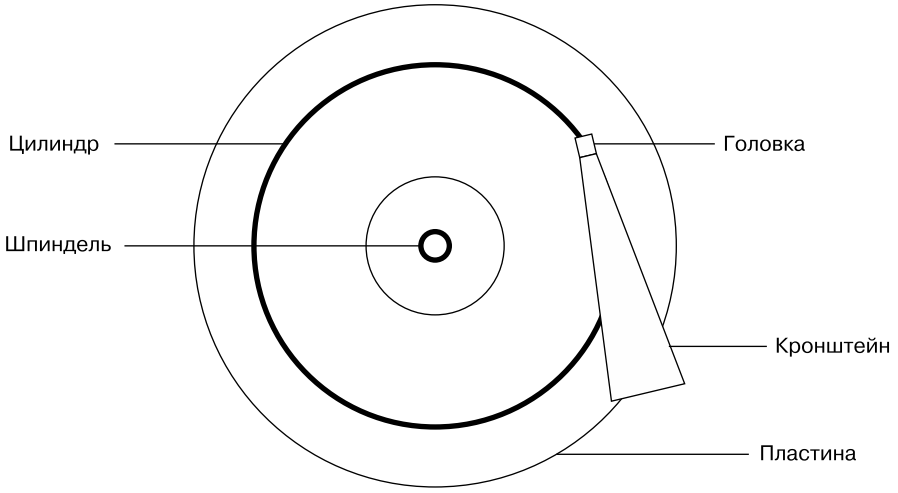


**Рис. 4.3.** Жесткий диск, вид сверху

ПРИМЕЧАНИЕ

Дорожка является частью цилиндра, к которой имеет доступ одна головка, поэтому на рис. 4.3 цилиндр является также и дорожкой.

Ядро и различные программы для работы с разделами могут сообщить вам о том, что́ из себя представляет диск как совокупность цилиндров (и *секторов*, которые являются частями цилиндров). Однако для современных жестких дисков *сообщаемые значения являются фиктивными*! Традиционная схема адресации, которая использует параметры CHS, не вписывается в современное аппаратное обеспечение жестких дисков. Она также не принимает в расчет тот факт, что в одних цилиндрах можно разместить больше данных, чем в других. Дисковые аппаратные средства поддерживают *блочную адресацию LBA (Logical Block Addressing)*, чтобы просто обращаться к какому-либо месту диска по номеру блока. Однако следы системы CHS еще присутствуют. Например, таблица разделов MBR содержит информацию CHS, а также ее LBA-эквивалент, и некоторые загрузчики системы по-прежнему довольно глупы, чтобы доверять значениям CHS (но не беспокойтесь — в большинстве загрузчиков Linux используются значения LBA).

Тем не менее понятие о цилиндрах оказалось важным для работы с разделами, поскольку цилиндры являются идеальными границами для разделов. Чтение потока данных с цилиндра происходит очень быстро, так как головка может непрерывно считывать данные по мере вращения диска. Раздел, который организован как набор смежных цилиндров, также позволяет получить быстрый доступ

к данным, поскольку головке не приходится перемещаться слишком далеко между цилиндрами.

Некоторые программы для работы с разделами выражают недовольство, если вы не размечаете разделы точно по границам цилиндров. Игнорируйте это. Вы мало чем сможете помочь, поскольку значения CHS для современных дисков попросту недостоверны. Схема LBA гарантирует вам то, что разделы окажутся именно там, где вы предполагали.

### 4.1.4. Твердотельные накопители (диски SSD)

Устройства хранения без движущихся частей, такие как *твердотельные накопители* (*SSD*), совершенно отличны от вращающихся дисков, если говорить о характеристиках доступа к данным. Для них произвольный доступ не является проблемой, так как отсутствует перемещающаяся вдоль пластины головка. Однако некоторые факторы отражаются на производительности.

Одним из наиболее значимых факторов, влияющих на производительность дисков SSD, является *выравнивание разделов*. Когда вы считываете данные с диска SSD, чтение происходит фрагментарно — как правило, порциями по 4096 байт за один прием, — причем такое чтение должно начинаться с числа, кратного этому размеру. Поэтому, если раздел и данные в нем не располагаются в пределах 4096-байтной зоны, вам может понадобиться выполнить две небольшие операции чтения вместо одной, например чтения содержимого каталога.

Многие утилиты для работы с разделами (например, `parted` и `gparted`) содержат средства для размещения вновь созданных разделов с правильными отступами от начала диска, и вам никогда не придется беспокоиться о неверном выравнивании разделов. Однако, если вам любопытно узнать, где начинаются ваши разделы, чтобы убедиться в том, что они начинаются от границ, можно легко это выяснить, заглянув в каталог `/sys/block`. Вот пример раздела для устройства `/dev/sdf2`:

```
$ cat /sys/block/sdf/sdf2/start
1953126
```

Этот раздел начинается на расстоянии 1 953 126 байт от начала диска. Поскольку это число не делится нацело на 4096, работа с таким разделом не достигала бы оптимальной производительности, если бы он был расположен на диске SSD.

## 4.2. Файловые системы

Последним звеном между ядром и пространством пользователя для дисков обычно является *файловая система*. С ней вы привыкли взаимодействовать, когда запускали такие команды, как `ls` и `cd`. Как отмечалось ранее, файловая система является разновидностью базы данных; она поддерживает структуру, призванную трансформировать простое блочное устройство в замысловатую иерархию файлов и подкаталогов, которую пользователи способны понять.

В свое время файловые системы, располагавшиеся на дисках и других физических устройствах, использовались исключительно для хранения данных. Однако

древовидная структура каталогов, а также интерфейс ввода-вывода довольно гибки, поэтому теперь файловые системы выполняют множество задач, например роль системных интерфейсов, которые вы можете увидеть в каталогах `/sys` и `/proc`. Файловые системы традиционно реализованы внутри ядра, однако инновационный протокол 9P из операционной системы Plan 9 (http://plan9.bell-labs.com/sys/doc/9.html) способствовал разработке файловых систем в пространстве пользователя. Функция *FUSE* (File System in User Space, файловая система в пространстве пользователя) позволяет применять такие файловые системы в Linux.

Слой абстракции *VFS* (виртуальная файловая система) завершает реализацию файловой системы. Во многом подобно тому, как подсистема SCSI стандартизирует связь между различными типами устройств и управляющими командами ядра, слой VFS обеспечивает поддержку стандартного интерфейса всеми реализациями файловых систем, чтобы приложения из пространства пользователя одинаковым образом обращались с файлами и каталогами. Виртуальная файловая система позволяет Linux поддерживать невообразимо большое число файловых систем.

## 4.2.1. Типы файловых систем

В Linux включена поддержка таких файловых систем, как «родные» разработки, оптимизированные для Linux, «чужеродные» типы, например семейство Windows FAT, универсальные файловые системы вроде ISO 9660 и множество других. В приведенном ниже списке перечислены наиболее распространенные типы файловых систем для хранения данных. Имена типов систем, как их определяет Linux, приведены в скобках после названия файловых систем.

- *Четвертая расширенная файловая система* (ext4) является текущей реализацией в линейке «родных» для Linux файловых систем. *Вторая расширенная файловая система* (ext2) долгое время была системой по умолчанию в системах Linux, которые испытывали влияние традиционных файловых систем Unix, таких как файловая система Unix (UFS, Unix File System) и быстрая файловая система (FFS, Fast File System). В *третьей расширенной файловой системе* (ext3) появился режим журналирования (небольшой кэш за пределами нормальной структуры данных файловой системы) для улучшения целостности данных и ускорения загрузки системы. Файловая система ext4 является дальнейшим улучшением, с поддержкой файлов большего размера по сравнению с допустимым в системах ext2 или ext3, а также большего количества подкаталогов.

  Среди расширенных файловых систем присутствует некоторая доля обратной совместимости. Например, можно смонтировать систему ext2 как ext3 или наоборот, а также смонтировать файловые системы ext2 и ext3 как ext4, однако нельзя смонтировать файловую систему ext4 как ext2 или ext3.

- *Файловая система ISO 9660* (iso9660) — это стандарт для дисков CD-ROM. Большинство дисков CD-ROM использует какой-либо вариант стандарта ISO 9660.

- *Файловые системы FAT* (msdos, vfat, umsdos) относятся к системам Microsoft. Простой тип msdos поддерживает весьма примитивное унылое многообразие систем MS-DOS. Для большинства современных файловых систем Windows следует использовать тип vfat, чтобы получить возможность полного доступа

из ОС Linux. Редко используемый тип umsdos представляет интерес для Linux: в нем есть поддержка таких особенностей Unix, как символические ссылки, которые находятся над файловой системой MS-DOS.

- *Tun HFS+* (hfsplus) является стандартом Apple, который используется в большинстве компьютеров Macintosh.

Хотя расширенные файловые системы были абсолютно пригодны для применения обычными пользователями, в технологии файловых систем были произведены многочисленные улучшения, причем такие, что даже система ext4 не может ими воспользоваться в силу требований обратной совместимости. Эти улучшения относятся главным образом к расширяемости системы, как то: очень большое количество файлов, файлы большого объема и другие подобные вещи. Новые файловые системы Linux, такие как Btrfs, находятся в разработке и могут прийти на смену расширенным файловым системам.

## 4.2.2. Создание файловой системы

Когда вы завершите работу с разделами, которая описана выше (см. раздел 4.1), можно создавать файловую систему. Как и для разделов, это выполняется в пространстве пользователя, поскольку процесс из пространства пользователя может напрямую обращаться к блочному устройству и работать с ним. Утилита `mkfs` способна создать многие типы файловых систем. Например, можно создать раздел типа ext4 в устройстве `/dev/sdf2` с помощью такой команды:

```
# mkfs -t ext4 /dev/sdf2
```

Команда `mkfs` автоматически определяет количество блоков в устройстве и устанавливает некоторые разумные параметры по умолчанию. Если вы не в полной мере представляете, что делаете, или если не любите читать подробную документацию, не меняйте эти настройки.

При создании файловой системы команда `mkfs` осуществляет диагностический вывод, включая и тот, который относится к *суперблоку*. Суперблок является ключевым компонентом, расположенным на верхнем уровне базы данных файловой системы. Он настолько важен, что утилита `mkfs` создает для него несколько резервных копий на случай утраты оригинала. Постарайтесь записать несколько номеров резервных копий суперблока во время работы команды `mkfs`, они могут вам понадобиться, когда придется восстанавливать суперблок после ошибки диска (см. подраздел 4.2.11).

ВНИМАНИЕ

Создание файловой системы — это задача, которую необходимо выполнять только после добавления нового диска или изменения разделов на существующем. Файловую систему следует создавать лишь один раз для каждого нового раздела, на котором еще нет данных (или который содержит данные, подлежащие удалению). Создание новой файловой системы поверх уже существующей фактически уничтожает старые данные.

Оказывается, утилита `mkfs` является только «лицевой стороной» набора команд для создания файловых систем. Эти команды называются `mkfs.`*`fs`*, где вместо *`fs`* подставлен тип файловой системы. Таким образом, когда вы запускаете команду `mkfs -t ext4`, утилита `mkfs`, в свою очередь, запускает команду `mkfs.ext4`.

Но двуличности здесь еще больше. Исследуйте файлы, которые скрываются за обозначениями `mkfs.*`, и вы увидите следующее:

```
$ ls -l /sbin/mkfs.*
-rwxr-xr-x 1 root root 17896 Mar 29 21:49 /sbin/mkfs.bfs
-rwxr-xr-x 1 root root 30280 Mar 29 21:49 /sbin/mkfs.cramfs
lrwxrwxrwx 1 root root     6 Mar 30 13:25 /sbin/mkfs.ext2 -> mke2fs
lrwxrwxrwx 1 root root     6 Mar 30 13:25 /sbin/mkfs.ext3 -> mke2fs
lrwxrwxrwx 1 root root     6 Mar 30 13:25 /sbin/mkfs.ext4 -> mke2fs
lrwxrwxrwx 1 root root     6 Mar 30 13:25 /sbin/mkfs.ext4dev -> mke2fs
-rwxr-xr-x 1 root root 26200 Mar 29 21:49 /sbin/mkfs.minix
lrwxrwxrwx 1 root root     7 Dec 19  2011 /sbin/mkfs.msdos -> mkdosfs
lrwxrwxrwx 1 root root     6 Mar  5  2012 /sbin/mkfs.ntfs -> mkntfs
lrwxrwxrwx 1 root root     7 Dec 19  2011 /sbin/mkfs.vfat -> mkdosfs
```

Файл `mkfs.ext4` является лишь символической ссылкой на `mke2fs`. Об этом важно помнить, если вы натолкнетесь на какую-либо систему без специальной команды `mkfs` или же когда станете искать документацию по какой-либо файловой системе. Каждой утилите для создания файловой системы посвящена особая страница в руководстве, например, mke2fs(8). В большинстве версий ОС это не создаст проблем, поскольку при попытке доступа к странице mkfs.ext4(8) руководства вы будете перенаправлены на страницу mke2fs(8). Просто имейте это в виду.

## 4.2.3. Монтирование файловой системы

В Unix процесс присоединения файловой системы называется *монтированием*. Когда система загружается, ядро считывает некоторые конфигурационные данные и на их основе монтирует корневой каталог (/).

Чтобы выполнить монтирование файловой системы, вы должны знать следующее:

- устройство для размещения файловой системы (например, раздел диска; на нем будут располагаться актуальные данные файловой системы);
- тип файловой системы;
- *точку монтирования*, то есть место в иерархии каталогов текущей системы, куда будет присоединена файловая система. Точка монтирования всегда является обычным каталогом. Например, можно использовать каталог `/cdrom` в качестве точки монтирования для приводов CD-ROM. Точка монтирования не обязана находиться именно в корневом каталоге, в системе она может быть где угодно.

Для монтирования файловой системы применяется терминология «смонтировать устройство в точке монтирования». Чтобы узнать статус текущей файловой системы, запустите команду `mount`. Результат будет выглядеть примерно так:

```
$ mount
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
none on /sys/fs/fuse/connections type fusectl (rw)
none on /sys/kernel/debug type debugfs (rw)
```

```
none on /sys/kernel/security type securityfs (rw)
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
--snip--
```

Каждая строка соответствует одной файловой системе, смонтированной в настоящее время. Перечислены следующие элементы:

- устройство, например `/dev/sda3`. Обратите внимание на то, что некоторые устройства в действительности не являются таковыми (например, `proc`), а играют роль заместителей для имен реальных устройств, поскольку таким файловым системам специального назначения не нужны устройства;
- слово `on`;
- точка монтирования;
- слово `type`;
- тип файловой системы, как правило, в виде краткого идентификатора;
- параметры монтирования (в скобках) (см. подробности в подразделе 4.2.6).

Чтобы смонтировать файловую систему, используйте приведенную ниже команду `mount`, указав тип файловой системы, устройство и желаемую точку монтирования:

```
# mount -t type device mountpoint
```

Чтобы, например, смонтировать четвертую расширенную файловую систему `/dev/sdf2` в точке `/home/extra`, используйте такую команду:

```
# mount -t ext4 /dev/sdf2 /home/extra
```

Обычно не требуется указывать параметр `-t`, поскольку команда `mount` способна догадаться о нем сама. Однако иногда бывает необходимо сделать различие между сходными типами файловых систем, таких как FAT, например.

В подразделе 4.2.6 можно увидеть еще несколько более длинных параметров монтирования. Чтобы демонтировать (открепить) файловую систему, воспользуйтесь командой `umount`:

```
# umount mountpoint
```

Можно также демонтировать файловую систему вместе с ее устройством, а не с точкой монтирования.

## 4.2.4. Файловая система UUID

Метод монтирования файловых систем, рассмотренный в предыдущем разделе, зависит от названий устройств. Однако имена устройств могут измениться, поскольку они зависят от порядка их обнаружения ядром. Чтобы справиться с этой проблемой, можно идентифицировать и монтировать файловые системы по их *идентификатору UUID* (Universally Unique Identifier, универсальный уникальный идентификатор), который является стандартом в программном обеспечении. Идентификатор UUID — это своего рода серийный номер, причем каждый такой номер уникален. Команды

для создания файловых систем, такие как mke2fs, присваивают идентификатор UUID при инициализации структуры данных файловой системы.

Чтобы просмотреть список устройств, соответствующих им файловых систем, а также идентификаторы UUID, используйте команду blkid (block ID):

```
# blkid
/dev/sdf2: UUID="a9011c2b-1c03-4288-b3fe-8ba961ab0898" TYPE="ext4"
/dev/sda1: UUID="70ccd6e7-6ae6-44f6-812c-51aab8036d29" TYPE="ext4"
/dev/sda5: UUID="592dcfd1-58da-4769-9ea8-5f412a896980" TYPE="swap"
/dev/sde1: SEC_TYPE="msdos" UUID="3762-6138" TYPE="vfat"
```

В этом примере команда blkid обнаружила четыре раздела с данными: два из них с файловой системой ext4, один с сигнатурой области подкачки (см. раздел 4.3) и один с файловой системой семейства FAT. Все собственные разделы Linux снабжены стандартными идентификаторами UUIDs, однако у раздела FAT он отсутствует. К разделу FAT можно обратиться с помощью серийного номера тома FAT (в данном случае это 3762-6138).

Чтобы смонтировать файловую систему по ее идентификатору UUID, используйте синтаксис UUID=. Например, для монтирования первой файловой системы из приведенного выше списка в точке /home/extra введите такую команду:

```
# mount UUID=a9011c2b-1c03-4288-b3fe-8ba961ab0898 /home/extra
```

Как правило, монтировать файловые системы вручную по их идентификаторам не придется, поскольку вам, вероятно, известно устройство, а смонтировать устройство по его имени гораздо проще, чем использовать безумный номер UUID. Однако все же важно понимать суть идентификаторов UUID. С одной стороны, они являются предпочтительным средством для автоматического монтирования файловых систем в точке /etc/fstab во время загрузки системы (см. раздел 4.2.8). Помимо этого, многие версии ОС используют идентификатор UUID в качестве точки монтирования, когда вы вставляете сменный носитель данных. В приведенном выше примере файловая система FAT находится на флеш-карте. Ubuntu, если какой-либо пользователь зашел в нее, смонтирует данный раздел в точке /media/3762-6138 после вставки носителя. Демон udevd, описанный в главе 3, обрабатывает начальное событие для вставки устройства.

Если необходимо, можно изменить идентификатор UUID для файловой системы (например, если вы скопировали всю файловую систему куда-либо еще, и теперь вам необходимо отличать ее от оригинала). Обратитесь к странице tune2fs(8) руководства, чтобы узнать о том, как это выполнить в файловых системах ext2/ext3/ext4.

## 4.2.5. Буферизация диска, кэширование и файловые системы

Система Linux, подобно другим версиям Unix, выполняет буферизацию при записи на диск. Это означает, что ядро обычно не сразу же вносит изменения в файловую систему, когда процессы запрашивают их. Вместо этого ядро хранит такие изменения в оперативной памяти до тех пор, пока ядро не сможет с удобством

выполнить реальные изменения на диске. Такая система буферизации очевидна для пользователя и улучшает производительность.

Когда вы демонтируете файловую систему с помощью команды `umount`, ядро автоматически синхронизируется с диском. В любой другой момент времени можно выполнить принудительную запись изменений из буфера ядра на диск, запустив команду `sync`. Если по каким-либо причинам невозможно демонтировать файловую систему до выхода из операционной системы, обязательно запустите сначала команду `sync`.

Кроме того, ядро располагает рядом механизмов, использующих оперативную память, чтобы автоматически кэшировать блоки, считанные с диска. Следовательно, если один или несколько процессов часто обращаются к какому-либо файлу, то ядру не приходится снова и снова получать доступ к диску — оно может просто выполнить чтение из кэша, экономя время и системные ресурсы.

## 4.2.6. Параметры монтирования файловой системы

Существует множество способов изменить режим работы команды `mount`, поскольку часто бывает необходимо поработать со съемными накопителями или выполнить обслуживание системы. Общее число параметров команды поражает. Исчерпывающее руководство на странице mount(8) является хорошей справкой, но при этом трудно понять, с чего следует начать, а чем можно пренебречь. В данном разделе вы увидите наиболее полезные параметры.

Параметры разделены на две категории:

- общие параметры. Содержат флаг `-t` для указания типа файловой системы;
- параметры, зависящие от файловой системы. Относятся только к определенным типам файловых систем.

Чтобы задействовать параметр для какой-либо файловой системы, используйте перед ним флаг `-o`. Например, параметр `-o norock` отключает расширения Rock Ridge в файловой системе ISO 9660, однако для любой другой файловой системы он не имеет смысла.

### Короткие параметры

Наиболее важные общие параметры таковы.

- `-r` — монтирует файловую систему в режиме «только для чтения». Это может пригодиться в разных случаях: начиная с защиты от записи и заканчивая самозагрузкой. Нет необходимости указывать данный параметр при доступе к такому устройству, как CD-ROM, система сделает это за вас (а также уведомит о том, что статус устройства — только для чтения).
- `-n` — гарантирует то, что команда `mount` не будет пытаться обновить исполняемую системную базу данных монтирования `/etc/mtab`. Операция монтирования прерывается, если она не может производить запись в данный файл, а это важно во время загрузки системы, поскольку корневой раздел (и, следовательно, системная база данных монтирования) поначалу доступен только для чтения. Этот параметр может быть полезен, когда вы будете пытаться исправить

системную ошибку в режиме одиночного пользователя, поскольку в этот момент системная база данных монтирования не будет доступна.

- `-t` — задает тип файловой системы.

### Длинные параметры

Короткие параметры, вроде `-r`, слишком коротки для постоянно увеличивающегося количества параметров монтирования. К тому же в алфавите не так много букв, чтобы обозначить ими все возможные параметры. Короткие параметры могут также вызвать проблемы, поскольку на основе одной буквы сложно определить значение параметра. Для многих общих параметров, а также для всех параметров, которые зависят от файловой системы, используется более длинный и гибкий формат параметров.

Чтобы применять длинные параметры для утилиты `mount` в командной строке, начните с флага `-o` и добавьте несколько ключевых слов. Вот пример полной команды, снабженной длинными параметрами после флага `-o`:

```
# mount -t vfat /dev/hda1 /dos -o ro,conv=auto
```

Здесь присутствуют два длинных параметра: `ro` и `conv=auto`. Параметр `ro` задает режим «только чтение» и эквивалентен короткому параметру `-r`. Параметр `conv=auto` дает ядру указание об автоматической конвертации определенных текстовых файлов из формата DOS с переводом строки в формат Unix (совсем скоро вы узнаете об этом).

Наиболее полезны следующие длинные параметры:

- `exec`, `noexec` — включает или отключает исполнение команд над файловой системой;
- `suid`, `nosuid` — включает или отключает команды `setuid` (установка идентификатора пользователя);
- `ro` — монтирует файловую систему в режиме «только чтение» (подобно короткому параметру `-r`);
- `rw` — монтирует файловую систему в режиме «чтение-запись»;
- `conv=`*`rule`* (для файловых систем на основе FAT) — конвертирует содержащиеся в файлах символы перевода строки, в зависимости от атрибута *`rule`*, который может принимать значения `binary`, `text` или `auto`. По умолчанию установлено значение `binary`, при котором отключена конвертация символов. Чтобы трактовать все файлы как текстовые, используйте значение `text`. Если указать значение `auto`, конвертация файлов будет происходить на основе их расширения. Например, файл `.jpg` обрабатываться не будет, а файл `.txt` пройдет специальную обработку. Будьте осторожны с этим параметром, поскольку он может повредить файлы. Постарайтесь применять его в режиме «только чтение».

## 4.2.7. Демонтирование файловой системы

Иногда может возникнуть необходимость заново присоединить недавно смонтированную файловую систему к той же точке монтирования, изменив при этом параметры монтирования. Чаще всего это случается, когда вам необходимо открыть доступ на запись в файловой системе во время восстановления после сбоя.

Следующая команда заново монтирует корневой каталог в режиме «чтение-запись» (параметр `-n` необходим, поскольку команда `mount` не может вести запись в системную базу данных монтирования, если корневой каталог находится в режиме «только чтение»):

```
# mount -n -o remount /
```

Эта команда подразумевает, что корректный перечень устройств для корневого каталога расположен в каталоге `/etc/fstab` (о чем будет сказано в следующем разделе). Если это не так, следует указать устройство.

## 4.2.8. Таблица файловой системы /etc/fstab

Чтобы смонтировать файловые системы во время загрузки, а также избавить команду `mount` от нудной работы, Linux постоянно хранит список файловых систем и их параметров в таблице `/etc/fstab`. Это файл в обычном текстовом формате, достаточно простом, как можно увидеть из примера 4.1.

**Пример 4.1.** Список файловых систем и их параметров в файле /etc/fstab

```
proc /proc proc nodev,noexec,nosuid 0 0
UUID=70ccd6e7-6ae6-44f6-812c-51aab8036d29 / ext4 errors=remount-ro 0 1
UUID=592dcfd1-58da-4769-9ea8-5f412a896980 none swap sw 0 0
/dev/sr0 /cdrom iso9660 ro,user,nosuid,noauto 0 0
```

Каждая строка, содержащая шесть полей, соответствует одной файловой системе. Ниже перечислены эти поля (слева направо).

- **Устройство или идентификатор UUID.** Большинство современных систем Linux больше не использует устройство в файле `/etc/fstab`, предпочитая идентификатор UUID. Обратите внимание на то, что запись `/proc` содержит устройство-заместитель с именем `proc`.
- **Точка монтирования.** Указывает, где присоединяется файловая система.
- **Тип файловой системы.** Скорее всего, вам незнаком параметр `swap` в данном перечне; это раздел подкачки (см. раздел 4.3).
- **Параметры.** Использованы длинные параметры, разделенные запятыми.
- **Информация о резервной копии для использования командой сброса.** В этом поле всегда следует указывать значение `0`.
- **Порядок проверки целостности системы.** Чтобы команда `fsck` всегда начинала работу с корневого каталога, устанавливайте в этом поле значение `1` для корневой файловой системы и значение `2` для остальных файловых систем на жестком диске. Используйте значение `0`, чтобы отключить при запуске проверку чего-либо еще, включая приводы CD-ROM, область подкачки и файловую систему `/proc` (о команде `fsck` можно узнать в подразделе 4.2.11).

При использовании команды `mount` можно применять некоторые обходные пути, если файловая система, с которой вы желаете работать, есть в таблице `/etc/fstab`. Если бы, например, вы использовали систему из листинга 4.1 и монтировали CD-ROM, можно было бы просто запустить команду `mount /cdrom`.

Можно также попытаться смонтировать разом все компоненты, перечисленные в таблице /etc/fstab (если они не снабжены параметром noauto), с помощью такой команды:

```
# mount -a
```

Пример 4.1 содержит несколько новых параметров, а именно: errors, noauto и user, поскольку они не применяются вне файла /etc/fstab. Кроме того, вам часто будет встречаться здесь параметр defaults. Перечисленные параметры означают следующее.

- defaults. Используются параметры mount команды по умолчанию: режим «чтение-запись», применение файлов устройств, исполняемых файлов, бита setuid и т. п. Используйте этот параметр, когда вам не нужно специальным образом настраивать файловую систему, однако необходимо заполнить все поля в таблице /etc/fstab.
- errors. Этот параметр, относящийся к файловой системе ext2, определяет поведение ядра, когда операционная система испытывает сложности при монтировании файловой системы. По умолчанию обычно указан вариант errors=continue, который означает, что ядро должно возвратить код ошибки и продолжить работу. Чтобы заставить ядро выполнить монтирование заново в режиме «только чтение», используйте вариант errors=remount-ro. Вариант errors=panic говорит ядру (и вашей системе) о том, что необходимо выполнить останов, когда возникают проблемы с монтированием.
- noauto. Этот параметр сообщает команде mount -a, что данную запись следует игнорировать. Используйте его, чтобы предотвратить во время загрузки системы монтирование сменных накопителей, например дисков CD-ROM или флоппи-дисков.
- user. Данный параметр позволяет пользователям без специальных прав доступа запускать команду mount для какой-либо отдельной записи, что может быть удобно для предоставления доступа к приводам CD-ROM. Поскольку пользователи могут разместить корневой файл setuid на сменном носителе с другой системой, данный параметр устанавливает также атрибуты nosuid, noexec и nodev (чтобы исключить специальные файлы устройств).

## 4.2.9. Альтернативы таблицы /etc/fstab

Хотя файл /etc/fstab традиционно применяется для представления файловых систем и их точек монтирования, появилось два альтернативных способа. Первый — это каталог /etc/fstab.d, который содержит отдельные файлы конфигурации файловой системы (по одному на каждую файловую систему). Идея очень похожа на многие другие конфигурационные каталоги, которые встретятся вам в этой книге.

Второй способ — конфигурирование модулей демона systemd для файловых систем. Подробности о демоне systemd и его модулях вы узнаете из главы 6. Тем не менее конфигурация модуля systemd часто исходит из таблицы /etc/fstab (или основана на ней), поэтому в вашей системе могут встретиться некоторые частичные совпадения.

## 4.2.10. Мощность файловой системы

Чтобы увидеть размеры и степень использования смонтированных в данный момент файловых систем, воспользуйтесь командой `df`. Результат ее работы может выглядеть так:

```
$ df
Filesystem    1024-blocks     Used    Available  Capacity Mounted on
/dev/sda1         1011928    71400      889124      7%   /
/dev/sda3        17710044  9485296     7325108     56%   /usr
```

Приведу краткое описание полей в этом выводе:

- `Filesystem` — устройство, на котором расположена файловая система;
- `1024-blocks` — общая мощность файловой системы в блоках по 1024 байта;
- `Used` — количество занятых блоков;
- `Available` — количество свободных блоков;
- `Capacity` — процент использованных блоков;
- `Mounted on` — точка монтирования.

Легко заметить, что эти две файловые системы занимают приблизительно 1 и 17,5 Гбайт. Однако значения мощности могут выглядеть немного странно, поскольку при сложении 71 400 и 889 124 не получается 1 011 928, а 9 485 296 не составляет 56 % от 17 710 044. В обоих случаях 5 % от общей мощности не учтены. На самом деле это пространство присутствует, но оно спрятано в *зарезервированных* блоках. Следовательно, только пользователь superuser может использовать все пространство файловой системы, если остальная часть раздела окажется заполненной. Такая особенность предотвращает немедленный отказ в работе системных серверов, когда заканчивается свободное пространство.

Если ваш диск заполнен и вы желаете знать, где расположены все эти пожирающие пространство медиафайлы, воспользуйтесь командой `du`. При запуске без аргументов эта команда выводит статистику использования диска для каждого каталога в иерархии каталогов, начиная с текущего рабочего каталога. Запустите команду `cd /;`, чтобы понять суть, остановите сочетанием клавиш Ctrl+C. Команда `du -s` работает в режиме общего подсчета и выводит только итоговую сумму. Чтобы проверить какой-либо один каталог, перейдите в него и запустите команду `du -s *`.

ПРИМЕЧАНИЕ

Стандарт POSIX (Portable Operating System Interface for Unix, переносимый интерфейс операционных систем Unix) определяет размер блока равным 512 байтам. Однако такой размер сложнее воспринимается при чтении, поэтому по умолчанию результаты работы команд df и du в большинстве версий Linux выражены в 1024-байтных блоках. Если вы настаиваете на отображении значений в виде 512-байтных блоков, задайте переменную окружения POSIXLY_CORRECT. Чтобы явно указать блоки размером 1024 байта, используйте параметр -k (его поддерживают обе утилиты). У команды df есть также параметр -m, чтобы отображать мощность в блоках размером 1 Мбайт, и параметр -h, который пытается выбрать наиболее удобное представление для чтения.

## 4.2.11. Проверка и восстановление файловых систем

Предлагаемые файловыми системами Unix оптимизации возможны благодаря замысловатому устройству базы данных. Чтобы файловые системы работали бесперебойно, ядро должно быть уверено в том, что в смонтированной файловой системе нет ошибок. Если они присутствуют, может произойти потеря данных и сбой в работе системы.

Ошибки в файловой системе обычно возникают в результате того, что пользователь грубым образом выходит из системы (например, выдергивая кабель электропитания). В подобных случаях кэш файловой системы в памяти может не соответствовать данным на диске, к тому же система может выполнять изменение файловой системы, когда вы подвергаете компьютер «встряске». Хотя новые поколения файловых систем снабжены журналированием, чтобы сделать их повреждение менее частым, вам всегда следует выходить из системы корректным образом. Вне зависимости от используемой файловой системы для поддержания ее стабильной работы необходимо регулярно выполнять проверки.

Инструмент для проверки файловой системы называется `fsck`. Подобно команде `mkfs`, у него существуют различные версии для каждого типа файловой системы, поддерживаемого Linux. Например, когда вы применяете команду `fsck` для расширенных файловых систем (ext2/ext3/ext4), она распознает тип файловой системы и запускает утилиту `e2fsck`. Следовательно, вам, как правило, не придется вручную вводить `e2fsck`, если только команда `fsck` не сможет выяснить тип файловой системы или вы ищете страницу руководства по команде `e2fsck`.

Информация, представленная в этом разделе, относится к расширенным файловым системам и команде `e2fsck`.

Чтобы запустить команду `fsck` в режиме интерактивного руководства, укажите в качестве аргумента устройство или точку монтирования (как они приведены в таблице `/etc/fstab`). Например, так:

```
# fsck /dev/sdb1
```

ВНИМАНИЕ

Никогда не используйте команду fsck для смонтированной файловой системы, поскольку ядро может изменить данные на диске во время работы проверки. Это вызовет несоответствия во время исполнения, которые могут привести к сбою системы и повреждению файлов. Есть всего одно исключение: если вы монтируете корневой раздел только для чтения в режиме единственного пользователя, то в этом разделе можно запустить команду fsck.

В режиме руководства команда `fsck` выводит подробные сообщения о проходах проверки, которые в случае отсутствия ошибок могут выглядеть так:

```
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information /dev/sdb1: 11/1976 files (0.0% non-
contiguous), 265/7891 blocks
```

Если в режиме руководства команда `fsck` обнаружит ошибку, она остановится и задаст вам вопрос, относящийся к устранению проблемы. Подобные вопросы касаются внутренней структуры файловой системы, например повторного подключения неприкрепленных дескрипторов `inode` или очистки блоков (дескриптор `inode` является строительным блоком файловой системы; о работе этих дескрипторов подробнее в разделе 4.5). Когда команда `fsck` спрашивает вас о повторном подключении дескриптора `inode`, это означает, что обнаружен файл, у которого, по-видимому, нет имени. При подключении такого файла команда `fsck` помещает его в каталог `lost+found` файловой системы, указав число в качестве имени файла. Если такое происходит, вам потребуется определить его имя, основываясь на содержимом файла; исходное имя файла будет, вероятно, утрачено.

Не имеет смысла дожидаться окончания процесса восстановления, если вы всего лишь некорректно вышли из системы, поскольку команда `fsck` может обнаружить большое число ошибок, подлежащих устранению. К счастью, у команды `e2fsck` есть параметр `-p`, который выполняет автоматическое исправление типичных проблем, не задавая вопросов, и прерывает работу только в случае серьезной ошибки. На самом деле версии Linux вовремя запускают какой-либо вариант команды `fsck -p`. Вам может также встретиться команда `fsck -a`, которая выполняет то же самое.

Если вы подозреваете, что в вашей системе есть такие проблемы, как отказ аппаратных средств или неправильная конфигурация устройств, вам необходимо определить порядок действий, поскольку команда `fsck` способна попросту превратить в кашу файловую систему с многочисленными ошибками. Одним из явных признаков того, что у вашей системы есть серьезные проблемы, является обилие вопросов, которые задает команда `fsck` в ручном режиме работы.

Попробуйте запустить команду `fsck -n`, чтобы проверить файловую систему, ничего не изменяя. Если возникла проблема в конфигурации устройства и вы думаете, что ее можно исправить (например, неправильное число блоков в таблице разделов или неплотно подключенные кабели), сделайте это до запуска команды `fsck` в реальном режиме. В противном случае вы можете потерять большое количество данных.

Если вы думаете, что поврежден лишь суперблок (например, когда кто-либо выполнил запись в начале дискового раздела), можно попробовать восстановить файловую систему на основе одной из резервных копий, созданных командой `mkfs`. Используйте команду `fsck -b` *num*, чтобы заменить поврежденный суперблок альтернативным блоком *num*.

Если вы не знаете, где искать резервную копию суперблока, можно запустить команду `mkfs -n` на данном устройстве, чтобы просмотреть список номеров резервных копий суперблока, не повреждая данные. Опять-таки убедитесь в том, что вы используете флаг `-n`, чтобы не разнести файловую систему на куски.

## Проверка файловых систем ext3 и ext4

Обычно вам не понадобится вручную проверять файловые системы ext3 и ext4, поскольку целостность данных обеспечивается журналированием. Однако вам может потребоваться смонтировать «поломанную» файловую систему ext3 или ext4 в режиме ext2, поскольку ядро не станет монтировать такие файловые системы с непустым журналом. Если выход из системы был совершен некорректно, то журнал может

содержать какие-либо данные. Чтобы очистить журнал в файловой системе ext3 или ext4, приведя ее к стандартной базе данных, запустите следующую команду:

```
# e2fsck -fy /dev/disk_device
```

### Наихудший случай

Самые серьезные дисковые проблемы оставляют вам не много вариантов для выбора.

- Можно попробовать извлечь из диска образ всей файловой системы с помощью команды `dd` и переместить ее в раздел другого диска с таким же размером.
- Можно попытаться «залатать» файловую систему, насколько это возможно, смонтировать ее в режиме «только чтение» и спасти что удастся.
- Можно попробовать команду `debugfs`.

В первом и во втором случаях вам все же понадобится исправить файловую систему до ее монтирования, если только вы не являетесь любителем ручного разбора сырых данных. Если желаете, можно ответить `y` на все вопросы команды `fsck`, запустив ее в таком виде: `fsck -y`. Однако используйте этот вариант в последнюю очередь, поскольку во время процесса восстановления могут обнаружиться ошибки, которые лучше исправлять вручную.

Инструмент `debugfs` позволяет вам просматривать файлы в файловой системе и копировать их куда-либо. По умолчанию он работает с файловыми системами в режиме «только чтение». Если вы восстанавливаете данные, вероятно, было бы неплохо оставить файлы неприкосновенными, чтобы избежать дальнейшей неразберихи.

Если же вы пришли в полное отчаяние в результате катастрофического сбоя и отсутствия резервных копий, вам остается только надеяться на то, что профессиональный сервис сможет «соскоблить данные с пластин».

## 4.2.12. Файловые системы специального назначения

Не все файловые системы представляют хранилища на физических носителях. В большинстве версий Unix есть файловые системы, которые играют роль системных интерфейсов. Такие файловые системы не только служат средством хранения данных на устройстве, но способны также представлять системную информацию, например идентификаторы процессов и диагностические сообщения ядра. Эта идея восходит к механизму `/dev`, который является ранней моделью использования файлов для интерфейсов ввода-вывода. Идея применения каталога `/proc` берет начало из восьмого издания экспериментальной версии Unix, которая была реализована Томом Дж. Киллианом (Tom J. Killian) и усовершенствована сотрудниками лаборатории Bell Labs (включая многих первичных разработчиков Unix), создавшими Plan 9 — экспериментальную операционную систему, которая вывела файловую систему на новый уровень абстракции (http://plan9.bell-labs.com/sys/doc/9.html).

Специальные типы файловых систем, которые широко применяются в Linux, включают следующие.

- **proc**. Смонтирована в каталоге `/proc`. Имя `proc` является сокращением слова *process* («процесс»). Каждый нумерованный каталог внутри `/proc` — это идентификатор происходящего в системе процесса; файлы в этих каталогах отражают различные характеристики процессов. Файл `/proc/self` представляет текущий процесс. Файловая система `proc` в Linux содержит внушительное количество дополнительной информации о ядре и аппаратных средствах в файлах вроде `/proc/cpuinfo`. Информация, которая не относится к процессам, перенесена из каталога `/proc` в каталог `/sys`.
- **sysfs**. Смонтирована в каталоге `/sys` (его вы встречали в главе 3).
- **tmpfs**. Смонтирована в каталоге `/run` и других. С помощью файловой системы tmpfs вы можете использовать физическую память и область подкачки в качестве временного хранилища. Например, можно смонтировать tmpfs там, где вам нравится, применяя длинные параметры `size` и `nr_blocks` для контроля максимального размера. Будьте осторожны и не помещайте постоянно данные в систему tmpfs, поскольку в итоге у операционной системы может закончиться память и программы начнут выходить из строя. Так, корпорация Sun Microsystems годами применяла вариант файловой системы tmpfs для каталога `/tmp`, что вызывало проблемы на компьютерах, работающих в течение долгого времени.

# 4.3. Область подкачки

Не каждый раздел диска содержит файловую систему. Пополнять оперативную память компьютера можно также за счет дискового пространства. Если оперативная память на исходе, система виртуальной памяти в Linux может автоматически перемещать фрагменты памяти на дисковое хранилище и обратно. Этот процесс называется *подкачкой* (свопингом), поскольку участки бездействующих команд перекачиваются на диск в обмен на активные фрагменты памяти, расположенные на диске. Область диска, используемая для хранения страниц памяти, называется *областью подкачки* (или сокращенно *swap*).

Результатом работы команды `free` является следующая информация о применении подкачки (в килобайтах):

```
$ free
             total       used       free
--snip--
Swap:        514072     189804     324268
```

## 4.3.1. Использование раздела диска в качестве области подкачки

Чтобы полностью применять раздел диска для подкачки, выполните следующее.

1. Убедитесь в том, что раздел пуст.
2. Запустите команду `mkswap` *dev*, в которой в качестве параметра *dev* укажите устройство с необходимым разделом. Эта команда помещает в данный раздел сигнатуру области подкачки.
3. Запустите команду `swapon dev`, чтобы зарегистрировать область с помощью ядра.

После создания раздела подкачки можно внести новую запись о нем в файл `/etc/fstab`, чтобы система смогла применять область подкачки уже при загрузке компьютера. Вот пример такой записи, в которой в качестве раздела подкачки использовано устройство `/dev/sda5`:

```
/dev/sda5 none swap sw 0 0
```

Принимайте во внимание то, что теперь многие системы применяют идентификаторы UUID вместо простых названий устройств.

## 4.3.2. Использование файла в качестве области подкачки

Можно применять обычный файл в качестве области подкачки, если возникнет ситуация, когда вы будете вынуждены изменить разделы диска, чтобы создать область подкачки. Вы не должны испытать при этом какие-либо сложности.

Воспользуйтесь приведенными ниже командами, чтобы создать пустой файл, инициализировать его для подкачки, а затем добавить в пул подкачки:

```
# dd if=/dev/zero of=swap_file bs=1024k count=num_mb
# mkswap swap_file
# swapon swap_file
```

Здесь параметр *swap_file* задает имя нового файла подкачки, а параметр *num_mb* — желаемый размер в мегабайтах.

Чтобы удалить раздел или файл подкачки из активного пула ядра, используйте команду `swapoff`.

## 4.3.3. Какой объем области подкачки необходим

В свое время, согласно традиционной мудрости Unix, полагали, что всегда следует резервировать для подкачки по меньшей мере в два раза больший объем памяти по сравнению с оперативной. Теперь этот вопрос не столь однозначен, поскольку стали доступны огромные объемы дискового пространства и оперативной памяти, а также изменились способы использования системы. С одной стороны, дисковое пространство настолько обширно, что возникает искушение выделить больше памяти, чем двойной размер оперативной памяти. С другой — вам, вероятно, никогда не придется использовать область подкачки, когда в наличии имеется столько реальной памяти.

Правило о «двойном размере реальной памяти» возникло в то время, когда к одному компьютеру могло быть подключено одновременно несколько пользователей. Не все они могли быть активны, поэтому было удобно переводить в область

подкачки те участки памяти, которые были выделены для неактивных пользователей, когда активному пользователю требовалось больше памяти.

Это может по-прежнему быть верным для компьютера с одним пользователем. Если вы запускаете много процессов, как правило, будет неплохо переместить в область подкачки части неактивных процессов или даже неактивные фрагменты активных процессов. Тем не менее, если вы постоянно применяете область подкачки, поскольку многие активные процессы желают сразу же использовать память, вы будете испытывать серьезные сложности с производительностью, так как дисковый ввод-вывод происходит слишком медленно и ему не угнаться за остальной частью системы. Выходы таковы: приобрести дополнительную память или завершить некоторые процессы.

Иногда ядро Linux может перевести в область подкачки какой-либо процесс с целью получения дополнительного дискового кэша. Чтобы это предотвратить, администраторы конфигурируют отдельные системы вообще без области подкачки. Например, высокопроизводительным сетевым серверам не следует использовать область подкачки и по возможности избегать обращения к диску.

ПРИМЕЧАНИЕ

Это опасно выполнять для компьютера общего назначения. Если на компьютере полностью будут исчерпаны оперативная память и область подкачки, ядро Linux запускает подавитель OOM (out-of-memory, нехватка памяти), чтобы прервать процесс и освободить некоторое количество памяти. Вы, безусловно, не захотите, чтобы это случилось с вашими приложениями. С другой стороны, высокопроизводительные серверы содержат сложные системы слежения и выравнивания нагрузки, чтобы никогда не оказаться в опасной зоне.

Из главы 8 вы узнаете подробнее о том, как работает система памяти.

## 4.4. Заглядывая вперед: диски и пространство пользователя

В относящихся к дискам компонентах системы Unix границы между пространством пользователя и пространством ядра довольно сложно определить. Как вы уже видели, ядро оперирует блочным вводом-выводом от устройств, а инструменты из пространства пользователя способны использовать блочный ввод-вывод с помощью файлов устройств. Тем не менее пространство пользователя, как правило, применяет блочный ввод-вывод только при инициализации таких операций, как создание разделов, файловых систем и области подкачки. В нормальном режиме пространство пользователя задействует только поддержку файловой системы, которая обеспечивается ядром в верхнем слое блочного ввода-вывода. Подобным образом ядро управляет и множеством мелких деталей, когда оно имеет дело с областью подкачки в системе виртуальной памяти.

В следующей части этой главы вкратце рассказано про внутренние части файловой системы Linux. Это более сложный материал, и вам определенно нет необходимости знать его, чтобы продолжить чтение книги. Переходите к следующей главе, чтобы начать изучение процесса загрузки системы Linux.

# 4.5. Внутри традиционной файловой системы

Традиционная файловая система Unix содержит два основных компонента: пул блоков данных, где можно хранить данные, и базу данных, которая управляет пулом данных. В основу базы данных положена структура данных inode. *Дескриптор inode* — это набор данных, который описывает конкретный файл, включая его тип, права доступа и (что, возможно, наиболее важно) расположение его данных в пуле данных. Дескрипторы inode распознаются по номерам, перечисленным в соответствующей таблице.

Имена файлов и каталогов также реализованы в виде дескрипторов inode. Дескриптор каталога содержит перечень имен файлов и соответствующих ссылок на другие дескрипторы.

Чтобы привести реальный пример, я создал новую файловую систему, смонтировал ее и сменил каталог на точку монтирования. После этого добавил несколько файлов и каталогов с помощью таких команд (попробуйте выполнить это самостоятельно на флеш-накопителе):

```
$ mkdir dir_1
$ mkdir dir_2
$ echo a > dir_1/file_1
$ echo b > dir_1/file_2
$ echo c > dir_1/file_3
$ echo d > dir_2/file_4
$ ln dir_1/file_3 dir_2/file_5
```

Обратите внимание на то, что я создал каталог `dir_2/file_5` в виде жесткой ссылки на каталог `dir_1/file_3`. Это означает, что данные два имени файлов на самом деле представляют один и тот же файл.

Если вы рассмотрите каталоги в этой файловой системе, то ее содержимое выглядело бы так, как показано на рис. 4.4. Реальная разметка файловой системы, как показано на рис. 4.5, не выглядит настолько ясной, как представление на уровне пользователя.

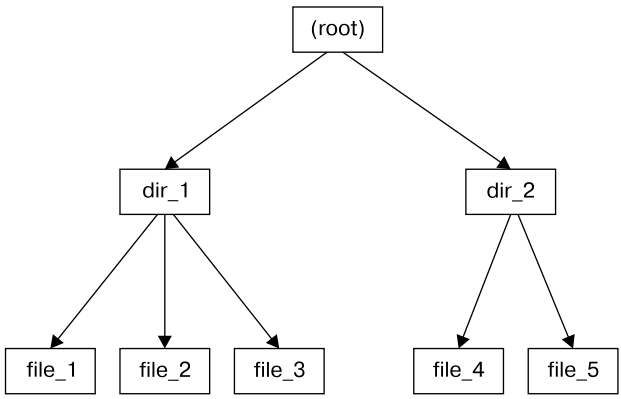


**Рис. 4.4.** Представление файловой системы на уровне пользователя

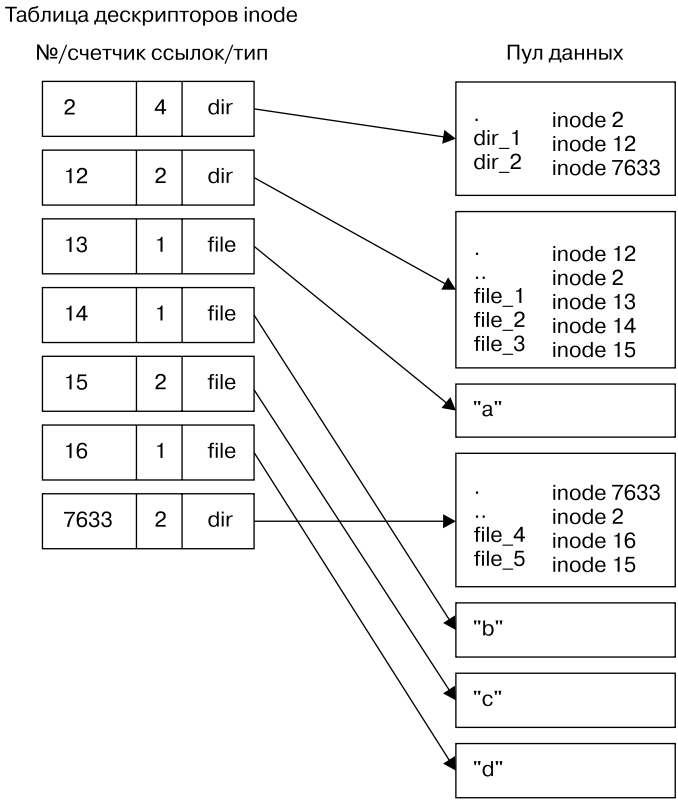


**Рис. 4.5.** Структура дескрипторов файловой системы, показанной на рис. 4.4

В любой расширенной файловой системе (ext2/3/4) нумерация дескрипторов начинается с 2 — *корневого дескриптора inode*. Из таблицы дескрипторов на рис. 4.5 можно заметить, что это дескриптор каталога (dir), поэтому можно перейти по стрелке к пулу данных, где вы увидите содержимое корневого каталога: два элемента с именами `dir_1` и `dir_2`, которые соответствуют дескрипторам inode 12 и 7633. Чтобы разобраться с этими элементами, вернитесь в таблицу дескрипторов и посмотрите на любой из них.

Для проверки ссылки `dir_1/file_2` в этой файловой системе ядро выполняет следующие действия.

1. Определяет компоненты пути: за каталогом `dir_1` следует компонент с именем `file_2`.
2. Переходит к корневому дескриптору и его данным о каталогах.
3. Отыскивает в данных о каталогах у дескриптора inode 2 имя `dir_1`, которое указывает на дескриптор с номером 12.
4. Ищет дескриптор inode 12 в таблице дескрипторов и проверяет, является ли он дескриптором каталога.

5. Следует по ссылке дескриптора inode 12 к информации о каталоге (это второй сверху контейнер в пуле данных).
6. Обнаруживает второй компонент пути (`file_2`) в данных о каталогах дескриптора inode 12. Эта запись указывает на дескриптор inode с номером 14.
7. Отыскивает дескриптор inode 14 в таблице каталогов. Это дескриптор файла.

К этому моменту ядро знает свойства файла и может открыть его, проследовав по ссылке на данные дескриптора inode 14.

Такая система дескрипторов, указывающих на структуры данных каталогов, и структур данных, указывающих на дескрипторы, позволяет создать иерархию файловой системы, к которой вы привыкли. Кроме того, обратите внимание на то, что дескрипторы каталогов содержат записи для текущего каталога (`.`) и родительского (`..`), исключая лишь корневой каталог. С их помощью можно легко получить точку отсчета при переходе на уровень выше в структуре каталогов.

## 4.5.1. Просмотр деталей дескрипторов inode

Чтобы увидеть значения дескриптора для любого каталога, используйте команду `ls -i`. Вот что получится, если применить ее к корневому каталогу нашего примера. Для получения более детальной информации воспользуйтесь командой `stat`.

```
$ ls -i
  12 dir_1 7633 dir_2
```

Теперь вас, вероятно, заинтересует счетчик ссылок. Он уже встречался в результатах работы обычной команды `ls -l`. Как счетчик ссылок относится к файлам, показанным на рис. 4.5, и, в частности, к «жестко связанному» файлу `file_5`? Поле со счетчиком ссылок содержит общее количество записей в каталоге (по всем каталогам), которые указывают на дескриптор inode. У большинства файлов счетчик ссылок равен 1, поскольку они появляются лишь один раз среди записей каталога. Этого и следовало ожидать: в большинстве случаев при создании файла вы создаете новую запись в каталоге и соответствующий ей новый дескриптор inode. Однако дескриптор 15 появляется дважды: сначала он создан как дескриптор для файла `dir_1/file_3`, а затем связан с файлом `dir_2/file_5`. Жесткая ссылка является лишь созданной вручную записью в каталоге, связанной с уже существующим дескриптором inode. Команда `ln` (без параметра `-s`) позволяет вручную создавать новые ссылки.

По этой причине удаление файла иногда называется *разъединением*. Если запустить команду `rm dir_1/file_2`, ядро начнет поиск записи с именем `file_2` в каталоге дескриптора inode 12. Обнаружив, что `file_2` соответствует дескриптору inode 14, ядро удаляет эту запись из каталога, а затем вычитает 1 из счетчика ссылок для дескриптора inode 14. В результате этот счетчик ссылок становится равным 0 и ядро будет знать о том, что с данным дескриптором inode не связаны никакие имена. Следовательно, этот дескриптор и все относящиеся к нему данные можно удалить.

Однако, если запустить команду `rm dir_1/file_3`, в результате получится, что счетчик ссылок для дескриптора inode 15 изменится с 2 на 1 (поскольку ссылка `dir_2/file_5` все еще указывает на него) и ядро узнает о том, что данный дескриптор не следует удалять.

Счетчик ссылок для каталогов устроен во многом так же. Заметьте, что счетчик ссылок дескриптора inode 12 равен 2, поскольку присутствуют две ссылки на этот дескриптор: одна для каталога *dir_1* в записях каталога для дескриптора inode 2, а вторая ссылка в собственных записях каталога (`.`) ведет на саму себя. Если создать новый каталог *dir_1/dir_3*, счетчик ссылок для дескриптора inode 12 станет равным 3, поскольку новый катлог будет включать запись о родительском каталоге (`..`), который связан с дескриптором inode 12, подобно тому, как родительская ссылка дескриптора inode 12 указывает на дескриптор inode 2.

Есть одно небольшое исключение. У корневого дескриптора inode 2 счетчик ссылок равен 4. Однако на рис. 4.5 показаны только три ссылки на записи каталогов. «Четвертая» ссылка находится в суперблоке файловой системы, поскольку именно он указывает, где отыскать корневой дескриптор inode.

Не бойтесь экспериментировать со своей системой. Создание структуры каталогов с последующим применением команды `ls -i` или `stat` для исследования различных частей является безопасным. Вам не придется выполнять перезагрузку системы (если только вы не смонтировали новую файловую систему).

Тем не менее один фрагмент еще отсутствует: каким образом файловая система при назначении блоков из пула данных для нового файла узнает о том, какие блоки использованы, а какие свободны? Один из самых простых способов состоит в применении дополнительной структуры управления данными, которая называется *битовой картой блоков*. В этой схеме файловая система резервирует последовательность байтов, в которой каждый бит соответствует одному блоку в пуле данных. Если бит равен 0, это означает, что данный блок свободен, а если 1, то блок используется. Таким образом, в основу распределения блоков положено переключение состояния битов.

Проблемы в файловой системе возникают тогда, когда данные из таблицы дескрипторов inode не соответствуют данным о размещении блоков или когда счетчики ссылок неверные; это может произойти, если работа системы была завершена некорректно. Тогда при проверке файловой системы (как было рассказано в подразделе 4.2.11) команда `fsck` будет просматривать таблицу дескрипторов inode и структуру каталогов, чтобы определить новые счетчики ссылок и создать новую карту размещения блоков (такую как битовая карта блоков), после чего она сравнит только что созданные данные с файловой системой на диске. Если обнаружатся несоответствия, команда `fsck` должна исправить счетчики ссылок и определить, как поступить с теми дескрипторами inode и/или данными, которые не были обнаружены при просмотре структуры каталогов. Большинство команд `fsck` помещает такие «осиротевшие» новые файлы в каталог `lost+found`.

## 4.5.2. Работа с файловыми системами в пространстве пользователя

При работе с файлами и каталогами в пространстве пользователя вы можете не беспокоиться о том, как это реализовано на нижнем уровне. От вас ожидают, что доступ к содержимому файлов и каталогов смонтированной файловой системы будет осуществляться с помощью системных вызовов ядра. Любопытно, что при

этом у вас все же есть доступ к определенной системной информации, которая не очень вписывается в пространство пользователя, в частности, системный вызов `stat()` возвращает номера дескрипторов inode и счетчики ссылок.

Если вы не заняты обслуживанием файловой системы, следует ли волноваться насчет дескрипторов inode и счетчиков ссылок? Как правило, нет. Эти данные доступны для команд из пространства пользователя в основном в целях обратной совместимости. К тому же не все файловые системы, доступные в Linux, располагают необходимой для них «начинкой». Слой интерфейса VFS обеспечивает то, что системные вызовы всегда возвращают значения дескрипторов inode и счетчиков ссылок, однако такие числа не обязаны что-либо значить.

В нетрадиционных файловых системах вы можете быть лишены возможности выполнять обычные для Unix операции с файловой системой. Например, нельзя использовать команду `ln`, чтобы создать жесткую ссылку в смонтированной файловой системе VFAT, поскольку в ней совершенно другая структура записей о каталогах.

Системные вызовы, которые доступны в пространстве пользователя Unix/Linux, обеспечивают достаточный уровень абстракции для безболезненного доступа к файлам: вам не обязательно знать что-либо о том, как он реализован, чтобы работать с файлами. Более того, гибкий формат имен файлов и поддержка использования разного регистра символов облегчают возможность применения других файловых систем с иерархической структурой.

Помните о том, что ядро не обязано поддерживать специфические файловые системы. В файловых системах с пространством пользователя ядро лишь выступает в роли проводника для системных вызовов.

## 4.5.3. Эволюция файловых систем

Даже в самой простой файловой системе имеется множество различных компонентов, требующих обслуживания. В то же время предъявляемые к файловым системам требования неуклонно возрастают по мере появления новых задач, технологий и возможностей хранения данных. Нынешний уровень производительности, целостности данных и требований безопасности намного превосходит ранние реализации файловых систем, поскольку технология файловых систем постоянно меняется. Мы уже упоминали в качестве примера о Btrfs — файловой системе нового поколения (см. подраздел 4.2.1).

Одним из примеров того, как изменяются файловые системы, является использование новыми файловыми системами раздельных структур данных для представления каталогов и имен файлов, а не дескрипторов inode, описанных здесь. Они по-другому ссылаются на блоки данных. Кроме того, в процессе развития находятся файловые системы, оптимизированные для дисков SSD. Постоянные изменения в развитии файловых систем являются нормой, однако имейте в виду, что эволюция файловых систем не изменяет их предназначения.

# 5 Как происходит загрузка ядра Linux

Теперь вы знаете о физической и логической структуре системы Linux, что такое ядро и как работать с процессами. В этой главе вы получите информацию о том, как ядро начинает работу или загружается, иначе говоря, как ядро перемещается в память до того момента, где начинается первый пользовательский процесс.

Упрощенная схема процесса загрузки выглядит так.

1. Система BIOS или прошивка загрузки загружают и запускают загрузчик системы.
2. Загрузчик системы отыскивает образ ядра на диске, загружает его в память и запускает.
3. Ядро выполняет инициализацию устройств и их драйверов.
4. Ядро монтирует корневую файловую систему.
5. Ядро запускает команду `init` с идентификатором процесса 1. Эта точка является *началом пространства пользователя*.
6. Команда `init` приводит в действие остальные системные процессы.
7. В определенный момент команда `init` запускает процесс, позволяющий вам войти в систему. Обычно это происходит в конце или незадолго до окончания загрузки системы.

В этой главе рассмотрены шаги с первого по четвертый, основное внимание уделено загрузчикам ядра и системы. В главе 6 продолжается рассказ о загрузке пространства пользователя.

Способность определять каждую стадию процесса загрузки окажется неоценимой, когда вам придется устранять проблемы при загрузке, а также поможет понять систему в целом. Однако принятый по умолчанию режим загрузки во многих версиях Linux зачастую затрудняет (если не делает невозможным) идентификацию некоторых первых этапов, и вам, вероятно, удастся посмотреть на них только по их завершении, после входа в систему.

## 5.1. Сообщения при запуске

Традиционные системы Unix во время загрузки выводят множество диагностических сообщений, чтобы проинформировать вас о ходе загрузки. Эти сообщения

исходят сначала от ядра, а затем от процессов и процедур инициализации, которые запускает команда init. Однако такие сообщения не очень привлекательны и последовательны, а в некоторых случаях они даже не слишком информативны. В большинстве современных версий Linux стараются скрыть их с помощью экранов загрузки, заполнителя и параметров загрузки. Кроме того, улучшенные аппаратные средства привели к тому, что ядро загружается намного быстрее, чем раньше, поэтому сообщения проскакивают настолько быстро, что было бы трудно уяснить, что происходит.

Существуют два способа увидеть сообщения ядра о загрузке и оперативной диагностике. Вы можете:

- заглянуть в системный журнал ядра. Обычно он находится в файле /var/log/kern.log, но в зависимости от конфигурации вашей системы может также оказаться вместе с другими системными журналами в каталоге /var/log/messages или где-либо еще;
- воспользоваться командой dmesg, не забыв при этом указать параметр less, поскольку результаты займут намного больше места, чем один дисплей. Команда dmesg использует циклический буфер ядра, размер которого ограничен, но в большинстве новых версий ядра буфер достаточно велик, чтобы хранить сообщения в течение длительного времени.

Вот пример того, что можно ожидать в результате работы команды dmesg:

```
$ dmesg
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 3.2.0-67-generic-pae (buildd@toyol) (gcc version 4.
6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5) ) #101-Ubuntu SMP Tue Jul 15 18:04:54 UTC
2014
 (Ubuntu 3.2.0-67.101-generic-pae 3.2.60)
[    0.000000] KERNEL supported cpus:
--snip--
[     2.986148] sr0: scsi3-mmc drive: 24x/8x writer dvd-ram cd/rw xa/form2 cdda
tray
[    2.986153] cdrom: Uniform CD-ROM driver Revision: 3.20
[    2.986316] sr 1:0:0:0: Attached scsi CD-ROM sr0
[    2.986416] sr 1:0:0:0: Attached scsi generic sg1 type 5
[    3.007862] sda: sda1 sda2 < sda5 >
[    3.008658] sd 0:0:0:0: [sda] Attached SCSI disk
--snip--
```

Процедура запуска пространства пользователя часто оставляет сообщения после запуска ядра. Эти сообщения будет сложнее увидеть и просмотреть, поскольку в большинстве систем они не находятся в одном файле журнала. Сценарии запуска, как правило, выводят собщения в консоль, а по завершении процесса загрузки они стираются. Тем не менее это не является проблемой, поскольку каждый сценарий обычно ведет собственный журнал. Определенные версии команды init, например Upstart и systemd, способны перехватывать диагностические сообщения процесса загрузки и обычной работы, которые в обычном случае отображаются в консоли.

## 5.2. Инициализация ядра и параметры загрузки

Во время запуска ядро системы Linux выполняет инициализацию в следующем порядке.

1. Проверка центрального процессора.
2. Проверка оперативной памяти.
3. Обнаружение шины устройств.
4. Обнаружение устройств.
5. Настройка вспомогательной подсистемы ядра (сеть и т. п.).
6. Монтирование корневой файловой системы.
7. Запуск пространства пользователя.

Первые шаги не слишком примечательны, но зато, когда ядро добирается до устройств, возникает вопрос о зависимостях. Например, драйверы дисковых устройств могут зависеть от поддержки шины и подсистемы SCSI.

Далее в ходе инициализации ядро должно смонтировать корневую файловую систему до запуска команды `init`. Как правило, вам не придется беспокоиться об этих процессах, исключая тот случай, когда необходимые компоненты являются загружаемыми модулями ядра, а не частями основного ядра. На некоторых компьютерах вам может потребоваться загрузить такие модули ядра до монтирования реальной корневой файловой системы. Мы рассмотрим этот вопрос и обходные пути его решения в разделе 6.8.

На момент написания книги ядро не выводит каких-либо специальных сообщений, когда оно готово к запуску первого пользовательского процесса. Тем не менее приведенные ниже сообщения об управлении памятью являются верным признаком того, что скоро вступит в игру пространство пользователя, так как именно сейчас ядро защищает собственную память от процессов из пространства пользователя:

```
Freeing unused kernel memory: 740k freed
Write protecting the kernel text: 5820k
Write protecting the kernel read-only data: 2376k
NX-protecting the kernel data: 4420k
```

Вы можете также увидеть сообщение о том, что в данный момент происходит монтирование корневой файловой системы.

ПРИМЕЧАНИЕ

Можете спокойно переходить к главе 6, чтобы узнать об особенностях запуска пространства пользователя и команде init, которую ядро запускает в качестве первого процесса. Далее в данной главе приводятся подробности запуска ядра.

## 5.3. Параметры ядра

При запуске ядра Linux загрузчик передает ему набор текстовых *параметров ядра*, которые говорят ядру о том, как оно должно быть запущено. Эти параметры описы-

вают различные варианты действий, такие как, например, величина выполняемого ядром диагностического вывода и варианты настроек, зависящие от драйверов устройств.

Параметры ядра при загрузке вашей системы можно увидеть в файле `/proc/cmdline`:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-3.2.0-67-generic-pae root=UUID=70ccd6e7-6ae6-44f6-
  812c-51aab8036d29 ro quiet splash vt.handoff=7
```

Эти параметры являются либо простыми однословными флагами, вроде `ro` и `quiet`, либо парами *key=value*, например `vt.handoff=7`. Многие из этих параметров не являются существенными, например флаг `splash`, отвечающий за отображение экрана загрузки. Действительно важным является параметр `root`. Он задает местоположение корневой файловой системы, без него ядро не может отыскать команду `init`, а следовательно, и выполнить запуск пространства пользователя.

Корневая файловая система может быть определена как файл устройства, например так:

```
root=/dev/sda1
```

Однако в большинстве современных ПК более распространенным является применение идентификаторов UUID (см. подраздел 4.2.4):

```
root=UUID=70ccd6e7-6ae6-44f6-812c-51aab8036d29
```

Параметр `ro` стандартен; он указывает ядру на то, что корневую файловую систему при запуске пространства пользователя следует монтировать в режиме «только чтение». Этот режим гарантирует возможность безопасной проверки корневой файловой системы с помощью команды `fsck`. По окончании проверки процесс загрузки выполняет повторное монтирование корневой файловой системы в режиме «чтение-запись».

При обнаружении какого-либо непонятного параметра ядро Linux сохраняет его, а затем передает команде `init` при выполнении запуска пространства пользователя. Например, если вы добавите к параметрам ядра флаг `-s`, оно передаст его команде `init`, и это будет означать, что запуск следует выполнить в режиме одиночного пользователя.

Теперь рассмотрим механику того, как загрузчики системы запускают ядро.

# 5.4. Загрузчики системы

В начале процесса загрузки, до запуска ядра и команды `init`, загрузчик системы запускает ядро. Задача загрузчика системы выглядит просто: он загружает ядро в память, а затем запускает его с определенными параметрами. Рассмотрим вопросы, на которые должен ответить загрузчик системы.

- Где находится ядро?
- Какие параметры ядра должны быть переданы ему при запуске?

Ответ (как правило) такой: ядро и его параметры обычно располагаются в корневой файловой системе. Кажется, что параметры ядра отыскать несложно, но ведь само ядро еще не запущено, поэтому оно не может «пройтись» по файловой системе в поисках необходимых файлов. Кроме того, драйверы устройств, которые ядро обычно использует для доступа к диску, также недоступны. Можно увидеть в этом нечто вроде проблемы «яйцо или курица».

Начнем разбираться с драйверами. На персональных компьютерах загрузчики системы используют систему BIOS или интерфейс UEFI, чтобы получить доступ к дискам. Практически все дисковые аппаратные средства снабжаются прошивками, которые позволяют системе BIOS получать доступ к присоединенным устройствам хранения с помощью *блочной адресации LBA* (Linear Block Addressing). Хотя производительность этого режима невысока, он все же позволяет получить универсальный доступ к дискам. Загрузчики системы часто являются единственными программами, которые используют систему BIOS для доступа к дискам; ядро применяет собственные высокопроизводительные драйверы.

Вопрос о файловой системе более сложен. Многие современные загрузчики системы способны читать таблицы разделов, а также имеют встроенную поддержку доступа к файловым системам в режиме «только чтение». Следовательно, они могут находить и читать файлы. Такая возможность значительно облегчает динамическую конфигурацию и улучшение загрузчика системы. Загрузчики системы Linux не всегда обладали этой возможностью; без нее настройка загрузчика была намного более сложной.

## 5.4.1. Задачи загрузчика системы

Основные функции загрузчика системы Linux содержат следующие возможности:

- выбор среди нескольких ядер;
- переключение между наборами параметров ядра;
- разрешение ручного переопределения и редактирование имен образов ядра и его параметров пользователем (например, чтобы войти в режим одиночного пользователя);
- поддержка загрузки других операционных систем.

Загрузчики системы стали более сложными по сравнению с начальными вариантами ядра Linux. Появились такие функции, как история и меню, однако главной потребностью всегда остается гибкость образа ядра и выбор его параметров. Интересным фактом является то, что некоторые возможности были ослаблены. Поскольку теперь вы можете, например, выполнять аварийную или восстановительную загрузку частично или полностью с USB-накопителя, вам, вероятно, не придется беспокоиться о том, чтобы вручную вводить параметры ядра или переходить в режим одиночного пользователя. К тому же современные загрузчики предлагают больше возможностей, чем когда-либо ранее. Это может быть чрезвычайно удобно, если вы разрабатываете пользовательское ядро или желаете настроить параметры.

## 5.4.2. Общий обзор загрузчиков системы

Приведем перечень основных загрузчиков системы, которые могут вам встретиться, в порядке их популярности:

- **GRUB** — почти универсальный стандарт для систем Linux;
- **LILO** — один из первых загрузчиков системы Linux. Версия ELILO предназначена для интерфейса UEFI;
- **SYSLINUX** — может быть настроен для запуска в множестве различных типов файловых систем;
- **LOADLIN** — загружает ядро из оболочки MS-DOS;
- **efilinux** — загрузчик с интерфейсом UEFI, призванный быть моделью и эталоном для других загрузчиков с интерфейсом UEFI;
- **coreboot** (ранее назывался **LinuxBIOS**) — высокопроизводительная замена системы BIOS в персональных компьютерах. Может содержать ядро;
- **Linux Kernel EFISTUB** — плагин ядра для загрузки напрямую из системного раздела ESP (EFI/UEFI System Partition), имеющегося в современных системах.

В этой книге речь пойдет исключительно о загрузчике GRUB. Основная причина для использования других загрузчиков системы заключается либо в более простом их конфигурировании, по сравнению с GRUB, либо в быстроте работы.

Чтобы ввести имя ядра и параметры, вам сначала необходимо узнать, как попасть в строку приглашения загрузчика. К сожалению, иногда это бывает сложно выяснить, поскольку различные версии Linux по-разному настраивают поведение загрузчика и его отображение.

В следующих разделах вы узнаете о том, как оказаться в строке приглашения загрузчика, чтобы ввести имя ядра и параметры. Когда вы освоитесь с этим, вы изучите настройку и установку загрузчика системы.

# 5.5. Первое знакомство с загрузчиком GRUB

Аббревиатура *GRUB* означает *Grand Unified Boot Loader (Большой унифицированный загрузчик системы)*. Мы рассмотрим версию GRUB 2; есть также более старая версия GRUB Legacy, использование которой постепенно сходит на нет.

Одной из наиболее важных возможностей загрузчика GRUB является навигация по файловой системе, которая позволяет намного проще выбирать ядро и настраивать параметры. Лучший способ увидеть это в действии и узнать в общих чертах о загрузчике GRUB — заглянуть в его меню. Интерфейс снабжен простой навигацией, однако есть большая вероятность того, что вы никогда его раньше не видели. Системы Linux изо всех сил стараются скрыть от вас свой загрузчик.

Чтобы получить доступ к меню загрузчика GRUB, нажмите клавишу Shift и удерживайте ее, когда появится начальный экран системы BIOS или прошивки. В противном случае процесс настройки загрузчика может не сделать паузы перед

загрузкой ядра. На рис. 5.1 показано меню загрузчика GRUB. Нажмите клавишу Esc, чтобы временно отключить автоматическое истечение времени ожидания загрузки после появления меню.

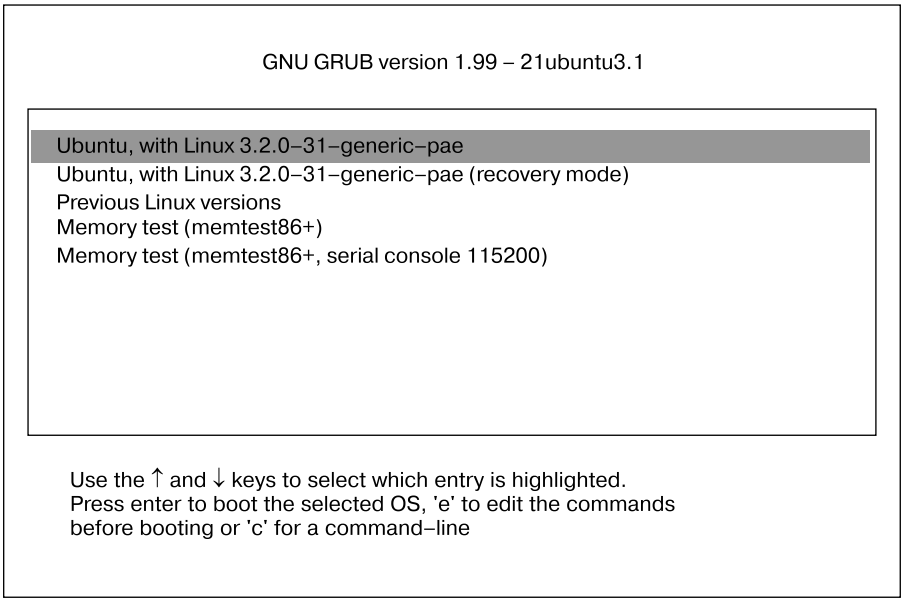


**Рис. 5.1.** Меню загрузчика GRUB

Попробуйте следующим образом исследовать загрузчик системы.

1. Перезагрузите или включите компьютер с Linux.
2. Удерживайте нажатой клавишу Shift при появлении начального экрана системы BIOS или самопроверки прошивки и/или экрана загрузки, чтобы попасть в меню загрузчика GRUB.
3. Нажмите клавишу E, чтобы увидеть команды конфигурирования загрузчика для принятого по умолчанию варианта загрузки. Должно появиться что-то вроде показанного на рис. 5.2.

Этот экран говорит нам о том, что для данной конфигурации корневой каталог установлен с помощью идентификатора UUID, образ ядра называется `/boot/vmlinuz-3.2.0-31-generic-pae`, а параметры ядра содержат `ro`, `quiet` и `splash`. Начальной файловой системой оперативной памяти является `/boot/initrd.img-3.2.0-31-generic-pae`. Если вы раньше никогда не встречали подобного рода конфигурацию, она может показаться немного запутанной. Почему несколько раз встречается параметр `root`, причем в разных вариантах? Почему здесь указан параметр `insmod`? Разве эта функция ядра не запускается обычно менеджером udevd?

Двойные записи оправданны, поскольку загрузчик GRUB в действительности не *использует* ядро Linux — он его *запускает*. Конфигурация, которую вы видите, полностью состоит из внутренних команд загрузчика GRUB. Этот загрузчик представляет собой совершенно особый мир.

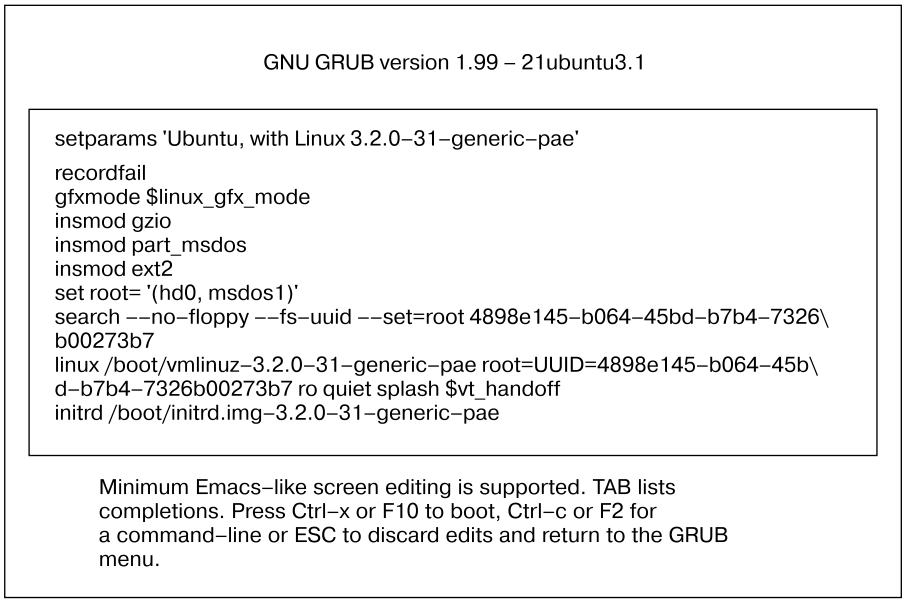


**Рис. 5.2.** Редактор конфигурации загрузчика GRUB

Путаница возникает вследствие того, что загрузчик GRUB заимствует терминологию из нескольких источников. У него есть собственное «ядро» и собственная команда `insmod`, чтобы динамически загружать модули GRUB, совершенно независимо от ядра Linux. Многие команды загрузчика GRUB похожи на команды оболочки Unix; есть даже команда `ls` для отображения списка файлов.

Больше всего путаницы возникает из-за применения слова `root`. Чтобы прояснить ситуацию, следуйте одному простому правилу при отыскании корневой файловой системы: *только* корневой параметр *ядра* будет корневой файловой системой при загрузке.

В приведенной конфигурации загрузчика GRUB такой корневой параметр располагается где-то после имени образа в команде `linux`. Любая другая ссылка на `root` в этой конфигурации относится к корневому каталогу загрузчика GRUB, который существует только внутри загрузчика. «Корень» загрузчика GRUB — это файловая система, в которой загрузчик отыскивает файлы образов ядра и файловой системы оперативной памяти.

На рис. 5.2 корневой каталог загрузчика GRUB сначала настроен на специфичное для загрузчика устройство (`hd0, msdos1`). В следующей команде загрузчик отыскивает конкретный идентификатор UUID для раздела. Если такой идентификатор находится, то корневой каталог загрузчика настраивается на этот раздел.

В завершение первый аргумент команды `linux (/boot/vmlinuz-...)` задает расположение файла образа ядра Linux. Загрузчик GRUB загружает этот файл из своего корневого каталога. Команда `initrd` действует подобным образом, отвечая за файл для начальной файловой системы оперативной памяти.

Эту конфигурацию можно редактировать внутри загрузчика GRUB; обычно это самый простой способ временно исправить ошибку при загрузке. Чтобы полностью устранить такую ошибку, вам потребуется изменить конфигурацию (см. подраздел 5.5.2), но сейчас копнем поглубже и исследуем некоторые внутренние части загрузчика GRUB с помощью интерфейса командной строки.

## 5.5.1. Выявление устройств и разделов с помощью командной строки загрузчика GRUB

Как можно заметить на рис. 5.2, загрузчик GRUB обладает собственной схемой адресации устройств. Например, первый обнаруженный жесткий диск назван `hd0`, следующий — `hd1` и т. д. Однако назначения устройств могут измениться. К счастью, загрузчик GRUB может выполнить поиск идентификаторов UUID во всех разделах, чтобы обнаружить именно тот, на котором расположено ядро, как вы только что убедились на примере команды `search`.

### Перечисление устройств

Чтобы получить представление о том, как загрузчик GRUB ссылается на устройства в вашей системе, получите доступ к командной строке загрузчика, нажав клавишу C в меню загрузки или в редакторе конфигурации. Вы должны увидеть такое приглашение загрузчика GRUB:

```
grub>
```

Здесь можно ввести любую команду, которую вы видите в конфигурации, но для начала попробуйте диагностическую команду `ls`. Если не указывать аргументы, результатом работы станет перечень устройств, известных загрузчику GRUB:

```
grub> ls
(hd0) (hd0,msdos1) (hd0,msdos5)
```

В данном случае присутствует одно основное дисковое устройство, обозначенное как `(hd0)`, а также разделы `(hd0,msdos1)` и `(hd0,msdos5)`. Префикс `msdos` в названиях разделов говорит о том, что такой диск содержит таблицу разделов MBR; для таблицы GBT название начиналось бы с префикса `gpt`. Могут также встретиться более сложные комбинации с третьим идентификатором, когда карта разметки диска располагается внутри раздела; но вам, как правило, не придется об этом беспокоиться, если вы не запускаете несколько операционных систем на одном компьютере.

Чтобы получить более детальную информацию, используйте команду `ls -l`. Эта команда может оказаться исключительно полезной, поскольку она отображает идентификаторы UUID для разделов диска. Например, так:

```
grub> ls -l
Device hd0: Not a known filesystem - Total size 426743808 sectors
       Partition hd0,msdos1: Filesystem type ext2 - Last modification time
         2015-09-18 20:45:00 Friday, UUID 4898e145-b064-45bd-b7b4-7326b00273b7 -
Partition start at 2048 - Total size 424644608 sectors
       Partition hd0,msdos5: Not a known filesystem - Partition start at
         424648704 - Total size 2093056 sectors
```

Данный конкретный диск имеет файловую систему Linux ext2/3/4 в первом разделе MBR, а также сигнатуру области подкачки в разделе 5, что является довольно распространенной конфигурацией. Хотя на основе этого вывода нельзя сказать, что раздел (hd0,msdos5) является областью подкачки.

## Файловая навигация

Рассмотрим навигационные возможности файловой системы загрузчика GRUB. Определите корневой каталог загрузчика с помощью команды echo (вспомните, что именно в нем загрузчик ожидает найти ядро системы):

```
grub> echo $root
hd0,msdos1
```

Чтобы использовать команду ls загрузчика GRUB для перечисления файлов и каталогов в этом корневом каталоге, следует снабдить ее после названия раздела символом косой черты:

```
grub> ls (hd0,msdos1)/
```

Запоминать и набирать полное имя корневого раздела неудобно, поэтому для экономии времени используйте переменную root:

```
grub> ls ($root)/
```

Результатом работы будет краткий перечень имен файлов и каталогов файловой системы данного раздела, например etc/, bin/ и dev/. Следует понимать, что теперь это совершенно другая функция ls загрузчика GRUB: до этого перечислялись устройства, таблицы разделов и, возможно, некоторая заголовочная информация о файловой системе. Теперь же вы на самом деле смотрите на содержимое файловых систем.

Подобным образом можно заглянуть чуть глубже в файлы и каталоги раздела. Например, чтобы исследовать каталог /boot, начните с такой команды:

```
grub> ls ($root)/boot
```

ПРИМЕЧАНИЕ

Используйте клавиши ↑ и ↓, чтобы перемещаться по истории команд загрузчика GRUB, а также клавиши ← и →, чтобы редактировать командную строку. Стандартные комбинации клавиш (Ctrl+N, Ctrl+P и т. п.) также применимы.

Можно просмотреть все установленные для загрузчика переменные с помощью команды set:

```
grub> set
?=0
color_highlight=black/white
color_normal=white/black
--snip--
prefix=(hd0,msdos1)/boot/grub
root=hd0,msdos1
```

Одной из самых важных переменных является $prefix, которая определяет файловую систему и каталог, в котором загрузчик GRUB ожидает отыскать свою

конфигурацию и дополнительную поддержку. Об этом пойдет речь в следующем разделе.

По окончании работы с командной строкой загрузчика GRUB введите команду `boot`, чтобы загрузить текущую конфигурацию, или нажмите клавишу Esc для возврата в меню загрузчика. В любом случае загрузите систему: мы собираемся исследовать конфигурацию загрузчика GRUB, а это лучше всего делать тогда, когда система доступна полностью.

## 5.5.2. Конфигурация загрузчика GRUB

Конфигурационный каталог загрузчика GRUB содержит центральный файл конфигурации (`grub.cfg`) и многочисленные загружаемые модули с суффиксами `.mod`. С выходом новых версий загрузчика GRUB такие модули будут перемещены в подкаталоги, например `i386-pc`. Каталог, как правило, называется `/boot/grub` или `/boot/grub2`. Мы не будем напрямую изменять файл `grub.cfg`, вместо этого воспользуемся командой `grub-mkconfig` или командой `grub2-mkconfig` в Fedora.

### Обзор файла Grub.cfg

Сначала бегло просмотрите файл `grub.cfg`, чтобы понять, как загрузчик GRUB инициализирует свое меню и параметры ядра. Вы увидите, что файл `grub.cfg` состоит из команд загрузчика GRUB, которые обычно начинаются с нескольких шагов инициализации, а затем продолжаются рядом пунктов меню для различных конфигураций ядра и загрузки. Инициализация не является сложной: это набор определений функций и команд настройки видео, вроде этих:

```
if loadfont /usr/share/grub/unicode.pf2 ; then
  set gfxmode=auto
  load_video
  insmod gfxterm
  --snip--
```

Далее в этом файле вы должны увидеть доступные конфигурации загрузки, каждая из которых начинается с команды `menuentry`. Вы должны быть способны прочитать и понять данный пример, основываясь на том, что вы узнали из предыдущего раздела:

```
menuentry 'Ubuntu, with Linux 3.2.0-34-generic-pae' --class ubuntu --class gnu-linux
--class gnu
--class os {
        recordfail
        gfxmode $linux_gfx_mode
        insmod gzio
        insmod part_msdos
        insmod ext2
        set root='(hd0,msdos1)'
           search --no-floppy --fs-uuid --set=root 70ccd6e7-6ae6-44f6-812c-
51aab8036d29
         linux   /boot/vmlinuz-3.2.0-34-generic-pae root=UUID=70ccd6e7-6ae6-44f6-
812c-51aab8036d29
```

```
        ro   quiet splash $vt_handoff
        initrd  /boot/initrd.img-3.2.0-34-generic-pae
}
```

Присмотритесь к командам submenu. Если ваш файл grub.cfg содержит множество команд menuentry, большинство из них, вероятно, свернуто внутри команды submenu. В старых версиях ядра это сделано для того, чтобы они не переполняли меню загрузчика GRUB.

## Создание нового файла конфигурации

Если вы желаете внести изменения в конфигурацию загрузчика GRUB, не следует редактировать файл grub.cfg напрямую, поскольку он создается автоматически и система время от времени его перезаписывает. Необходимо дополнить конфигурацию где-либо в другом месте, а затем запустить команду grub-mkconfig, чтобы создать новый файл.

Чтобы понять, как происходит создание конфигурации, посмотрите в самое начало файла grub.cfg. Там должны быть строки с комментариями, например такими:

```
### BEGIN /etc/grub.d/00_header ###
```

Продолжая осмотр, вы обнаружите, что каждый файл в каталоге /etc/grub.d является сценарием оболочки, который производит фрагмент файла grub.cfg. Да и сама команда grub-mkconfig — это сценарий оболочки, который запускает все, что находится в каталоге /etc/grub.d.

Попробуйте запустить ее самостоятельно с правами корневого пользователя. Не беспокойтесь о том, что ваша текущая конфигурация будет перезаписана: данная команда всего лишь отображает конфигурацию в стандартном выводе.

```
# grub-mkconfig
```

Как быть, если вам необходимо добавить пункты меню и другие команды в конфигурацию загрузчика GRUB? Если отвечать кратко — следует указать необходимые вам настройки в новом файле custom.cfg и поместить его в каталог с конфигурацией загрузчика, например /boot/grub/custom.cfg.

Полный ответ выглядит немного сложнее. Конфигурационный каталог /etc/grub.d предлагает вам на выбор два варианта: 40_custom и 41_custom. Первый, 40_custom, является сценарием, который вы можете редактировать самостоятельно, но это, вероятно, наименее надежно: обновление пакета удалит все внесенные вами изменения. Сценарий 41_custom проще. Это всего лишь последовательность команд, загружающих файл custom.cfg при запуске загрузчика GRUB. Имейте в виду, что, если вы выберете второй вариант, изменения не отображаются, когда вы создаете файл конфигурации.

Эти варианты для создания пользовательского файла конфигурации не исчерпывают все возможности. Дополнения вы найдете в каталоге /etc/grub.d вашей версии системы. Так, например, Ubuntu добавляет в конфигурацию загрузки параметры для проверки памяти (memtest86+).

Чтобы записать и установить вновь созданный файл конфигурации GRUB, можно записать этот файл в каталог загрузчика GRUB, указав флаг -o в команде grub-mkconfig:

```
# grub-mkconfig -o /boot/grub/grub.cfg
```

Если же вы работаете в Ubuntu, просто запустите команду `install-grub`. В любом случае создайте резервную копию предыдущей конфигурации, убедитесь, что установка производится в правильный каталог, и продолжайте.

Теперь мы приступаем к рассмотрению дополнительных подробностей загрузчика GRUB и загрузчиков системы. Если вы устали от загрузчиков и от ядра, можете спокойно переходить к главе 6.

## 5.5.3. Установка загрузчика GRUB

Установка загрузчика GRUB является более сложной по сравнению с его настройкой. К счастью, вам, как правило, не придется об этом беспокоиться, поскольку ваше ПО должно выполнить это за вас. Однако если вы пытаетесь продублировать или восстановить загружаемый диск, а также подготавливаете собственную последовательность загрузки, вам может потребоваться установить загрузчик самостоятельно.

Прежде чем продолжать чтение, загляните в подраздел 5.8.3, чтобы получить представление о том, как загружается компьютер, и определить, какой тип загрузки вы применяете: MBR или EFI. Затем соберите ПО загрузчика GRUB и определите, где будет расположен каталог загрузчика; по умолчанию используется путь `/boot/grub`. Нет необходимости выполнять сборку загрузчика GRUB, если ваш дистрибутив делает это за вас. Если вы займетесь этим, загляните в главу 16, чтобы узнать о том, как выполняется сборка ПО на основе исходного программного кода. Убедитесь в том, что вы собираете верный целевой объект: он различен для загрузок MBR и UEFI (имеются также различия для 32- и 64-битного вариантов EFI).

### Установка загрузчика GRUB в систему

При установке загрузчика системы необходимо, чтобы вы или установщик определили следующее.

- Целевой каталог загрузчика GRUB — каким его видит работающая в данный момент система. Обычно это каталог `/boot/grub`, но он может быть и другим, если вы устанавливаете загрузчик GRUB на другой диск или в другую операционную систему.
- Текущее устройство для целевого диска загрузчика GRUB.
- Для загрузки UEFI — текущую точку монтирования загрузочного раздела UEFI.

Помните о том, что загрузчик GRUB является модульной системой, но для загрузки модулей он должен читать файловую систему, которая содержит каталог загрузчика GRUB. Ваша задача заключается в создании версии загрузчика GRUB, которая способна читать эту файловую систему, чтобы иметь возможность загрузки остальной части своей конфигурации (файл `grub.cfg`) и любых необходимых модулей. В Linux это обычно означает, что нужно собрать версию загрузчика GRUB с предварительно загруженным модулем `ext2.mod`. Как только у вас будет такая версия, вам потребуется лишь поместить ее в загружаемую часть диска, а остальные необходимые файлы — в каталог `/boot/grub`.

Загрузчик GRUB снабжен утилитой под названием `grub-install` (не смешивайте с командой `install-grub` в Ubuntu), которая выполняет за вас основную часть работы по установке файлов загрузчика GRUB и его конфигурации. Например, если ваш текущий диск расположен в `/dev/sda` и вы желаете установить загрузчик GRUB на этот диск с текущим каталогом `/boot/grub`, используйте такую команду установки загрузчика GRUB для таблицы MBR:

```
# grub-install /dev/sda
```

ВНИМАНИЕ

Неправильная установка загрузчика GRUB может нарушить последовательность загрузки вашей системы, поэтому не относитесь к этой команде легкомысленно. Если же такое произошло, прочитайте о том, как создать резервную копию MBR с помощью команды dd, сделайте резервную копию любого другого установленного в данный момент каталога GRUB и убедитесь в том, что у вас есть план аварийной загрузки.

## Установка загрузчика GRUB на внешнем устройстве хранения данных

Чтобы установить загрузчик GRUB на устройстве хранения вне текущей системы, необходимо вручную указать каталог загрузчика на таком устройстве, каким его видит ваша система. Допустим, у вас есть целевое устройство `/dev/sdc`, а корневая/загрузочная файловая система этого устройства (например, `/dev/sdc1`) смонтирована в каталоге `/mnt` вашей нынешней системы. Это подразумевает, что при установке загрузчика GRUB ваша система будет видеть файлы загрузчика в каталоге `/mnt/boot/grub`. При запуске команды `grub-install` сообщите ей, где расположены эти файлы, таким образом:

```
# grub-install --boot-directory=/mnt/boot /dev/sdc
```

## Установка загрузчика GRUB в интерфейсе UEFI

Предполагается, что установка в интерфейсе UEFI происходит проще, поскольку вам потребуется лишь скопировать загрузчик системы на место. Помимо этого, необходимо также «известить» прошивку о загрузчике системы с помощью команды `efibootmgr`. Если команда `grub-install` доступна, она выполнит это, поэтому теоретически для установки загрузчика в раздел с интерфейсом UEFI вам необходимо запустить такую команду:

```
# grub-install --efi-directory=efi_dir --bootloader-id=name
```

Здесь параметр `efi_dir` задает расположение каталога UEFI, как он виден вашей системе (обычно это `/boot/efi/efi`, поскольку раздел UEFI часто монтируется в точке `/boot/efi`), а параметр `name` является идентификатором загрузчика системы, как рассказано в разделе 5.8.2. Загрузка в интерфейсе UEFI.

К сожалению, при установке загрузчика в интерфейсе UEFI может возникнуть множество проблем. Например, если вы устанавливаете его на диск, который в итоге окажется в другой системе, вы должны выяснить, как объявить данный загрузчик прошивке новой системы. Есть также отличия в процедуре установки для сменных накопителей.

Одной из самых больших проблем является безопасная загрузка интерфейса UEFI.

## 5.6. Проблемы с безопасной загрузкой UEFI

Одной из новейших проблем, влияющих на установку систем Linux, является функция безопасной загрузки, которая присутствует на современных компьютерах. Если она активна, то в интерфейсе UEFI данный механизм требует, чтобы загрузчики системы для возможности своего запуска были снабжены надежной цифровой подписью. Компания Microsoft требует, чтобы поставщики системы Windows 8 использовали безопасную загрузку. В результате, если вы попытаетесь установить неподписанный загрузчик системы (а в большинстве современных версий Linux это именно так), он не станет работать.

Простейший обходной путь для тех, кого не интересует Windows, состоит в отключении безопасной загрузки в настройках интерфейса EFI. Однако это не будет четко работать в системах с двухвариантной загрузкой, а также вряд ли устроит всех пользователей. По этой причине в дистрибутивах Linux предлагаются подписанные загрузчики системы. Одни представляют собой всего лишь внешний интерфейс к загрузчику GRUB, другие же предлагают полностью подписанную загрузочную последовательность (от загрузчика системы до ядра); есть также абсолютно новые загрузчики (некоторые основаны на интерфейсе efilinux).

## 5.7. Передача управления загрузчикам других операционных систем

Интерфейс UEFI позволяет сравнительно легко осуществить поддержку загрузки других операционных систем, поскольку в разделе EFI можно установить несколько загрузчиков системы. Тем не менее «старый стиль» MBR не поддерживает это, и даже при наличии интерфейса UEFI вам может потребоваться отдельный раздел с загрузчиком системы, который использует MBR. Можно сделать так, чтобы загрузчик GRUB загрузил и запустил другой загрузчик системы в определенном разделе диска с помощью *передачи управления загрузчику*.

Для передачи управления создайте новый пункт меню в конфигурации загрузчика GRUB (используя один из методов пункта «Обзор файла Grub.cfg» в подразделе 5.5.2). Вот пример для установленной в третьем разделе диска Windows:

```
menuentry "Windows" {
        insmod chain
        insmod ntfs
        set root=(hd0,3)
        chainloader +1
}
```

Параметр `+1` в строке `chainloader` сообщает загрузчику, чтобы он загрузил то, что находится в первом секторе раздела. Можно также напрямую указать загрузку

файла, использовав строку, подобную приведенной ниже для загрузчика `io.sys` системы MS-DOS:

```
menuentry "DOS" {
        insmod chain
        insmod fat
        set root=(hd0,3)
        chainloader /io.sys
}
```

# 5.8. Детали загрузчика системы

Теперь мы рассмотрим некоторые внутренние части загрузчика системы. Можете спокойно переходить к следующей главе, если данный материал вам неинтересен.

Чтобы понять, как работают загрузчики системы, подобные GRUB, сначала исследуем, как загружается персональный компьютер, когда вы его включаете. Вследствие неадекватности традиционных механизмов загрузки ПК есть несколько вариантов, однако основных схем две: MBR и UEFI.

## 5.8.1. Загрузка с применением таблицы MBR

В дополнение к информации о разделе, которая описана в разделе 4.1, таблица *MBR* (Master Boot Record, *главная загрузочная запись*) содержит небольшую область (441 байт), которую система BIOS выполняет по завершении проверки *POST* (Power-On Self-Test, *самотестирование при включении питания*). К сожалению, эта область слишком мала, чтобы разместить загрузчик системы, поэтому необходимо дополнительное пространство. В результате возникает то, что иногда называют *многоэтапным загрузчиком системы*. В этом случае начальный фрагмент кода в таблице MBR не делает ничего, кроме загрузки оставшейся части кода загрузчика системы. Эти фрагменты загрузчика обычно помещаются в область между таблицей MBR и первым разделом диска.

Это не слишком безопасно, поскольку данный код может перезаписать кто угодно. Однако большинство загрузчиков, включая версии GRUB, поступают именно так. Более того, эта схема не будет работать для диска с разделами GPT при использовании системы BIOS для загрузки, поскольку информация таблицы GPT размещена в области после таблицы MBR. Таблица GPT не затрагивает традиционную таблицу MBR в целях обратной совместимости.

Обходной путь для таблицы GPT состоит в создании небольшого *загрузочного раздела BIOS* со специальным идентификатором UUID, чтобы предоставить место для размещения полного кода загрузчика системы. Однако таблица GPT обычно используется с интерфейсом UEFI, а не с традиционной системой BIOS, это приводит нас к схеме загрузки с применением интерфейса UEFI.

## 5.8.2. Загрузка с применением интерфейса UEFI

Производители компьютеров и компании, занимающиеся разработкой ПО, осознали, что традиционная система BIOS является весьма ограниченной, поэтому они

решили разработать замену под названием *расширенный интерфейс прошивки* (*EFI*, Extensible Firmware Interface). Этому интерфейсу потребовалось некоторое время, чтобы прижиться в ПК, а теперь он достаточно распространен. Текущим стандартом является *унифицированный интерфейс UEFI* (Unified EFI), содержащий такие функции, как встроенная оболочка, а также возможность чтения таблиц разделов и навигация по файловым системам. Схема GPT является частью стандарта UEFI.

Загрузка систем с интерфейсом UEFI происходит совершенно иначе, и понять ее гораздо проще. Вместо исполняемого кода загрузки, который расположен вне файловой системы, здесь всегда присутствует специальная файловая система под названием *«системный раздел EFI»* (*ESP*, EFI System Partition), содержащий каталог `efi`. Каждый загрузчик системы обладает идентификатором и соответствующим подкаталогом, например `efi/microsoft`, `efi/apple` или `efi/grub`. Файл загрузчика системы снабжен расширением `.efi` и располагается в одном из таких подкаталогов вместе с дополнительными файлами поддержки.

ПРИМЕЧАНИЕ

Раздел ESP отличается от загрузочного раздела BIOS, описанного в подразделе 5.8.1, и обладает другим идентификатором UUID.

Однако есть здесь некоторая тонкость: нельзя просто взять и поместить код старого загрузчика системы в раздел ESP, поскольку этот код был написан для интерфейса BIOS. Вместо этого вы должны предоставить загрузчик системы, написанный для интерфейса UEFI. Например, при использовании загрузчика GRUB вы должны установить UEFI-версию загрузчика GRUB, а не BIOS-версию. Помимо этого, следует «известить» прошивку о новых загрузчиках системы.

Как отмечалось в разделе 5.6, есть сложности с «безопасной загрузкой».

## 5.8.3. Как работает загрузчик GRUB

Подведем итог нашему обзору загрузчика GRUB, рассмотрев то, как он выполняет свою работу.

1. Система BIOS или прошивка ПК инициализирует аппаратные средства и выполняет поиск кода загрузки в устройствах хранения в указанной загрузочной последовательности.
2. Обнаружив код загрузки, система BIOS или прошивка загружают и исполняют его. Именно здесь в дело вступает загрузчик GRUB.
3. Загружается ядро загрузчика GRUB.
4. Происходит инициализация ядра. К этому моменту загрузчик GRUB получает доступ к дискам и файловым системам.
5. Загрузчик GRUB идентифицирует свой загрузочный раздел и загружает в него конфигурацию.
6. Загрузчик GRUB дает пользователю возможность изменить конфигурацию.
7. По истечении времени ожидания или после ответных действий пользователя загрузчик GRUB выполняет конфигурирование (последовательность команд, о которых говорилось в подразделе 5.5.2).

8. В процессе конфигурирования загрузчик GRUB может загрузить дополнительный код (модули) в загрузочный раздел.
9. Загрузчик GRUB исполняет команду `boot`, чтобы загрузить и исполнить ядро, как определено командой `linux` в конфигурации.

Шаги 3 и 4 в приведенной схеме, когда загружается ядро загрузчика GRUB, могут оказаться сложнее вследствие повторяющейся неадекватности традиционных механизмов загрузки ПК. Самый главный вопрос: где расположено ядро загрузчика GRUB? Существует три основные возможности:

- по частям размещено в таблице MBR и перед началом первого раздела;
- в обычном разделе;
- в специальном загрузочном разделе: загрузочном разделе GPT, системном разделе EFI (ESP) или еще где-либо.

Во всех случаях, исключая наличие раздела ESP, система BIOS загружает 512 байт из таблицы MBR, и именно здесь начинается загрузчик GRUB. Этот небольшой фрагмент (отщепленный от файла `boot.img` из каталога загрузчика GRUB) еще не является ядром, но он содержит исходную точку для ядра, с которой начинается его загрузка.

Однако при наличии раздела ESP ядро загрузчика GRUB присутствует здесь в виде файла. Прошивка может просмотреть раздел ESP и напрямую исполнить ядро загрузчика GRUB или загрузчика любой другой операционной системы, размещенного здесь.

Для большинства систем эта картина не является полной. Загрузчику системы может также понадобиться загрузить начальный образ файловой системы оперативной памяти до загрузки и исполнения ядра. Именно это определяет конфигурационный параметр `initrd` в разделе 6.8. Прежде чем вы узнаете о начальной файловой системе оперативной памяти, вам необходимо изучить запуск пространства пользователя. Именно с этого начинается следующая глава.

# 6 Как запускается пространство пользователя

Момент, когда ядро запускает первый процесс из пространства пользователя, `init`, важен не только потому, что именно сейчас оперативная память и центральный процессор окончательно готовы к нормальной работе, но и потому, что здесь вы можете увидеть, каким образом остальная часть системы выстраивается как целое. До этого момента ядро исполняет хорошо контролируемую последовательность действий, которая определена сравнительно малым количеством разработчиков ПО. Пространство пользователя в гораздо большей степени обладает модульной структурой. Здесь намного проще увидеть, что происходит в пространстве пользователя при его запуске и работе. Смелым пользователям также довольно просто изменить настройки запуска пространства пользователя, поскольку для этого не требуются навыки программирования на низком уровне.

В общих чертах, пространство пользователя запускается следующим образом.

1. Команда `init`.
2. Важнейшие низкоуровневые службы, такие как `udevd` и `syslogd`.
3. Сетевая конфигурация.
4. Службы среднего и высокого уровня (крон, печать и т. д.).
5. Приглашение ко входу в систему, пользовательский интерфейс и другие приложения высокого уровня.

## 6.1. Знакомство с командой init

Команда `init` — это команда из пространства пользователя, подобная любой другой команде системы Linux. Ее можно найти в каталоге `/sbin` вместе с другими системными исполняемыми файлами. Основное назначение этой команды — запуск и останов важнейших служебных процессов системы, хотя новые версии обладают дополнительными возможностями.

В дистрибутивах Linux существует три основные реализации команды `init`.

- **System V init.** Обычная последовательная команда `init` (`Sys V`, обычно произносится sys-five). Версия Red Hat Enterprise Linux и некоторые другие используют этот вариант.

- **systemd.** Набирающий силу стандарт команды `init`. Во многих дистрибутивах осуществлен переход на команду `systemd`, а в тех, где это еще не сделано, такой переход планируется.
- **Upstart.** Команда `init` для версий Ubuntu. Тем не менее к моменту написания книги в Ubuntu также запланирован переход на команду `systemd`.

Существуют также различные другие версии команды `init`, в особенности на встроенных платформах. Например, платформа Android обладает собственной командой `init`. В BSD также есть собственная версия команды `init`, но вряд ли вы встретите ее на современном компьютере с Linux. В некоторых дистрибутивах конфигурация команды `System V init` изменена, чтобы напоминать стиль BSD.

Различные реализации команды `init` существуют потому, что команда `System V init` и более ранние версии основаны на последовательности, в которой в один момент времени выполняется только одна задача запуска. Такая схема довольно легко позволяет разбираться с зависимыми процессами, однако производительность при этом не впечатляюще хороша, поскольку две части последовательности загрузки обычно не могут работать одновременно. Еще одним ограничением является возможность запуска лишь небольшого набора служб, определенных в последовательности загрузки: когда вы подключаете новое аппаратное средство или вам необходима еще не запущенная служба, не существует стандартизованного способа координации новых компонентов с командой `init`. Команды `systemd` и `Upstart` пытаются улучшить производительность, позволяя параллельный запуск нескольких служб, тем самым ускоряя процесс загрузки. Однако реализованы они совершенно по-разному.

- Команда `systemd` является ориентированной на цель. Вы определяете цель, которую необходимо достичь, а также зависящие от нее процессы и момент, когда цель должна быть достигнута. Команда `systemd` удовлетворяет все зависящие процессы и выполняет цель. Эта команда может также отложить запуск какой-либо службы, если она не является абсолютно необходимой.
- Команда `Upstart` выполняет ответную реакцию. Она реагирует на события и, основываясь на них, запускает задачи, которые, в свою очередь, порождают новые события, побуждающие команду `Upstart` запускать дополнительные задачи, и т. д.

Команды `systemd` и `Upstart` предлагают также более развитые способы запуска и отслеживания служб. В традиционных вариантах команды `init` ожидается, что демоны служб запускаются сами из сценариев. Сценарий запускает команду-демон, которая отделяется от сценария и работает автономно. Чтобы выяснить идентификатор PID для демона службы, необходимо использовать команду `ps` или какой-либо дугой способ, специфичный для данной службы. И напротив, команды `Upstart` и `systemd` могут изначально управлять отдельными демонами служб, предоставляя пользователю больше возможностей и точное понимание происходящего в системе.

Поскольку новые варианты команды `init` не завязаны на сценариях, конфигурирование служб для них также производится проще. В частности, сценарии команды `System V init` обычно содержат множество похожих команд, предназначенных для запуска, останова и повторного запуска служб. Подобная избыточность не понадобится вам в командах `systemd` и `Upstart`, которые позволяют сконцентрироваться на самих службах, а не на их сценариях.

Наконец, обе команды, systemd и Upstart, предлагают некоторую степень использования служб по запросу. Вместо того чтобы запускать все службы, которые могут понадобиться при загрузке системы (как сделала бы команда System V init), эти команды запускают службы только по мере необходимости. Эта идея не нова, она использована в традиционном демоне inetd, однако новые реализации являются более развитыми.

Команды systemd и Upstart предлагают некоторую долю обратной совместимости с командой System V. Например, обе поддерживают концепцию уровней запуска.

# 6.2. Уровни запуска команды System V

В каждый конкретный момент времени в системе Linux задействован некоторый набор процессов (таких как crond и udevd). Для команды System V init такое состояние компьютера называется *уровнем запуска* и обозначается числом от 0 до 6. Основную часть времени система проводит на одном уровне запуска, но когда вы выключаете компьютер, команда init переключается на другой уровень запуска, чтобы должным образом остановить системные службы и дать ядру команду останова.

Узнать уровень запуска вашей системы можно с помощью команды who -r. Система, в которой запущена команда Upstart, ответит чем-либо подобным:

```
$ who -r
run-level 2  2015-09-06 08:37
```

Этот результат сообщает нам о том, что текущий уровень запуска равен 2. Указаны также дата и время перехода на этот уровень запуска.

Уровни запуска служат различным целям, наиболее часто они используются для различения состояний запуска и выключения системы, режима одиночного пользователя и консоли. Так, например, системы семейства Fedora традиционно используют уровни запуска с 2 по 4 для текстовой консоли; уровень запуска 5 означает, что будет загружен графический интерфейс пользователя для входа в систему.

Однако уровни запуска постепенно уходят в прошлое. Несмотря на то что все три версии команды init, о которых рассказано в этой книге, поддерживают эти уровни, команды systemd и Upstart считают уровни запуска устаревшими для использования в качестве определяющих состояний системы. Для команд systemd и Upstart уровни запуска существуют главным образом для запуска служб, которые поддерживают только сценарии версии System V init. К тому же реализация уровней настолько различна, что даже если вы хорошо знакомы с одним из типов команды init, то это совсем не значит, что вы знаете, как работать с другим.

# 6.3. Определяем тип команды init

Прежде чем продолжить, вам необходимо определить, какая версия команды init используется в вашей системе. Если вы не вполне уверены, выполните следующие проверки.

- Если в вашей системе есть каталоги /usr/lib/systemd и /etc/systemd, то вы пользуетесь командой systemd. Переходите к разделу 6.4.
- Если каталог /etc/init содержит несколько файлов с расширением .conf, вы, вероятно, работаете с командой Upstart (за исключением Debian 7: в этом случае у вас, видимо, команда System V init). Переходите к разделу 6.5.
- Если ни один из приведенных вариантов не подходит, однако есть файл /etc/inittab, вероятно, вы используете команду System V init. Переходите к разделу 6.6.

Если в вашей системе установлено руководство, просмотр страницы init(8) должен помочь вам при определении версии команды init.

# 6.4. Команда systemd

Версия systemd команды init — одна из новейших реализаций команды init для Linux. Помимо работы с обычным процессом загрузки, команда systemd призвана также включить некоторые стандартные службы Unix, такие как cron и inetd. В этом можно заметить некоторое влияние команды launchd компьютеров Apple. Одной из ее самых существенных функций является возможность отложить запуск служб и некоторых функций операционной системы до тех пор, пока они не понадобятся.

У команды systemd так много функций, что довольно трудно понять, с чего начать изучение основ. В общих чертах опишу, что происходит, когда во время загрузки системы запускается команда systemd.

1. Команда systemd загружает свою конфигурацию.
2. Команда systemd определяет цель загрузки, которая обычно называется default.target.
3. Команда systemd определяет все зависимости для цели загрузки по умолчанию, зависимости зависимостей и т. д.
4. Команда systemd активизирует зависимые процессы и цель загрузки.
5. После загрузки команда systemd может реагировать на системные события (такие как uevents) и активизировать дополнительные компоненты.

При запуске служб команда systemd не придерживается жесткой последовательности. Как и у других современных версий команды init, процесс загрузки с помощью команды systemd довольно гибок. В большинстве вариантов конфигурации команды systemd намеренно пытаются избегать какой бы то ни было стартовой последовательности, предпочитая использовать другие методы для устранения жестких зависимостей.

## 6.4.1. Модули и типы модулей

Одним из самых интересных свойств команды systemd является то, что она не только управляет процессами и службами, но способна также монтировать файловые системы, отслеживать сетевые сокеты, запускать таймеры и многое другое. Каждый тип таких возможностей называется *типом модуля*, а каждая конкретная способность — *модулем*. Когда вы задействуете какой-либо модуль, вы *активизируете* его.

Вместо того чтобы описывать все типы модулей (вы найдете их на странице руководства systemd(1)), рассмотрим лишь некоторые из них, которые выполняют задачи запуска в любой системе Unix.

- **Модули служб.** Контролируют традиционные демоны служб в системе Unix.
- **Модули монтирования.** Контролируют присоединение файловых систем.
- **Целевые модули.** Контролируют другие модули, как правило группируя их.

По умолчанию целью загрузки обычно является целевой модуль, который группирует несколько служб и монтирует модули в качестве зависимостей. В результате довольно легко получить частичное представление о том, что происходит при загрузке системы. Можно даже составить дерево зависимостей с помощью команды `systemctl dot`. Вы обнаружите, что это дерево достаточно обширное, поскольку многие модули не запускаются по умолчанию.

На рис. 6.1 приведена часть дерева зависимостей для модуля `default.target`, который можно найти в системе Fedora. При активизации этого модуля все расположенные под ним модули также активизируются.

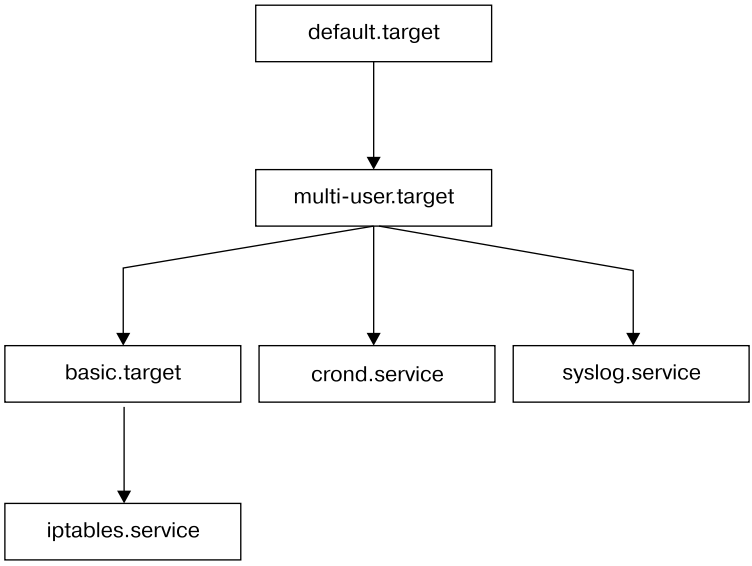


**Рис. 6.1.** Дерево зависимостей модуля

## 6.4.2. Зависимости команды systemd

Зависимости при загрузке системы и ее работе гораздо сложнее, чем могут показаться на первый взгляд, поскольку жесткие зависимости не допускают изменений. Например, вы решили, что приглашение ко входу в систему будет отображаться после запуска сервера базы данных, поэтому вы определяете зависимость приглашения от сервера базы данных. Однако если с сервером базы данных случится сбой, то в результате указанной зависимости приглашение ко входу в си-

стему также не появится, и вы не сможете даже войти в систему, чтобы исправить ошибку.

Задачи запуска системы Unix довольно терпимо относятся к ошибкам и зачастую сбои в задачах не вызывают серьезных проблем в стандартных службах. Например, если диск с данными для системы будет удален, но запись о нем в таблице `/etc/fstab` останется, то начальное монтирование файловой системы не осуществится. Однако такой сбой обычно не вызовет серьезных последствий в стандартной работе системы.

Чтобы приспособиться к требованиям гибкости и устойчивости к сбоям, команда `systemd` предлагает огромное количество типов и стилей зависимостей. Обозначим их по ключевым словам синтаксиса, но подробно этот синтаксис будет рассмотрен в подразделе 6.4.3. Посмотрим сначала на главные типы.

- **Requires.** Жесткие зависимости. При активизации модуля с зависимостью Requires команда `systemd` пытается активизировать модуль зависимости. Если модуль зависимости дает сбой, то команда `systemd` деактивизирует зависимый модуль.
- **Wants.** Зависимости, предназначенные только для активизации. Во время активизации какого-либо модуля команда `systemd` активизирует его Wants-зависимости, но не обращает внимания, если они дают сбой.
- **Requisite.** Модули, которые уже должны быть активными. Перед активизацией модуля с зависимостью Requisite команда `systemd` сначала проверяет состояние зависимости. Если такая зависимость еще не была активизирована, команда `systemd` дает сбой при активизации модуля с этой зависимостью.
- **Conflicts.** Противоположные зависимости. При активизации модуля с зависимостью Conflict команда `systemd` автоматически деактивизирует такую зависимость, если она активна. Одновременная активизация двух конфликтующих модулей вызовет сбой.

ПРИМЕЧАНИЕ

Тип зависимостей Wants особенно важен, поскольку он не распространяет ошибки на другие модули. В документации к команде systemd заявлено, что именно таким образом следует по возможности определять зависимости. Это позволит создать гораздо более устойчивую систему, подобную той, которая применяет традиционную команду init.

Зависимости могут быть также подключены «реверсивно». Например, чтобы добавить модуль А в качестве Wants-зависимости для модуля Б, не обязательно добавлять зависимость Wants в конфигурацию модуля Б. Вместо этого можно указать его как WantedBy-зависимость в конфигурации модуля А. То же самое верно и для зависимости RequiredBy. Конфигурация (а также результат) зависимости “By” немного сложнее, чем простое редактирование файла с настройками (см. пункт «Подключение модулей и секция [Install]» в подразделе 6.4.3).

Увидеть зависимости какого-либо модуля можно с помощью команды `systemctl`. С ее помощью можно также указать тип зависимости, например Wants или Requires:

```
# systemctl show -p type unit
```

### Порядок следования

Ни один из вариантов синтаксиса, который вы видели, не определяет явным образом порядок следования модулей. По умолчанию активизация модуля с зависимостью Requires или Wants ведет к тому, что команда `systemd` активизирует одновременно все такие зависимости в качестве первого модуля. Это оптимально, поскольку необходимо запустить по возможности максимальное количество служб, причем максимально быстро, чтобы сократить время загрузки системы. Однако бывают ситуации, когда один модуль должен быть запущен после другого. Например, в системе, которая изображена на рис. 6.1, настроен запуск модуля `default.target` после модуля `multi-user.service` (этот порядок следования не отображен на иллюстрации).

Чтобы активизировать модули в определенном порядке, можно использовать следующие модификаторы зависимостей.

- **Before.** Текущий модуль будет активизирован до указанного модуля или модулей. Например, если в модуле `foo.target` будет инструкция `Before=bar.target`, команда `systemd` активизирует модуль `foo.target` перед модулем `bar.target`.
- **After.** Текущий модуль будет активизирован после перечисленного модуля или модулей.

### Условные зависимости

Некоторые ключевые слова для условных зависимостей применяются в различных состояниях операционной системы вместо модулей команды `systemd`. Например:

- `ConditionPathExists=`*p*`:` — истинно, если путь (файла) *p* существует в системе;
- `ConditionPathIsDirectory=`*p*`:` — истинно, если *p* является каталогом;
- `ConditionFileNotEmpty=`*p*`:` — истинно, если *p* является файлом ненулевой длины.

Если условная зависимость в модуле не является истинной, когда команда `systemd` пытается активизировать данный модуль, то этот модуль не активизируется. Но это распространяется только на модуль, в котором есть условие. Следовательно, если вы активизируете модуль, в котором есть условная зависимость, а также некоторые другие зависимости этого модуля, команда `systemd` попытается активизировать их, не обращая внимания на истинность или ложность условия.

Другие зависимости являются главным образом вариантами перечисленных. Например, зависимость RequiresOverridable похожа на зависимость Requires, если режим работы нормальный, но она начинает вести себя подобно зависимости Wants, если модуль активизирован вручную. Полный перечень зависимостей можно увидеть на странице systemd.unit(5) руководства.

После того как вы увидели несколько фрагментов конфигурации команды `systemd`, посмотрим на реальные файлы модулей и на то, как они работают.

## 6.4.3. Конфигурация команды systemd

Файлы конфигурации команды `systemd` рассеяны по множеству каталогов системы, так что вы, как правило, не сможете найти файлы для всех модулей в одном месте. Однако имеется два основных каталога для конфигурации команды `systemd`: ката-

лог *системных модулей* (настраивается глобально, обычно это /usr/lib/systemd/system) и каталог *системной конфигурации* (локальные определения, обычно это /etc/systemd/system).

Во избежание недоразумений следуйте правилу: не вносите изменения в каталог модулей, поскольку ваша система позаботится об этом за вас. Локальные изменения вносите в каталог системной конфигурации. Итак, если вам будет предоставлен выбор между изменениями чего-либо в каталогах /usr и /etc, всегда изменяйте каталог /etc.

ПРИМЕЧАНИЕ

Можно выяснить текущий путь поиска для команды systemd (включая приоритет) с помощью такой команды:

```
# systemctl -p UnitPath show
```

Однако этот частный параметр исходит из третьего источника — настроек pkg-config. Чтобы увидеть каталоги системных модулей и конфигурации вашей системы, используйте следующие команды:

```
$ pkg-config systemd --variable=systemdsystemunitdir
$ pkg-config systemd --variable=systemdsystemconfdir
```

## Файлы модулей

Происхождение файлов модулей восходит к спецификации записей XDG (для файлов с расширением .desktop, которые очень похожи на файлы с расширением .ini в системах Microsoft), в которых названия секций заключены в скобки ([]), а в каждой из секций указаны переменные с присвоенными им значениями (параметрами).

Рассмотрим как пример файл модуля media.mount из каталога /usr/lib/systemd/system, который является стандартным для версии Fedora. Этот файл представляет файловую систему tmpfs, каталог /media выступает в роли контейнера для монтирования съемных носителей.

```
[Unit]
Description=Media Directory
Before=local-fs.target

[Mount]
What=tmpfs
Where=/media
Type=tmpfs
Options=mode=755,nosuid,nodev,noexec
```

Здесь присутствуют две секции. Секция [Unit] сообщает некоторые подробности о модуле и содержит описание и сведения о зависимости. В частности, этот модуль настроен так, чтобы его активизация происходила до модуля local-fs.target.

Секция [Mount] описывает данный модуль в роли модуля монтирования, а также сообщает детали о точке монтирования, типе файловой системы и параметрах монтирования, описанных в подразделе 4.2.6. Переменная What= идентифицирует устройство или идентификатор UUID устройства, предназначенного для монтирования. Здесь ему присвоено значение tmpfs, поскольку у этой файловой системы

нет устройства. Полный перечень параметров модуля монтирования можно увидеть на странице systemd.mount(5) руководства.

Многие другие файлы конфигурации модулей такие же простые. Например, файл модуля службы sshd.service задействует безопасный вход в оболочку:

```
[Unit]
Description=OpenSSH server daemon
After=syslog.target network.target auditd.service

[Service]
EnvironmentFile=/etc/sysconfig/sshd
ExecStartPre=/usr/sbin/sshd-keygen
ExecStart=/usr/sbin/sshd -D $OPTIONS
ExecReload=/bin/kill -HUP $MAINPID

[Install]
WantedBy=multi-user.target
```

Поскольку эта цель является службой, в секции [Service] вы найдете подробности, сообщающие о том, как подготовить, запустить и перезагрузить данную службу. Полный перечень можно увидеть на странице systemd.service(5) (а также на странице systemd.exec(5)) руководства. Отслеживание процессов будет рассмотрено в подразделе 6.4.6.

## Подключение модулей и секция [Install]

Секция [Install] в файле модуля sshd.service важна, поскольку она помогает понять, как использовать параметры зависимостей WantedBy и RequiredBy в команде systemd. Фактически это является механизмом подключения модулей без изменения каких-либо файлов конфигурации. Во время нормальной работы команда systemd игнорирует секцию [Install]. Однако представьте ситуацию, когда в вашей системе отключена служба sshd.service и вы желаете ее включить. Когда вы *включаете* модуль, команда systemd читает секцию [Install]; в данном случае включение модуля sshd.service приводит к тому, что команда systemd видит зависимость WantedBy для модуля multi-user.target. В ответ на это команда systemd следующим образом создает в каталоге системной конфигурации символическую ссылку на модуль sshd.service:

```
ln -s '/usr/lib/systemd/system/sshd.service' '/etc/systemd/system/multi-user.
target.wants/sshd.service'
```

Обратите внимание на то, что символическая ссылка помещена в подкаталог, соответствующий зависимому модулю (в данном случае это multi-user.target).

Секция [Install] обычно отвечает за каталоги .wants и .requires в каталоге системной конфигурации (/etc/systemd/system). Тем не менее каталоги .wants присутствуют также в каталоге конфигурации модулей (/usr/lib/systemd/system), и вы можете также добавлять ссылки, которые не соответствуют секциям [Install], в файлы модулей. Такие вносимые вручную дополнения являются простым способом добавить зависимость, не изменяя файл модуля, который может быть перезаписан в будущем (например, во время обновления ПО).

ПРИМЕЧАНИЕ

Подключение модуля отличается от его активизации. При подключении модуля вы указываете его в конфигурации команды systemd, внося полупостоянные изменения, которые сохраняются после перезагрузки системы. Однако не обязательно каждый раз явным образом подключать модуль. Если в файле модуля есть секция [Install], вы должны подключить его с помощью инструкции systemctl enable; в противном случае достаточно уже одного наличия такого файла для подключения модуля. Когда вы активизируете модуль с помощью systemctl start, вы просто задействуете его в текущем рабочем окружении. Кроме того, подключение модуля не активизирует его.

### Переменные и спецификаторы

Файл модуля `sshd.service` демонстрирует также применение переменных, а именно переменных окружения `$OPTIONS` и `$MAINPID`, которые переданы командой `systemd`. Значения переменной `$OPTIONS` являются параметрами, которые можно передать команде `sshd` при активизации данного модуля командой `systemctl`, а значение переменной `$MAINPID` — это отслеживаемый процесс службы (см. подраздел 6.4.6).

*Спецификатор* — это еще одно похожее на переменную средство, которое часто можно увидеть в файлах модулей. Спецификаторы начинаются с символа процента (`%`). Например, спецификатор `%n` представляет имя текущего модуля, а спецификатор `%H` — имя текущего хоста.

ПРИМЕЧАНИЕ

Имя модуля может содержать некоторые интересные спецификаторы. Можно параметризовать единичный файл модуля, чтобы породить несколько копий какой-либо службы, например процессов getty, работающих в терминалах tty1, tty2 и т. д. Чтобы использовать эти спецификаторы, добавьте символ @ в конец имени модуля. Для процесса getty создайте файл модуля с именем getty@.service, который позволит вам обращаться к таким модулям, как getty@tty1 и getty@tty2. Все, что следует за символом @, называется экземпляром, и при обработке файла модуля команда systemd развертывает спецификатор %I в имя экземпляра. Увидеть это в действии можно на файлах getty@.service, которые входят в большинство систем, использующих команду systemd.

## 6.4.4. Работа команды systemd

С командой `systemd` вы будете взаимодействовать главным образом с помощью команды `systemctl`, которая позволяет активизировать и деактивизировать службы, выводить статус, перезагружать конфигурацию и многое другое.

Наиболее существенные основные команды имеют дело с получением информации о модулях. Например, чтобы увидеть список активных модулей в вашей системе, воспользуйтесь командой `list-units`. На самом деле эта команда работает по умолчанию при запуске команды `systemctl`, поэтому часть `list-units` не нужна:

```
$ systemctl list-units
```

Формат вывода типичен для информационных команд Unix. Так, например, выглядит заголовок и строка для модуля `media.mount`:

```
UNIT              LOAD   ACTIVE SUB     JOB DESCRIPTION
media.mount       loaded active mounted Media Directory
```

Эта команда выводит много сведений, поскольку в типичной системе большое количество активных модулей, но даже при этом список сокращен, поскольку

команда systemctl обрезает все действительно длинные названия модулей. Чтобы увидеть полные имена модулей, используйте параметр --full, а чтобы увидеть все модули (а не только активные) — параметр --all.

Чрезвычайно полезной функцией команды systemctl является получение статуса модуля. Вот, например, типичная команда запроса статуса и результат ее работы:

```
$ systemctl status media.mount
media.mount - Media Directory
          Loaded: loaded (/usr/lib/systemd/system/media.mount; static)
          Active: active (mounted) since Wed, 13 May 2015 11:14:55 -0800;
37min ago
           Where: /media
            What: tmpfs
          Process: 331 ExecMount=/bin/mount tmpfs /media -t tmpfs -o
mode=755,nosuid,nodev,noexec (code=exited, status=0/SUCCESS)
          CGroup: name=systemd:/system/media.mount
```

Обратите внимание на то, что информации здесь гораздо больше, чем можно увидеть в любой традиционной системе init. Вы узнаете не только статус модуля, но также и точное название команды, использованной при его монтировании, его идентификатор PID, а также его конечный статус.

Одним из самых интересных фрагментов этого вывода является имя группы управления. В приведенном примере группа управления не содержит никакой информации, кроме имени systemd:/system/media.mount, поскольку процессы модуля уже остановлены. Однако, если вы запросите статус модуля службы, например NetworkManager.service, вы увидите также дерево процессов группы управления. Можно увидеть группы управления без сопутствующего статуса модуля с помощью команды systemd-cgls. О группах управления вы узнаете больше из подраздела 6.4.6.

Команда статуса отображает также последнюю информацию из журнала модуля (этот журнал записывает диагностическую информацию для каждого модуля).

Полный журнал модуля можно увидеть с помощью такой команды:

```
$ journalctl _SYSTEMD_UNIT=unit
```

Синтаксис немного странный, поскольку команда journalctl способна получать доступ не только к модулю systemd.

Для активизации, деактивизации и перезапуска модулей используйте команды systemd start, stop и restart. Однако если вы изменили файл конфигурации модуля, можно перезагрузить такой файл одним из двух способов:

- systemctl reload unit — перезагружает только конфигурацию модуля unit;
- systemctl daemon-reload — перезагружает конфигурацию всех модулей.

Запросы на активизацию, повторную активизацию и перезапуск модулей известны как *задания* для команды systemd, они являются, по сути, изменениями состояния модулей. Узнать о текущих заданиях в системе можно с помощью команды:

```
$ systemctl list-jobs
```

Если система уже работает некоторое время, то вполне разумно ожидать, что в ней не будет активных заданий, поскольку все процессы активизации должны быть завершены. Однако во время загрузки системы иногда можно войти в нее настолько быстро, чтобы успеть заметить несколько очень медленно запускающихся модулей, которые еще не полностью активны. Например:

```
JOB UNIT                      TYPE         STATE
  1 graphical.target          start        waiting
  2 multi-user.target         start        waiting
 71 systemd-...nlevel.service start        waiting
 75 sm-client.service         start        waiting
 76 sendmail.service          start        running
120 systemd-...ead-done.timer start        waiting
```

В данном случае задание 76, запуск модуля sendmail.service, действительно происходит довольно долго. Остальные перечисленные задания находятся в ждущем режиме, скорее всего, потому, что все они ждут задание 76. Когда служба `sendmail.service` завершит запуск и станет полностью активна, задание 76 и все остальные задания также будут завершены, а список заданий станет пустым.

ПРИМЕЧАНИЕ

Термин задание может сбивать с толку еще и потому, что другая версия команды init, Upstart, описанная в данной главе, использует слово «задание» (в общих чертах) применительно к тому, что команда systemd называет модулем. Важно помнить о том, что, если задание команды systemd, связанное с каким-либо модулем, будет завершено, сам модуль останется активным и в дальнейшем работающим, особенно в случае модулей служб.

Обратитесь к разделу 6.7, чтобы узнать о том, как выключать и перезапускать систему.

## 6.4.5. Добавление модулей в команду systemd

Добавление модулей заключается в создании, активизации и возможном редактировании файлов модулей. Обычно пользовательские файлы модулей следует помещать в каталог системной конфигурации `/etc/systemd/system`, чтобы не перепутать их с чем-либо, входящим в состав вашей версии системы, и чтобы система не перезаписала их при обновлении.

Поскольку довольно просто создать целевые модули, которые ничего не делают и ни на что не влияют, давайте попробуем. Ниже приведена процедура создания двух целевых модулей, один из которых зависит от другого.

1. Создайте файл модуля с именем `test1.target`:

```
[Unit]
Description=test 1
```

2. Создайте файл `test2.target` с зависимостью от файла `test1.target`:

```
[Unit]
Description=test 2
Wants=test1.target
```

3. Активизируйте модуль test2.target (помня о том, что зависимость в файле test2.target вынуждает команду systemd активизировать при этом и модуль test1.target):

```
# systemctl start test2.target
```

4. Убедитесь в том, что оба модуля активны:

```
# systemctl status test1.target test2.target
test1.target - test 1
           Loaded: loaded (/etc/systemd/system/test1.target; static)
           Active: active since Thu, 12 Nov 2015 15:42:34 -0800; 10s ago
test2.target - test 2
           Loaded: loaded (/etc/systemd/system/test2.target; static)
          Active: active since Thu, 12 Nov 2015 15:42:34 -0800; 10s ago
```

ПРИМЕЧАНИЕ

Если в файле модуля есть секция [Install], подключите модуль до его активизации:

```
# systemctl enable unit
```

Опробуйте это на предыдущем примере. Удалите зависимость из файла test2.target и добавьте секцию [Install] в файл test1.target, содержащую строку WantedBy=test2.target.

**Удаление модулей.** Чтобы удалить модуль, выполните следующее.

1. Если необходимо, деактивизируйте модуль:

```
# systemctl stop unit
```

2. Если в модуле есть секция [Install], отключите модуль, чтобы удалить все зависимые символические ссылки:

```
# systemctl disable unit
```

3. Удалите файл модуля, если желаете.

## 6.4.6. Отслеживание процессов и синхронизация в команде systemd

Команде systemd необходимы разумное количество информации и степень контроля над каждым запускаемым процессом. Основная проблема заключается в том, что службы могут быть запущены различными способами, они могут ответвляться в виде новых экземпляров и даже становиться демонами и открепляться от исходного процесса.

Чтобы свести к минимуму объем работы, который программист или администратор должен выполнить для создания работающего модуля, команда systemd использует *группы управления* (cgroups) — необязательную функцию ядра Linux, которая предусматривает более точное отслеживание иерархии процессов. Если вы уже работали до этого с командой Upstart, вы знаете, что необходимо проделывать небольшую дополнительную работу, чтобы выяснить, какой процесс является

главным для какой-либо службы. В случае с командой systemd вам не надо беспокоиться о том, сколько раз ветвится процесс, важно лишь то, ветвится ли он. Используйте параметр Type в файле модуля для службы, чтобы указать ее поведение при запуске. Существуют два основных стиля запуска.

- Type=simple — процесс службы не ветвится.
- Type=forking — служба ветвится, и команда systemd ожидает завершения исходного процесса службы. По его завершении команда systemd предполагает, что данная служба готова.

Параметр Type=simple не учитывает тот факт, что службе может потребоваться некоторое время на настройку, а команда systemd не знает, когда запускать зависимости, для которых абсолютно необходима готовность данной службы. Один из способов справиться с этим — использовать отложенный запуск (см. подраздел 6.4.7). Однако с помощью некоторых стилей параметра Type можно указать, чтобы служба сама известила команду systemd о своей готовности:

- Type=notify — когда служба готова, она отправляет уведомление специально для команды systemd (с помощью вызова функции sd_notify());
- Type=dbus — когда служба готова, она регистрирует себя в шине D-Bus (шина рабочего стола).

Еще один вариант запуска задается параметром Type=oneshot. Здесь процесс службы завершается полностью по окончании своей работы. Для служб подобного типа непременно следует добавлять параметр RemainAfterExit=yes, чтобы команда systemd по-прежнему рассматривала данную службу как активную даже после завершения ее процессов.

И наконец, последний стиль: Type=idle. Он просто указывает команде systemd не запускать службу, пока в системе есть активные задания. Идея заключается в простом откладывании запуска службы до тех пор, пока не будут запущены другие службы, чтобы снизить нагрузку на систему или избежать перекрывания выводов различных служб. Помните: как только служба запущена, задание команды systemd, которое запустило службу, завершается.

## 6.4.7. Запуск по запросу и распараллеливание ресурсов в команде systemd

Одной из самых важных функций команды systemd является ее способность откладывать запуск модуля до тех пор, пока он не станет абсолютно необходим. Обычно это выглядит так.

1. Создается в обычном порядке модуль systemd (назовем его модуль A) для системной службы, которую вы собираетесь обеспечить.
2. Определяется системный ресурс (например, сетевой порт/сокет, файл или устройство), который модуль A использует для предоставления своих служб.
3. Создается еще один модуль systemd, модуль R, для предоставления данного ресурса. Эти модули являются специальными: модули сокетов, модули путей и модули устройств.

На деле это происходит следующим образом.

1. После активизации модуля R команда systemd контролирует его ресурс.
2. Если что-либо пытается получить доступ к этому ресурсу, команда systemd блокирует ресурс, а ввод в данный ресурс буферизуется.
3. Команда systemd активизирует модуль A.
4. Когда служба модуля A будет готова, она получает управление ресурсом, считывает содержимое буфера и переходит в нормальный режим работы.

Есть моменты, о которых следует позаботиться.

- Вы должны убедиться в том, что модуль ресурсов охватывает все ресурсы, предоставляемые службой. Это не составляет сложности, поскольку у большинства служб всего одна точка доступа.
- Следует убедиться в том, что модуль ресурсов прикреплен к модулю службы, которую он представляет. Это может быть сделано неявно или явно. В некоторых случаях разные параметры определяют различные способы, с их помощью команда systemd передает управление модулю службы.
- Не все серверы знают, как выполнить стыковку с модулями, которые может предоставить команда systemd.

Если вы уже знакомы с тем, как работают утилиты наподобие inetd, xinetd и automount, вы увидите много сходного. Действительно, сама концепция не нова (и в действительности команда systemd включает поддержку модулей automount). Мы перейдем к примеру модуля сокета в пункте «Пример модуля сокета и службы» далее. Однако сначала рассмотрим, какую помощь оказывают модули ресурсов при загрузке системы.

## Оптимизация загрузки и вспомогательные модули

При обычном способе активизации модулей в команде systemd предпринимаются попытки упростить порядок следования зависимостей и ускорить время загрузки системы. Это напоминает запуск по запросу тем, что модуль службы и вспомогательный модуль представляют ресурс, предлагаемый модулем службы, только в данном случае команда systemd запускает модуль службы, как только она активизирует вспомогательный модуль.

В основе данной схемы лежит то, что таким важным модулям служб времени загрузки, как syslog и dbus необходимо некоторое время для запуска, а многие другие модули зависят от них. Однако команда systemd способна предоставить важный для модуля ресурс (например, модуль сокета) очень быстро, после чего она может немедленно активизировать не только важный модуль, но также и любые другие модули, которые зависят от данного важного ресурса. Как только важный модуль будет готов, он забирает управление ресурсом.

На рис. 6.2 показано, как такая схема могла бы работать в традиционной системе. В данной последовательности загрузки служба E предоставляет важный ресурс R. Службы A, B и C зависят от этого ресурса и должны ожидать запуска службы E. Во время загрузки такой системе потребуется довольно много времени, чтобы добраться до запуска службы C.

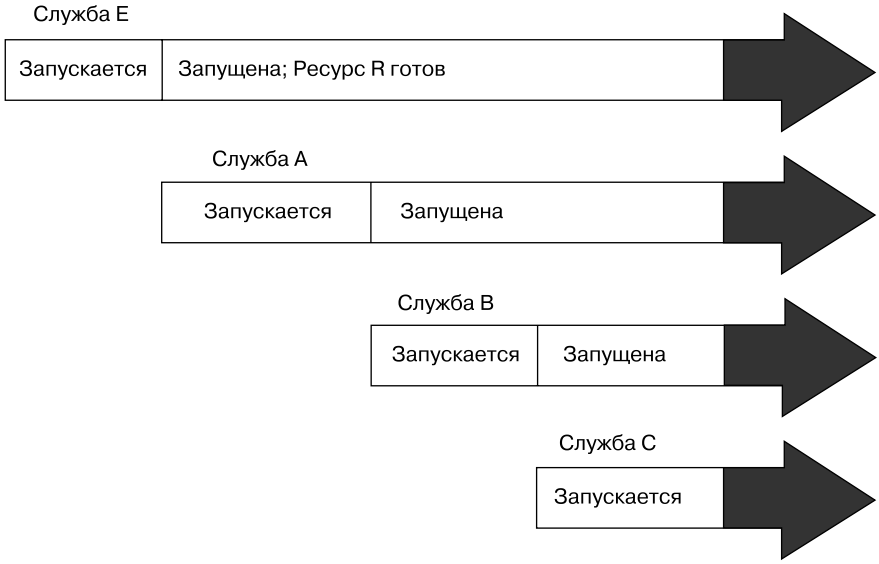


**Рис. 6.2.** Временна́я последовательность загрузки. Указана зависимость ресурсов

На рис. 6.3 показана эквивалентная конфигурация загрузки с помощью команды `systemd`. Службы представлены модулями A, B и C, а новый модуль R представляет ресурс, который предоставляет модуль E. Поскольку команда `systemd` способна обеспечить интерфейс для модуля R, пока запускается модуль E, модули A, B, C и E могут быть все запущены одновременно. Модуль E заступает на место модуля R, когда будет готов. Интересным моментом здесь является то, что модулям A, B и C может не понадобиться явный доступ к модулю R до завершения их запуска — это демонстрирует на схеме модуль B.

ПРИМЕЧАНИЕ

При подобном распараллеливании запуска есть вероятность того, что система на некоторое время замедлит свою работу вследствие большого количества одновременно стартующих модулей.

В конечном счете, хотя вы и не создаете в данном случае запуск модуля по запросу, вы используете те же функции, которые делают его возможным. Чтобы увидеть примеры из реальной жизни, загляните в модули конфигурации syslog и D-Bus на компьютере, использующем команду systemd; очень возможно, что они будут распараллелены подобным же образом.

## Пример модуля сокета и службы

Рассмотрим теперь в качестве примера простую службу сетевого эхо-запроса, которая использует модуль сокета. Этот материал достаточно сложный, и вы, вероятно, сможете понять его только после прочтения глав 9 и 10, в которых рассмотрены протокол TCP, порты, прослушивание и сокеты. По этой причине сейчас можно его пропустить и вернуться к нему позже.


**Рис. 6.3.** Временна́я последовательность загрузки с помощью systemd и модуля ресурсов

Идея, заложенная в эту службу, заключается в том, что, когда сетевой клиент подключается к данной службе, она повторяет все, что отправляет клиент. Модуль будет прослушивать TCP порт 22222. Мы назовем его службой эхо-запроса и начнем с модуля сокета, представленного следующим файлом модуля `echo.socket`:

```
[Unit]
Description=echo socket

[Socket]
ListenStream=22222
Accept=yes
```

Обратите внимание на то, что внутри файла модуля нет упоминания о том, какой модуль службы поддерживается данным сокетом. Каков же тогда соответствующий файл модуля службы?

Он называется `echo@.service`. Эта ссылка составлена на основе соглашения о присвоении имен: если у файла модуля службы такой же префикс, что и у файла `.socket` (в данном случае `echo`), команда `systemd` знает о том, что надо активизировать данный модуль службы, когда в модуле сокета возникает активность. В данном случае команда `systemd` создает экземпляр `echo@.service`, когда возникает активность сокета `echo.socket`.

Вот файл модуля службы `echo@.service`:

```
[Unit]
Description=echo service

[Service]
```

```
ExecStart=-/bin/cat
StandardInput=socket
```

ПРИМЕЧАНИЕ

Если вам не нравится неявная активизация модулей на основе префиксов или же вам необходимо создать механизм активизации между двумя модулями с разными префиксами, можно использовать вариант явного присвоения в модуле, который определяет ресурс. Применяйте, например, запись Socket=bar.socket внутри файла foo.service, чтобы модуль bar.socket предоставлял свой сокет службе foo.service.

Чтобы данная служба заработала, необходимо запустить после нее сокет `echo.socket`:

```
# systemctl start echo.socket
```

Теперь можно проверить эту службу, подключившись к локальному порту 22222. Когда приведенная ниже команда `telnet` установит соединение, наберите что-либо и нажмите клавишу Enter. Служба возвратит вам набранный текст.

```
$ telnet localhost 22222
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi there.
Hi there.
```

Когда захотите закончить, нажмите сочетание клавиш Ctrl+] в самой строке, а затем сочетание клавиш Ctrl+D. Чтобы остановить службу, остановите модуль сокета:

```
# systemctl stop echo.socket
```

## Экземпляры и передача управления

Поскольку модуль `echo@.service` поддерживает несколько одновременных экземпляров, в его имени присутствует символ `@` (вспомните, что такой символ означает параметризацию). Однако зачем может понадобиться несколько экземпляров? Причина в том, что у вас может оказаться более одного сетевого клиента, подключенного к этой службе в данный момент времени, и каждое соединение должно иметь собственный экземпляр.

В указанном случае модуль службы должен поддерживать несколько экземпляров, поскольку в файле `echo.socket` есть параметр `Accept`. Этот параметр указывает команде `systemd`, чтобы она не только прослушивала порт, но и принимала входящие соединения, а затем передавала входящие соединения модулю службы, создавая для каждого соединения отдельный экземпляр. Каждый экземпляр считывает данные из соединения как стандартный ввод, но при этом экземпляру не обязательно знать о том, что эти данные исходят из сетевого соединения.

ПРИМЕЧАНИЕ

Для большинства сетевых соединений требуется больше гибкости по сравнению с простым шлюзом стандартного ввода/вывода, поэтому не рассчитывайте на то, что сможете создавать сетевые службы с помощью модулей, подобных описанному здесь echo@.service.

Хотя модуль службы мог бы выполнить всю работу по принятию соединения, он не может содержать в своем имени символ @. Иначе он смог бы полностью контролировать сокет, и команда systemd не стала бы пытаться прослушивать сетевой порт, пока этот модуль службы не завершит работу.

Множество различных ресурсов и параметров передачи управления модулям служб не позволяют составить краткий обзор. К тому же документация к параметрам занимает несколько страниц руководства. Модулям, ориентированным на ресурсы, посвящены страницы systemd.socket(5), systemd.path(5) и systemd.device(5). О модулях служб есть также документ systemd.exec(5), который часто упускают из виду. Он содержит информацию о том, как модуль службы может рассчитывать на получение ресурса после активизации.

## 6.4.8. Совместимость команды systemd со сценариями System V

Одно свойство, которое отличает команду systemd от других init-систем нового поколения, заключается в том, что эта команда стремится выполнять более полную работу по отслеживанию служб, запущенных сценариями, которые совместимы со стандартом System V. Это работает следующим образом.

1. Сначала команда systemd активизирует модуль runlevel<*N*>.target, где *N* является уровнем запуска.
2. Для каждой символической ссылки в файле /etc/rc<*N*>.d команда systemd идентифицирует сценарий в каталоге /etc/init.d.
3. Команда systemd ассоциирует название сценария с модулем службы (например, для сценария /etc/init.d/foo это будет foo.service).
4. Команда systemd активизирует модуль службы и запускает сценарий с аргументом start или stop, в зависимости от его имени в файле rc<*N*>.d.
5. Команда systemd пытается ассоциировать любые процессы сценария с модулями служб.

Поскольку команда systemd создает ассоциацию с именем модуля службы, можно использовать команду systemctl для перезапуска службы или просмотра ее состояния. Однако не ожидайте чудес от режима совместимости со стандартом System V. Он по-прежнему должен запускать init-сценарии последовательно.

## 6.4.9. Команды, дополняющие systemd

Начав работу с командой systemd, вы можете заметить исключительно большое количество команд в каталоге /lib/systemd. В основном это вспомогательные программы для модулей. Например, команда udevd является частью systemd, и здесь вы обнаружите ее под именем systemd-udevd. Еще одна команда, systemd-fsck, выступает посредником между командами systemd и fsck.

Многие из этих команд существуют потому, что они содержат механизмы уведомления, которые отсутствуют в стандартных системных утилитах. Зачастую они просто запускают такие утилиты и уведомляют команду systemd о результатах.

В конечном счете было бы глупо пытаться заново реализовать все команды fsck внутри systemd.

ПРИМЕЧАНИЕ

Еще одним интересным свойством этих команд является то, что они написаны на языке C, поскольку одна из целей команды systemd — уменьшить число сценариев оболочки в системе. Довольно спорный вопрос, хорошо ли это (ведь многие из этих команд могли быть написаны в качестве сценариев оболочки), но пока все работает и делает свое дело надежно, безопасно и достаточно быстро, нет смысла беспокоить спорящие стороны.

Когда вы встретите незнакомую команду в каталоге /lib/systemd, загляните в руководство. Вполне вероятно, что страница справки не только даст описание утилиты, но также сообщит и о типе модуля, который она призвана дополнять.

Если вы не пользуетесь (или не интересуетесь) вариантом Upstart, переходите к разделу 6.6, чтобы получить представление об init-процессе версии System V.

# 6.5. Команда Upstart

Команда init в варианте Upstart основана на *заданиях* и *событиях*. Задания — это действия в момент запуска и в процессе работы, которые выполняет команда Upstart (например, системные службы и конфигурация), а события — это сообщения, которые команда Upstart получает от себя или от других процессов (например, udevd). Работа команды Upstart состоит в запуске заданий в ответ на события.

Чтобы получить представление о том, как это работает, рассмотрим задание udev для запуска демона udevd. Его конфигурация обычно содержится в файле /etc/init/udev.conf, в котором есть следующие строки:

```
start on virtual-filesystems
stop on runlevel [06]
```

Эти строки означают, что команда Upstart запускает задание udev после возникновения события virtual-filesystems, а остановка этого задания будет выполнена после возникновения события runlevel с аргументом 0 или 6.

Существуют многочисленные варианты событий и аргументов. Например, команда Upstart может реагировать на события, возникшие как ответ на состояние задания, например на событие started udev, которое было вызвано заданием udev, упомянутым выше. Однако прежде чем детально рассмотреть задания, приведем обзор того, как работает команда Upstart на высоком уровне.

## 6.5.1. Процедура инициализации команды Upstart

Во время запуска команда Upstart выполняет следующее.

1. Загружает файл своей конфигурации, а также файл конфигурации заданий в каталог /etc/init.
2. Вызывает событие startup.
3. Запускает задания, которые предназначены для выполнения при появлении события startup.

4. Начальные задания приводят к возникновению новых событий, которые дают начало другим заданиям и событиям.

По завершении всех заданий, связанных с нормальным запуском, команда Upstart продолжает отслеживать события и реагировать на них во время функционирования системы в целом.

Большинство версий Upstart работает следующим образом.

1. Самым важным заданием, которое команда Upstart запускает в ответ на событие startup, является задание mountall. Это задание монтирует все необходимые для запущенной в данный момент ОС локальные и виртуальные файловые системы, чтобы могло функционировать все остальное.
2. Задание mountall порождает несколько событий, в число которых входят среди прочих filesystem, virtual-filesystems, local-filesystems, remote-filesystems и all-swaps. Данные события сообщают о том, что главные файловые системы в этой ОС смонтированы и готовы к работе.
3. В ответ на эти события команда Upstart запускает несколько важных служебных заданий. Например, задание udev запускается в ответ на событие virtual-filesystems, а задание dbus — в ответ на событие local-filesystems.
4. Среди главнейших служебных заданий команда Upstart запускает задание network-interfaces, как правило, в ответ на событие local-filesystems и готовность к работе демона udevd.
5. Задание network-interfaces порождает событие static-network-up.
6. Команда Upstart запускает задание rc-sysinit в ответ на события filesystem и static-network-up. Это задание отвечает за обслуживание текущего уровня запуска в системе, и при первом запуске без указания уровня запуска оно переводит систему на уровень запуска по умолчанию, порождая событие runlevel.
7. Команда Upstart запускает бóльшую часть остальных заданий запуска системы в ответ на событие runlevel и новый уровень запуска.

Этот процесс может усложниться, поскольку не всегда понятно, где возникают события. Команда Upstart порождает лишь несколько событий, а все остальные исходят от заданий. Файлы конфигурации заданий обычно объявляют события, которые они будут порождать, но подробности того, как задание порождает события, как правило, отсутствуют в таких файлах.

Чтобы добраться до сути, приходится изрядно покопаться. Рассмотрим, например, событие static-network-up. Файл конфигурации задания network-interface.conf сообщает о том, что он порождает данное событие, но не уточняет где. Выясняется, что данное событие является результатом команды ifup, которую данное задание запускает при инициализации сетевого интерфейса в сценарии /etc/network/if-up.d/upstart.

ПРИМЕЧАНИЕ

Хотя все описанное содержится в документации (ссылка на страницу interfaces(5) с описанием каталога ifup.d идет со страницы ifup(8)), может оказаться довольно трудно выяснить, как все работает, просто читая ее. Обычно быстрее будет применить команду grep, указав название события, для нескольких файлов конфигурации, посмотреть результаты, а затем попробовать на основе фрагментов составить полную картину.

Одним из недостатков команды Upstart является отсутствие возможности простого отслеживания событий. Можно перевести ее журнал в режим отладки, в результате чего в нем будут отображены все входящие события (как правило, этот журнал хранится в файле /var/log/syslog), однако обилие посторонней информации в этом файле затрудняет определение контекста события.

## 6.5.2. Задания команды Upstart

Каждый файл в каталоге конфигурации /etc/init команды Upstart соответствует какому-либо заданию, а главный файл конфигурации для каждого задания снабжен расширением .conf. Например, файл /etc/init/mountall.conf определяет задание mountall.

Существуют два первичных типа заданий Upstart.

- **Задачи (Task jobs).** Это задания с четким окончанием. Например, задание mountall является таковым, поскольку оно завершается по окончании монтирования файловых систем.
- **Службы (Service jobs).** У таких заданий нет определенного окончания. Серверы (демоны), такие как udevd, серверы баз данных и веб-серверы, являются заданиями-службами.

Третьий тип заданий — *абстрактное задание*. Его можно представить как своего рода виртуальную службу. Абстрактные задания существуют только для команды Upstart и сами по себе ничего не запускают, но они иногда используются в качестве инструментов управления другими заданиями, поскольку другие задания могут быть запущены или остановлены на основе событий, исходящих от абстрактного задания.

### Просмотр заданий

Просмотреть задания команды Upstart, а также их статус можно с помощью команды initctl. Чтобы получить обзор того, что происходит в вашей системе, запустите такую команду:

```
$ initctl list
```

Результат работы будет довольно обширным, поэтому посмотрим лишь на два задания, которые могут быть найдены в типичном листинге. Вот простой пример состояния задачи:

```
mountall stop/waiting
```

Он сообщает о том, что задание mountall находится в статусе «останов/ожидание», и это означает, что оно не работает. К сожалению (на момент написания книги), вы не сможете использовать статус, чтобы определить, запущено ли уже задание или еще нет, поскольку статус «останов/ожидание» применяется также и к никогда не запускавшимся заданиям.

Службы, с которыми связаны процессы, будут отображаться в перечне статусов следующим образом:

```
tty1 start/running, process 1634
```

Эта строка говорит о том, что задание `tty1` запущено и процесс с идентификатором ID 1634 выполняет его. Не все службы обладают связанными процессами.

ПРИМЕЧАНИЕ

Если вам известно имя задания job, можно просмотреть его статус напрямую с помощью команды initctl status job.

Информация о статусе в результатах вывода команды `initctl` (например, `stop/waiting`) может сбивать с толку. Левая часть (до символа /) является *целью*, или тем, по направлению к чему призвано работать данное задание (например, запуск или останов). Правая часть сообщает текущее *состояние задания* или то, что данное задание выполняет именно сейчас (например, ожидание или работа). В приведенном выше листинге задание `tty1` обладает статусом start/running, это значит, что его целью является запуск. Состояние работы говорит о том, что задание запущено успешно. Для служб состояние работы является номинальным.

Случай с заданием `mountall` немного другой, поскольку задачи не остаются в рабочем состоянии. Статус «останов/ожидание» обычно говорит о том, что данное задание стартовало и завершило свою задачу. По завершении задачи цель меняется с запуска на останов, и теперь задание ожидает дальнейших указаний команды `Upstart`.

К сожалению, как отмечалось ранее, поскольку задания, которые никогда не запускались, также обладают статусом «останов/ожидание», невозможно установить, запускалось ли какое-либо задание, если вы не включили режим отладки и не заглянули в журналы, как описано в подразделе 6.5.5.

ПРИМЕЧАНИЕ

Вы не увидите те задания, которые были запущены в системе с помощью команды Upstart в режиме совместимости со стандартом System V.

## Переходы между состояниями заданий

Существует множество состояний заданий, однако для переключения между ними установлен четкий порядок. Вот как, например, запускается типичное задание.

1. Все задания начинаются в статусе «останов/ожидание».
2. Когда пользователь или системное событие запускают задание, цель задания меняется с останова на запуск.
3. Команда `Upstart` изменяет состояние задания с ожидающего на стартующее, поэтому теперь его статус — «запуск/запуск» (start/starting).
4. Команда `Upstart` порождает событие `starting` *`job`*.
5. Задание выполняет все, что необходимо сделать для состояния запуска.
6. Команда `Upstart` изменяет состояние задания со стартующего на предстартовое и порождает событие `pre-start` *`job`*.
7. Задание выполняется своим чередом и проходит еще через несколько состояний, пока не достигает работающего состояния.
8. Команда `Upstart` порождает событие `started`.

Завершение задания содержит похожий набор изменений состояния и событий. Обратитесь к странице руководства upstart-events(7), чтобы узнать подробности обо всех состояниях и переходах для обеих целей.

## 6.5.3. Конфигурация команды Upstart

Исследуем два файла конфигурации: один для задачи `mountall`, а другой — для сервиса `tty1`. Подобно другим файлам конфигурации, данные файлы расположены в каталоге `/etc/init` и называются `mountall.conf` и `tty1.conf`. Файлы конфигурации составлены из небольших фрагментов, которые называются *строфами*. Каждая строфа начинается с заглавного ключевого слова, например `description` или `start`.

Для начала откройте на своем компьютере файл `mountall.conf`. Отыщите в первой строфе нечто подобное приведенной ниже строке:

```
description     "Mount filesystems on boot"
```

Эта строфа дает краткое текстовое описание задания.

Далее вы увидите несколько строф, описывающих процесс запуска задания `mountall`:

```
start on startup
stop on starting rcS
```

Здесь первая строка говорит команде `Upstart` о запуске задания при возникновении события `startup` (это начальное событие, порождаемое командой `Upstart`). Вторая строка сообщает команде `Upstart` о том, что задание следует остановить при возникновении события `rcS`, когда система перейдет в режим одиночного пользователя.

Следующие строки говорят команде `Upstart` о том, как себя ведет задание `mountall`:

```
expect daemon
task
```

Строфа `task` говорит команде `Upstart` о том, что данное задание является задачей, и поэтому в какой-то момент должно быть завершено. Строфа `expect` чуть сложнее. Она означает, что задание `mountall` породит демон, который будет действовать независимо от исходного сценария задания. Команде `Upstart` необходимо знать о том, когда демон прекращает работу, чтобы дать правильный сигнал о завершении задания `mountall`. Более подробно мы рассмотрим это в пункте «Отслеживание процессов и строфа expect команды Upstart» далее.

Файл `mountall.conf` продолжается далее несколькими строфами `emits`, указывающими на события, которые порождает данное задание:

```
emits virtual-filesystems
emits local-filesystems
emits remote-filesystems
emits all-swaps
emits filesystem
emits mounting
emits mounted
```

ПРИМЕЧАНИЕ

Как отмечалось в подразделе 6.5.1, присутствующие здесь строки не сообщают о настоящих источниках событий. Чтобы их отыскать, вам придется тщательно просмотреть сценарий задания.

Вам может также встретиться строфа `console`, определяющая, куда должна направлять вывод команда `Upstart`:

```
console output
```

Если указан параметр `output`, команда `Upstart` отправляет вывод задания `mountall` в системную консоль.

Теперь вы увидите подробности самого задания — в данном случае это строфа `script`:

```
script
    . /etc/default/rcS
    [ -f /forcefsck ] && force_fsck="--force-fsck"
    [ "$FSCKFIX" = "yes" ] && fsck_fix="-fsck-fix"

    # set $LANG so that messages appearing in plymouth are translated
    if [ -r /etc/default/locale ]; then
        . /etc/default/locale
        export LANG LANGUAGE LC_MESSAGES LC_ALL
    fi

    exec mountall --daemon $force_fsck $fsck_fix
end script
```

Это сценарий оболочки (см. главу 11), основная часть которого является подготовительной: происходит настройка языка сообщений, а также определяется необходимость использования утилиты `fsck`. Реальные действия происходят в команде `exec mountall`, в нижней части этого сценария. Эта команда монтирует файловые системы и порождает события по окончании данного задания.

## Служба: tty1

Служба `tty1` намного проще, она контролирует строку приглашения в виртуальной консоли. Полный файл конфигурации `tty1.conf` выглядит так:

```
start on stopped rc RUNLEVEL=[2345] and (
            not-container or
            container CONTAINER=lxc or
            container CONTAINER=lxc-libvirt)

stop on runlevel [!2345]

respawn
exec /sbin/getty -8 38400 tty1
```

Самой сложной частью данного задания является момент его запуска, но пропустите пока строки, содержащие слово `container`, и сосредоточьтесь на следующем фрагменте:

```
start on stopped rc RUNLEVEL=[2345]
```

Эта часть сообщает команде Upstart, чтобы она активизировала данное задание после возникновения события stopped rc, когда отработает и будет завершена задача rc. Чтобы условие стало истинным, задание rc должно также установить для переменной окружения RUNLEVEL значение от 2 до 5 (см. подраздел 6.5.6).

ПРИМЕЧАНИЕ

Другие задания, которые работают с уровнями запуска, не столь требовательны. Вы можете встретить, например, такую запись: start on runlevel [2345]. Единственное существенное различие между двумя приведенными строфами start заключается в выборе момента времени; данный пример активизирует задание, как только будет установлен уровень запуска, а в предыдущем примере ожидается завершение всех процессов System V.

Здесь присутствует конфигурация контейнера, поскольку команда Upstart работает не только напрямую поверх ядра системы Linux с реальными аппаратными средствами, но способна также работать и в виртуальных средах или контейнерах. Некоторые из таких сред не располагают виртуальными консолями, и вам не придется запускать процесс getty для несуществующей консоли.

Остановка задания tty1 происходит просто:

```
stop on runlevel [!2345]
```

Строфа stop говорит команде Upstart о том, что задание следует остановить, если уровень запуска выйдет из диапазона значений от 2 до 5 (например, во время выключения системы).

Строфа exec в нижней части является командой, которую следует запустить:

```
exec /sbin/getty -8 38400 tty1
```

Эта строфа очень похожа на строфу script, которую вы видели в задании mountall, за исключением того, что для запуска задания tty1 не требуется сложная настройка — оно просто запускается одной строкой. В данном случае мы запускаем команду выдачи строки приглашения getty в устройстве /dev/tty1, которое является первой виртуальной консолью (той, в которой вы окажетесь, когда нажмете в графическом режиме сочетание клавиш Ctrl+Alt+F1).

Строфа respawn дает распоряжение команде Upstart о перезапуске задания tty1, если оно завершается. В данном случае команда Upstart запускает новый процесс getty для новой строки приглашения, когда вы выходите из виртуальной консоли.

Это были основы конфигурации команды Upstart. Подробности вы сможете найти на странице руководства init(5), а также в онлайн-источниках. Одной строфе следует уделить особое внимание. Это строфа expect, и о ней пойдет речь дальше.

## Отслеживание процессов и строфа expect команды Upstart

Поскольку команда Upstart отслеживает процессы в заданиях, как только они запущены (чтобы она могла эффективно останавливать и перезапускать их), ей необходимо знать, какие процессы относятся к каждому из заданий. Эта задача может оказаться сложной, поскольку в традиционной схеме запуска системы Unix процессы ответвляются друг от друга, превращаясь в демоны, и основной процесс для какого-либо задания может начаться после одного-двух ветвлений. Без четкого

отслеживания процессов команда Upstart будет неспособна завершить запуск задания или же неправильно определит идентификатор PID для задания.

Поведение задания сообщается команде Upstart с помощью строфы expect. Существует четыре основные возможности:

- отсутствие строфы expect — основной процесс задания не ветвится, следует отслеживать основной процесс;
- expect fork — процесс ветвится один раз, следует отслеживать ответвившийся процесс;
- expect daemon — процесс ветвится дважды, следует отслеживать второе ответвление;
- expect stop — основной процесс задания подаст сигнал SIGSTOP, чтобы сообщить о своей готовности. Такое бывает редко.

Для команды Upstart и для других современных вариантов команды init, таких как systemd, идеальным вариантом является первый (отсутствие строфы expect), поскольку основному процессу задания не приходится задействовать никаких собственных механизмов запуска и завершения. Другими словами, ему не приходится заботиться об ответвлении или откреплении себя от текущего терминала, с которыми разработчикам систем Unix приходится сталкиваться годами.

Во многие традиционные служебные демоны уже включены параметры стиля отладки, которые дают указание главному процессу, чтобы он не ветвился. В качестве примера можно привести демон защищенной оболочки sshd с параметром -D. Заглянув в строфы запуска в файле /etc/init/ssh.conf, можно обнаружить простую конфигурацию запуска демона sshd, которая предотвращает быстрое повторное ветвление и исключает ложный вывод в стандартную ошибку:

```
respawn
respawn limit 10 5
umask 022

# 'sshd -D' leaks stderr and confuses things in conjunction with 'console log'
console none
--snip--

exec /usr/sbin/sshd -D
```

Для заданий, которым необходимо наличие строфы expect, самым распространенным является вариант expect fork. Вот, например, пусковой фрагмент файла /etc/init/cron.conf:

```
expect fork
respawn

exec cron
```

Простой запуск задания, подобный вышеприведенному, обычно указывает на хорошо себя ведущий стабильный демон.

ПРИМЕЧАНИЕ

Стоит прочитать дополнительную информацию о строфе expect на сайте upstart.ubuntu.com, поскольку она напрямую относится к продолжительности действия процессов. Можно, например, отслеживать активность процесса и его системные вызовы, включая fork(), с помощью команды strace.

## 6.5.4. Управление командой Upstart

В дополнение к командам `list` и `status`, описанным в подразделе 6.5.2, можно также использовать команду `initctl`, чтобы управлять командой `Upstart` и ее заданиями. Вам потребуется прочитать страницу руководства initctl(8), но сейчас рассмотрим основы.

Чтобы запустить задание `Upstart`, используйте команду `initctl start`:

```
# initctl start job
```

Чтобы остановить задание, применяйте команду `initctl stop`:

```
# initctl stop job
```

Чтобы перезапустить задание:

```
# initctl restart job
```

Если вам необходимо породить событие для команды `Upstart`, это можно выполнить вручную с помощью такой команды:

```
# initctl emit event
```

Можно также добавить к порожденному событию переменные окружения, указав после события *event* параметры в виде пар *key=value*.

ПРИМЕЧАНИЕ

Невозможно запускать и останавливать отдельные службы, запущенные в режиме совместимости команды Upstart со стандартом System V. Из подраздела 6.6.1 вы больше узнаете о том, как это выполняется в сценариях System V init.

Существует множество способов отключить задание `Upstart`, чтобы оно не стартовало при загрузке системы, однако самым управляемым является следующий: определите имя файла конфигурации задания (обычно это файл `/etc/init/<job>.conf`), а затем создайте новый файл с именем `/etc/init/<job>.override`, который содержит всего одну строку:

```
manual
```

После этого единственным способом запуска указанного задания станет применение команды `initctl start job`.

Основным преимуществом данного метода является его простая обратимость. Чтобы заново задействовать задание при загрузке системы, удалите файл `.override`.

## 6.5.5. Журналы команды Upstart

Существуют два основных типа журналов команды `Upstart`: журналы служб и диагностические общения, которые создает сама команда `Upstart`. Журналы служб

записывают стандартный вывод и стандартную ошибку сценариев и демонов, которые запускают службы. Такие сообщения, хранящиеся в каталоге /var/log/upstart, являются дополнением к стандартным сообщениям, которые может выдавать служба syslog (о сообщениях syslog вы узнаете подробнее в главе 7). Сложно систематизировать записи, попадающие в эти журналы, поскольку нет стандартов. Наиболее часто встречаются сообщения о запуске и остановке, а также сообщения об аварийных ошибках. Многие службы вообще не оставляют никаких сообщений, поскольку они отправляют все в журнал syslog или в собственное средство записи событий.

Собственный диагностический журнал команды Upstart может содержать информацию о том, когда она была запущена и перезагружена, а также некоторую информацию о заданиях и событиях. Этот диагностический журнал поступает в утилиту ядра syslog. В ОС этот журнал можно, как правило, найти в файле /var/log/kern.log, а также во всеохватном файле /var/log/syslog.

В то же время команда Upstart по умолчанию заносит в журнал очень мало записей или совсем ничего, поэтому, если вы желаете увидеть что-либо в журналах, необходимо изменить приоритет журнала команды Upstart. По умолчанию имя приоритета равно message. Чтобы заносить в журнал сведения о событиях и изменениях заданий в работающей системе, поменяйте значение приоритета на info:

```
# initctl log-priority info
```

Помните о том, что это изменение не станет постоянным и будет сброшено после перезагрузки. Чтобы команда Upstart при своем запуске заносила в журнал все, добавьте в качестве параметра загрузки ключ --verbose, как описано в разделе 5.5.

## 6.5.6. Уровни запуска команды Upstart и совместимость со стандартом System V

К настоящему моменту мы затронули несколько областей, в которых команда Upstart поддерживает идею уровней запуска System V, а также отметили, что она обладает возможностью запуска сценариев System V в качестве заданий. Приведу более детальный обзор того, как это выглядит в Ubuntu.

1. Запускается задание rc-sysinit обычно после возникновения событий filesystem и static-network-up. До запуска этого задания уровни запуска отсутствуют.
2. Задание rc-sysinit определяет, на какой уровень запуска перейти. Как правило, это уровень запуска по умолчанию, однако задание может проверить также «старый» файл /etc/inittab или взять значения уровня запуска из параметра ядра (в файле /proc/cmdline).
3. Задание rc-sysinit запускает команду telinit, чтобы изменить уровень запуска. Эта команда порождает событие runlevel, которое задает значение уровня запуска в переменной окружения RUNLEVEL.
4. Команда Upstart получает сигнал о событии runlevel. Несколько заданий настроено на запуск при возникновении события runlevel и установке определенного уровня запуска, вследствие чего команда Upstart приводит их в действие.

5. Одно из заданий, активизированных по уровню запуска, rc, отвечает за запуск системы System V. Чтобы это выполнить, задание rc запускает сценарий /etc/init.d/rc, подобно тому как это выполнила бы команда System V init (см. раздел 6.6).
6. По завершении задания rc команда Upstart может запустить другие задания, когда будет получен сигнал о событии stopped rc (например, задание tty1, о котором шла речь в пункте «Служба: tty1» подраздела 6.5.3).

Обратите внимание на то, что, хотя команда Upstart обходится с уровнями запуска так же, как и с любым другим событием, многие файлы конфигурации заданий в большинстве систем Upstart опираются на уровни запуска.

В любом случае есть критическая точка во время загрузки системы, когда монтируются файловые системы и производится наиболее важная часть инициализации ОС. В этот момент система готова к запуску высокоуровневых системных служб, таких как менеджеры графического дисплея и серверы баз данных. Событие runlevel удобно, как метка для этого момента. Хотя можно было бы настроить команду Upstart на использование любого события в качестве триггера. Затруднение возникает при попытке определить, какие службы запускаются в качестве заданий команды Upstart, а какие запускаются в режиме совместимости с версией System V. Простейший способ выяснить это — заглянуть в ферму ссылок уровня запуска System V (см. подраздел 6.6.2). Например, если уровень запуска равен 2, посмотрите каталог /etc/rc2.d. Все, что в нем находится, работает, вероятно, в режиме совместимости с версией System V.

ПРИМЕЧАНИЕ

Замешательство может вызвать наличие фиктивных сценариев в каталоге /etc/init.d. Для любой службы команды Upstart здесь может также находиться сценарий для такой службы в стиле System V, однако этот сценарий не делает ничего другого, кроме как сообщает вам о том, что служба была конвертирована в задание команды Upstart. Не будет также ссылки на этот сценарий в каталоге ссылок System V. Если вам встретится фиктивный сценарий, выясните имя задания Upstart, а затем используйте команду initctl для управления этим заданием.

# 6.6. Команда System V init

Реализация команды System V init восходит к «юности» Linux. Ее основная идея заключается в поддержке упорядоченной загрузки системы на различные уровни запуска с помощью тщательно подобранной последовательности запуска процессов. Хотя вариант System V сейчас не распространен в большинстве версий ОС для ПК, вы можете встретить команду System V init в версии Red Hat Enterprise Linux, а также во внедренных средах Linux (например, для роутеров и смартфонов).

В типичный состав команды System V init входят два главных компонента: центральный файл конфигурации и большой набор сценариев загрузки системы, дополненный фермой символических ссылок. Все начинается с конфигурационного файла /etc/inittab. Если вы работаете с вариантом System V init, отыщите строку, подобную приведенной ниже, в файле inittab:

```
id:5:initdefault:
```

Она сообщает о том, что по умолчанию устанавливается уровень запуска 5.

Все строки файла inittab построены по следующему шаблону, в котором четыре поля отделяются двоеточиями:

- уникальный идентификатор (короткая строка, как id в приведенном примере);
- применимое значение уровня запуска (или несколько значений);
- действие, которое следует выполнить команде init (в приведенном примере — установить по умолчанию значение 5 для уровня запуска);
- команда на выполнение (необязательна).

Чтобы увидеть, как работают команды в файле inittab, рассмотрим следующую строку:

```
l5:5:wait:/etc/rc.d/rc 5
```

Эта особая строка важна, поскольку она приводит в действие бóльшую часть системной конфигурации и служб. Здесь действие wait определяет, когда и как команда System V init запускает команду: она должна запустить один раз команду /etc/rc.d/rc 5 при переходе на уровень запуска 5, а затем, прежде чем делать что-либо еще, дождаться завершения ее работы. Чтобы «сократить историю», команда rc 5 выполняет все, что находится в каталоге /etc/rc5.d и начинается с числа (в порядке следования чисел).

Ниже приводится еще несколько распространенных действий из файла inittab в дополнение к initdefault and wait.

## Действие respawn

Действие respawn говорит команде init о том, чтобы она запустила следующую дальше команду и, если выполнение команды завершилось, запустила ее снова. Вам, вероятно, встретится подобная строка в файле inittab:

```
1:2345:respawn:/sbin/mingetty tty1
```

Утилиты getty обеспечивают приглашение на вход. Приведенная выше строка используется для первой виртуальной консоли (/dev/tty1), которую вы видите, когда нажимаете сочетание клавиш Alt+F1 или Ctrl+Alt+F1 (см. подраздел 3.4.4). Действие respawn заново выводит приглашение на вход, когда вы выйдете из системы.

## Действие ctrlaltdel

Действие ctrlaltdel управляет тем, что выполняет система, когда вы нажимаете сочетание клавиш Ctrl+Alt+Delete в виртуальной консоли. В большинстве систем это команда перезагрузки, использующая команду shutdown (о которой рассказано в разделе 6.7).

## Действие sysinit

Действие sysinit команда должна выполнить в самую первую очередь, перед переходом на какой-либо из уровней запуска.

ПРИМЕЧАНИЕ

О других доступных действиях можно узнать на странице руководства inittab(5).

## 6.6.1. Команда System V init: командная последовательность запуска

Теперь вы готовы узнать, как команда `System V init` запускает системные службы, прежде чем она позволяет вам войти в систему. Вспомните приведенную выше строку из файла `inittab`:

```
l5:5:wait:/etc/rc.d/rc 5
```

Эта небольшая строка приводит в действие множество других команд. На самом деле символы `rc` являются сокращением от *run commands (запустить команды)*, под которыми многие подразумевают сценарии, программы и службы. Но где расположены эти команды?

Число 5 в этой строке сообщает о том, что речь идет об уровне запуска 5. Вероятно, команды будут находиться в каталоге `/etc/rc.d/rc5.d` или `/etc/rc5.d`. Уровень запуска 1 использует каталог `rc1.d`, уровень запуска 2 — `rc2.d` и т. д. Например, вы можете встретить подобные файлы в каталоге `rc5.d`:

```
S10sysklogd     S20ppp          S99gpm
S12kerneld      S25netstd_nfs   S99httpd
S15netstd_init  S30netstd_misc  S99rmnologin
S18netbase      S45pcmcia       S99sshd
S20acct         S89atd
S20logoutd      S89cron
```

Команда `rc 5` запускает утилиты из каталога `rc5.d`, выполняя команды в такой последовательности:

```
S10sysklogd start
S12kerneld start
S15netstd_init start
S18netbase start
--snip--
S99sshd start
```

Обратите внимание на аргумент `start` в каждой команде. Прописная буква `S` в имени команды означает, что данная команда должна работать в режиме запуска, а число (от 00 до 00) определяет местоположение команды `rc` в последовательности команд. Команды из каталогов `rc*.d` обычно являются сценариями оболочки, запускающими команды из каталога `/sbin` или `/usr/sbin`.

Обычно можно выяснить, что делает конкретная команда, просмотрев ее сценарий с помощью команды `less` или аналогичной ей.

ПРИМЕЧАНИЕ

Некоторые каталоги rc*.d содержат команды, которые начинаются с символа K (для режима останова). В таком случае команда rc запускает команду с аргументом stop вместо аргумента start. Чаще всего команды с символом K будут встречаться вам на уровнях запуска, отвечающих за выключение системы.

Такие команды можно запускать вручную. Однако обычно это осуществляется не с помощью каталогов `rc*.d`, а с использованием каталога `init.d`, о котором речь пойдет ниже.

## 6.6.2. Ферма ссылок команды System V init

Содержимое каталогов rc*.d в действительности является символическими ссылками на файлы, расположенные в другом каталоге, init.d. Если вы намерены взаимодействовать со службами из каталогов rc*.d, добавлять, удалять или изменять службы, вам необходимо понимать такие символические ссылки. Полный список содержимого такого каталога, как rc5.d, обнаруживает подобную структуру:

```
lrwxrwxrwx . . . S10sysklogd -> ../init.d/sysklogd
lrwxrwxrwx . . . S12kerneld -> ../init.d/kerneld
lrwxrwxrwx . . . S15netstd_init -> ../init.d/netstd_init
lrwxrwxrwx . . . S18netbase -> ../init.d/netbase
--snip--
lrwxrwxrwx . . . S99httpd -> ../init.d/httpd
--snip--
```

Большое количество символических ссылок в различных подкаталогах, подобное приведенному, называется *фермой ссылок*. Система Linux содержит эти ссылки, чтобы использовать одинаковые сценарии запуска на всех уровнях запуска. Такая договоренность не является требованием, она лишь упрощает организацию системы.

### Запуск и останов служб

Чтобы вручную запустить или остановить службы, используйте сценарий из каталога init.d. Например, одним из способов ручного запуска веб-сервера httpd является запуск сценария init.d/httpd start. Подобным же образом для остановки работающей службы можно применять аргумент stop (httpd stop, например).

### Изменение последовательности загрузки системы

Изменение последовательности запуска системы в команде System V init обычно выполняется с помощью изменения фермы ссылок. Самое распространенное изменение — запрет работы какой-либо команды из каталога init.d на определенном уровне запуска. Следует внимательно относиться к тому, как это осуществлено. Например, вы могли бы решить удалить символическую ссылку из соответствующего каталога rc*.d. Но будьте осторожны: если вам когда-нибудь понадобится вернуть эту ссылку обратно, у вас могут возникнуть трудности при указании точного имени ссылки. Одним из лучших способов является добавление символа подчеркивания (_) в начало имени ссылки, например так:

```
# mv S99httpd _S99httpd
```

Это приводит к тому, что команда rc игнорирует файл _S99httpd, поскольку его имя не начинается с символа S или K, однако при этом исходное имя по-прежнему очевидно.

Чтобы добавить службу, создайте сценарий, подобный тем, которые находятся в каталоге init.d, а затем создайте символическую ссылку в правильном каталоге rc*.d. Проще всего это выполнить с помощью копирования и изменения одного из сценариев, который вы понимаете (в главе 11 содержится дополнительная информация о сценариях оболочки).

При добавлении службы для ее запуска выберите подходящее место в последовательности загрузки системы. Если служба будет запущена слишком рано, она может не функционировать вследствие зависимости от какой-либо другой службы. Для несущественных служб большинство системных администраторов предпочитает номера из девятого десятка. Тогда службы оказываются размещены после большинства основных системных служб.

## 6.6.3. Утилита run-parts

Механизм, который команда `System V init` использует для запуска сценариев из каталога `init.d`, находит себе применение в различных версиях Linux, вне зависимости от того, используют ли они команду `System V init`. Это утилита `run-parts`, единственной ее задачей является запуск подборки исполняемых команд из указанного каталога в некотором предсказуемом порядке. Можно представлять себе эту команду как исполнителя, запускающего для некоторого каталога команду `ls`, а затем просто выполняющего все команды, которые видит в результатах вывода.

По умолчанию запускаются все команды из каталога, однако зачастую у вас есть возможность выбрать определенные команды и игнорировать остальные. В некоторых версиях системы у вас не будет большого контроля над работающими программами. Например, Fedora поставляется с очень простым вариантом утилиты `run-parts`.

Другие версии, такие как Debian и Ubuntu, располагают более сложной командой `run-parts`. Их функции содержат возможность запуска команд на основе регулярных выражений (например, используя выражение `S[0-9]{2}` для запуска всех сценариев «запуск» из каталога уровня запуска `/etc/init.d`), а также для передачи аргументов командам. Такие возможности позволяют вам запускать и останавливать уровни запуска System V с помощью единственной команды.

Нет необходимости разбираться в деталях использования утилиты `run-parts`; многие даже и не знают о ее существовании. Главное — помнить о том, что она время от времени возникает в сценариях и служит только для того, чтобы запускать команды из указанного каталога.

## 6.6.4. Управление командой System V init

Иногда вам может понадобиться слегка «подтолкнуть» команду `init`, чтобы она переключила уровень запуска, заново считала свою конфигурацию или выключила систему. Для управления командой `System V init` используйте команду `telinit`. Например, для переключения на уровень запуска 3 введите:

```
# telinit 3
```

При переключении уровней запуска команда `init` пытается завершить все процессы, которых нет в файле `inittab` для нового уровня запуска, поэтому будьте бдительны при смене уровня запуска.

Когда необходимо добавить или удалить задания, а также выполнить какие-либо изменения в файле `inittab`, следует сообщить команде `init` о таком изменении

и побудить ее к перезагрузке этого файла. Для этого используется следующая команда:

```
# telinit q
```

Можно также применять команду `telinit s` для переключения в режим одиночного пользователя (см. раздел 6.9).

## 6.7. Выключение системы

Команда `init` управляет выключением и перезапуском системы. Команды для выключения системы одни и те же, вне зависимости от используемого варианта команды `init`. Корректный способ выключения компьютера с Linux заключается в применении команды `shutdown`.

Существуют два основных способа использования команды `shutdown`. При *останове* системы она выключает компьютер и оставляет его в выключенном состоянии. Чтобы немедленно остановить компьютер, запустите следующую команду:

```
# shutdown -h now
```

На большинстве компьютеров и во многих версиях Linux при останове прекращается подача электропитания. Можно также выполнить *перезагрузку* компьютера. Для перезагрузки используйте флаг `-r` вместо `-h`.

Процесс выключения занимает несколько секунд. Никогда не выполняйте сброс или отключение питания на этой стадии.

В приведенном примере параметр `now` задает время выключения. Данный параметр является обязательным, есть много способов указать это время. Если, например, вы желаете, чтобы компьютер был выключен через определенный промежуток времени, можно использовать параметр *+n*, где число *n* задает количество минут, которое должна подождать команда `shutdown`, прежде чем выполнить свою работу. Другие параметры можно увидеть на странице руководства shutdown(8).

Чтобы перезагрузить систему через 10 минут, введите такую команду:

```
# shutdown -r +10
```

В Linux команда `shutdown` уведомляет пользователя о том, что компьютер будет выключен, но реальной работы она выполняет немного. Если вы укажете время, отличающееся от `now`, команда `shutdown` создаст файл с именем `/etc/nologin`. Когда такой файл присутствует, система не позволяет выполнить вход никому, кроме пользователя superuser.

Когда наступает момент выключения системы, команда `shutdown` дает знать команде `init` о начале процесса выключения. В версии systemd это означает активизацию модулей выключения; в версии Upstart — порождение событий выключения; а в версии System V init — изменение уровня запуска с 0 на 6. Вне зависимости от реализации команды `init` или ее конфигурации процедура обычно проходит следующим образом.

1. Команда `init` просит каждый процесс корректно завершиться.
2. Если какой-либо процесс не отвечает через некоторое время, команда `init` прерывает его, попробовав сначала сигнал TERM.
3. Если сигнал TERM не сработал, команда `init` использует сигнал KILL для всех затянувшихся процессов.
4. Система блокирует системные файлы на своих местах и выполняет другую подготовку к выключению.
5. Система демонтирует все файловые системы, кроме корневой.
6. Система заново монтирует корневую файловую систему в режиме «только чтение».
7. Система записывает все буферизованные данные в файловую систему с помощью команды `sync`.
8. На последнем шаге она дает указание ядру о перезагрузке или останове с помощью системного вызова `reboot(2)`. Это может быть выполнено с помощью команды `init` или какой-либо вспомогательной команды вроде `reboot`, `halt` или `poweroff`.

Команды `reboot` и `halt` ведут себя по-разному, в зависимости от того, как они вызваны. По умолчанию эти команды вызывают команду `shutdown` с параметром `-r` или `-h`. Однако если система уже находится на уровне запуска, который соответствует останову или перезагрузке, эти команды приказывают ядру немедленно отключиться. Если вам действительно необходимо спешно выключить компьютер, несмотря на возможные повреждения в результате неорганизованного выключения, используйте параметр `-f`.

## 6.8. Начальная файловая система оперативной памяти

Несмотря на то что процесс загрузки системы Linux довольно прост, один компонент постоянно приводит в замешательство: *initramfs*, или *начальная файловая система оперативной памяти*. Представьте его в виде маленького вставного пространства пользователя, которое появляется перед запуском нормального пользовательского режима. Но сначала поговорим о том, для чего оно существует.

Проблема заключается в доступности различных типов аппаратных средств для хранения данных. Как вы помните, ядро системы Linux не взаимодействует с интерфейсами BIOS или EFI для получения данных с дисков, поэтому для монтирования корневой файловой системы ему нужны драйверы, поддерживающие механизм хранения, заложенный в основу системы. Если, например, корневая файловая система расположена на дисковом массиве RAID, подключенном к контроллеру от стороннего разработчика, то ядру в первую очередь понадобится драйвер для такого контроллера. К сожалению, существует такое множество драйверов для контроллеров устройств хранения данных, что в ядро системы невозможно включить их все, поэтому многие драйверы поставляются в качестве загружаемых модулей. Однако загружаемые модули являются файлами, и, если у ядра не будет смонтированной

в первую очередь файловой системы, оно не сможет загрузить необходимые модули драйверов.

Обходной путь заключается в объединении небольшой подборки модулей драйверов для ядра, а также некоторого количества других утилит в виде архива. Загрузчик системы загружает этот архив в память перед запуском ядра. Во время запуска ядро считывает содержимое архива во временную файловую систему оперативной памяти (initramfs), монтирует ее в корневой каталог и в пользовательском режиме выполняет передачу управления команде `init` в файловой системе initramfs. Затем включенные в состав initramfs утилиты позволяют ядру загрузить необходимые модули драйверов для реальной корневой файловой системы. Наконец, утилиты монтируют реальную корневую файловую систему и запускают настоящую команду `init`.

Реализации этого процесса различны и более сложны. В некоторых версиях команда `init` в файловой системе initramfs — это всего лишь простой сценарий оболочки, который запускает демон `udevd` для загрузки драйверов, а затем монтирует реальный корневой каталог и выполняет в нем команду `init`. В версиях, использующих вариант systemd, как правило, можно увидеть полноценную сборку systemd, которая не использует файлы конфигурации модулей и содержит совсем немного файлов конфигурации демона `udevd`.

Одной из основных характеристик начальной файловой системы оперативной памяти, которая (пока) остается неизменной, является возможность обойти ее, если она вам не нужна. То есть, если ядро содержит все необходимые для монтирования корневой файловой системы драйверы, вы можете изъять начальную файловую систему оперативной памяти из конфигурации загрузчика системы. Если это удается сделать, время загрузки системы сокращается, как правило, на несколько секунд. Попробуйте выполнить это самостоятельно во время загрузки системы, использовав редактор меню загрузчика GRUB для удаления строки `initrd`. Лучше не экспериментировать с изменениями файла конфигурации загрузчика GRUB, поскольку можно совершить ошибку, которую будет трудно исправить. С недавних пор обойти начальную файловую систему оперативной памяти стало труднее, поскольку такие функции, как монтирование по идентификатору UUID, могут быть недоступны для обобщенных версий ядра.

Содержимое начальной файловой системы оперативной памяти легко увидеть, поскольку в большинстве современных систем это всего лишь архивы `cpio` (см. страницу руководства cpio(1)), сжатые с помощью утилиты `gzip`. Сначала отыщите файл архива, заглянув в конфигурацию загрузчика системы (используйте, например, команду `grep` для поиска строк `initrd` в файле конфигурации `grub.cfg`). Затем примените команду `cpio`, чтобы выгрузить содержимое архива в какой-либо временный каталог и исследовать результаты. Например:

```
$ mkdir /tmp/myinitrd
$ cd /tmp/myinitrd
$ zcat /boot/initrd.img-3.2.0-34 | cpio -i --no-absolute-filenames
--snip--
```

Особый интерес представляет «опорная точка» почти в самом конце процесса `init` для начальной файловой системы оперативной памяти. Этот участок отвечает за

удаление содержимого временной файловой системы (чтобы освободить пространство) и окончательное переключение на реальную корневую файловую систему.

Как правило, вам не потребуется создавать пользовательскую начальную файловую систему оперативной памяти, поскольку это довольно кропотливая процедура. Существует множество утилит, предназначенных для создания образов такой файловой системы, и в вашей версии ОС наверняка есть какая-либо из них.

Наиболее распространенными являются утилиты `dracut` и `mkinitramfs`.

ПРИМЕЧАНИЕ

Термин начальная файловая система оперативной памяти (initramfs) относится к такой реализации, которая использует архив cpio в качестве источника для временной файловой системы. Существует устаревшая версия под названием «начальный диск оперативной памяти», или initrd, которая применяет образ диска в качестве основы для временной файловой системы. Этот вариант выходит из употребления, поскольку гораздо проще работать с архивом cpio. Тем не менее вам часто будет встречаться термин initrd применительно к начальной файловой системе оперативной памяти на основе архива cpio. Зачастую (как в приведенном примере) имена файлов и файлы конфигурации по-прежнему содержат слово initrd.

# 6.9. Аварийная загрузка системы и режим одиночного пользователя

Когда в системе происходит что-то неладное, первым средством помощи обычно является загрузка системы с использованием «живого» образа дистрибутива (в большинстве версий образ дистрибутива играет также вторую роль «живого» образа) или выделенного аварийного образа, такого как SystemRescueCd, который можно разместить на сменном носителе. Обычные действия по восстановлению системы заключаются в следующем:

- проверка файловых систем после сбоя;
- переустановка забытого пароля для корневого пользователя;
- устранение ошибок в важных файлах, таких как `/etc/fstab` и `/etc/passwd`;
- восстановление с помощью резервных копий после сбоя системы.

Еще одним вариантом быстрой загрузки до рабочего состояния является *режим одиночного пользователя*. Идея заключается в том, что система быстро загружается до корневой оболочки вместо тщательного запуска всех служб. В версии System V init режиму одиночного пользователя обычно соответствует уровень запуска 1; можно также указать этот режим загрузчику системы с помощью флага `-s`. Чтобы войти в режим одиночного пользователя, вам может потребоваться ввести пароль корневого пользователя.

Самой большой проблемой режима одиночного пользователя является то, что он не слишком комфортен для работы. Почти наверняка не будет доступна сеть (а если и будет, то воспользоваться ею сложно), графический интерфейс пользователя будет отключен, и даже терминал может работать некорректно. По этой причине практически всегда предпочтительнее использовать «живые» образы.

# 7 Конфигурация системы: журнал, системное время, пакетные задания и пользователи

Когда вы в первый раз заглянете в каталог /etc, вы испытаете потрясение. Практически все файлы в определенной степени влияют на работу системы, а часть из них являются фундаментальными.

Основной материал этой главы охватывает те части системы, которые делают доступной инфраструктуру, описанную в главе 4, для инструментов уровня пользователя, рассмотренных в главе 2. В частности, мы будем исследовать следующее:

- конфигурационные файлы, к которым обращаются системные библиотеки для получения информации о сервере и пользователе;
- серверные приложения (иногда называемые *демонами*), запускающиеся при загрузке системы;
- конфигурационные утилиты, которые можно использовать для изменения настроек серверных приложений и файлов конфигурации;
- утилиты администрирования.

Как и в предыдущих главах, здесь практически нет материала о сети, поскольку сеть является отдельным «строительным блоком» системы. В главе 9 вы увидите, куда встраивается сеть.

## 7.1. Структура каталога /etc

Большинство файлов конфигурации Linux располагается в каталоге /etc. Исторически сложилось так, что для каждого приложения здесь один или несколько файлов конфигурации, а поскольку пакетов программ в системе Unix довольно много, каталог /etc быстро наполняется файлами.

Такой подход сталкивается с двумя проблемами: в работающей системе сложно найти конкретные файлы конфигурации и выполнять обслуживание системы, сконфигурированной подобным образом. Если, например, вы желаете изменить конфигурацию системного журнала, необходимо отредактировать файл /etc/syslog.conf.

Однако после этого, когда произойдет обновление системы, ваши пользовательские настройки могут быть утрачены.

Основной тенденцией последних лет является размещение файлов системной конфигурации в подкаталогах каталога /etc, как вы уже видели на примере каталогов загрузки системы (/etc/init для варианта Upstart и /etc/systemd для systemd). В каталоге /etc по-прежнему есть несколько отдельных файлов конфигурации, но если вы запустите команду ls -F /etc, вы увидите, что эти элементы теперь в основном являются подкаталогами.

Чтобы справиться с проблемой перезаписывания файлов конфигурации, можно поместить пользовательские настройки в отдельных файлах внутри подкаталогов конфигурации вроде тех, которые присутствуют в каталоге /etc/grub.d.

Какие типы файлов конфигурации можно обнаружить в каталоге /etc? Основной принцип такой: настраиваемая конфигурация для одного компьютера, например информация о пользователе (/etc/passwd) и параметры сети (/etc/network), попадает в каталог /etc. Однако общие параметры приложений, например установки по умолчанию для пользовательского интерфейса, не располагаются в каталоге /etc. Зачастую файлы системной конфигурации, которые не подлежат настройке, находятся где-либо в другом месте, как это сделано для подготовленных файлов модулей systemd в каталоге /usr/lib/systemd.

Вы уже видели некоторые файлы конфигурации, которые имеют отношение к загрузке системы. Сейчас мы рассмотрим типичную системную службу и способы просмотра и настройки ее конфигурации.

# 7.2. Системный журнал

Большинство системных приложений передает свой диагностический вывод в службу syslog. Традиционный демон syslogd ожидает сообщения и в зависимости от типа полученного сообщения направляет вывод в файл, на экран, пользователям или в виде какой-либо комбинации перечисленного, но может и игнорировать сообщение.

## 7.2.1. Системный регистратор

Системный регистратор является одной из важнейших частей системы. Когда что-либо происходит не так и вы не знаете, с чего начать, загляните сначала в файлы системного журнала. Вот пример сообщения из него:

```
Aug 19 17:59:48 duplex sshd[484]: Server listening on 0.0.0.0 port 22.
```

В большинстве версий Linux используется новая версия службы syslogd под названием rsyslogd, которая выполняет намного больше, чем простая запись сообщений в файлы. Например, можно использовать ее для загрузки модуля, чтобы отправлять сообщения журнала в базу данных. Однако, приступая к изучению системных журналов, проще всего начать с файлов журналов, обычно хранящихся в каталоге /var/log. Просмотрите несколько таких файлов — когда вы будете знать, как они выглядят, вы будете готовы выяснить, как они здесь появились.

Многие из файлов в каталоге /var/log обслуживаются не с помощью системного регистратора. Единственный способ точно установить, какие из них принадлежат службе rsyslogd, — посмотреть ее файл конфигурации.

## 7.2.2. Файлы конфигурации

Основным файлом конфигурации службы rsyslogd является /etc/rsyslog.conf, но некоторые настройки вы обнаружите также в других каталогах, например /etc/rsyslog.d. Формат конфигурации представляет собой смесь обычных правил и специфичных для службы rsyslog расширений. Один из признаков такой: если что-либо начинается с символа доллара ($), то это расширение.

Традиционное правило состоит из *селектора* и *действия*, которые описывают, как перехватывать сообщения журнала и куда их направлять (пример 7.1).

**Пример 7.1.** Правила службы syslog

```
kern.*                      /dev/console
*.info;authpriv.none❶       /var/log/messages
authpriv.*                  /var/log/secure,root
mail.*                      /var/log/maillog
cron.*                      /var/log/cron
*.emerg                     *❷
local7.*                    /var/log/boot.log
```

Селектор располагается слева. Это тип информации, которая должна быть занесена в журнал. Список в правой части содержит действия: куда отправлять журнал. Большинство действий из примера 7.1 — обычные файлы, но есть некоторые исключения. Так, например, действие /dev/console ссылается на специальное устройство для системной консоли, действие root означает отправку сообщения пользователю superuser, если он подключен, а действие * означает отправку сообщения всем пользователям, находящимся сейчас в системе. Можно также отправлять сообщения другому сетевому хосту с помощью параметра @*host*.

### Источник и приоритет

Селектор является шаблоном, которому удовлетворяют *источник* и *приоритет* сообщений журнала. Источник — это общая категория сообщения (см. полный список источников на странице rsyslog.conf(5) руководства).

Функция большинства источников достаточно хорошо видна из их имен. Файл конфигурации, приведенный в примере 7.1, относится к сообщениям, источниками которых являются службы kern, authpriv, mail, cron и local7. В этом же списке звездочкой отмечен (❷) джокерный символ, который перехватывает вывод, относящийся ко всем источникам.

*Приоритет* следует после точки ( . ) за источником. Приоритеты располагаются в таком порядке, от низшего к высшему: debug, info, notice, warning, err, crit, alert или emerg.

ПРИМЕЧАНИЕ

Чтобы исключить сообщения журнала от какого-либо источника, укажите в файле rsyslog.conf приоритет none, как отмечено символом ❶ в примере 7.1.

Когда вы указываете приоритет в селекторе, служба rsyslogd отправляет сообщения с данным приоритетом *и приоритетами выше указанного* по назначению в данной строке. То есть в примере 7.1 селектор *.info в строке с символом ❶ в действительности перехватывает большинство сообщений журнала и помещает их в файл /var/log/messages, поскольку приоритет info является низким.

## Расширенный синтаксис

Синтаксис службы rsyslogd расширяет традиционный синтаксис службы syslogd. Расширения конфигурации называются *директивами* и обычно начинаются с символа $. Одно из самых распространенных расширений позволяет вам загружать дополнительные файлы конфигурации. Попробуйте найти в вашем файле rsyslog.conf директиву, подобную приведенной ниже. Она вызывает загрузку всех файлов конфигурации .conf, расположенных в каталоге /etc/rsyslog.d:

```
$IncludeConfig /etc/rsyslog.d/*.conf
```

Названия большинства других расширенных директив не требуют дополнительных объяснений. Например, такие директивы относятся к работе с пользователями и правами доступа:

```
$FileOwner syslog
$FileGroup adm
$FileCreateMode 0640
$DirCreateMode 0755
$Umask 0022
```

ПРИМЕЧАНИЕ

Дополнительные расширения в файлах конфигурации службы rsyslogd определяют шаблоны и каналы вывода. Если вам необходимо их использовать, страница руководства rsyslogd(5) предоставит достаточно объемную информацию, однако онлайн-документация является более полной.

## Устранение неполадок

Один из простейших способов проверить системный регистратор — вручную отправить сообщение журнала с помощью команды logger, как показано ниже:

```
$ logger -p daemon.info something bad just happened
```

Со службой rsyslogd возникает мало проблем. Наиболее часто это случается, когда конфигурация не соответствует некоторому источнику или приоритету, а также при заполнении разделов диска файлами журналов. В большинстве версий системы Linux файлы в каталоге /var/log укорачиваются с помощью автоматического вызова утилиты logrotate или подобной ей, однако, если в течение короткого интервала времени возникнет слишком много сообщений, по-прежнему есть вероятность заполнить ими весь диск или очень сильно загрузить систему.

ПРИМЕЧАНИЕ

Журналы, которые перехватывает служба rsyslogd, — не единственные записываемые различными частями системы. Мы обсуждали в главе 6 сообщения журнала запуска, которые отслеживают команды systemd и Upstart, однако вы обнаружите множество других источников, например веб-сервер Apache, который, как правило, ведет собственный журнал доступа и ошибок. Чтобы найти такие журналы, загляните в конфигурацию сервера.

### Журналы: прошлое и будущее

Служба `syslog` развивается с течением времени. Ранее, например, существовал демон `klogd`, который перехватывал диагностические сообщения ядра для службы `syslogd`. Именно эти сообщения вы можете увидеть с помощью команды `dmesg`. Эта возможность была внедрена в версию `rsyslogd`.

Практически наверняка системные журналы Linux в будущем изменятся. В Unix для системных журналов никогда не существовало настоящего стандарта. Сейчас предпринимаются попытки изменить эту ситуацию.

# 7.3. Файлы управления пользователями

Системы Unix позволяют независимо работать нескольким пользователям. На уровне ядра пользователи являются всего лишь идентификаторами *(пользовательскими ID)*, но поскольку гораздо проще запомнить имя, чем номер, то при управлении Linux вы будете иметь дело с *именами пользователей* (или *зарегистрированными именами*). Имена пользователей существуют только в пространстве пользователя, поэтому любому приложению, работающему с именем пользователя, как правило, необходима возможность сопоставления имени пользователя его идентификатору ID, если приложение собирается сослаться на пользователя при обращении к ядру.

## 7.3.1. Файл /etc/passwd

Простой текстовый файл `/etc/passwd` содержит соответствия имен пользователей идентификаторам ID. Он может выглядеть так (пример 7.2).

**Пример 7.2.** Перечень пользователей в файле `/etc/passwd`

```
root:x:0:0:Superuser:/root:/bin/sh
daemon:*:1:1:daemon:/usr/sbin:/bin/sh
bin:*:2:2:bin:/bin:/bin/sh
sys:*:3:3:sys:/dev:/bin/sh
nobody:*:65534:65534:nobody:/home:/bin/false
juser:x:3119:1000:J. Random User:/home/juser:/bin/bash
beazley:x:143:1000:David Beazley:/home/beazley:/bin/bash
```

Каждая строка представляет одного пользователя и состоит из семи полей, разделенных двоеточиями. Эти поля таковы.

- Имя пользователя.
- Зашифрованный пароль пользователя. В большинстве систем Linux пароль в действительности не хранится в файле `passwd`, а помещается в файл `shadow` (см. подраздел 7.3.3). Формат файла `shadow` похож на формат файла `passwd`, однако у обычных пользователей нет разрешения на чтение файла `shadow`. Второе поле в файле `passwd` или `shadow` является зашифрованным паролем, он выглядит как нечитаемый набор символов, например `d1CVEWiB/oppc`. Пароли в системе Unix никогда не хранятся в виде простого текста.

  Символ `x` во втором поле файла `passwd` говорит о том, что зашифрованный пароль хранится в файле `shadow`. Звездочка (*) сообщает, что этот пользователь не может

совершать вход в систему, а если это поле пусто (то есть вы видите два двоеточия подряд, ::), то для входа в систему пароля не требуется. Остерегайтесь пустых паролей. Никогда не следует регистрировать пользователя без пароля.

- *Идентификатор пользователя* (UID), который представляет данного пользователя ядру. Можно создать две записи с одинаковым идентификатором пользователя, но это будет сбивать с толку и вас, и ваше программное обеспечение. Старайтесь использовать уникальные идентификаторы.
- *Идентификатор группы* (GID) представляет собой одну из нумерованных записей в файле /etc/group. Группы задают права доступа к файлам и кое-что еще. Данная группа называется также *первичной группой* пользователя.
- Реальное имя пользователя (часто называется полем *GECOS*). В этом поле вы можете встретить запятые, которые отделяют номер комнаты и номер телефона.
- Домашний каталог пользователя.
- Оболочка пользователя (команда, которая запускается при работе пользователя в терминале).

На рис. 7.1 обозначены различные поля в одной из записей примера 7.2.

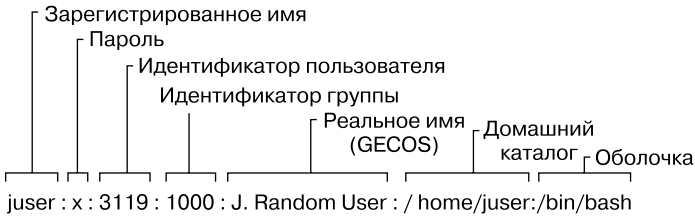


**Рис. 7.1.** Запись из файла паролей

Синтаксис файла /etc/passwd является довольно строгим и не допускает применения комментариев или пустых строк.

ПРИМЕЧАНИЕ

Пользователь, указанный в файле /etc/passwd, и соответствующий ему домашний каталог известны под обобщающим названием «учетная запись».

## 7.3.2. Особые пользователи

В файле /etc/passwd можно найти несколько особых пользователей. Пользователь superuser (корневой) всегда обладает идентификаторами UID 0 и GID 0, как в примере 7.2. У некоторых пользователей, таких как *демоны*, нет привилегий на вход в систему. Пользователь nobody («никто») является непривилегированным. Некоторые процессы запускаются как пользователь nobody, поскольку такой пользователь не может записывать что-либо в систему.

Пользователи, которые не могут входить в систему, именуются *псевдопользователями*. Хотя они и не могут войти в систему, система может запускать процессы с их идентификаторами. Такие пользователи, как nobody, обычно создаются в целях безопасности.

## 7.3.3. Файл /etc/shadow

Файл паролей shadow (/etc/shadow) в системе Linux обычно содержит информацию об аутентификации пользователя, включая зашифрованные пароли и сведения об окончании срока действия паролей, соответствующих пользователям из файла /etc/passwd.

Файл shadow был введен для обеспечения более гибкого (и более защищенного) способа хранения паролей. Он содержит набор библиотек и утилит, многие из которых скоро будут заменены на модули PAM (см. раздел 7.10). Вместо того чтобы вводить совершенно новый набор файлов для системы Linux, стандарт PAM использует файл /etc/shadow, а не конкретные файлы конфигурации, такие как /etc/login.defs.

## 7.3.4. Управление пользователями и паролями

Обычные пользователи взаимодействуют с файлом /etc/passwd с помощью команды passwd. По умолчанию команда passwd изменяет пароль данного пользователя, но можно также указать флаг -f, чтобы изменить реальное имя пользователя, или флаг -s, чтобы изменить оболочку пользователя на одну из перечисленных в файле /etc/shells. Можно также использовать команды chfn и chsh для изменения реального имени и оболочки. Команда passwd относится к разряду suid-root-команд, поскольку только пользователь superuser может изменять файл /etc/passwd.

### Изменение файла /etc/passwd с правами пользователя superuser

Поскольку файл /etc/passwd является простым текстовым файлом, пользователь superuser может выполнить изменения в любом текстовом редакторе. Чтобы создать пользователя, просто добавьте соответствующую строку и создайте домашний каталог для нового пользователя; чтобы его удалить, выполните обратные действия. Однако для редактирования этого файла лучше использовать команду vipw, которая в целях дополнительной предосторожности создает резервную копию файла /etc/passwd и блокирует его, пока вы занимаетесь редактированием. Чтобы отредактировать файл /etc/shadow вместо файла /etc/passwd, воспользуйтесь командой vipw -s. Хотя вам, скорее всего, это никогда не потребуется.

В большинстве организаций неодобрительно относятся к прямому редактированию файла passwd, поскольку при этом очень легко совершить ошибку. Гораздо проще (и безопаснее) выполнять изменения с помощью отдельных команд, доступных в терминале или в графическом интерфейсе пользователя.

Например, чтобы указать пароль для пользователя, запустите команду passwd *user* с правами администратора. Используйте команды adduser и userdel для добавления или удаления пользователей.

## 7.3.5. Работа с группами

*Группы* в Unix предназначены для организации совместного доступа определенных пользователей к файлам, при этом доступ всем остальным пользователям запрещен. Идея заключается в установке битов на чтение или запись для конкретной

группы, исключая кого-либо еще. В свое время такая функция была важна, поскольку одним компьютером пользовалось несколько человек, однако за последние годы эта функция становится менее важной, так как рабочие станции реже предоставляются для совместного доступа.

Файл /etc/group определяет идентификаторы групп (подобные тем, которые находятся в файле /etc/passwd file). Ниже приведен пример такого файла (пример 7.3).

**Пример 7.3.** Пример файла /etc/group

```
root:*:0:juser
daemon:*:1:
bin:*:2:
sys:*:3:
adm:*:4:
disk:*:6:juser,beazley
nogroup:*:65534:
user:*:1000:
```

Так же как в файле /etc/passwd, каждая строка файла /etc/group является набором полей, разделенных двоеточиями. Эти поля в каждой записи следуют в таком порядке.

- **Имя группы.** Отображается, если вы запустите такую команду, как ls -l.
- **Пароль группы.** Практически не используется, поэтому вам не следует его применять (вместо него воспользуйтесь командой sudo). Указывайте * или любое другое значение по умолчанию.
- **Идентификатор группы (число).** Идентификатор GID должен быть уникальным в файле group. Это число указывается также в поле группы пользователя в записи файла /etc/passwd для данного пользователя.
- **Необязательный перечень пользователей, принадлежащих данной группе.** В дополнение к перечисленным здесь пользователям в данную группу будут также включены пользователи с соответствующим идентификатором группы, указанном в файле passwd.

На рис. 7.2 отмечены поля в записи из файла group.

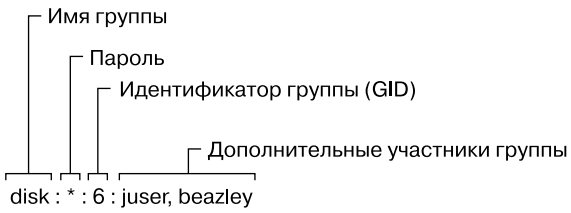


**Рис. 7.2.** Запись из файла group

Чтобы увидеть, в какие группы вы входите, запустите команду groups.

ПРИМЕЧАНИЕ

В Linux для каждого добавляемого пользователя часто создается новая группа, имя которой совпадает с именем пользователя.

# 7.4. Команды getty и login

Команда getty прикрепляется к терминалам для отображения строки приглашения. В большинстве версий Linux команда getty не усложнена, поскольку система использует ее только для входа в виртуальных терминалах. В списке процессов она обычно выглядит следующим образом (например, после запуска в терминале /dev/tty1):

```
$ ps ao args | grep getty
/sbin/getty 38400 tty1
```

В этом примере число 38400 является значением скорости двоичной передачи в бодах. Для некоторых команд getty значение этой скорости необязательно. Виртуальные терминалы игнорируют его; оно присутствует только для обратной совместимости с программным обеспечением, которое подключено к реальным последовательным каналам.

После ввода зарегистрированного имени команда getty сменится на команду login, которая запросит пароль. Если пароль указан верно, команда login сменится с помощью команды exec() на оболочку. В противном случае будет выдано сообщение о некорректном входе в систему.

Теперь вы знаете, что делают команды getty и login, но вам, вероятно, никогда не приходилось конфигурировать или изменять их. На самом деле вам даже вряд ли понадобится их использовать, поскольку теперь большинство пользователей входит в систему либо с помощью графического интерфейса, вроде gdm, либо удаленно, с помощью оболочки SSH, и ни в одном из этих способов не задействованы команды getty или login. Основная доля работы по аутентификации при входе в систему осуществляется с использованием стандарта PAM (см. раздел 7.10).

# 7.5. Настройка времени

Компьютеры с Unix зависят от точного хронометрирования. Ядро обслуживает *системные часы*, с которыми сверяются, например, при запуске команды date. Системные часы можно также настроить с помощью команды date, но такая идея является, как правило, пагубной, поскольку вы никогда не сможете узнать время абсолютно точно. Системные часы должны быть близки к точному времени, насколько это возможно.

Аппаратные средства ПК содержат *часы реального времени* (RTC, real-time clock) с питанием от батареи. Это не самые точные часы в мире, но они лучше, чем совсем ничего. Ядро обычно устанавливает время на основе показаний RTC во время загрузки системы, и можно выполнить сброс показания системных часов до текущего значения аппаратного времени с помощью команды hwclock. Настройте аппаратные часы на время UTC (Universal Coordinated Time, всеобщее скоординированное время), чтобы избежать различных сложностей, связанных с часовыми поясами или переходом на летнее время. Можно настроить часы RTC в соответствии с UTC-часами ядра с помощью следующей команды:

```
# hwclock --hctosys --utc
```

К сожалению, ядро хранит время еще хуже, чем часы RTC, и поскольку компьютеры Unix часто работают в течение нескольких месяцев или лет после единственной загрузки, возрастает смещение по времени. *Смещение по времени* — это текущая разность между временем ядра и истинным временем (которое определено по атомным или каким-либо еще очень точным часам).

Не следует пытаться исправлять смещение по времени с помощью команды `hwclock`, поскольку системные события, основанные на времени, могут быть потеряны или искажены. Можно было бы запустить утилиту вроде `adjtimex`, чтобы аккуратно обновить показания часов, но обычно правильность системного времени лучше всего поддерживать с помощью сетевого демона времени (см. подраздел 7.5.2).

## 7.5.1. Представление времени в ядре и часовые пояса

Системные часы ядра представляют текущее время в виде количества секунд, протекших с полуночи 1 января 1970 года по времени UTC. Чтобы увидеть это значение для данного момента, запустите такую команду:

```
$ date +%s
```

Чтобы представить это число в приемлемом для человека формате, команды из пространства пользователя переводят его в местное время с учетом перехода на летнее время, а также других необычных обстоятельств (таких как проживание в штате Индиана[1]). Местный часовой пояс настраивается с помощью файла `/etc/localtime`. Не пытайтесь заглянуть в него, поскольку этот файл является двоичным.

Файлы часовых поясов для вашей системы расположены в каталоге `/usr/share/zoneinfo`. Этот каталог содержит множество файлов часовых поясов и псевдонимов для них. Чтобы настроить часовой пояс вручную, скопируйте один из таких файлов из каталога `/usr/share/zoneinfo` в каталог `/etc/localtime`, или создайте символическую ссылку, или же измените его с помощью инструмента для работы с часовыми поясами. Команда `tzselect` может помочь вам при определении файла часового пояса.

Чтобы использовать лишь на один сеанс оболочки часовой пояс, который отличается от установленного в системе по умолчанию, укажите в переменной окружения `TZ` имя файла из каталога `/usr/share/zoneinfo` и проверьте изменения, например, так:

```
$ export TZ=US/Central
$ date
```

Как и в случае с другими переменными окружения, можно указать часовой пояс только на время работы единственной команды:

```
$ TZ=US/Central date
```

---

[1] Бóльшая часть штата, включая столицу, входит в Восточную зону времени, где принято так называемое Северо-Американское восточное стандартное время. В некоторых округах на западе штата принято так называемое Центральное стандартное время. — *Примеч. пер.*

### 7.5.2. Сетевое время

Если ваш компьютер постоянно подключен к сети Интернет, можно запустить демон NTP (Network Time Protocol, протокол сетевого времени), чтобы настраивать время с помощью удаленного сервера. Во многие версии ОС встроена поддержка демона NTP, однако он может быть не включен по умолчанию. Может потребоваться установка пакета `ntpd`, чтобы привести его в действие.

Если вам необходимо выполнить конфигурацию вручную, справочную информацию можно найти на основной странице NTP (http://www.ntp.org/), но если вы предпочитаете не копаться в документации, выполните следующее.

1. Отыщите ближайший к вам сервер NTP, узнав его от поставщика интернет-услуг или на странице ntp.org.
2. Поместите имя сервера времени в файл `/etc/ntpd.conf`.
3. Запустите во время загрузки системы команду `ntpdate` *`server`*.
4. После команды `ntpdate` запустите во время загрузки системы команду `ntpd`.

Если ваш компьютер не подключен к Интернету постоянно, можно использовать демон вроде `chronyd`, чтобы поддерживать время, когда подключение отсутствует.

Можно также настроить аппаратные часы на основе сетевого времени, чтобы обеспечить связность отсчета времени в системе при ее перезагрузке. Во многих версиях ОС это происходит автоматически. Чтобы выполнить это, возьмите системное время из сети с помощью команды `ntpdate` (или `ntpd`), а затем запустите команду, которую вы уже видели ранее:

```
# hwclock --systohc --utc
```

## 7.6. Планирование повторяющихся задач с помощью службы cron

Служба `cron` в Unix повторно запускает команды на основе постоянного расписания. Большинство опытных администраторов считают службу `cron` особо важной для системы, поскольку она способна выполнять автоматическое обслуживание системы. Например, она выполняет запуск утилит для чистки файлов журналов, чтобы ваш жесткий диск не переполнялся старыми файлами журналов. Вам следует знать, как использовать службу `cron`, поскольку она, безусловно, полезна.

С помощью службы `cron` можно запустить любую команду в любое удобное для вас время. Команда, запущенная с помощью службы `cron`, называется *заданием* `cron`. Чтобы создать задание cron, необходимо внести запись в *файл* `crontab`, обычно с помощью команды `crontab`.

Например, для настройки ежедневного запуска команды `/home/juser/bin/spmake` в 9:15 утра запись выглядит так:

```
15 09 * * * /home/juser/bin/spmake
```

Пять разделенных пробелами полей в начале этой строки определяют запланированное время (см. также рис. 7.3). Эти поля обозначают следующее:

- минута (от 0 до 59). Приведенное выше задание `cron` настроено на запуск в 15-ю минуту;
- час (от 0 до 23). Задание настроено на запуск в 9-й час;
- день месяца (от 1 до 31);
- месяц (от 1 до 12);
- день недели (от 0 до 7). Числа 0 и 7 соответствуют воскресенью.

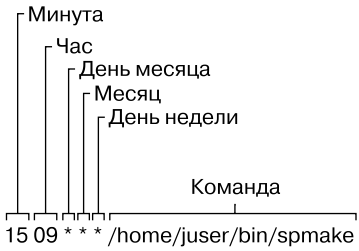


**Рис. 7.3.** Запись из файла crontab

Звездочка (*) в любом поле соответствует любому значению. В приведенном примере команда `spmake` запускается ежедневно, поскольку поля для дня месяца, месяца и дня недели заполнены звездочками, которые служба `cron` интерпретирует как «запускать данное задание каждый день, каждый месяц и каждый день недели».

Чтобы запускать команду `spmake` только на 14-й день каждого месяца, можно воспользоваться такой строкой в файле `crontab`:

```
15 09 14 * * /home/juser/bin/spmake
```

Можно указывать более одного значения в каждом поле. Чтобы, например, запускать команду по 5-м и 14-м числам каждого месяца, следует ввести числа `5,14` в третье поле:

```
15 09 5,14 * * /home/juser/bin/spmake
```

ПРИМЕЧАНИЕ

Если задание cron создает стандартный вывод, ошибку или некорректно завершает работу, служба cron отправит вам по электронной почте сообщение об этом. Если такие электронные письма вам надоедают, перенаправьте вывод в устройство /dev/null или в какой-либо файл журнала.

Страница руководства crontab(5) содержит полную информацию о формате файла `crontab`.

## 7.6.1. Установка файлов crontab

У каждого пользователя может быть свой файл `crontab`. Это значит, что в системе может быть несколько таких файлов, которые, как правило, располагаются в каталоге `/var/spool/cron/crontabs`. Обычным пользователям не разрешено выполнять запись в данный каталог. Пользовательский файл `crontab` можно разместить, просмотреть, отредактировать и удалить с помощью команды `crontab`.

Простейший способ установить файл crontab — поместить записи в какой-либо файл, а затем воспользоваться командой crontab *file*, чтобы установить этот файл в качестве текущего файла crontab. Команда crontab проверяет формат файла, чтобы убедиться в отсутствии ошибок в нем. Чтобы вывести список заданий cron, запустите команду crontab -l. Чтобы удалить файл crontab, воспользуйтесь командой crontab -r.

Тем не менее, после того как вы создали начальный файл crontab, использование временных файлов при дальнейшем его редактировании может вызвать неудобства. Вместо этого можно отредактировать и установить файл crontab за один шаг с помощью команды crontab -e. Если вы допустите ошибку, команда crontab сообщит вам о том, где она находится, и предложит вам повторно отредактировать файл.

## 7.6.2. Системные файлы crontab

Чтобы при планировании повторяющихся системных задач не использовать файл crontab для пользователя с правами superuser, в системах Linux обычно предусмотрен файл /etc/crontab. Не применяйте команду crontab для редактирования этого файла, поскольку в его записях присутствует дополнительное поле, вставленное перед командой, предназначенной для запуска. В этом поле указан пользователь, который должен запустить задание. Вот, например, задание cron, которое определено в файле /etc/crontab и будет запускаться в 6:42 утра с правами корневого пользователя (root, отмечен символом ❶):

```
42 6 * * * root❶ /usr/local/bin/cleansystem > /dev/null 2>&1
```

ПРИМЕЧАНИЕ

В некоторых версиях системные файлы crontab хранятся в каталоге /etc/cron.d. Такие файлы могут быть названы как угодно, но все они обладают тем же форматом, что и файл /etc/crontab.

## 7.6.3. Будущее службы cron

Утилита cron является одним из старейших компонентов системы Linux, она используется уже десятки лет (задолго до самой Linux). За эти годы формат конфигурации изменился ненамного. Сегодня пытаются выполнить замену данной утилиты.

Предлагаемые замены на самом деле являются лишь частями новых версий команды init: для варианта systemd это модули таймера, а для варианта Upstart идея заключается в возможности создания повторяющихся событий для запуска заданий. В конечном итоге оба варианта команды init могут запускать задачи от имени любого пользователя; они обладают также некоторыми преимуществами, такими как специальный вход в систему.

Однако реальность такова, что ни версия systemd, ни версия Upstart не обладают в данный момент всеми возможностями утилиты cron. Более того, когда они будут способны к этому, потребуется обратная совместимость для поддержки всего, что основано на службе cron. По этим причинам формат cron вряд ли исчезнет в ближайшее время.

# 7.7. Планирование единовременных задач с помощью службы at

Чтобы запустить задание в будущем один раз без помощи службы `cron`, воспользуйтесь службой `at`. Например, чтобы запустить команду `myjob` в 22:30 вечера, введите такую команду:

```
$ at 22:30
at> myjob
```

Завершите ввод, нажав сочетание клавиш Ctrl+D. Утилита `at` считывает команды из стандартного ввода.

Чтобы убедиться в том, что задание запланировано, используйте команду `atq`. Чтобы его удалить, запустите команду `atrm`. Можно также указать день для выполнения задания, добавив дату в формате ДД.ММ.ГГ, например, так: `22:30 30.09.15`.

О команде `at` больше нечего добавить. Хотя она используется нечасто, она может пригодиться тогда, когда вам необходимо сообщить системе, чтобы она выключилась в будущем.

# 7.8. Идентификаторы пользователей и переключение между пользователями

Мы рассказывали о том, каким образом setuid-команды вроде `sudo` и `su` позволяют вам сменить пользователя, а также упомянули о системных компонентах типа `login`, которые контролируют пользовательский доступ. Возможно, вам интересно, как работают эти составляющие и какую роль играет ядро в переключении между пользователями.

Существует два способа изменить идентификатор пользователя, и оба они используются ядром. Первый способ — с помощью исполняемого файла `setuid`, о котором рассказано в разделе 2.17. Второй способ — используя семейства системных вызовов `setuid()`. Есть несколько различных версий таких вызовов, которые охватывают всевозможные идентификаторы пользователей, связанные с процессами, как вы узнаете далее.

Ядро обладает набором правил относительно того, что дозволено процессу, а что — нет. Приведу три основных правила.

- Процесс, запущенный как корневой (`userid 0`), может использовать команду `setuid()`, чтобы стать любым другим пользователем.
- На процесс, запущенный не в качестве корневого, накладываются строгие ограничения по использованию команды `setuid()`; в большинстве случаев он не может ее использовать.
- Любой процесс может выполнить setuid-команду, если у него есть соответствующие права доступа к файлам.

ПРИМЕЧАНИЕ

Переключение между пользователями никак не затрагивает пароли или имена пользователей. Эти понятия относятся исключительно к пространству пользователя, как вы уже видели на примере файла /etc/passwd в подразделе 7.3.1. Дополнительные подробности о том, как это работает, — в разделе 7.9.

**Принадлежность процессов, эффективный, реальный и сохраненный идентификатор пользователя.** Наш рассказ об идентификаторах пользователя до сего момента был упрощенным. В действительности каждый процесс снабжен несколькими идентификаторами пользователя. Мы описали *эффективный идентификатор пользователя* (euid), который определяет права доступа для процесса. Второй идентификатор пользователя, *реальный идентификатор пользователя* (ruid), указывает на инициатора процесса. При запуске setuid-команды система Linux устанавливает для владельца команды значение эффективного идентификатора пользователя во время исполнения, но она сохраняет исходный идентификатор в качестве реального идентификатора пользователя.

В современных системах различие между эффективным и реальным идентификаторами пользователя приводит к такой путанице, что большая часть документации, посвященной принадлежности процессов, является неверной.

Представляйте себе эффективный идентификатор пользователя как *исполнителя*, а реальный идентификатор — как *владельца*. Реальный идентификатор пользователя определяет того пользователя, который может взаимодействовать с запущенным процессом, и, что наиболее важно, пользователя, который может прерывать процесс и отправлять ему сигналы. Если, например, пользователь A запускает новый процесс от имени пользователя B (на основе разрешений setuid), то пользователь A по-прежнему владеет этим процессом и может его прервать.

В обычных системах Linux у большинства процессов совпадают эффективный и реальный идентификаторы пользователя. По умолчанию команда `ps` и другие команды диагностики системы показывают эффективный идентификатор пользователя. Чтобы увидеть оба идентификатора в своей системе, попробуйте ввести приведенную ниже команду, но не удивляйтесь, если вы обнаружите, что для всех процессов окажутся одинаковыми два столбца с идентификаторами пользователя:

```
$ ps -eo pid,euser,ruser,comm
```

Чтобы создать исключение только для того, чтобы увидеть различные значения в столбцах, попробуйте поэкспериментировать: создайте setuid-копию команды `sleep`, запустите эту копию на несколько секунд, а затем выполните приведенную выше команду `ps` в другом окне, прежде чем копия прекратит работу.

Чтобы усугубить путаницу, в дополнение к реальному и эффективному идентификаторам пользователя есть также *сохраненный идентификатор пользователя* (для которого обычно не используют сокращение). Во время выполнения процесс может переключить свой эффективный идентификатор пользователя на реальный или сохраненный. Чтобы вещи стали еще более сложными, в системе Linux присутствует *идентификатор пользователя файловой системы* (fsuid), который определяет пользователя, имеющего доступ к файловой системе. Однако этот идентификатор используется редко.

**Типичное поведение команды setuid.** Идея, заложенная в реальный идентификатор пользователя, может противоречить вашему предшествующему опыту. Почему вам не приходится часто иметь дело с другими идентификаторами пользователя? Например, если после запуска процесса с помощью команды `sudo` вам необходимо его остановить, вы также используете команду `sudo`; обычный пользователь не может остановить процесс. Не должен ли в таком случае обычный пользователь обладать реальным идентификатором пользователя, чтобы получить правильные права доступа?

Причина такого поведения заключается в том, что команда `sudo` и многие другие setuid-команды явным образом изменяют эффективный и реальный идентификаторы пользователя с помощью системных вызовов команды `setuid()`. Эти команды поступают так потому, что зачастую возникают непреднамеренные побочные эффекты и проблемы с доступом, если не совпадают все идентификаторы пользователя.

ПРИМЕЧАНИЕ

Если вам интересны подробности и правила, относящиеся к переключению идентификаторов пользователя, прочитайте страницу руководства setuid(2), а также загляните на страницы, перечисленные в секции SEE ALSO («см. также»). Для различных ситуаций существует множество разных системных вызовов.

Некоторым командам не нравится корневой реальный идентификатор пользователя. Чтобы запретить команде `sudo` изменение реального идентификатора пользователя, добавьте следующую строку в файл `/etc/sudoers` (и остерегайтесь побочных эффектов на другие команды, которые вы желаете запускать с корневыми правами!):

```
Defaults      stay_setuid
```

**Привлечение функций безопасности.** Поскольку ядро Linux обрабатывает все переключения между пользователями (и как результат права доступа к файлам) с помощью команд `setuid` и последующих системных вызовов, системные программисты и администраторы должны быть исключительно внимательны по отношению:

- к командам, у которых есть права доступа к `setuid`;
- к тому, что эти команды выполняют.

Если вы создадите копию оболочки `bash`, которая будет наделена корневыми правами `setuid`, любой локальный пользователь сможет получить полное управление системой. Это действительно просто. Более того, даже специальная команда, которой предоставлены корневые права `setuid`, способна создать риск, если в ней есть ошибки. Использование слабых мест программы, работающей с корневыми правами, является основным методом вторжения в систему, и количество попыток такого использования очень велико.

Поскольку существует множество способов проникновения в систему, предотвращение вторжения является многосторонней задачей. Один из самых важных способов избежать нежелательной активности в системе состоит в обязательной аутентификации пользователей с помощью имен пользователей и паролей.

# 7.9. Идентификация и аутентификация пользователей

Многопользовательская система должна обеспечивать базовую поддержку безопасности пользователей в терминах идентификации и аутентификации. *Идентификация* отвечает на вопрос, *что* за пользователи перед нами. При *аутентификации* система просит пользователей *доказать*, что они являются теми, кем они себя называют. Наконец, *авторизация* используется для определения границ того, что *дозволено* пользователям.

Когда дело доходит до идентификации пользователя, ядро системы Linux знает только численный идентификатор пользователя для процесса и владения файлами. Ядро знает о правилах авторизации, относящихся к тому, как запускать исполняемые файлы `setuid` и как совершать системные вызовы из семейства `setuid()`, чтобы выполнить переход от одного пользователя к другому. Однако ядро ничего не знает об аутентификации: именах пользователей, паролях и т. п. Практически все, что относится к аутентификации, происходит в пространстве пользователя.

В подразделе 7.3.1 мы рассматривали соответствие между идентификаторами пользователей и паролями. Сейчас я объясню, как пользовательские процессы получают доступ к этому соответствию. Начнем с предельно упрощенного случая, при котором пользовательский процесс желает узнать свое имя пользователя (имя, которое соответствует эффективному идентификатору пользователя). В традиционной системе Unix процесс мог бы выполнить для этого что-либо подобное.

1. Процесс спрашивает у ядра свой эффективный идентификатор пользователя с помощью системного вызова `geteuid()`.
2. Процесс открывает файл `/etc/passwd` и начинает его чтение с самого начала.
3. Процесс читает строки в файле `/etc/passwd`. Если читать больше нечего, попытка поиска имени пользователя завершается неудачей.
4. Процесс выполняет анализ строки по полям (вытаскивая все, что находится между двоеточиями). Третье поле является идентификатором пользователя в текущей строке.
5. Процесс сравнивает идентификатор, полученный на четвертом шаге, с тем, который был получен на первом шаге. Если они совпадают, первое поле, найденное на четвертом шаге, является искомым именем пользователя; процесс может прекратить поиски и воспользоваться данным именем.
6. Процесс переходит к следующей строке файла `/etc/passwd` и возвращается к третьему шагу.

Процедура довольно длинная, а в реальности она обычно гораздо более сложная.

**Применение библиотек для получения информации о пользователе.** Если каждому разработчику, которому потребовалось узнать текущее имя пользователя, приходилось бы создавать весь код, который вы только что видели, система стала бы ужасающе несвязной, раздутой и совершенно неуправляемой. К счастью, мы можем использовать стандартные библиотеки, чтобы выполнять повторяющиеся задачи. Чтобы узнать имя пользователя, необходимо лишь вызвать функ-

цию вроде `getpwuid()` из стандартной библиотеки после получения ответа от команды `geteuid()`. Обратитесь к страницам руководства по этим вызовам, чтобы узнать о том, как они работают.

Когда стандартная библиотека является используемой совместно, можно осуществить значительные изменения в реализации системы, не меняя никаких других команд. Например, можно полностью уйти от применения файла `/etc/passwd` для ваших пользователей и применять вместо него сетевую службу, подобную LDAP (Lightweight Directory Access Protocol, облегченный (упрощенный) протокол доступа к (сетевым) каталогам).

Такой подход прекрасно работает при идентификации имен пользователей, связанных с идентификаторами, однако для паролей дела обстоят сложнее. В подразделе 7.3.1 описано, каким обычно образом зашифрованный пароль становится частью файла `/etc/passwd`, поэтому, если бы вам понадобилось проверить введенный пользователем пароль, пришлось бы шифровать все, что вводит пользователь, и сравнивать с содержимым файла `/etc/passwd`.

Традиционная реализация обладает следующими ограничениями.

- Не устанавливается общесистемный стандарт на протокол шифрования.
- Предполагается, что у вас есть доступ к зашифрованному паролю.
- Предполагается, что пользователю будет предлагаться ввести пароль всякий раз, когда он будет пытаться получить доступ к чему-либо, требующему аутентификации.
- Предполагается, что вы намерены использовать именно пароли. Если вы желаете применять одноразовые жетоны, смарт-карты, биометрическую или какую-либо еще аутентификацию пользователя, вам придется добавлять такую поддержку самостоятельно.

Некоторые ограничения, посодействовавшие развитию пакета shadow-паролей, рассмотрены в подразделе 7.3.3. Этот файл сделал первый шаг к тому, чтобы разрешить конфигурирование паролей на уровне системы в целом. Однако решение большинства проблем пришло вместе с разработкой и реализацией стандарта PAM.

## 7.10. Стандарт PAM

Чтобы обеспечить гибкость при аутентификации пользователей, в 1995 году корпорация Sun Microsystems предложила новый стандарт под названием *PAM* (Pluggable Authentication Modules, *подключаемые модули аутентификации*) — систему совместно используемых библиотек для аутентификации (рабочие предложения Open Source Software Foundation, выпуск 86.0, октябрь 1995 года). Для аутентификации пользователя приложение «вручает» пользователя утилите PAM, чтобы определить, может ли пользователь успешно идентифицировать себя. Таким образом сравнительно легко добавить поддержку дополнительных методов, таких как двухфакторная аутентификация или ключи на материальных носителях. В дополнение к поддержке механизма аутентификации стандарт PAM также обеспечивает ограниченную

степень контроля авторизации для служб (если, например, вы желаете, чтобы у некоторых пользователей не было службы cron).

Поскольку существуют различные типы аутентификации, стандарт PAM использует несколько динамически загружаемых *модулей аутентификации*. Каждый модуль выполняет специальную задачу; например, модуль pam_unix.so проверяет пароль пользователя.

Занятие это довольно сложное. Программный интерфейс непростой, и неясно, действительно ли стандарт PAM справляется со всеми существующими проблемами. Однако поддержка стандарта PAM присутствует практически в каждой команде, для которой требуется аутентификация в системе Linux, и большинство версий системы используют этот стандарт. Поскольку он работает поверх интерфейса API для аутентификации, существующего в Unix, интеграция поддержки в клиент требует незначительной (а то и совсем никакой не требует) дополнительной работы.

## 7.10.1. Конфигурация PAM

Мы изучим основы стандарта PAM, рассмотрев его конфигурацию. Файлы конфигурации PAM обычно можно найти в каталоге /etc/pam.d (в старых системах может использоваться единственный файл /etc/pam.conf). Во многих версиях содержится несколько файлов, и может быть неясно, с чего начать. Определенные имена файлов должны соответствовать частям системы, которые вы уже знаете, например cron и passwd.

Поскольку специальная конфигурация в таких файлах может сильно различаться в разных версиях системы, трудно подыскать общий пример. Рассмотрим образец строки конфигурации, который вы можете обнаружить для команды chsh (команда смены оболочки):

```
auth       requisite     pam_shells.so
```

Эта строка говорит о том, что оболочка пользователя должна располагаться в каталоге /etc/shells, чтобы пользователь мог успешно пройти аутентификацию команды chsh. Посмотрим, каким образом. Каждая строка конфигурации содержит три поля в таком порядке: тип функции, управляющий аргумент и модуль. Вот что они означают для данного примера.

- **Тип функции.** Это функция, которую пользовательское приложение просит выполнить утилиту PAM. В данном случае это команда auth, задание аутентификации пользователя.
- **Управляющий аргумент.** Этот параметр контролирует то, что будет делать PAM-приложение *после* успешного или неудачного завершения действия в данной строке (requisite в этом примере). Скоро мы перейдем к этому.
- **Модуль.** Это модуль аутентификации, который запускается для данной строки и определяет, что именно делает строка. Здесь модуль pam_shells.so проверяет, упоминается ли текущая оболочка пользователя в файле /etc/shells.

Конфигурация PAM детально описана на странице руководства pam.conf(5). Рассмотрим некоторые основные части.

## Типы функций

Пользовательское приложение может попросить PAM-утилиту выполнить одну из четырех перечисленных ниже функций:

- `auth` — выполнить аутентификацию пользователя (проверить, является ли пользователь тем, кем он себя называет);
- `account` — проверить статус учетной записи пользователя (авторизован ли, например, пользователь на выполнение каких-либо действий);
- `session` — выполнить что-либо только для текущего сеанса пользователя (например, отобразить сообщение дня);
- `password` — изменить пароль пользователя или другую информацию в учетной записи.

Для любой строки конфигурации модуль и функция совместно определяют действие PAM-утилиты. У модуля может быть несколько типов функции, поэтому при определении назначения строки конфигурации всегда помните о том, что функцию и модуль следует рассматривать в виде пары. Например, модуль `pam_unix.so` проверяет пароль, когда выполняет функцию `auth`, но если он выполняет функцию `password`, то пароль устанавливается.

## Управляющие аргументы и стек правил

Одним важным свойством стандарта PAM является то, что правила, которые определены в строках конфигурации, *образуют стек*, это значит, что можно применять несколько правил при выполнении функции. Именно поэтому важен управляющий аргумент: успешное или неудачное завершение действия в одной строке может повлиять на следующие строки или привести к тому, что сама функция завершит работу корректно или нет.

Существуют два типа управляющих аргументов: с простым и более сложным синтаксисом. Приведу три главных управляющих аргумента с простым синтаксисом, которые вы можете найти в правиле.

- `sufficient`. Если данное правило выполняется успешно, аутентификация также проходит успешно и PAM-утилите не нужно проверять следующие правила. Если правило не выполняется, утилита PAM переходит к дополнительным правилам.
- `requisite`. Если правило выполняется успешно, PAM-утилита переходит к следующим правилам. Если правило не выполняется, аутентификация завершается неудачей и PAM-утилите не нужно проверять следующие правила.
- `required`. Если данное правило выполняется успешно, PAM-утилита переходит к следующим правилам. Если правило не выполняется, PAM-утилита переходит к следующим правилам, но всегда вернет отрицательный результат аутентификации, вне зависимости от результатов выполнения дополнительных правил.

Продолжая предыдущий пример, приведу образец стека для функции аутентификации `chsh`:

```
auth       sufficient      pam_rootok.so
auth       requisite       pam_shells.so
auth       sufficient      pam_unix.so
auth       required        pam_deny.so
```

В соответствии с этой конфигурацией, после того как команда `chsh` запрашивает PAM-утилиту о выполнении функции аутентификации, утилита выполняет следующее (см. блок-схему на рис. 7.4).

1. Модуль `pam_rootok.so` проверяет, не пытается ли пройти аутентификацию корневой пользователь. Если это так, то происходит немедленное успешное завершение и дальнейшая аутентификация не предпринимается. Это срабатывает, поскольку для управляющего аргумента установлено значение `sufficient`, которое означает, что успешного завершения данного действия достаточно, чтобы утилита PAM немедленно возвратила результат команде `chsh`. В противном случае она переходит ко второму шагу.
2. Модуль `pam_shells.so` проверяет, есть ли оболочка пользователя в файле `/etc/shells`. Если ее там нет, модуль возвращает отказ, а управляющий аргумент `requisite` говорит о том, что утилита PAM должна немедленно сообщить об отказе команде `chsh` и не пытаться продолжать аутентификацию. В противном случае, когда оболочка есть в файле `/etc/shells`, модуль возвращает результат, который удовлетворяет флагу `required`, переходя к третьему шагу.
3. Модуль `pam_unix.so` запрашивает у пользователя пароль и проверяет его. Для управляющего аргумента указано значение `sufficient`, поэтому успешной проверки (пароль верный) достаточно для утилиты PAM, чтобы она сообщила об этом команде `chsh`. Если пароль неверный, утилита PAM переходит к четвертому шагу.
4. Модуль `pam_deny.so` всегда вызывает отказ, а поскольку присутствует управляющий аргумент `required`, утилита PAM возвращает отказ команде `chsh`. Этот вариант применяется по умолчанию в тех случаях, когда больше нечего пробовать. Обратите внимание на то, что управляющий аргумент `required` не приводит к моментальному отказу функции PAM — она обработает все остальные строки стека, — однако в результате в приложение всегда вернется отказ.

ПРИМЕЧАНИЕ

Не смешивайте термины «функция» и «действие», когда работаете с утилитой PAM. Функция является целью верхнего уровня: что ожидает пользовательское приложение от утилиты PAM (например, выполнить аутентификацию пользователя). Действие является отдельным шагом, который выполняет утилита, чтобы достичь цели. Запомните лишь то, что сначала пользовательское приложение вызывает функцию, а утилита PAM заботится обо всех деталях, связанных с действиями.

Сложный синтаксис управляющего аргумента, помещаемый внутри квадратных скобок (`[]`), позволяет вручную контролировать реакцию, основываясь на специальном значении, которое возвращает модуль (не только успех или отказ). Подробности см. на странице pam.conf(5) руководства. Когда вы разберетесь с простым синтаксисом, у вас не возникнет затруднений и со сложным.

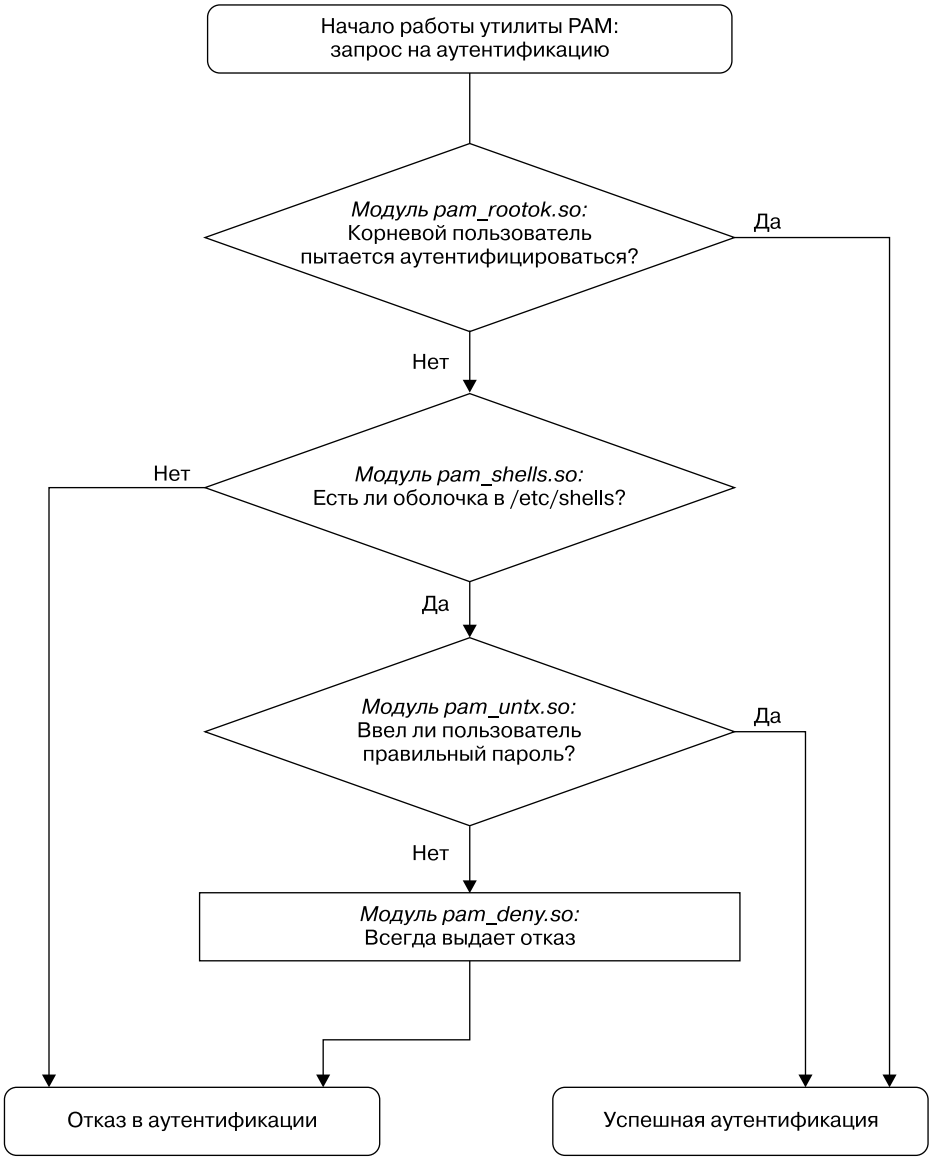


**Рис. 7.4.** Процесс исполнения правила PAM

## Аргументы модуля

Модули PAM могут задействовать аргументы после имени модуля. Вам часто будет встречаться такой вариант для модуля `pam_unix.so`:

```
auth      sufficient    pam_unix.so    nullok
```

Здесь аргумент `nullok` сообщает о том, что у пользователя может не быть пароля (по умолчанию в таком случае возник бы отказ).

## 7.10.2. Примечания о стандарте PAM

Благодаря возможности управления процессом и модульному синтаксису аргументов синтаксис PAM-конфигурации обладает многими чертами языка программирования и некоторыми функциональными возможностями. Мы лишь слегка затронули эту тему, однако приведу еще несколько моментов, относящихся к стандарту PAM.

- Чтобы выяснить, какие модули PAM присутствуют в вашей системе, воспользуйтесь командой `man -k pam_` (обратите внимание на символ подчеркивания). Может оказаться непросто выяснить местоположение модулей. Попробуйте команду `locate unix_pam.so` и посмотрите, к чему это приведет.
- Страницы руководства содержат описание функций и аргументов для каждого модуля.
- Многие версии системы автоматически генерируют некоторые файлы конфигурации PAM, поэтому неразумно изменять их напрямую в каталоге `/etc/pam.d`. Прочитайте комментарии в этих файлах перед их редактированием; если они были сгенерированы, комментарии скажут вам, откуда взялись файлы.
- Конфигурационные файлы в каталоге `/etc/pam.d/other` определяют конфигурацию по умолчанию для любого приложения, у которого нет собственного файла конфигурации. Часто по умолчанию на все следует отказ.
- Существуют разные способы включения дополнительных файлов конфигурации в файл конфигурации PAM. Синтаксис `@include` загружает весь файл конфигурации, но можно также использовать управляющий аргумент, чтобы загрузить только конфигурацию для конкретной функции. Способ использования разный для различных версий системы.
- Конфигурация PAM не ограничивается лишь аргументами модуля. Некоторые модули могут иметь доступ к дополнительным файлам в каталоге `/etc/security`, как правило, для настройки ограничений для отдельного пользователя.

## 7.10.3. Стандарт PAM и пароли

В результате эволюции системы проверки паролей в Linux сохранилось несколько артефактов, которые иногда могут привести к путанице. Во-первых, следует опасаться файла `/etc/login.defs`. Это файл конфигурации для исходного набора shadow-паролей. Он содержит информацию об алгоритме шифрования, использованном для файла `shadow`, однако редко применяется в современных системах с установленными модулями PAM, поскольку конфигурация PAM уже содержит данную информацию. Учитывая сказанное, алгоритм шифрования, указанный в файле `/etc/login.defs`, должен совпадать с конфигурацией PAM — для тех редких случаев, когда вам встретится приложение, не поддерживающее стандарт PAM.

Откуда утилита PAM получает информацию о схеме шифрования пароля? Вспомните о том, что она может взаимодействовать с паролями двумя способами: с помощью функции `auth` (для проверки пароля) и функции `password` (для установ-

ки пароля). Проще отследить параметр, устанавливающий пароль. Лучшим способом будет, вероятно, простое использование команды grep:

```
$ grep password.*unix /etc/pam.d/*
```

Соответствующие строки должны содержать имя pam_unix.so и выглядеть, например, так:

```
password      sufficient     pam_unix.so obscure sha512
```

Аргументы obscure и sha512 говорят утилите PAM о том, что следует делать при установке пароля. Сначала утилита проверяет, является ли пароль достаточно «скрытным» (то есть, среди прочего, не напоминает ли он предыдущий пароль), а затем использует алгоритм SHA512, чтобы зашифровать новый пароль.

Однако так происходит *только* тогда, когда пользователь *устанавливает* пароль, а не когда утилита PAM *проверяет* его. Как же ей узнать, какой алгоритм использовать при аутентификации? К сожалению, конфигурация не сообщит ничего; у модуля pam_unix.so нет аргументов шифрования для функции auth.

Оказывается (на момент написания книги), модуль pam_unix.so просто пытается угадать алгоритм, как правило призывая на выполнение этой грязной работы библиотеку libcrypt, которая перебирает множество различных вариантов, пока что-либо не сработает или будет нечего пробовать. Следовательно, вам не придется заботиться об алгоритме шифрования при верификации.

## 7.11. Заглядывая вперед

Рассмотрев многие из жизненно важных блоков системы Linux, сейчас мы прошли половину этой книги. Обсуждение процесса входа в систему и управления пользователями Linux познакомило вас с тем, чтó позволяет разбивать службы и задачи на маленькие независимые фрагменты, которые до определенной степени обладают способностью взаимодействовать.

Данная глава практически полностью была посвящена пространству пользователя, теперь необходимо уточнить наши представления о процессах из пространства пользователя и о потребляемых ими ресурсах. Для этого в главе 8 мы возвращаемся обратно в ядро.

# 8 Подробное рассмотрение процессов и использования ресурсов

Эта глава более подробно расскажет вам о взаимоотношениях между процессами, ядром и системными ресурсами. Существует три основных типа аппаратных ресурсов: центральный процессор, память и устройства ввода/вывода. Процессы соперничают за эти ресурсы, и задачей ядра является их честное распределение. Само ядро также является ресурсом — программным ресурсом, который процессы используют, чтобы выполнять такие задачи, как создание новых процессов и взаимодействие с другими процессами.

Многие из инструментов, которые вы увидите в этой главе, часто считают инструментами для слежения за производительностью. Они чрезвычайно полезны, если ваша система начинает «тормозить» и вы пытаетесь выяснить причину этого. Тем не менее не следует уделять излишнего внимания производительности: попытки оптимизировать систему, которая и так работает хорошо, зачастую являются лишь пустой тратой времени. Вместо этого сосредоточьтесь на понимании того, что́ эти инструменты измеряют на самом деле, и тогда вы получите прекрасное представление о том, как работает ядро.

## 8.1. Отслеживание процессов

Из раздела 2.16 вы узнали о том, как использовать команду `ps`, чтобы увидеть перечень процессов, запущенных в системе в данный момент. Команда `ps` приводит список текущих процессов, но она может мало что сказать вам о том, как изменяются процессы с течением времени. Следовательно, она не поможет вам определить процесс, который использует слишком много ресурсов ЦПУ или оперативной памяти.

Команда `top` часто оказывается полезнее команды `ps`, поскольку она отображает текущее состояние системы, заодно со многими полями, которые есть в листинге команды `ps`, и при этом она обновляет результат каждую секунду. Вероятно, самым важным является то, что команда `top` показывает наиболее активные процессы (то есть те, которые в данный момент используют наибольшую часть процессорного времени) в верхней части списка.

Нажимая на клавиши, можно отправлять инструкции команде top. Приведу самые важные из них (табл. 8.1).

**Таблица 8.1.** Инструкции для команды top

| Инструкция | Действие |
|---|---|
| Клавиша Пробел | Немедленно обновить экран |
| M | Выполнить сортировку по количеству используемой резидентной памяти |
| T | Выполнить сортировку по общему (кумулятивному) применению ЦПУ |
| P | Выполнить сортировку по текущему использованию ЦПУ (по умолчанию) |
| u | Отобразить процессы только для одного пользователя |
| f | Выбрать другие параметры для отображения |
| ? | Отобразить статистику использования всех команд top |

Две другие утилиты Linux, подобные команде top, — atop и htop — предлагают расширенный набор вариантов просмотра и функций. Большинство дополнительных функций доступно в других утилитах. Например, команда htop обладает многими возможностями команды lsof, описанной в следующем разделе.

# 8.2. Поиск открытых файлов с помощью команды lsof

Команда lsof перечисляет открытые файлы и процессы, которые их используют. Поскольку Unix делает существенный акцент на файлах, команда lsof входит в число самых полезных инструментов для отыскания неполадок. Однако эта команда не ограничивается обычными файлами — она может перечислять сетевые ресурсы, динамические библиотеки, каналы и многое другое.

## 8.2.1. Чтение результатов вывода команды lsof

После запуска команды lsof в командной строке обычно появляется огромный список. Ниже приведен фрагмент того, что вы могли бы увидеть. Этот результат содержит файлы, открытые процессом init, а также запущенный процесс vi:

```
$ lsof
COMMAND PID  USER  FD TYPE DEVICE  SIZE     NODE NAME
init      1  root cwd  DIR    8,1  4096        2 /
init      1  root rtd  DIR    8,1  4096        2 /
init      1  root mem  REG    8,  47040  9705817 /lib/i386-linux-gnu/libnss_files-
                                                  2.15.so
init      1  root mem  REG    8,1 42652  9705821 /lib/i386-linux-gnu/libnss_nis-
                                                  2.15.so
init      1  root mem  REG    8,1 92016  9705833 /lib/i386-linux-gnu/libnsl-2.15.so
--snip--
vi    22728 juser cwd  DIR    8,1  4096 14945078 /home/juser/w/c
vi    22728 juser  4u  REG    8,1  1288  1056519 /home/juser/w/c/f
--snip--
```

Результат состоит из следующих полей (перечисленных в верхней строке).

- COMMAND. Командное имя для процесса, который удерживает дескриптор файла.
- PID. Идентификатор процесса.
- USER. Пользователь, запустивший процесс.
- FD. Это поле может содержать два типа элементов. В приведенном выше результате столбец FD показывает назначение файла. Это поле может также содержать *файловый дескриптор* открытого файла — число, которое процесс использует вместе с системными библиотеками и ядром, чтобы идентифицировать файл и работать с ним.
- TYPE. Тип файла (обычный файл, каталог, сокет и т. п.).
- DEVICE. Старший и младший номера устройства, которое удерживает данный файл.
- SIZE. Размер файла.
- NODE. Номер дескриптора inode для данного файла.
- NAME. Имя файла.

Страница руководства lsof(1) содержит полный перечень того, что вы можете встретить в каждом из полей, однако вы должны уметь определять, что перед вами, просто глядя на результат вывода. Посмотрите, например, на записи, у которых в поле FD указано значение cwd (выделено жирным шрифтом). В этих строках заданы текущие рабочие каталоги для процессов. Еще один пример содержится в самой последней строке: это файл, который в данный моммент редактируется пользователем с помощью команды vi.

## 8.2.2. Использование команды lsof

Есть два основных подхода к запуску команды lsof.

- Перечислить все, а затем перенаправить вывод в команду типа less и поискать то, что вам необходимо. На это может потребоваться некоторое время, в зависимости от результата вывода.
- Сузить список, создаваемый командой lsof, с помощью параметров командной строки.

Можно использовать параметры командной строки, чтобы передать имя файла в качестве аргумента и вынудить команду lsof перечислить только те записи, которые соответствуют этому аргументу. Например, следующая команда отображает записи для файлов, открытых в каталоге /usr:

```
$ lsof /usr
```

Чтобы вывести список файлов, открытых процессом с идентификатором PID, запустите такую команду:

```
$ lsof -p pid
```

Для вывода краткой справки о параметрах команды lsof запустите команду lsof -h. Большинство параметров относится к формату вывода (см. главу 10, в которой говорится о сетевых функциях команды lsof).

ПРИМЕЧАНИЕ

Команда lsof сильно зависит от информации о ядре. Если вы обновляете ядро, но при этом нерегулярно обновляете все остальное, вам может потребоваться обновление команды lsof. Более того, если вы применили обновление и для ядра, и для команды lsof, обновленная команда lsof может не запускаться до тех пор, пока вы не перезагрузите систему с новым ядром.

# 8.3. Отслеживание выполнения команд и системных вызовов

Инструменты, которые мы рассмотрели, исследуют активные процессы. Однако если вам непонятно, почему какая-либо программа закрывается практически сразу после запуска, то даже команда `lsof` вам не поможет. На самом деле у вас возникли бы сложности, если бы вы запустили команду `lsof` одновременно с командой, вызывающей отказ.

Команды `strace` (отслеживание системных вызовов) и `ltrace` (отслеживание библиотек) могут помочь выяснить, что пытается делать команда. Эти инструменты выводят чрезвычайно большие отчеты, но как только вы узнаете, что искать, в вашем распоряжении будут дополнительные инструменты для отслеживания проблем.

## 8.3.1. Команда strace

Вспомните о том, что *системный вызов* является привилегированной операцией, которую процесс из пространства пользователя просит у ядра выполнить (например, открытие файла и чтение данных из него). Утилита `strace` выводит список всех системных вызовов, которые осуществляет процесс. Чтобы увидеть это в действии, запустите такую команду:

```
$ strace cat /dev/null
```

Из главы 1 вы узнали о том, что, когда процесс собирается запустить другой процесс, он задействует системный вызов `fork()`, чтобы создать ответвленную копию, которая затем использует один из системных вызовов семейства `exec()`, чтобы запустить новую команду. Команда `strace` начинает работать с новым процессом (копией исходного процесса) сразу после вызова `fork()`. Следовательно, первые строки вывода данной команды должны показать команду `execve()` в действии, за которой следует вызов инициализации памяти, `brk()`, как приведено ниже:

```
execve("/bin/cat", ["cat", "/dev/null"], [/* 58 vars */]) = 0
brk(0)                                  = 0x9b65000
```

Следующая часть вывода относится главным образом к загрузке совместно используемых библиотек. Можете пропустить это, если вы не стремитесь узнать о том, что делает система совместно используемых библиотек.

```
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb77b5000
```

```
access("/etc/ld.so.preload", R_OK)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
--snip--
open("/lib/libc.so.6", O_RDONLY)       = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200^\1"..., 1024)= 1024
```

Кроме того, пропустите вывод до команды `mmap` включительно, пока не встретите строки, подобные следующим:

```
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 6), ...}) = 0
open("/dev/null", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 3), ...}) = 0
fadvise64_64(3, 0, 0, POSIX_FADV_SEQUENTIAL)= 0
read(3,"", 32768)                      = 0
close(3)                               = 0
close(1)                               = 0
close(2)                               = 0
exit_group(0)                          = ?
```

Эта часть вывода показывает команду в действии. Сначала посмотрите на вызов `open()`, который открывает файл. Число 3 — результат, означающий успешное завершение (это файловый дескриптор, который ядро возвращает после открытия файла). Под ним вы видите, где команда `cat` выполняет чтение из устройства `/dev/null` (вызов `read()`, который также обладает файловым дескриптором `3`). Считывать больше нечего, поэтому команда закрывает файловый дескриптор и выходит с помощью вызова `exit_group()`.

Что происходит, если возникает ошибка? Попробуйте запустить команду `strace cat not_a_file` и посмотрите на системный вызов `open()` в результатах вывода:

```
open("not_a_file", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
```

Поскольку команде `open()` не удалось открыть файл, она возвратила значение `-1`, чтобы сообщить об ошибке. Видно, что команда `strace` выводит название ошибки и дает ее краткое описание.

Отсутствующие файлы являются наиболее частым источником ошибок в командах Unix, поэтому если системный журнал и другая информация оказываются не слишком полезными, а обратиться больше не к чему, то команда `strace` может оказать существенную помощь. Ее можно применить даже для демонов, которые откреплены. Например, так:

```
$ strace -o crummyd_strace -ff crummyd
```

В данном примере параметр `-o` команды `strace` заносит в журнал действия любого дочернего процесса, который демон `crummy` породил в `crummyd_strace.`*`pid`*, где *`pid`* — это идентификатор дочернего процесса.

## 8.3.2. Команда ltrace

Команда `ltrace` отслеживает вызовы совместно используемых библиотек. Результаты ее работы напоминают вывод команды `strace`, и именно поэтому я упоминаю

о ней здесь, но она не отслеживает ничего на уровне ядра. Имейте в виду: вызовов совместно используемых библиотек *намного* больше, чем системных вызовов. Вам непременно понадобится фильтровать результаты, и у команды `ltrace` есть множество встроенных параметров, чтобы помочь вам в этом.

ПРИМЕЧАНИЕ

См. подраздел 15.1.4, содержащий дополнительную информацию. Команда ltrace не работает для статически связанных двоичных файлов.

# 8.4. Потоки

В Linux некоторые процессы разделены на части, называемые *потоками*. Поток очень похож на процесс: у него есть идентификатор (TID, или ID потока), и ядро планирует запуск потоков и запускает их так же, как и процессы. Однако в отличие от отдельных процессов, которые обычно не используют совместно с другими процессами такие системные ресурсы, как оперативная память и подключение к вводу/выводу, все потоки внутри какого-либо процесса совместно задействуют ресурсы системы и некоторую часть памяти.

## 8.4.1. Однопоточные и многопоточные процессы

Многие процессы обладают только одним потоком. Процесс с одним потоком является *однопоточным*, а процесс с несколькими потоками — *многопоточным*. Все процессы запускаются как однопоточные. Этот стартовый поток обычно называется *главным потоком*. Затем главный поток может запустить новые потоки, чтобы процесс стал многопоточным, подобно тому как процесс может вызвать команду `fork()` для запуска нового процесса.

ПРИМЕЧАНИЕ

Если процесс является однопоточным, то о потоках вообще довольно редко упоминают. В этой книге на потоки не обращается внимание, если многопоточные процессы не отражаются на том, что вы видите или осуществляете.

Основное преимущество многопоточных процессов таково: когда процесс должен выполнить много работы, потоки могут быть запущены одновременно на нескольких процессорах, что потенциально ускоряет вычисления. Хотя одновременные вычисления можно организовать и с помощью нескольких процессов, потоки запускаются быстрее процессов и потокам часто бывает проще и/или эффективнее взаимодействовать между собой при совместном использовании памяти по сравнению с процессами, которые взаимодействуют через сетевое соединение или канал.

Некоторые команды применяют потоки, чтобы обойти проблемы при управлении несколькими ресурсами ввода/вывода. Традиционно процесс использовал бы что-либо вроде команды `fork()`, чтобы запустить новый подпроцесс для работы с новым потоком ввода или вывода. Потоки предлагают похожий механизм без излишнего запуска нового процесса.

## 8.4.2. Просмотр потоков

По умолчанию в выводе команд ps и top отображаются только процессы. Чтобы показать информацию о потоке в команде ps, добавьте параметр m (пример 8.1).

**Пример 8.1.** Просмотр потоков с помощью команды ps m

```
$ ps m
  PID TTY      STAT   TIME COMMAND
 3587 pts/3    -      0:00 bash❶
    - -        Ss     0:00 -
 3592 pts/4    -      0:00 bash❷
    - -        Ss     0:00 -
12287 pts/8    -      0:54 /usr/bin/python /usr/bin/gm-notify❸
    - -        SL1    0:48 -
    - -        SL1    0:00 -
    - -        SL1    0:06 -
    - -        SL1    0:00 -
```

В примере 8.1 процессы показаны вместе с потоками. Каждая строка с номером в столбце PID (эти строки отмечены символами ❶, ❷ и ❸) представляет процесс как при обычном выводе команды ps. Строки с дефисами в столбце PID представляют потоки, связанные с данным процессом. В этом выводе у каждого из процессов ❶ и ❷ только один поток, а процесс 12287 (❸) является многопоточным и состоит из четырех потоков.

Если вы желаете просмотреть идентификаторы потоков с помощью команды ps, можно использовать специальный формат вывода. В примере 8.2 показаны только идентификаторы процессов и потоков, а также сама команда.

**Пример 8.2.** Отображение идентификаторов процессов и потоков с помощью команды ps m

```
$ ps m -o pid,tid,command
  PID   TID   COMMAND
 3587     -   bash
    -  3587   -
 3592     -   bash
    -  3592   -
 12287    -   /usr/bin/python /usr/bin/gm-notify
    - 12287   -
    - 12288   -
    - 12289   -
    - 12295   -
```

Приведенный в примере 8.2 вывод соответствует потокам, показанным в примере 8.1. Обратите внимание на то, что идентификаторы потоков для однопоточных процессов совпадают с идентификаторами процессов: это главные потоки. Для многопоточного процесса 12287 поток 12287 также является главным потоком.

ПРИМЕЧАНИЕ

Как правило, вам не придется взаимодействовать с отдельными потоками так, как вы это делали бы с процессами. Вам потребуется узнать довольно много о том, как была написана многопоточная команда, чтобы воздействовать на один из потоков в какой-либо момент, но даже в этом случае такая идея не слишком хороша.

Потоки могут вызвать путаницу при отслеживании ресурсов, поскольку отдельные потоки в многопоточном процессе могут одновременно пользоваться ресурсами. Например, команда `top` по умолчанию не отображает потоки; необходимо нажать клавишу H, чтобы включить их показ. Для большинства инструментов отслеживания ресурсов, о которых вы скоро узнаете, потребуется выполнить небольшую дополнительную работу, чтобы включить отображение потоков.

# 8.5. Введение в отслеживание ресурсов

Сейчас мы обсудим некоторые моменты, относящиеся к отслеживанию ресурсов, включая время центрального процессора, память и дисковый ввод/вывод. Мы рассмотрим использование ресурсов как в масштабе всей системы, так и для отдельных процессов.

Многие пользователи вникают в устройство ядра системы Linux в целях улучшения производительности. Однако большинство версий систем прекрасно работает с установками по умолчанию, и вы можете потратить несколько дней, пытаясь настроить производительность компьютера без существенных результатов, особенно если вы не знаете, что искать. По этой причине, когда вы будете экспериментировать с инструментами, описанными в этой главе, думайте не о производительности, а о том, как действует ядро, распределяя ресурсы между процессами.

# 8.6. Измерение процессорного времени

Чтобы отследить один или несколько процессов с течением времени, используйте параметр `-p` в команде `top` с таким синтаксисом:

```
$ top -p pid1 [-p pid2 ...]
```

Чтобы выяснить, какое количество процессорного времени применяет команда для своей работы, используйте команду `time`. В большинстве оболочек есть встроенная команда `time`, которая не приводит подробную статистику, поэтому может потребоваться запуск команды `/usr/bin/time`. Например, чтобы измерить процессорное время, использованное командой `ls`, запустите такую команду:

```
$ /usr/bin/time ls
```

По окончании работы команды `ls` команда `time` должна вывести результаты, подобные приведенным ниже. Ключевые поля выделены жирным шрифтом:

```
0.05user 0.09system 0:00.44elapsed 31%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (125major+51minor)pagefaults 0swaps
```

- **Время пользователя.** Количество секунд, которое центральный процессор потратил на выполнение собственного кода команды. В современных процессорах некоторые команды запускаются настолько быстро и процессорное время настолько мало, что значение округляется до нуля.
- **Время системы.** Какое количество времени ядро затрачивает на выполнение работы процесса (например, на чтение файлов и каталогов).

- **Время работы.** Общее количество времени, которое требуется на работу процесса, от его запуска до завершения, включая время, затраченное процессором на выполнение других задач. Эта величина, как правило, не слишком пригодна для измерения производительности, однако, если вычесть из нее значения времени пользователя и времени системы, можно получить общее представление о том, как долго процесс пребывает в ожидании системных ресурсов.

Остальная часть вывода содержит главным образом подробности об использовании памяти и ресурсов ввода/вывода. Подробнее об ошибках отсутствия страницы вы прочитаете в разделе 8.9.

# 8.7. Настройка приоритетов процессов

Можно изменить распорядок, который ядро назначает процессам, чтобы предоставить какому-либо процессу больше или меньше процессорного времени по сравнению с другими процессами. Ядро запускает каждый процесс в соответствии с назначенным ему *приоритетом*, который является числом от –20 до 20, причем –20 означает высший приоритет. (Да, это сбивает с толку!)

Команда `ps -l` выводит текущий приоритет процесса, однако немного проще увидеть приоритеты в действии с помощью команды `top`, как показано здесь:

```
$ top
Tasks: 244 total,  2 running, 242 sleeping,   0 stopped,   0 zombie
Cpu(s): 31.7%us, 2.8%sy,  0.0%ni, 65.4%id,  0.2%wa,  0.0%hi,  0.0%si,   0.0%st
Mem:   6137216k total, 5583560k used,   553656k free,    72008k buffers
Swap:  4135932k total,  694192k used, 3441740k  free,   767640k cached
  PID  USER      PR NI  VIRT  RES  SHR S %CPU %MEM    TIME+ COMMAND
28883  bri       20  0 1280m 763m  32m S   58 12.7 213:00.65 chromium-
browse
 1175  root      20  0  210m  43m  28m R   44  0.7  14292:35 Xorg
 4022  bri       20  0  413m 201m  28m S   29  3.4   3640:13 chromium-browse
 4029  bri       20  0  378m 206m  19m S    2  3.5  32:50.86 chromium-browse
 3971  bri       20  0  881m 359m  32m S    2  6.0 563:06.88 chromium-browse
 5378  bri       20  0  152m  10m 7064 S    1  0.2  24:30.21 compiz
 3821  bri       20  0  312m  37m  14m S    0  0.6  29:25.57 soffice.bin
 4117  bri       20  0  321m 105m  18m S    0  1.8  34:55.01 chromium-browse
 4138  bri       20  0  331m  99m  21m S    0  1.7 121:44.19 chromium-browse
 4274  bri       20  0  232m  60m  13m S    0  1.0  37:33.78 chromium-browse
 4267  bri       20  0 1102m 844m 11m S    0 14.1  29:59.27 chromium-browse
 2327  bri       20  0  301m   43m 16m S    0  0.7 109:55.65 unity-2d-shell
```

В приведенном отчете команды `top` столбец `PR` («приоритет») содержит текущий приоритет ядра для запуска данного процесса. Чем больше число, тем меньше вероятность того, что ядро запланирует этот процесс, если процессорное время необходимо другим процессам. Однако один лишь приоритет запуска не определяет решение ядра о предоставлении процессорного времени процессу, к тому же он часто меняется во время выполнения команды в соответствии с количеством процессорного времени, потребляемого процессом.

За столбцом приоритета следует столбец, содержащий *значение относительного приоритета* (NI), которое дает рекомендацию планировщику в ядре. Об этом значении следует позаботиться, если вы пытаетесь повлиять на решение ядра. Ядро прибавляет значение относительного приоритета к текущему приоритету, чтобы определить следующий квант времени для данного процесса.

По умолчанию значение относительного приоритета равно 0. Допустим, что вы запускаете в фоновом режиме большой объем вычислений и желаете, чтобы он не замедлял вашу работу в интерактивном сеансе. Чтобы такой процесс занял последнее место по отношению к другим процессам и работал лишь тогда, когда остальным задачам нечего делать, можно установить для него значение относительного приоритета равным 20 с помощью команды `renice` (здесь параметр *pid* — идентификатор процесса, который вы желаете изменить):

```
$ renice 20 pid
```

Если вы пользователь-superuser, можно указать отрицательное значение относительного приоритета, но такая идея практически всегда является вредной, поскольку для системных процессов может не оказаться достаточного количества процессорного времени. На самом деле вам, вероятно, не потребуется часто менять значения относительного приоритета, поскольку во многих системах Linux всего один пользователь, который не выполняет большие объемы вычислений. Значение относительного приоритета было гораздо более важным тогда, когда на одном компьютере работало несколько пользователей.

# 8.8. Средние значения загрузки

Производительность процессора — один из самых простых параметров для измерения. *Среднее значение загрузки* является средним количеством процессов, которые в данный момент готовы к запуску. То есть это оценка числа процессов, способных использовать процессор в конкретный момент времени. Осмысляя это значение, имейте в виду, что большинство процессов в системе обычно ожидает ввода (с клавиатуры, с помощью мыши или из сети, например), и это означает, что большинство процессов не готовы к запуску и ничего не вносят в среднее значение загрузки. Только те процессы, которые действительно что-либо выполняют, влияют на среднее значение загрузки.

## 8.8.1. Использование команды uptime

Команда `uptime` сообщает три средних значения загрузки в дополнение к тому, как долго работает ядро:

```
$ uptime
... up 91 days, ... load average: 0.08, 0.03, 0.01
```

Три числа, выделенных жирным шрифтом, являются средними значениями загрузки за последние 1, 5 и 15 минут соответственно. Как видите, система не очень занята: за последние 15 минут на всех процессорах работало в среднем только

0,01 процесса. Другими словами, если бы у вас был всего один процессор, то он запускал бы приложения из пространства пользователя лишь 1 % времени за последние 15 минут. Традиционно в большинстве ПК среднее значение загрузки — около 0, если вы заняты чем-либо отличным от компилирования программы или компьютерной игры. Нулевое значение обычно является хорошим признаком, поскольку оно означает, что ваш процессор не перегружен и вы экономите мощность.

ПРИМЕЧАНИЕ

В современных ПК компоненты пользовательского интерфейса стремятся использовать больше ресурсов процессора, чем это было раньше. Например, в системах Linux плагин Flash для браузера приобрел печальную известность как пожиратель ресурсов, а Flash-приложения могут с легкостью занять большую часть процессорного времени и памяти вследствие некачественной реализации в целом.

Если среднее значение загрузки достигает 1, то, вероятно, один процесс практически постоянно использует центральный процессор. Чтобы выяснить, что это за процесс, воспользуйтесь командой `top`; этот процесс будет показан на экране в самой верхней части списка.

В большинстве современных систем присутствует несколько процессорных ядер или процессоров, поэтому несколько процессов могут легко работать одновременно. Если ядер два, то среднее значение загрузки 1 означает, что только одно из ядер активно в любой момент времени, а среднее значение 2 говорит о том, что оба ядра работают.

## 8.8.2. Высокие значения загрузки

Высокое среднее значение загрузки не обязательно свидетельствует о том, что система испытывает сложности. Система, у которой достаточное количество оперативной памяти и ресурсов ввода/вывода, способна легко справиться с несколькими запущенными процессами. Если среднее значение загрузки высоко, но при этом система нормально откликается, не волнуйтесь: множество процессов совместно используют процессор. Процессы вынуждены соперничать друг с другом за процессорное время, в результате им требуется больше времени на выполнение своих вычислений по сравнению с тем случаем, когда им позволено применять процессор постоянно. Еще один случай, при котором можно высокое среднее значение загрузки является нормальным, это веб-сервер, в котором процессы могут запускаться и прекращаться настолько быстро, что функция измерения средней загрузки не может эффективно работать.

Тем не менее, если вы чувствуете, что система «тормозит» и значение средней загрузки велико, вероятно, присутствуют проблемы с производительностью оперативной памяти. Когда в системе мало памяти, ядро может начать быстро подкачивать с диска память для процессов. Когда такое происходит, многие процессы становятся готовы к запуску, но при этом для них может быть недоступна память, и тогда они остаются в состоянии готовности (и вносят вклад в среднее значение загрузки) намного дольше, чем при нормальном режиме работы.

Сейчас мы более подробно рассмотрим оперативную память.

# 8.9. Память

Один из самых простых способов проверить состояние системной памяти в целом — запустить команду `free` или посмотреть файл `/proc/meminfo`, чтобы понять, сколько реальной памяти используется для кэша и буферов. Как мы только что отмечали, проблемы с производительностью могут быть вызваны недостатком памяти. Если используется немного памяти для кэша/буфера (и вместо этого расходуется реальная память), вам может понадобиться дополнительная память. Однако было бы чересчур просто полагать, что проблемы с производительностью вызваны лишь недостатком памяти.

## 8.9.1. Как работает память

В процессоре присутствует модуль управления памятью (MMU), который переводит виртуальные адреса памяти, используемые процессами, в реальные. Ядро помогает модулю MMU, разбивая память на маленькие фрагменты, называемые *страницами*. Ядро содержит структуру данных, которая называется *таблицей страниц* и содержит схему соответствия виртуальных адресов страниц реальным адресам страниц в памяти. Когда процесс получает доступ к памяти, модуль MMU переводит виртуальные адреса, используемые процессом, в реальные адреса на основе таблицы страниц ядра.

В действительности пользовательскому процессу для работы не нужны сразу все его страницы. Обычно ядро загружает и распределяет страницы по мере их необходимости для процесса; такая система работы известна как *вызов страниц по запросу* или *листание по запросу*. Чтобы понять, как это устроено, рассмотрим запуск и работу команды в качестве нового процесса.

1. Ядро загружает начало кода с инструкциями команды в страницы памяти.
2. Ядро может выделить несколько страниц рабочей памяти для нового процесса.
3. Во время своей работы процесс может дойти до такого момента, когда следующей инструкции не окажется ни в одной из страниц, загруженных ядром изначально. Тогда ядро вступает в действие, загружает необходимые страницы в память и позволяет команде продолжить выполнение.
4. Подобным же образом, если команде необходимо больше рабочей памяти, чем было выделено изначально, ядро решает эту задачу, отыскивая свободные страницы (или освобождая пространство) и назначая их данному процессу.

## 8.9.2. Ошибки из-за отсутствия страниц

Если страница памяти не готова, когда процессу необходимо ее использовать, процесс вызывает *ошибку из-за отсутствия страницы*. При возникновении такого события ядро забирает у процесса управление процессором, чтобы подготовить страницу. Существуют два типа ошибок из-за отсутствия страниц: малые и большие.

## Малые ошибки

Малая ошибка из-за отсутствия страницы возникает тогда, когда желаемая страница фактически находится в основной памяти, но модуль MMU не знает, где она. Такое может произойти, когда процесс требует больше памяти или когда модуль MMU не обладает достаточным пространством для хранения всех местоположений страниц для процесса. В таком случае ядро сообщает модулю MMU данные о странице и позволяет процессу продолжить работу. Малые ошибки из-за отсутствия страницы не столь уж существенны, и многие из них возникают во время работы процесса. Если вам не требуется максимальная производительность какой-либо программы, интенсивно обращающейся к памяти, вероятно, не стоит беспокоиться об этих ошибках.

## Большие ошибки

Большая ошибка из-за отсутствия страницы возникает тогда, когда желаемая страница памяти не находится в основной памяти и, значит, ядро должно загрузить ее с диска или из какого-либо другого медленного хранилища данных. Большое количество таких ошибок сильно замедлит работу системы, поскольку ядро должно выполнить довольно солидную работу по снабжению процессов страницами, лишая нормальные процессы возможности работать.

Некоторых больших ошибок невозможно избежать: например, когда код загружается с диска при запуске программы в первый раз. Самые серьезные проблемы возникают, когда памяти становится недостаточно и ядро начинает подкачивать страницы из рабочей памяти на диск, чтобы освободить место для новых страниц.

## Отслеживание ошибок из-за отсутствия страниц

Можно отследить ошибки страниц для отдельных процессов с помощью команд `ps`, `top` и `time`. Следующая команда показывает простой пример того, как команда `time` отображает ошибки страниц. Результаты работы команды `cal` не имеют значения, поэтому мы их отключили, перенаправив в устройство `/dev/null`.

```
$ /usr/bin/time cal > /dev/null
0.00user 0.00system 0:00.06elapsed 0%CPU (0avgtext+0avgdata 3328maxresident)k
648inputs+0outputs (2major+254minor)pagefaults 0swaps
```

Как можно заметить из выделенного жирным шрифтом текста, при работе программы произошли две большие и 254 малые ошибки из-за отсутствия страниц. Большие ошибки возникли, когда ядру потребовалось загрузить команду с диска в первый раз. Если бы вы запустили эту команду повторно, то больших ошибок, вероятно, не было бы, поскольку ядро выполнило кэширование страниц с диска.

Если вы хотите увидеть ошибки для работающих процессов, воспользуйтесь командой `top` или `ps`. При запуске команды `top` используйте флаг `f`, чтобы изменить отображаемые поля, и флаг `u`, чтобы отобразить количество больших ошибок. Результаты будут показаны в новом столбце, `nFLT`. Количество малых ошибок вы не увидите.

При использовании команды `ps` можно применить специальный формат вывода, чтобы увидеть ошибки для конкретного процесса. Вот пример для процесса с идентификатором ID 20365:

```
$ ps -o pid,min_flt,maj_flt 20365
  PID  MINFL  MAJFL
20365 834182     23
```

Столбцы MINFL и MAJFL показывают число малых и больших ошибок из-за отсутствия страниц. Конечно, можно сочетать все это с любыми другими параметрами выбора процессов, как рассказано на странице руководства ps(1).

Просмотр страниц ошибок по процессам может помочь вам сконцентрироваться на отдельных проблематичных компонентах. Тем не менее, если вы заинтересованы в производительности системы в целом, вам необходим инструмент, позволяющий вывести итог по использованию процессора и памяти всеми процессами.

# 8.10. Отслеживание производительности процессора и памяти с помощью команды vmstat

Среди множества инструментов, доступных для отслеживания производительности, команда vmstat является одним из самых старых, содержащих минимум необходимой информации. Она пригодится для получения высокоуровневого представления о том, как часто ядро выполняет подкачку страниц, насколько загружен процессор и как используются ресурсы ввода/вывода.

Хитрость в овладении мощью команды vmstat состоит в понимании ее отчета. Вот, например, результаты работы команды vmstat 2, которая сообщает статистику каждые две секунды:

```
$ vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff    cache  si  so   bi    bo  in  cs us sy id wa
 2  0 320416 3027696 198636 1072568   0   0    1     1   2   0 15  2 83  0
 2  0 320416 3027288 198636 1072564   0   0    0  1182 407 636  1  0 99  0
 1  0 320416 3026792 198640 1072572   0   0    0    58 281 537  1  0 99  0
 0  0 320416 3024932 198648 1074924   0   0    0   308 318 541  0  0 99  1
 0  0 320416 3024932 198648 1074968   0   0    0     0 208 416  0  0 99  0
 0  0 320416 3026800 198648 1072616   0   0    0     0 207 389  0  0 100 0
```

Этот вывод распределяется по таким категориям: procs — для процессов, memory — для использования памяти, swap — для страниц, которые перемещаются в область подкачки и из нее, io — для использования диска, system — для количества переключений ядра на его код и cpu — для количества времени, затраченного различными частями системы.

Приведенный выше пример типичен для систем, которые не выполняют много работы. Обычно следует начинать просмотр со второй строки — в первой содержатся средние значения за все время работы системы. Например, в данном случае система переместила на диск (swpd) 320 416 Кбайт памяти, при этом свободно около 3 025 000 Кбайт (3 Гбайт) реальной памяти. Хотя некоторая часть области подкачки использована, нулевые значения в столбцах si (swap-in, «входящая» подкачка) и so (swap-out, «выходящая» подкачка) говорят о том, что в данный момент

ядро не занято никаким из видов подкачки с диска. Столбец `buff` сообщает объем памяти, который ядро использует для дисковых буферов (см. подраздел 4.2.5).

В правом столбце с заголовком `CPU` можно увидеть распределение процессорного времени (столбцы `us`, `sy`, `id` и `wa`). Они сообщают соответственно процентное соотношение времени, которое процессор тратит на задачи пользователя, системные задачи (задачи ядра), бездействие и ожидание ввода/вывода. В приведенном примере запущено не так много пользовательских процессов (они используют не более 1 % процессорного времени); ядро не делает практически ничего, в то время как процессор находится в бездействии 99 % всего времени.

Теперь взгляните, что происходит, если через некоторое время запускается большая команда (первые две строки появились перед самым запуском программы) (пример 8.3).

**Пример 8.3.** Активность памяти

```
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
r b     swpd   free   buff     cache  si  so    bi   bo   in   cs us sy id wa
1 0   320412  2861252  198920 1106804   0   0     0    0 2477 4481 25 2 72 0❶
1 0   320412  2861748  198924 1105624   0   0     0   40 2206 3966 26 2 72 0
1 0   320412  2860508  199320 1106504   0   0   210   18 2201 3904 26 2 71 1
1 1   320412  2817860  199332 1146052   0   0 19912    0 2446 4223 26 3 63 8
2 2   320284  2791608  200612 1157752 202   0  4960  854 3371 5714 27 3 51 18❷
1 1   320252  2772076  201076 1166656  10   0  2142 1190 4188 7537 30 3 53 14
0 3   320244  2727632  202104 1175420  20   0  1890  216 4631 8706 36 4 46 14
```

Как следует из примера 8.3 (маркер ❶), процессор используется в течение продолжительного периода, в особенности пользовательскими процессами. Поскольку свободной памяти достаточно, объем использованного кэша и буфера начинает возрастать, так как ядро применяет диск сильнее.

Чуть позже можно увидеть интересное (маркер ❷): ядро извлекает в память страницы из области подкачки (столбец `si`). Это означает, что команда, которая только что запустилась, запросила некоторые из страниц, используемых совместно с другим процессом. Такое встречается часто, многие процессы применяют код из определенных общих библиотек только при своем запуске.

Обратите также внимание на то, что столбец `b` сообщает о том, что некоторые процессы *блокированы* (им не разрешен запуск) в ожидании страниц памяти. В целом количество свободной памяти уменьшается, но до ее нехватки еще очень далеко. Наблюдается также значительное количество дисковой активности, что отмечено увеличением значений в столбцах `bi` (blocks in, блоки «на входе») и `bo` (blocks out, блоки «на выходе»).

Результат будет совсем другим, если возникнет нехватка памяти. По мере уменьшения свободного пространства будут уменьшаться и размеры буфера с кэшем, поскольку ядру все в большей степени требуется пространство для пользовательских процессов. Когда не останется совсем ничего, вы увидите активность в столбце `so` («выходящая» подкачка), так как ядро начинает перемещать страницы на диск. В этот момент практически все остальные столбцы вывода изменятся, чтобы отобразить количество выполняемой ядром работы. Вы заметите, что увеличилось системное время, больше данных перемещается

на диск и с него, а также больше процессов заблокировано, поскольку память, которую они намерены использовать, недоступна (она перемещена в область подкачки).

Я объяснил не все столбцы вывода команды `vmstat`. Узнать подробности вы можете на странице руководства vmstat(8). Чтобы лучше их понимать, сначала может потребоваться узнать больше о том, как ядро управляет памятью: из лекций или книги вроде *Operating System Concepts* («Общие представления об операционных системах»), 9-е издание (Wiley, 2012).

# 8.11. Отслеживание ввода/вывода

По умолчанию команда `vmstat` выводит некоторую общую статистику ввода/вывода. Хотя можно получить детализированные сведения об использовании ресурсов каждого раздела с помощью команды `vmstat -d`, в этом случае вывод будет довольно объемным. Попробуйте начать с инструмента, предназначенного только для статистики ввода/вывода, — команды `iostat`.

## 8.11.1. Использование команды iostat

Подобно команде `vmstat`, при запуске без параметров команда `iostat` показывает статистику за все время работы компьютера:

```
$ iostat
[kernel information]
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           4.46    0.01    0.67    0.31    0.00   94.55

Device:            tps   kB_read/s   kB_wrtn/s   kB_read   kB_wrtn
sda               4.67        7.28       49.86   9493727  65011716
sde               0.00        0.00        0.00      1230         0
```

Часть `avg-cpu` в верхней части сообщает ту же информацию об использовании процессора, что и другие утилиты, которые вы видели в этой главе. Перейдите к нижней части, которая показывает для каждого из устройств следующее.

| | |
|---|---|
| tps | Среднее количество пересылок данных в секунду |
| kB_read/s | Среднее количество считанных килобайтов в секунду |
| kB_wrtn/s | Среднее количество записанных килобайтов в секунду |
| kB_read | Общее количество считанных килобайтов |
| kB_wrtn | Общее количество записанных килобайтов |

Еще одно сходство с командой `vmstat` таково: можно передавать величину интервала как аргумент, например `iostat 2`, чтобы результаты обновлялись каждые 2 секунды. При использовании интервала может потребоваться отобразить отчет только об устройстве. Для этого применяется параметр `-d` (например, `iostat -d 2`).

По умолчанию в отчете команды `iostat` не приводится информация о разделах. Чтобы отобразить всю такую информацию, используйте параметр `-p ALL`. Поскольку

в типичной системе бывает несколько разделов, вы получите обширный отчет. Вот фрагмент того, что вы можете увидеть:

```
$ iostat -p ALL
--snip
--Device:              tps       kB_read/s       kB_wrtn/s    kB_read     kB_wrtn
--snip-
sda                  4.67            7.27           49.83    9496139    65051472
sda1                 4.38            7.16           49.51    9352969    64635440
sda2                 0.00            0.00            0.00          6           0
sda5                 0.01            0.11            0.32     141884      416032
scd0                 0.00            0.00            0.00          0           0
--snip--
sde                  0.00            0.00            0.00       1230           0
```

В этом примере все устройства `sda1`, `sda2` и `sda5` являются разделами диска `sda`, поэтому между столбцами, относящимися к чтению и записи, будет небольшое наложение данных. Однако сумма значений, относящихся к разделам, не обязательно должна равняться значению для диска. Несмотря на то что чтение с устройства `sda1` также рассматривается как чтение с диска `sda`, помните о том, что с диска `sda` можно выполнять считывание напрямую, например при чтении таблицы разделов.

## 8.11.2. Отслеживание использования ввода/вывода каждого процесса с помощью команды iotop

Если вам необходимо копнуть глубже, чтобы увидеть ресурсы ввода/вывода, используемые отдельными процессами, вам может помочь команда `iotop`. Применение этой команды похоже на работу с командой `top`. Появляется постоянно обновляемый отчет, который показывает процессы, использующие большую часть ресурсов ввода/вывода, а общий итог приведен вверху:

```
# iotop
Total DISK READ:         4.76 K/s | Total DISK WRITE:     333.31 K/s
  TID  PRIO  USER       DISK READ DISK  WRITE  SWAPIN     IO> COMMAND
  260 be/3 root          0.00 B/s  38.09 K/s  0.00 %  6.98 % [jbd2/sda1-8]
 2611 be/4 juser         4.76 K/s  10.32 K/s  0.00 %  0.21 % zeitgeist-daemon
 2636 be/4 juser         0.00 B/s  84.12 K/s  0.00 %  0.20 % zeitgeist-fts
 1329 be/4 juser         0.00 B/s  65.87 K/s  0.00 %  0.03 % soffice.b~ash-pipe=6
 6845 be/4 juser 0.00 B/s 812.63 B/s 0.00 % 0.00 % chromium-browser
19069 be/4 juser 0.00 B/s 812.63 B/s 0.00 % 0.00 % rhythmbox
```

Обратите внимание на то, что здесь наряду со столбцами сведений о пользователе, команде и чтении/записи присутствует столбец `TID` (идентификатор потока) вместо идентификатора процесса. Инструмент `iotop` — одна из немногих утилит, которые отображают потоки вместо процессов.

Столбец `PRIO` (приоритет) отображает приоритет ввода/вывода. Он похож на приоритет процессора, который вы уже видели, но он влияет на то, насколько быстро ядро распределяет операции чтения и записи для процесса. В таком приори-

тете, как be/4, часть be является *классом обслуживания*, а число задает уровень приоритета. Как и для приоритетов процессора, более важными являются меньшие числа. Например, ядро отводит больше времени на ввод/вывод для процесса с приоритетом be/3, чем для процесса с приоритетом be/4.

Ядро использует класс обслуживания, чтобы обеспечить дополнительное управление планированием ввода/вывода. Вы увидите следующие три класса обслуживания в команде iotop.

- be — наилучший объем работы. Ядро старается наиболее справедливо распределить время ввода/вывода для этого класса. Большинство процессов запускаются в этом классе обслуживания.
- rt — реальное время. Ядро планирует любой ввод/вывод в реальном времени перед любым другим классом ввода/вывода, каким бы он ни был.
- idle — бездействие. Ядро выполняет ввод/вывод для этого класса только тогда, когда не должен быть выполнен никакой другой ввод/вывод. Для этого класса обслуживания не указывается уровень приоритета.

Можно проверить и изменить приоритет ввода/вывода для процесса с помощью утилиты ionice; подробности см. на странице руководства ionice(1). Хотя вам вряд ли потребуется беспокоиться о приоритетах ввода/вывода.

# 8.12. Отслеживание процессов с помощью команды pidstat

Вы увидели, как можно отслеживать конкретные процессы с помощью таких утилит, как top и iotop. Однако эти результаты обновляются в реальном времени, при каждом обновлении предыдущий отчет стирается. Утилита pidstat позволяет вам отследить использование ресурсов процессом с течением времени в стиле команды vmstat. Вот простой пример, в котором с ежесекундным обновлением отслеживается процесс 1329:

```
$ pidstat -p 1329 1
Linux 3.2.0-44-generic-pae (duplex)     07/01/2015      _i686_ (4 CPU)
09:26:55 PM       PID    %usr %system %guest %CPU CPU Command
09:27:03 PM      1329    8.00  0.00     0.00 8.00   1 myprocess
09:27:04 PM      1329    0.00  0.00     0.00 0.00   3 myprocess
09:27:05 PM      1329    3.00  0.00     0.00 3.00   1 myprocess
09:27:06 PM      1329    8.00  0.00     0.00 8.00   3 myprocess
09:27:07 PM      1329    2.00  0.00     0.00 2.00   3 myprocess
09:27:08 PM      1329    6.00  0.00     0.00 6.00   2 myprocess
```

В отчете по умолчанию приведены процентные отношения для пользовательского и системного времени, а также общая процентная доля процессорного времени. Есть даже сведения о том, на каком из процессоров запущен процесс. Столбец %guest представляет нечто необычное: это процентное отношение времени, которое процесс потратил на выполнение чего-либо внутри виртуальной машины. Если вы не запускаете виртуальную машину, не беспокойтесь о нем.

Хотя команда pidstat по умолчанию показывает использование процессора, она может намного больше этого. Например, можно применять параметр -r, чтобы отслеживать память, или параметр -d, чтобы включить отслеживание диска. Попробуйте применить их, а затем загляните на страницу руководства pidstat(1), чтобы узнать еще больше подробностей о потоках, переключении контекста или о чем-либо еще, что обсуждалось в данной главе.

# 8.13. Дополнительные темы

Одна из причин, почему существует так много инструментов для измерения использования ресурсов, в том, что множество типов ресурсов потребляется различными способами. В этой главе вы видели, как ресурсы процессора, памяти, ввода/вывода и системы использовались процессами, потоками внутри процессов и ядром.

Еще одна причина — ограниченность ресурсов. Чтобы система работала, ее компоненты должны стремиться к потреблению меньшего количества ресурсов. В прошлом за одним компьютером работало несколько пользователей, поэтому было необходимо обеспечить каждого из них достаточной долей ресурсов. Сейчас, хотя современные ПК могут и не иметь нескольких пользователей, они по-прежнему имеют множество процессов, соревнующихся за ресурсы. Точно так же и для высокопроизводительных сетевых серверов необходимо тщательное отслеживание ресурсов.

Дополнительные темы, относящиеся к отслеживанию ресурсов и анализу производительности, включают следующее.

- sar (System Activity Reporter, обозреватель системной активности). Пакет sar содержит многие из функций для непрерывного отслеживания команды vmstat, но он также выполняет запись использования ресурсов с течением времени. С помощью пакета sar можно узнать, что делала ваша система в определенный момент времени. Это удобно, когда необходимо проанализировать системное событие, которое уже произошло.
- acct (учет процессов). Пакет acct может регистрировать процессы и использование ресурсов ими.
- **Квоты.** Многие системные ресурсы можно ограничить в зависимости от процесса или от пользователя. Некоторые параметры применения процессора и памяти содержатся в файле /etc/security/limits.conf; есть также страница руководства limits.conf(5). Это функция стандарта PAM, и процессы будут подчиняться ей только тогда, когда они были запущены из чего-либо, что использует стандарт PAM (например, из оболочки входа в систему). Можно также ограничить количество дискового пространства, которое может потреблять пользователь, с помощью системы quota.

Если вы заинтересованы настройкой системы и, в частности, ее производительностью, книга Брендана Грегга (Brendan Gregg) *Systems Performance: Enterprise and the Cloud* («Производительность системы: предприятие и облако») (Prentice Hall, 2013) содержит намного больше подробностей.

Мы еще не коснулись множества инструментов, которые могут быть использованы при отслеживании потребления сетевых ресурсов. Чтобы их применять, сначала необходимо понять, как устроена сеть. Именно к этому мы сейчас и перейдем.

# 9 Представление о сети и ее конфигурации

Создание сети — это установка соединения между компьютерами и пересылка данных между ними. Звучит достаточно просто, но, чтобы понять, как это работает, необходимо задать два главных вопроса.

- Каким образом компьютер, отправляющий данные, знает, *куда* их отправлять?
- Когда компьютер-адресат получает данные, как он догадывается о том, *что́* он только что получил?

Компьютер отвечает на эти вопросы, используя набор компонентов, каждый из которых ответственен за определенный аспект отправки, получения и идентификации данных. Эти компоненты организованы в виде групп, которые формируют *сетевые уровни*, расположенные один над другим, чтобы создать законченную систему. Ядро Linux обращается с сетью подобно тому, как оно работает с подсистемой SCSI, описанной в главе 3.

Поскольку каждый уровень стремится быть независимым, есть возможность построить сети с помощью различных комбинаций компонентов. Именно здесь конфигурация сети может стать очень сложной. По этой причине мы начнем данную главу с рассмотрения уровней в очень простых сетях. Вы узнаете о том, как просматривать параметры вашей сети, а когда усвоите основы работы каждого уровня, вы будете готовы к изучению того, как самостоятельно конфигурировать эти уровни. Наконец, мы перейдем к таким сложным темам, как построение собственных сетей и настройка брандмауэров. Пропустите этот материал, если ваши глаза начинают тускнеть; вы всегда сможете вернуться к нему позже.

## 9.1. Основные понятия о сети

Прежде чем разбираться с теорией сетевых уровней, взгляните на простую сеть, показанную на рис. 9.1.

Такой тип сети является повсеместным, подобным образом сконфигурировано большинство сетей в квартирах и небольших офисах. Каждый компьютер, подключенный к этой сети, называется *хостом*. Хосты подключены к *маршрутизатору*, который является хостом, способным передавать данные от одной сети к другой. Эти компьютеры (то есть хосты A, B и C) и маршрутизатор составляют локальную сеть (LAN, local area network). Подключения в локальной сети могут быть проводными или беспроводными.

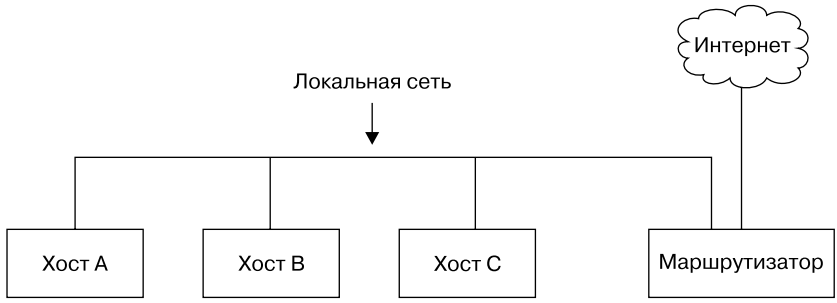


**Рис. 9.1.** Типичная локальная сеть с маршрутизатором, который обеспечивает доступ в Интернет

Маршрутизатор подключен также к Интернету, изображенному на рисунке в виде облака. Поскольку маршрутизатор подключен одновременно и к локальной сети, и к Интернету, все компьютеры локальной сети также имеют доступ к Интернету через маршрутизатор. Одной из целей данной главы является описание того, каким образом маршрутизатор обеспечивает такой доступ.

Наша исходная точка обзора будет располагаться в компьютере с Linux, таком как хост A в локальной сети на рис. 9.1.

**Пакеты.** Компьютер передает данные по сети в виде небольших порций, называемых *пакетами*. Они состоят из двух частей: *заголовка* и *полезной нагрузки*. Заголовок содержит такую идентифицирующую информацию, как хосты происхождения/назначения и основной протокол. С другой стороны, полезная нагрузка — это реальные данные, которые компьютер собирается передать (например, код HTML или изображение).

Пакеты позволяют хосту взаимодействовать с другими хостами «одновременно», поскольку хосты могут отправлять, получать и обрабатывать пакеты в любом порядке, вне зависимости от того, откуда они поступили и куда направляются. Разбиение сообщений на небольшие части также облегчает нахождение и устранение ошибок, возникших при передаче.

В большинстве случаев вам не придется заботиться о переводе пакетов в данные, которые используют ваши приложения, поскольку в операционной системе есть необходимые для этого средства. Однако полезно узнать о роли пакетов в сетевых уровнях, к чему мы и приступаем.

## 9.2. Сетевые уровни

Полностью функционирующая сеть содержит полный набор сетевых уровней, называемый *сетевым стеком*. В любой действующей сети есть стек. Типичный интернет-стек выглядит следующим образом, от верхнего уровня к нижнему.

- **Прикладной уровень.** Содержит «язык», с помощью которого общаются приложения и серверы. Как правило, это какой-либо протокол верхнего уровня. Самыми распространенными протоколами прикладного уровня являются:

HTTP (Hypertext Transfer Protocol, протокол передачи гипертекстовых файлов; используется во Всемирной паутине), SSL (Secure Socket Layer, протокол защищенных сокетов) и FTP (File Transfer Protocol, протокол передачи файлов). Протоколы прикладного уровня часто могут сочетаться. Так, например, протокол SSL обычно используется в соединении с протоколом HTTP.

- **Транспортный уровень.** Определяет характеристики передачи данных для прикладного уровня. Этот уровень содержит проверку целостности данных, порты источника и назначения, а также спецификации по разбиению данных приложения на пакеты (если прикладной уровень еще не выполнил это). Самыми распространенными протоколами транспортного уровня являются TCP (Transmission Control Protocol, протокол управления передачей) и UDP (User Datagram Protocol, протокол дейтаграмм пользователя). Транспортный уровень иногда также называют *уровнем протоколов*.
- **Сетевой или интернет-уровень.** Определяет, как перемещать пакеты от хоста-источника к хосту-назначению. Частные правила передачи пакетов через Интернет известны как протокол IP (Internet Protocol, интернет-протокол). Поскольку в этой книге речь пойдет только о сети Интернет, мы на самом деле будем говорить лишь об интернет-уровне. Тем не менее, так как сетевые уровни задуманы как не зависящие от аппаратных средств, можно одновременно настроить несколько независимых сетевых уровней (таких как IP, IPv6, IPX и AppleTalk) на одном хосте.
- **Физический уровень.** Определяет, как необработанные данные передаются через физический посредник, например сеть Ethernet или модем. Иногда этот уровень называют *связывающим уровнем* или *уровнем «хост-сеть»*.

Важно понимать структуру сетевого стека, поскольку данные должны пройти через эти уровни минимум дважды, прежде чем достигнут программу на месте назначения. Если, например, вы отправляете данные от хоста A к хосту B, как показано на рис. 9.1, байты покидают прикладной уровень хоста A и перемещаются через транспортный и сетевой уровни хоста A. Затем они попадают вниз, в физический посредник, передаются по нему, после чего поднимаются вверх через различные уровни до прикладного уровня хоста B очень сходным образом. Если что-либо отправляется на интернет-хост через маршрутизатор, то данные пройдут через некоторые (но, как правило, не все) уровни маршрутизатора и всего, что находится между ними.

Иногда уровни странным образом вклиниваются друг в друга, поскольку было бы неэффективно продвигаться по всем уровням последовательно. Например, устройства, которые исторически имели дело только с физическим уровнем, теперь иногда заглядывают в данные транспортного и интернет-уровней, чтобы быстро отфильтровать и проложить маршрут для данных. Не беспокойтесь об этом, пока вы изучаете основы.

Мы начнем с рассмотрения того, как компьютер с Linux подключается к сети, чтобы ответить на вопрос *«куда?»*, поставленный в начале этой главы. Это самая нижняя часть стека — физический и сетевой уровни. Далее мы рассмотрим два верхних уровня, чтобы ответить на вопрос *«что?»*.

ПРИМЕЧАНИЕ

Вы, наверное, слышали о другом наборе уровней, известном как модель OSI (Open Systems Interconnection Reference Model). Это модель сети, которая содержит семь уровней и часто используется при обучении и в разработке сетей, однако мы не будем рассматривать ее, поскольку вы будете напрямую работать с четырьмя уровнями, описанными здесь. Чтобы узнать об уровнях больше (и о сетях вообще), обратитесь к книге Эндрю С. Таненбаума (Andrew S. Tanenbaum) и Дэвида Дж. Уэзерола (David J. Wetherall) Computer Networks («Компьютерные сети»), 5-е издание (Prentice Hall, 2010).

# 9.3. Интернет-уровень

Вместо того чтобы начать с самого низа сетевого стека, физического уровня, мы начнем с сетевого уровня, поскольку его проще понять. Интернет, как мы уже знаем, основан на интернет-протоколе, начиная с версии 4 (IPv4) и заканчивая набирающей силу версией 6 (IPv6). Одним из самых важных аспектов интернет-уровня является то, что он призван быть сетью программного обеспечения, которое не предъявляет специальных требований к аппаратным средствам или операционным системам. Суть в том, что вы можете отправлять и получать интернет-пакеты с помощью любого аппаратного обеспечения, используя любую операционную систему.

Топология Интернета децентрализована; эта сеть составлена из более мелких сетей, называемых *подсетями*. Идея заключается в том, что все подсети каким-либо образом соединены между собой. Например, на рис. 9.1 локальная сеть обычно является единственной подсетью.

Хост может быть подключен к более чем одной подсети. Как вы видели в разделе 9.1, такой тип хоста называется маршрутизатором, если он может передавать данные из одной подсети в другую (еще один термин для маршрутизатора — *шлюз*). Рисунок 9.2 уточняет рис. 9.1 за счет идентификации локальной сети в качестве подсети, а также за счет добавления интернет-адресов для каждого хоста и для маршрутизатора. Маршрутизатор на этом рисунке обладает двумя адресами: 10.23.2.1 в локальной подсети, а также интернет-ссылкой (сейчас нам неважен адрес интернет-ссылки, поэтому она отмечена как «Адрес ссылки верхнего уровня»). Сначала мы рассмотрим адреса, а затем обозначение подсети.

Каждый интернет-хост обладает по крайней мере одним численным IP-адресом в виде *a.b.c.d*, например 10.23.2.37. Адрес в подобной записи называется *четверкой чисел, разделенных точками*. Если хост подключен к нескольким подсетям, у него есть хотя бы один IP-адрес для каждой подсети. Каждый IP-адрес хоста должен быть уникальным для сети Интернет в целом, однако, как вы увидите позже, частные сети и преобразование сетевых адресов (NAT) могут вызвать небольшую путаницу.

ПРИМЕЧАНИЕ

С технической точки зрения IP-адрес состоит из 4 байт (или 32 бит), abcd. Байты a и d являются числами от 1 до 254, а байты b и c — числами от 0 до 255. Компьютер обрабатывает IP-адреса в виде «сырых» байтов. Тем не менее человеку намного проще читать и записывать адрес в виде четверки чисел, разделенных точками, вроде 10.23.2.37, а не в виде неприглядного шестнадцатеричного числа 0x0A170225.

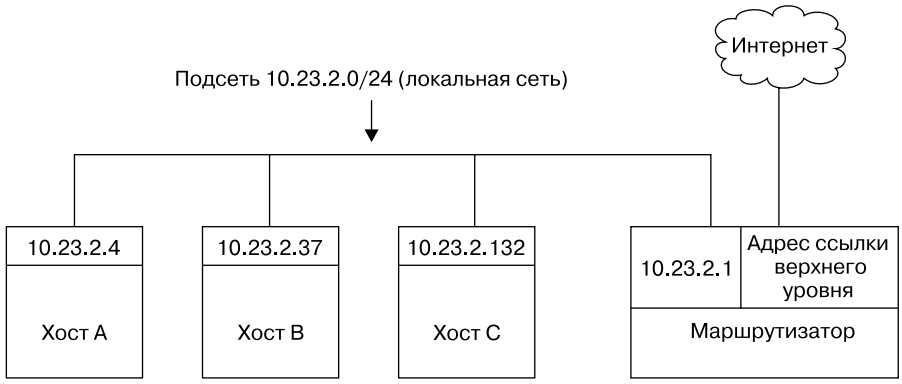


**Рис. 9.2.** Сеть с IP-адресами

IP-адреса напоминают почтовые адреса. Чтобы взаимодействовать с другим хостом, ваш компьютер должен знать IP-адрес этого хоста. Посмотрим на адреса в вашем компьютере.

## 9.3.1. Просмотр IP-адресов компьютера

У одного хоста может быть несколько IP-адресов. Чтобы увидеть адреса, которые активны на вашем компьютере с Linux, запустите такую команду:

```
$ ifconfig
```

Результат вывода окажется, вероятно, довольно обширным, но он должен содержать нечто подобное:

```
eth0      Link encap:Ethernet  HWaddr 10:78:d2:eb:76:97
          inet addr:10.23.2.4  Bcast:10.23.2.255  Mask:255.255.255.0
          inet6 addr: fe80::1278:d2ff:feeb:7697/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:85076006 errors:0 dropped:0 overruns:0 frame:0
          TX packets:68347795 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:86427623613 (86.4 GB)  TX bytes:23437688605 (23.4 GB)
          Interrupt:20 Memory:fe500000-fe520000
```

Вывод команды `ifconfig` включает множество деталей об интернет-уровне и о физическом уровне (иногда он даже совсем не содержит интернет-адреса!). Более подробно мы обсудим этот вывод позже, а сейчас сосредоточимся на второй строке, которая сообщает, что хост настроен на использование адреса IPv4 (`inet addr`) 10.23.2.4. В той же самой строке для параметра `Mask` указано значение 255.255.255.0. Это *маска подсети*, определяющая подсеть, к которой принадлежит IP-адрес. Посмотрим, как это устроено.

ПРИМЕЧАНИЕ

Команда ifconfig, подобно некоторым другим, которые вы встретите в этой главе (такие как route и arp), на техническом уровне вытеснена более новой командой ip. Команда ip способна выполнить

больше, чем старые команды, и она предпочтительнее при написании сценариев. Однако большинство пользователей по-прежнему использует старые команды при настройки сети вручную, и такие команды также могут быть применены в других версиях Unix. По этой причине мы будем применять команды «в старом стиле».

## 9.3.2. Подсети

*Подсеть* — это соединенная группа хостов, IP-адреса которых каким-либо образом упорядочены. Обычно такие хосты расположены в одной физической сети, как показано на рис. 9.2. Например, хосты между 10.23.2.1 и 10.23.2.254 могли бы составлять подсеть, равно как и хосты между 10.23.1.1 и 10.23.255.254.

Подсеть определяется с помощью двух фрагментов: *сетевого префикса* и *маски подсети* (вроде той, которую вы видели в отчете команды `ifconfig` в предыдущем разделе). Предположим, необходимо создать подсеть, содержащую IP-адреса от 10.23.2.1 до 10.23.2.254. Сетевой префикс — та часть, которая является общей для всех адресов данной подсети; в приведенном примере это 10.23.2.0, а маской подсети будет 255.255.255.0. Посмотрим, почему эти числа правильны.

Не сразу понятно, каким образом префикс и маска работают совместно, чтобы дать вам все возможные IP-адреса в подсети. Выяснить это поможет просмотр данных чисел в двоичном представлении. Маска отмечает положения битов в IP-адресе, которые являются общими для подсети. Вот, например, двоичная запись адресов 10.23.2.0 и 255.255.255.0:

```
10.23.2.0:        00001010 00010111 00000010 00000000
255.255.255.0:    11111111 11111111 11111111 00000000
```

Теперь выделим жирным шрифтом те положения битов в адресе 10.23.2.0, которые являются единицами в адресе 255.255.255.0:

```
10.23.2.0:        00001010 00010111 00000010 00000000
```

Посмотрите на биты, которые *не* выделены жирным шрифтом. Для любого из них можно установить значение 1, чтобы получить правильный IP-адрес для данной подсети, за исключением случая, когда все биты являются нулями или единицами.

Собирая все воедино, можно понять, как хост с IP-адресом 10.23.2.1 и маской подсети 255.255.255.0 оказывается в той же подсети, что и любой другой компьютер, IP-адрес которого начинается с 10.23.2. Можно обозначить в целом эту подсеть как 10.23.2.0/255.255.255.0.

## 9.3.3. Распространенные маски подсети и нотация CIDR

Если вам повезет, вы будете иметь дело в основном с простыми масками подсети вроде 255.255.255.0 или 255.255.0.0. В случае невезения вам может попасться адрес 255.255.255.192, для которого не так-то просто установить набор адресов, принадлежащих подсети. Более того, возможно, что вы встретите другую форму представления подсети, которая называется нотацией CIDR (Classless Inter-Domain Routing,

бесклассовая междоменная маршрутизация). В ней подсеть 10.23.2.0/255.255.255.0 будет записана в виде 10.23.2.0/24.

Чтобы понять, что это значит, посмотрите на маску в двоичной форме (как в примере, который вы видели в предыдущем разделе). Вы обнаружите, что практически все маски подсетей являются всего лишь набором единиц, за которыми следует набор нулей. Например, вы только что видели, что адрес 255.255.255.0 в двоичной записи представлен в виде 24 единичных бит, за которыми следуют 8 нулевых бит. Нотация CIDR идентифицирует маску подсети по количеству *ведущих* единиц в записи маски подсети. Следовательно, такая комбинация, как 10.23.2.0/24, содержит как префикс подсети, так и маску подсети.

В табл. 9.1 приведено несколько примеров масок подсети и их записи в форме CIDR.

**Таблица 9.1.** Маски подсети

| Длинная форма | Форма CIDR |
|---|---|
| 255.0.0.0 | 8 |
| 255.255.0.0 | 16 |
| 255.240.0.0 | 12 |
| 255.255.255.0 | 24 |
| 255.255.255.192 | 26 |

ПРИМЕЧАНИЕ

Если вы не очень хорошо знакомы с преобразованием чисел в десятичный, двоичный и шестнадцатеричный форматы, можно воспользоваться утилитой-калькулятором, например bc или dc, чтобы переводить числа с различными основаниями системы счисления. Например, в утилите bc можно запустить команду obase=2; 240, чтобы вывести число 240 в двоичной форме (основание равно 2).

Идентификация подсетей и их хостов является первым строительным блоком в понимании того, как работает Интернет. Но тем не менее подсети еще предстоит соединить.

## 9.4. Маршруты и таблица маршрутизации ядра

Соединение подсетей Интернет заключается в основном в идентификации хостов, подключенных к более чем одной подсети. Вернитесь к рис. 9.2 и поразмышляйте о хосте A с IP-адресом 10.23.2.4. Этот хост подключен к локальной сети 10.23.2.0/24 и может напрямую взаимодействовать с хостами этой сети. Чтобы добраться до остальной части Интернета, он должен «общаться» через маршрутизатор с адресом 10.23.2.1.

Как ядро Linux различает эти два типа назначений? Чтобы выбрать для себя образ действий, оно использует конфигурацию назначений, которая называется *таблицей маршрутизации*. Чтобы отобразить таблицу маршрутизации, применяйте команду `route -n`. Вот что вы могли бы увидеть для простого хоста, такого как хост с адресом 10.23.2.4:

```
$ route -n
Kernel IP routing table
Destination     Gateway     Genmask       Flags Metric Ref    Use Iface
0.0.0.0         10.23.2.1   0.0.0.0       UG    0      0        0 eth0
10.23.2.0       0.0.0.0     255.255.255.0 U     1      0        0 eth0
```

Две последние строки здесь содержат информацию о маршрутизации. Столбец `Destination` сообщает префикс сети, а столбец `Genmask` — маску, которая соответствует данной сети. В этом выводе определены две сети: 0.0.0.0/0 (которая соответствует каждому адресу в Интернете) и 10.23.2.0/24. У каждой из этих сетей в столбце `Flags` стоит символ `U`, который говорит о том, что данный маршрут активен («up»).

Различие между назначениями заложено в комбинации значений столбцов `Gateway` и `Flags`. Для адреса 0.0.0.0/0 в столбце `Flags` указан флаг `G`, который означает, что для данной сети связь должна проходить через шлюз, указанный в столбце `Gateway` (в данном случае 10.23.2.1). Однако для сети 10.23.2.0/24 в столбце `Flags` нет символа `G`, это говорит о том, что данная сеть подключена напрямую каким-либо способом. Здесь 0.0.0.0 используется в качестве заместителя значения в столбце `Gateway`. Не обращайте пока внимания на остальные столбцы вывода.

Есть некоторая хитрость: допустим, хост собирается отправить что-либо по адресу 10.23.2.132, который соответствует обоим правилам таблицы маршрутизации, 0.0.0.0/0 и 10.23.2.0/24. Как ядро узнает о том, что необходимо применить второй адрес? Оно выбирает самый длинный совпадающий префикс назначения. Именно здесь нотация CIDR становится чрезвычайно удобной: адрес 10.23.2.0/24 годится, и длина его префикса равна 24 битам; адрес 0.0.0.0/0 тоже подходит, но его префикс — нулевой длины (то есть у него нет префикса), поэтому берет верх правило для адреса 10.23.2.0/24.

ПРИМЕЧАНИЕ

Параметр -n просит команду route отобразить IP-адреса вместо имен хостов и сетей. Следует помнить об этом важном параметре, поскольку вы сможете использовать его в других командах, относящихся к работе с сетью, таких как netstat.

**Шлюз по умолчанию.** Запись для адреса 0.0.0.0/0 в таблице маршрутизации имеет особое значение, поскольку она соответствует любому адресу в Интернете. Это *маршрут по умолчанию*, и адрес, который указан в столбце `Gateway` (в результате вызова команды `route -n`) для маршрута по умолчанию, является *шлюзом по умолчанию*. Когда остальные правила не подходят, маршрут по умолчанию всегда годится, а в шлюз по умолчанию отправляются сообщения, если нет другого выбора. Можно настроить хост без шлюза по умолчанию, но он будет неспособен подключиться к тем хостам, назначения которых отсутствуют в таблице маршрутизации.

ПРИМЕЧАНИЕ

В большинстве сетей с маской 255.255.255.0 маршрутизатор обычно расположен по адресу подсети 1 (например, 10.23.2.1 в сети 10.23.2.0/24). Поскольку это просто договоренность, возможны исключения.

# 9.5. Основные инструменты, использующие протокол ICMP и службу DNS

Теперь пришло время рассмотреть некоторые основные утилиты, помогающие взаимодействовать с хостами. Эти инструменты используют два протокола, представляющих особый интерес: ICMP (Internet Control Message Protocol, протокол управляющих сообщений в Интернете), который может помочь в искоренении проблем с подключением и маршрутизацией, и систему DNS (Domain Name Service, служба доменных имен), которая сопоставляет имена с IP-адресами, чтобы вам не приходилось запоминать уйму чисел.

## 9.5.1. Команда ping

Команда `ping` (см. http://ftp.arl.mil/~mike/ping.html) является одним из главнейших инструментов для сетевой отладки. Она отправляет пакеты эхо-запроса по протоколу ICMP какому-либо хосту-адресату и просит вернуть его отправителю. Если принимающий хост получает пакет и настроен на ответ, он отправляет эхо-ответ по протоколу ICMP.

Допустим, например, что вы запустили команду `ping 10.23.2.1` и получили такой результат:

```
$ ping 10.23.2.1
PING 10.23.2.1 (10.23.2.1) 56(84) bytes of data.
64 bytes from 10.23.2.1: icmp_req=1 ttl=64 time=1.76 ms
64 bytes from 10.23.2.1: icmp_req=2 ttl=64 time=2.35 ms
64 bytes from 10.23.2.1: icmp_req=4 ttl=64 time=1.69 ms
64 bytes from 10.23.2.1: icmp_req=5 ttl=64 time=1.61 ms
```

Первая строка говорит о том, что вы отправляете пакеты из 56 байт (если учитывать заголовки, то из 84 байтов) по адресу 10.23.2.1 (по умолчанию один пакет в секунду); остальные строки приводят отклик от хоста с адресом 10.23.2.1. Самыми важными частями вывода являются порядковый номер (`icmp_req`) и время прохождения в прямом и обратном направлениях (`time`). Количество возвращенных байтов равно размеру отправленного пакета плюс 8. Содержимое пакетов не имеет значения.

Разрывы в порядковых номерах (такие как между 2 и 4) обычно свидетельствуют о каких-либо неполадках с подключением. Возможно также, что пакеты будут приходить в неправильном порядке. Если это так, то это тоже говорит о проблеме, так как команда `ping` отправляет только один пакет в секунду. Если на доставку ответа требуется больше секунды (1000 мс), то подключение является крайне медленным.

Время прохождения — это интервал времени с момента отправки пакета-запроса до момента прибытия пакета-ответа. Если достичь назначения не представляется возможным, конечный маршрутизатор, который видит данный пакет, возвращает команде `ping` ICMP-пакет host unreachable («хост недоступен»).

В проводной локальной сети следует ожидать полное отсутствие потери пакетов и очень малое значение времени прохождения. Приведенный выше пример взят

из беспроводной сети. Следует также ожидать отсутствие потери пакетов и устойчивую величину времени прохождения при передаче между вашей сетью и поставщиком интернет-услуг.

ПРИМЕЧАНИЕ

Из соображений безопасности не все хосты в Интернете отвечают на пакеты эхо-запросов, поэтому вы можете подключиться к какому-либо сайту на хосте, но не получить ответа на команду ping.

## 9.5.2. Команда traceroute

Основанная на протоколе ICMP команда `traceroute` окажется полезной, когда вы дойдете в этой главе до материала, посвященного уровню маршрутизации. Воспользуйтесь командой `traceroute` *host*, чтобы увидеть путь, который проходят пакеты до удаленного хоста. Команда `traceroute -n` *host* отключает поиск имени хоста.

Одной из самых приятных черт команды `traceroute` является то, что она возвращает значения времени прохождения для каждого участка маршрута, как показано в приведенном ниже фрагменте вывода:

```
 4  206.220.243.106  1.163 ms  0.997 ms  1.182 ms
 5  4.24.203.65  1.312 ms  1.12 ms  1.463 ms
 6  64.159.1.225  1.421 ms  1.37 ms  1.347 ms
 7  64.159.1.38  55.642 ms  55.625 ms  55.663 ms
 8  209.247.10.230  55.89 ms  55.617 ms  55.964 ms
 9  209.244.14.226  55.851 ms 55.726 ms   55.832 ms
10  209.246.29.174  56.419 ms 56.44 ms   56.423 ms
```

Поскольку в этом выводе видна большая задержка между шагами 6 и 7, вероятно, эта часть маршрута является каким-либо протяженным звеном.

Результаты команды `traceroute` могут быть непоследовательными. Например, на некотором шаге ответы могут прерваться, чтобы затем «заново возникнуть» на следующих шагах. Обычно причиной является отказ маршрутизатора на этом шаге вернуть отладочный вывод, который ожидает команда `traceroute`, но при этом маршрутизаторы следующих этапов благополучно возвращают результат. Более того, маршрутизатор мог бы назначить более низкий приоритет отладочному трафику по сравнению с нормальным.

## 9.5.3. Служба DNS и хост

IP-адреса трудно запомнить, к тому же они могут измениться. Именно поэтому мы обычно пользуемся вместо них именами вроде www.example.com. Библиотека службы DNS в вашей системе, как правило, автоматически выполняет это преобразование, но иногда вам может потребоваться вручную перевести имя в IP-адрес. Чтобы определить IP-адрес, стоящий за доменным именем, используйте такую команду `host`:

```
$ host www.example.com
www.example.com has address 93.184.216.119
www.example.com has IPv6 address 2606:2800:220:6d:26bf:1447:1097:aa7
```

Обратите внимание на то, что в этом примере есть как адрес версии IPv4 (93.184.216.119), так и более длинный адрес версии IPv6. Это означает, что данный хост обладает также адресом сети Интернет следующего поколения.

Можно также использовать команду `host` наоборот: введите IP-адрес вместо имени хоста, чтобы попытаться определить имя хоста, соответствующее данному IP-адресу. Не ожидайте, что это будет работать надежно. Многие имена хостов могут представлять один и тот же IP-адрес, и служба DNS не знает, как определить, которое из имен соответствует указанному адресу. Администратор домена должен вручную настраивать это обратное определение, но зачастую администраторы этого не делают. Помимо команды `host`, есть много других моментов, относящихся к службе DNS. Мы рассмотрим основную конфигурацию клиента позже, в разделе 9.12.

## 9.6. Физический уровень и сеть Ethernet

Об Интернете следует усвоить одну ключевую идею — он является сетью программного обеспечения. Ничто из рассмотренного нами до настоящего момента не привязано к какому-либо аппаратному средству, и даже более того: одной из причин успеха Интернета является его способность работать практически в любом типе компьютера, операционной системы и физической сети. Тем не менее по-прежнему необходимо помещать сетевой уровень поверх какого-либо аппаратного обеспечения, и такой интерфейс называется физическим уровнем.

В данной книге мы рассмотрим наиболее распространенный тип физического уровня — сеть Ethernet. Семейство стандартов IEEE 802 определяет различные типы сетей Ethernet, от проводных до беспроводных, но их все объединяют некоторые ключевые особенности.

- Все устройства сети Ethernet обладают адресом *MAC* (*Media Access Control*, управление доступом к среде передачи данных), который иногда называют *аппаратным адресом*. Этот адрес не зависит от IP-адреса хоста и является уникальным для сети Ethernet этого хоста (но не обязательно для более крупной программной сети, такой как Интернет). MAC-адрес может быть, например, таким: 10:78:d2:eb:76:97.
- Устройства в сети Ethernet отправляют сообщения в виде *кадров*, которые являются оболочкой вокруг набора данных. Кадр содержит MAC-адреса отправителя и назначения.

На самом деле сеть Ethernet не пытается выйти за рамки аппаратного обеспечения для одной сети. Если, например, у вас есть две различные сети Ethernet с одним хостом, подключенным к обеим сетям (и два различных устройства сетевого интерфейса), вы не сможете напрямую передать кадр из одной сети Ethernet в другую, если вы не настроите специальный Ethernet-мост. Именно здесь возникают более высокие сетевые уровни (такие как интернет-уровень). По договоренности каждая сеть Ethernet является обычно и подсетью Интернета. Даже если кадр не может покинуть физическую сеть, маршрутизатор способен извлечь данные из кадра, заново упаковать их и отправить хосту другой физической сети — именно это и происходит в сети Интернет.

# 9.7. Понятие о сетевых интерфейсах ядра

Физический и интернет-уровни должны быть соединены таким способом, который позволяет интернет-уровню сохранять свою не зависящую от аппаратных средств гибкость. Ядро Linux обеспечивает собственное разделение этих двух уровней и предоставляет стандарты коммуникации для их соединения под названием *«сетевой интерфейс (ядра)»*. Когда вы настраиваете сетевой интерфейс, вы соединяете настройки IP-адреса со стороны Интернета с идентификацией аппаратного средства со стороны физического устройства. Сетевые интерфейсы имеют имена, которые обычно отражают тип расположенного под ними аппаратного средства, например `eth0` (первая карта Ethernet в компьютере) и `wlan0` (беспроводной интерфейс).

В подразделе 9.3.1 вы узнали о наиболее важной команде для просмотра или ручной настройки параметров сетевого интерфейса: `ifconfig`. Вспомните такой результат ее работы:

```
eth0      Link encap:Ethernet  HWaddr 10:78:d2:eb:76:97
          inet addr:10.23.2.4  Bcast:10.23.2.255  Mask:255.255.255.0
          inet6 addr: fe80::1278:d2ff:feeb:7697/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500   Metric:1
          RX packets:85076006 errors:0 dropped:0 overruns:0 frame:0
          TX packets:68347795 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:86427623613 (86.4 GB)  TX bytes:23437688605 (23.4 GB)
          Interrupt:20 Memory:fe500000-fe520000
```

Для каждого сетевого интерфейса в левой части вывода отображается имя интерфейса, а правая часть содержит параметры и статистические данные об этом интерфейсе. В дополнение к уже рассмотренным нами частям интернет-уровня можно увидеть MAC-адрес на физическом уровне (`HWaddr`). Строки, которые содержат слова `UP` и `RUNNING`, сообщают о том, что такой интерфейс работает.

Хотя команда `ifconfig` и показывает некоторую информацию об аппаратном средстве (в данном случае даже параметры низкоуровневого устройства, такие как прерывание и использованная память), она предназначена в основном для просмотра и конфигурирования программных уровней, присоединенных к интерфейсам. Чтобы проникнуть вглубь аппаратного и физического уровней, стоящих за сетевым интерфейсом, используйте что-либо вроде команды `ethtool` для отображения или изменения параметров карт Ethernet. Мы кратко рассмотрим беспроводные сети в разделе 9.23.

# 9.8. Введение в конфигурирование сетевого интерфейса

Теперь вы узнали обо всех основных элементах, которые входят в нижние уровни сетевого стека: физический уровень, сетевой (интернет-) уровень и сетевые интерфейсы ядра Linux. Чтобы объединить эти части в целях подключения компьютера с Linux к Интернету, вам или какому-либо ПО необходимо выполнить следующее.

1. Подключить сетевое оборудование и убедиться в том, что у ядра есть драйверы для него. Если драйвер есть, то команда `ifconfig -a` отобразит сетевой интерфейс ядра, соответствующий этому аппаратному средству.
2. Выполнить дополнительную настройку физического уровня, например выбор имени сети или пароля.
3. Связать IP-адрес и маску сети с сетевым интерфейсом ядра, чтобы драйверы устройства (физический уровень) и подсистемы Интернета (интернет-уровень) могли общаться друг с другом.
4. Добавить дополнительные необходимые маршруты, включая шлюз по умолчанию.

Когда все компьютеры были большими стационарными ящиками, соединенными с помощью проводов, все было сравнительно просто: ядро выполняло первый шаг, второй шаг не требовался и вам следовало выполнить третий шаг с помощью команды `ifconfig` и четвертый шаг с помощью команды `route`.

Чтобы вручную указать IP-адрес и маску сети для сетевого интерфейса ядра, нужно было выполнить следующее:

```
# ifconfig interface address netmask mask
```

Здесь параметр *interface* является именем интерфейса, таким как `eth0`. Когда интерфейс заработает, вам следует быть готовыми к добавлению маршрутов, что, как правило, сводится к указанию шлюза по умолчанию, например, так:

```
# route add default gw gw-address
```

Параметр *gw-address* является IP-адресом вашего шлюза по умолчанию; он должен быть адресом в локально подключенной подсети, определенной адресом и маской одного из сетевых интерфейсов.

**Добавление и удаление маршрутов вручную.** Чтобы удалить шлюз по умолчанию, запустите команду:

```
# route del -net default
```

Можно легко переопределить шлюз по умолчанию с помощью других маршрутов. Допустим, ваш компьютер находится в подсети 10.23.2.0/24, вы желаете попасть в подсеть 192.168.45.0/24 и знаете о том, что адрес 10.23.2.44 может выступать в роли маршрутизатора для этой подсети. Запустите такую команду, чтобы отправить трафик, связанный с адресом 192.168.45.0 на этот маршрутизатор:

```
# route add -net 192.168.45.0/24 gw 10.23.2.44
```

Нет необходимости указывать маршрутизатор, чтобы удалить маршрут:

```
# route del -net 192.168.45.0/24
```

Вам следует знать о том, что работа с маршрутами зачастую намного сложнее, чем кажется. Для приведенного примера следует также убедиться в том, что маршруты для всех хостов сети 192.163.45.0/24 могут привести обратно в сеть 10.23.2.0/24, а иначе первый добавленный вами маршрут окажется бесполезным.

Обычно для своих клиентов вам следует настраивать сети по возможности проще, чтобы хостам требовался лишь один маршрут. Если вам необходимо несколько подсетей и возможность маршрутизации между ними, обычно для этого лучше настроить маршрутизаторы в качестве шлюзов по умолчанию, которые выполняют всю работу по маршрутизации между различными локальными подсетями. Пример вы увидите в разделе 9.17.

## 9.9. Конфигурация сети, активизируемая при загрузке системы

Мы рассмотрели способы ручной настройки сети. Традиционно для обеспечения правильной сетевой конфигурации компьютера требовалось, чтобы во время загрузки системы запускался сценарий ручной конфигурации. Это сводится к запуску таких инструментов, как `ifconfig` и `route`, где-либо среди последовательности событий загрузки. Многие серверы до сих пор поступают подобным образом.

Предпринимались многие попытки стандартизировать файлы конфигурации для настройки сети во время загрузки системы Linux. Инструменты `ifup` и `ifdown` выполняют это: например, сценарий загрузки может (теоретически) запустить команду `ifup eth0`, чтобы запустить корректные команды `ifconfig` и `route` для интерфейса `eth0`. К сожалению, в разных версиях системы различная реализация команд `ifup` и `ifdown`, в результате чего их файлы конфигурации также совершенно различны. Версия Ubuntu, например, использует вариант `ifupdown`, файлы конфигурации которого расположены в каталоге `/etc/network`, а версия Fedora пользуется собственным набором сценариев и конфигурацией в каталоге `/etc/sysconfig/network-scripts`.

Вам не обязательно знать подробности об этих файлах конфигурации, но, если вы настаиваете на ручной настройке в обход инструментов конфигурации вашей системы, можно посмотреть формат этих файлов на страницах руководства ifup(8) и interfaces(5). Важно знать о том, что этот тип конфигурации, активизируемой во время загрузки, часто не используется совсем. Чаще всего вы будете встречать его в сетевом интерфейсе локальных хостов (или `lo`, см. раздел 9.13) и более нигде, поскольку он недостаточно гибок, чтобы отвечать потребностям современных систем.

## 9.10. Проблемы, связанные с конфигурацией сети вручную и при активизации во время загрузки системы

Несмотря на то что в большинстве систем настройка сети была заложена в их механизм загрузки (а многие серверы до сих пор так устроены), динамичная природа современных сетей означает, что у большинства компьютеров нет статического (неизменного) IP-адреса. Вместо того чтобы хранить IP-адрес и другую сетевую информацию на компьютере, ваш компьютер получает эту информацию откуда-

либо из локальной физической сети, когда он в первый раз подключается к ней. Большинство обычных клиентских сетевых приложений не сильно заботит, какой IP-адрес использует компьютер, пока он работает. Инструменты с протоколом DHCP (Dynamic Host Configuration Protocol, протокол динамической конфигурации хоста; рассмотрен в разделе 9.16) выполняют основную конфигурацию сетевого уровня для типичных клиентов.

Но на этом история не заканчивается. Можно добавить, например, беспроводные сети и дополнительные измерения конфигурации интерфейса, такие как сетевые имена, аутентификация и методы шифрования. Когда вы отступите назад, чтобы увидеть картину в целом, вы поймете, что вашей системе необходимо отвечать на следующие вопросы.

- Если компьютер имеет несколько физических сетевых интерфейсов (например, ноутбук с проводным и беспроводным Ethernet-подключением), как выбрать тот, который следует использовать?
- Каким образом компьютер должен настроить физический интерфейс? Для беспроводных сетей для этого потребуется сканирование сетевых имен, выбор имени и проведение аутентификации.
- Когда интерфейс физической сети подключен, каким образом компьютер должен настроить программные сетевые уровни, такие как интернет-уровень?
- Как разрешить пользователю выбирать варианты подключения? Например, как позволить выбор беспроводной сети?
- Что должен предпринять компьютер в случае потери подключения к сетевому интерфейсу?

Ответ на эти вопросы, как правило, выше возможностей простых сценариев загрузки, а в ручной конфигурации практически неосуществим. Следует использовать системную службу, которая может отслеживать физические сети и выбирать (и автоматически конфигурировать) сетевые интерфейсы ядра на основе набора правил, которые понятны пользователю. Эта служба должна быть также способна отвечать на запросы пользователей, у которых должна быть возможность смены беспроводной сети без необходимости использования корневых привилегий только для того, чтобы «подкручивать» настройки сети всякий раз, когда что-либо изменится.

## 9.11. Менеджеры сетевой конфигурации

Есть несколько способов автоматического конфигурирования сетей в системах на основе Linux. Наиболее широко в ПК и на ноутбуках используется менеджер сети NetworkManager. Другие системы управления сетевой конфигурацией предназначены главным образом для небольших внедренных систем, например `netifd` для OpenWRT, служба ConnectivityManager для платформы Android, менеджеры ConnMan и Wicd.

Мы кратко рассмотрим менеджер NetworkManager, поскольку вам, вероятно, встретится именно он. Однако не станем углубляться в устрашающее количество деталей, поскольку после того, как вы увидите всю картину, менеджер NetworkManager и другие системы конфигурирования окажутся более прозрачными.

## 9.11.1. Работа менеджера NetworkManager

Менеджер NetworkManager — это демон, который запускается во время загрузки системы. Подобно другим демонам, он не зависит от запущенного компонента рабочего стола. Его задача состоит в прослушивании системных и пользовательских событий с последующим изменением конфигурации сети на основе набора правил.

Во время работы менеджер NetworkManager обслуживает два основных уровня конфигурации. Первый — это информация о доступных аппаратных средствах, обычно она извлекается из ядра и появляется при отслеживании демона `udev` через шину Desktop Bus (D-Bus). Второй уровень конфигурации представляет специальный перечень *подключений*: аппаратные средства и дополнительные параметры конфигурации физического и сетевого уровней. Например, беспроводная сеть может быть представлена как подключение.

Чтобы активизировать подключение, менеджер NetworkManager часто поручает задачи другим специализированным сетевым инструментам и демонам, например `dhclient`, которые узнают конфигурацию интернет-уровня из локально подключенной физической сети. Поскольку инструменты и схемы конфигурирования сети отличаются в разных версиях системы, менеджер NetworkManager использует плагины, чтобы осуществить стыковку с ними, а не навязывать собственный стандарт. Существуют, например, плагины как для конфигурации интерфейса Debian/Ubuntu, так и в стиле Red Hat.

Во время запуска менеджер NetworkManager собирает всю доступную информацию о сетевом устройстве, отыскивает его перечень подключений, а затем предпринимает попытку активизации какого-либо из них. Вот как это выглядит для интерфейсов Ethernet.

1. Если проводное подключение доступно, попытаться подключиться с его использованием. В противном случае применять беспроводные подключения.
2. Просмотреть список доступных беспроводных сетей. Если доступна сеть, к которой вы уже подключались ранее, менеджер NetworkManager попробует подключиться к ней снова.
3. Если будет доступно несколько беспроводных сетей, с которыми ранее было установлено соединение, выбрать ту, к которой подключались совсем недавно.

После установления подключения менеджер NetworkManager обслуживает его: пока оно не будет утрачено, пока не появится сеть с лучшими параметрами (например, вы подключили сетевой кабель, когда работает беспроводное соединение) или пока пользователь не выполнит изменения.

## 9.11.2. Взаимодействие с менеджером NetworkManager с помощью интерфейса

Большинство пользователей взаимодействует с менеджером NetworkManager с помощью апплета на рабочем столе: как правило, это значок в верхнем или нижнем правом углу, который сообщает статус соединения (проводное, беспроводное или

отсутствие подключения). Если щелкнуть кнопкой мыши на этом значке, появится возможность изменить подключение, например выбрать беспроводную сеть или отключиться от текущей сети. В каждой среде рабочего стола своя версия этого апплета, поэтому он выглядит немного по-разному.

В дополнение к этому апплету есть несколько инструментов, которые можно использовать для выполнения запросов к менеджеру и управления им из оболочки. Чтобы получить очень быстрый отчет о текущем статусе соединения, применяйте команду `nm-tool` без аргументов. Вы получите перечень интерфейсов и параметров конфигурации. Это в некоторой степени напоминает отчет команды `ifconfig`, за исключением того, что здесь больше подробностей, в особенности при просмотре беспроводных подключений.

Для управления менеджером NetworkManager из командной строки используйте команду `nmcli`. Эта команда довольно обширная; дополнительную информацию о ней см. на странице руководства nmcli(1).

Наконец, утилита `nm-online` сообщит вам о том, функционирует сеть или нет. Если сеть в порядке, команда возвращает нулевое значение кода завершения; в противном случае оно отличается от нуля. Подробности об использовании кода завершения в сценарии оболочки см. в главе 11.

## 9.11.3. Конфигурация менеджера NetworkManager

Основным каталогом конфигурации менеджера NetworkManager обычно является `/etc/NetworkManager`, и в нем присутствует несколько различных типов конфигурации. Главный файл конфигурации — `NetworkManager.conf`. Его формат подобен XDG-файлам `.desktop` и файлам `.ini` стандарта Microsoft: пары параметров «ключ — значение» распределены по различным секциям. Вы обнаружите, что практически каждый файл конфигурации содержит секцию `[main]`, которая определяет необходимые для использования плагины. Вот простой пример, в котором активизируется плагин ifupdown, применяемый в системах Ubuntu и Debian:

```
[main]
plugins=ifupdown,keyfile
```

Плагинами, которые зависят от версии ОС, являются ifcfg-rh (для семейства Red Hat) и ifcfg-suse (для SuSE). Плагин keyfile, который вы также видите здесь, поддерживает собственный файл конфигурации менеджера NetworkManager. При использовании этого плагина можно увидеть опознанные системой подключения в файле `/etc/NetworkManager/system-connections`.

В большинстве случаев вам не потребуется изменять файл `NetworkManager.conf`, поскольку специальные параметры конфигурации находятся в других файлах.

### Неуправляемые интерфейсы

Несмотря на то что менеджер NetworkManager может вам понадобиться для управления большинством сетевых интерфейсов, иногда возникают ситуации, когда необходимо игнорировать интерфейсы. Например, нет причин для того, чтобы большинству пользователей понадобился какой-либо тип динамической конфигурации интерфейса локального хоста (`lo`), так как эта конфигурация никогда не меняется.

Может также потребоваться настроить этот интерфейс на ранних стадиях процесса загрузки системы, поскольку основные системные службы часто зависят от него. В большинстве версий ОС менеджер NetworkManager не допускается к локальному хосту.

Можно дать указание менеджеру NetworkManager, чтобы он игнорировал какой-либо интерфейс, используя плагины. Если вы применяете плагин ifupdown (например, в Ubuntu и Debian), добавьте конфигурацию интерфейса в файл `/etc/network/interfaces`, а затем в секции `ifupdown` файла `NetworkManager.conf` установите для параметра `managed` значение `false`:

```
[ifupdown]
managed=false
```

Для плагина ifcfg-rh, который используется в Fedora и Red Hat, поищите строку, подобную приведенной ниже, в каталоге `/etc/sysconfig/network-scripts`, который содержит конфигурационные файлы `ifcfg-*`:

```
NM_CONTROLLED=yes
```

Если такой строки нет или установлено значение `no`, менеджер NetworkManager игнорирует данный интерфейс. Например, вы обнаружите, что он деактивизирован в файле `ifcfg-lo`. Можно также указать адрес аппаратного средства, которое следует игнорировать:

```
HWADDR=10:78:d2:eb:76:97
```

Если вы не используете ни одну из этих схем сетевой конфигурации, можно применить плагин keyfile, чтобы указать неуправляемое устройство прямо в файле `NetworkManager.conf` с помощью адреса MAC. Это могло бы выглядеть так:

```
[keyfile]
unmanaged-devices=mac:10:78:d2:eb:76:97;mac:1c:65:9d:cc:ff:b9
```

## Диспетчеризация

Одна из завершающих деталей конфигурации менеджера NetworkManager относится к указанию дополнительных системных действий для тех случаев, когда сетевой интерфейс становится активным или отключается. Например, некоторым сетевым демонам для корректной работы необходимо знать, когда начать или завершить прослушивание интерфейса (например, демону защищенной оболочки, о которой пойдет речь в следующей главе).

Когда статус сетевого интерфейса в системе меняется, менеджер NetworkManager запускает все, что находится в каталоге `/etc/NetworkManager/dispatcher.d` с каким-либо из аргументов, таким как `up` или `down`. Это сравнительно просто, однако во многих версиях ОС используются собственные сценарии управления сетью, поэтому в указанном каталоге нет отдельных сценариев диспетчеризации. В Ubuntu, например, применяется всего один сценарий — `01ifupdown`, который запускает все, что расположено в соответствующем подкаталоге каталога `/etc/network`, например в `/etc/network/if-up.d`.

Что касается остальной конфигурации менеджера NetworkManager, подробности сценариев не так уж важны; вам необходимо знать лишь о том, как определить

соответствующее местоположение, если вам потребуется внести изменения. И, как обычно, заглядывайте в сценарии собственной системы.

# 9.12. Разрешение имени хоста

Одной из заключительных задач любого сетевого конфигурирования является разрешение имени хоста с помощью службы DNS. Вы уже видели инструмент `host`, который переводит имя, такое как www.example.com, в IP-адрес вроде 10.23.2.132.

Служба DNS отличается от рассмотренных нами элементов сети, так как она расположена на прикладном уровне, полностью в пространстве пользователя. Технически она немного не на своем месте в данной главе, рядом с обсуждением интернет-уровня и физического уровня. Но при неправильной конфигурации DNS ваше интернет-подключение практически бесполезно. Никто не станет рекламировать IP-адреса вместо имен сайтов и адресов электронной почты, поскольку IP-адрес хоста может измениться, да и набор чисел запомнить непросто. Автоматические службы сетевой конфигурации, такие как DHCP, почти всегда содержат конфигурацию DNS.

Практически все сетевые приложения в Linux выполняют поиски DNS. Процесс разрешения имен обычно протекает следующим образом.

1. Приложение вызывает функцию, чтобы выяснить IP-адрес, который стоит за именем хоста. Эта функция находится в совместно используемой системной библиотеке, поэтому приложению не нужно знать подробности о том, как она работает, или об изменениях в ее реализации.
2. Когда эта функция запускается, она действует в соответствии с набором правил (расположенных в файле `/etc/nsswitch.conf`), чтобы установить план действий при поисках. Например, такие правила обычно говорят о том, что перед переходом к DNS следует проверить ручное переопределение в файле `/etc/hosts`.
3. Когда функция решает использовать службу DNS для поиска имени, она обращается к дополнительному файлу конфигурации, чтобы найти сервер имен DNS. Сервер имен представлен в виде IP-адреса.
4. Функция отправляет DNS-запрос на поиск (по сети) серверу имен.
5. Сервер имен сообщает в ответ IP-адрес имени хоста, а функция возвращает этот IP-адрес приложению.

Это упрощенный вариант. В типичной современной системе присутствует больше исполнителей, которые стремятся ускорить трансакцию и/или добавить гибкость. Пока проигнорируем их и более подробно рассмотрим основные составляющие.

## 9.12.1. Файл /etc/hosts

В большинстве систем можно переопределить параметры поиска имен хоста с помощью файла `/etc/hosts`. Обычно это выглядит так:

```
127.0.0.1       localhost
10.23.2.3       atlantic.aem7.net       atlantic
10.23.2.4       pacific.aem7.net        pacific
```

Практически всегда вы увидите здесь запись для локального хоста (см. раздел 9.13).

ПРИМЕЧАНИЕ

В старые недобрые времена был единственный центральный хост-файл, который каждый пользователь копировал на собственный компьютер, чтобы данные были самыми свежими (см. Рабочие предложения № 606, 608, 623 и 625), однако с развитием сетей ARPANET/Internet это быстро стало ненужным.

## 9.12.2. Файл resolv.conf

Традиционным файлом конфигурации для серверов DNS является файл `/etc/resolv.conf`. Когда все было проще, типичный пример мог выглядеть так (здесь 10.32.45.23 и 10.3.2.3 — это адреса серверов имен у поставщика интернет-услуг):

```
search mydomain.example.com example.com
nameserver 10.32.45.23
nameserver 10.3.2.3
```

Строка `search` определяет правила для неполных хост-имен (то есть для первой части имени хоста; например, `myserver` вместо `myserver.example.com`). Здесь библиотека с функцией разрешения имен попыталась бы поискать имена *host*`.mydomain.example.com` и *host*`.example.com`. Однако теперь все, как правило, не настолько просто. В конфигурации службы DNS сделано много улучшений и изменений.

## 9.12.3. Кэширование и службы DNS без конфигурирования

Традиционная конфигурация службы DNS имеет две основные проблемы. Во-первых, локальный компьютер не кэширует имя, которое возвращает сервер, поэтому частый повторяющийся доступ к сети может неоправданно замедлиться вследствие запросов к серверу имен. Чтобы справиться с этой проблемой, многие компьютеры (и маршрутизаторы, если они работают в качестве серверов имен) запускают промежуточный демон, чтобы перехватывать запросы к серверу имен и возвращать на них, если это возможно, кэшированный ответ. В противном случае запросы отправляются на реальный сервер имен. Два наиболее распространенных демона Linux для этой цели — `dnsmasq` и `nscd`. Можно также настроить демон BIND (стандартный демон сервера имена в Unix) в качестве кэша. Часто можно понять, запущен ли демон кэширования имен, если в файле конфигурации `/etc/resolv.conf` присутствует адрес 127.0.0.1 (localhost) или же в качестве имени сервера отображается 127.0.0.1, когда вы запускаете команду `nslookup -debug` *host*.

Может оказаться непросто выяснить конфигурацию, если вы используете демон кэширования имен. По умолчанию для демона `dnsmasq` применяется файл конфигурации `/etc/dnsmasq.conf`, но в вашей версии системы это может быть переопределено. Например, в Ubuntu, если вы вручную настроили интерфейс, который управляется менеджером NetworkManager, вы найдете его конфигурацию в соответствующем файле каталога `/etc/NetworkManager/system-connections`, поскольку менеджер NetworkManager при активизации соединения запускает также и демон `dnsmasq` с дан-

ной конфигурацией. Можно переопределить все это, если снять комментарии с записи в файле NetworkManager.conf, относящейся к демону dnsmasq.

Другой проблемой, связанной с традиционным устройством сервера имен, является то, что оно может оказаться чрезвычайно негибким, когда вам потребуется выяснить имена в вашей локальной сети, не вникая в детали сетевой конфигурации. Если, например, вы настраиваете сетевое устройство в своей сети, вам может понадобиться немедленно вызвать его по имени. Это часть идеи, которая заложена в такие службы имен, не требующие конфигурации, как mDNS (Multicast DNS, многоадресная служба DNS) и SSDP (Simple Service Discovery Protocol, простой протокол обнаружения службы). Если необходимо найти в локальной сети хост по его имени, то вы просто рассылаете запрос по этой сети; если требуемый хост в ней присутствует, то он возвращает в ответ свой адрес. Эти протоколы выходят за рамки разрешения имени хоста, предоставляя также информацию о доступных службах.

Наиболее широко используемая реализация mDNS для Linux называется Avahi. Вариант mdns часто указывается в качестве функции разрешения имен в файле /etc/nsswitch.conf, который мы сейчас рассмотрим более подробно.

## 9.12.4. Файл /etc/nsswitch.conf

Файл /etc/nsswitch.conf контролирует параметры старшинства, связанные с именами, такие как информация о пользователе и пароле. Однако мы будем говорить в данной главе лишь о параметрах DNS. В этом файле должна быть строка, подобная следующей:

```
hosts:      files dns
```

Параметр files помещен перед параметром dns, чтобы система искала запрашиваемый вами IP-адрес в файле /etc/hosts, прежде чем обратиться к серверу DNS. Обычно такой способ хорош (особенно при отыскании локального хоста, как описано ниже), но при этом файл /etc/hosts должен быть по возможности *коротким*. Не помещайте в него ничего для улучшения производительности, это только навредит вам впоследствии. Можно поместить данные о всех хостах небольшой частной локальной сети в файл /etc/hosts, однако общее правило здесь таково: если у какого-либо хоста есть запись в службе DNS, его не следует указывать в файле /etc/hosts. Файл /etc/hosts полезен также для разрешения имен на ранних этапах загрузки системы, когда сеть может оказаться недоступной.

ПРИМЕЧАНИЕ

Тема, посвященная DNS, достаточно обширна. Если вы каким-либо образом ответственны за доменные имена, прочитайте книгу Крикета Лиу (Cricket Liu) и Пола Альбитца (Paul Albitz) DNS and BIND («Службы DNS и BIND»), 5-е издание (O'Reilly, 2006).

# 9.13. Локальный хост

Если запустить команду ifconfig, можно заметить интерфейс lo:

```
lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
```

```
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:16436 Metric:1
```

Интерфейс `lo` является виртуальным сетевым интерфейсом, который называется *возвратной петлей*, поскольку он «закольцован» сам на себя. Результат таков, что при подключении к адресу 127.0.0.1 происходит подключение к компьютеру, которым вы пользуетесь в данный момент. Когда исходящие данные для локального хоста доходят до сетевого интерфейса `lo` в ядре, ядро просто заново упаковывает их как входящие данные и отправляет обратно через интерфейс `lo`.

Интерфейс возвратной петли `lo` часто является единственным местом, где можно увидеть статическую сетевую конфигурацию в сценариях загрузки системы. Так, например, команда `ifup` в Ubuntu читает файл `/etc/network/interfaces`, а в Fedora используется файл `/etc/sysconfig/network-interfaces/ifcfg-lo`. Часто можно обнаружить конфигурацию какого-либо устройства с возвратной петлей, если поискать в каталоге `/etc` с помощью команды `grep`.

# 9.14. Транспортный уровень: протоколы TCP, UDP и службы

До сих пор мы видели лишь то, как пакеты перемещаются от хоста к хосту по сети Интернет, другими словами, ответ на вопрос *«куда?»* из начала этой главы. Теперь начнем отвечать на вопрос *«что?»*. Важно знать, как ваш компьютер представляет данные пакета, которые он получает от других хостов, своим работающим процессам. Для команд из пространства пользователя трудно и неудобно иметь дело с набором пакетов таким образом, как с ними может работать ядро. В особенности важна гибкость: сразу несколько приложений должны иметь возможность одновременного обращения к сети (например, вам может потребоваться запустить почтовый клиент и браузер).

Протоколы *транспортного уровня* заполняют разрыв между необработанными пакетами интернет-уровня и «рафинированными» потребностями приложений. Двумя самыми популярными транспортными протоколами являются TCP (Transmission Control Protocol, протокол управления передачей) и UDP (User Datagram Protocol, протокол передачи дейтаграмм пользователя). Мы сосредоточимся на протоколе TCP, так как он безоговорочно является наиболее используемым, и вкратце рассмотрим и протокол UDP.

## 9.14.1. Порты TCP и соединения

Протокол TCP предоставляется нескольким сетевым приложениям на одном компьютере с помощью сетевых *портов*. Порт — это просто число. Если IP-адрес можно уподобить почтовому индексу какого-либо жилого дома, то порт похож на номер почтового ящика: это дальнейшее деление на более мелкие части.

При использовании протокола TCP приложение открывает *соединение* (не смешивайте с подключениями в менеджере NetworkManager) между одним из портов данного компьютера и каким-либо портом удаленного хоста. Например, такое

приложение, как браузер, могло бы открыть соединение между портом 36406 компьютера и портом 80 удаленного хоста. С точки зрения приложения порт 36406 является локальным портом, а порт 80 — удаленным портом.

Можно идентифицировать соединение с помощью пары, составленной из IP-адреса и номера порта. Чтобы увидеть соединения, которые в данный момент открыты на компьютере, воспользуйтесь командой `netstat`. Приведем пример, в котором показаны TCP-соединения (параметр `-n` отключает разрешение имен (DNS), а параметр `-t` ограничивает результаты только протоколом TCP):

```
$ netstat -nt
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.23.2.4:47626        10.194.79.125:5222      ESTABLISHED
tcp        0      0 10.23.2.4:41475        172.19.52.144:6667      ESTABLISHED
tcp        0      0 10.23.2.4:57132        192.168.231.135:22      ESTABLISHED
```

Поля `Local Address` и `Foreign Address` показывают соединения с точки зрения компьютера. Следовательно, для этого компьютера интерфейс настроен на IP-адрес 10.23.2.4, а локальные порты 47626, 41475 и 57132 подключены. Здесь первое соединение установлено между портом 47626 и портом 5222 по адресу 10.194.79.125.

## 9.14.2. Установление TCP-соединений

Чтобы установить соединение с помощью транспортного уровня, процесс на какомлибо хосте инициирует подключение одного из своих локальных портов ко второму хосту с помощью специальной серии пакетов. Чтобы распознать входящее соединение и ответить на него, второй хост должен обладать процессом, который *прослушивает* правильный порт. Как правило, подключающийся процесс называется *клиентом*, а прослушивающий — *сервером* (подробности изложены в главе 10).

О портах важно знать следующее: клиент выбирает такой порт со своей стороны, который в настоящее время не используется, но он практически всегда соединяется с каким-либо хорошо известным портом на стороне сервера. Вернитесь к выводу команды `netstat` из предыдущего раздела:

```
Proto Recv-Q Send-Q Local Address      Foreign Address        State
tcp        0      0 10.23.2.4:47626   10.194.79.125:5222     ESTABLISHED
```

С небольшой подсказкой можно понять, что это соединение с удаленным сервером было, по-видимому, инициировано локальным клиентом, так как порт локальной стороны (47626) выглядит как динамически присвоенное число, в то время как удаленный порт (5222) является хорошо известной службой (службой сообщений Jabber или XMPP, если быть конкретнее).

ПРИМЕЧАНИЕ

Динамически назначаемый порт называется эфемерным портом.

Однако если локальный порт в этом выводе хорошо известен, то соединение, вероятно, было инициировано удаленным хостом. В приведенном ниже примере удаленный хост 172.24.54.234 подключился к порту 80 (веб-порт по умолчанию) локального хоста.

```
Proto Recv-Q Send-Q Local Address      Foreign Address        State
tcp        0      0 10.23.2.4:80      172.24.54.234:43035    ESTABLISHED
```

Удаленный хост, подключающийся к хорошо известному порту вашего компьютера, подразумевает, что сервер локального компьютера прослушивает данный порт. Чтобы убедиться в этом, выведите список всех TCP-портов, которые прослушивает ваш компьютер, с помощью команды `netstat`:

```
$ netstat -ntl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address        Foreign Address        State
tcp        0      0 0.0.0.0:80          0.0.0.0:*              LISTEN
tcp        0      0 127.0.0.1:53        0.0.0.0:*              LISTEN
--snip--
```

Строка, которая содержит значение 0.0.0.0:80 как локальный адрес, говорит о том, что локальный компьютер прослушивает подключения к порту 80 от удаленных компьютеров. Сервер может ограничить доступ к некоторым интерфейсам, как показано в последней строке, где нечто прослушивает соединения только в интерфейсе локального хоста. Чтобы узнать еще больше подробностей, воспользуйтесь командой `lsof` для идентификации процесса, выполняющего прослушивание (как рассказано в подразделе 10.5.1).

## 9.14.3. Номера портов и файл /etc/services

Как узнать, является ли порт хорошо известным? Однозначно сказать нельзя, но начать стоит с просмотра файла `/etc/services`, который переводит значения хорошо известных портов в имена. Это простой текстовый файл. Вы можете увидеть в нем записи вроде:

```
ssh         22/tcp          # SSH Remote Login Protocol
smtp        25/tcp
domain      53/udp
```

Первый столбец содержит имя, а во втором указаны номер порта и относящийся к нему протокол транспортного уровня (который может отличаться от TCP).

ПРИМЕЧАНИЕ

В дополнение к файлу /etc/services по адресу http://www.iana.org/ существует онлайн-реестр портов, который регулируется документом RFC6335 о сетевых стандартах.

В Linux только те процессы, которые запущены с корневыми (superuser) правами, могут использовать порты с 1 по 1023. Все пользовательские процессы могут выполнять прослушивание и создавать соединения, применяя порты с 1024 и далее.

## 9.14.4. Характеристики протокола TCP

Протокол TCP популярен как протокол транспортного уровня, поскольку он требует сравнительно немного со стороны приложения. Процессу приложения надо

знать лишь о том, как открывать (или прослушивать), считывать, записывать и закрывать соединение. Для приложения это напоминает входящие и исходящие потоки данных; процесс почти так же прост, как работа с файлом.

Однако за всем этим стоит большая работа. Для начала протоколу TCP необходимо знать о том, как разбивать исходящий от процесса поток данных на пакеты. Сложнее узнать, как преобразовать серию входящих пакетов во входной поток данных для процесса, в особенности тогда, когда входящие пакеты не обязательно приходят в правильном порядке. В дополнение к этому хост, использующий протокол TCP, должен выполнять проверку на ошибки: при пересылке через Интернет пакеты могут быть потеряны или повреждены, протокол TCP должен обнаружить и исправить подобные ситуации. На рис. 9.3 приведена упрощенная схема того, как хост может использовать протокол TCP для отправки сообщения.

К счастью, вам не нужно знать обо всем этом практически ничего, кроме того, что в Linux протокол TCP реализован главным образом в ядре, а утилиты, которые работают с транспортным уровнем, стремятся использовать структуры данных ядра. Одним из примеров является система фильтрации пакетов с помощью IP-таблиц, которая рассмотрена в разделе 9.21.

## 9.14.5. Протокол UDP

Протокол UDP является намного более простым транспортным уровнем по сравнению с протоколом TCP. Он определяет передачу только для отдельных сообщений, потока данных здесь нет. В то же время, в отличие от протокола TCP, протокол UDP не выполняет коррекцию утерянных или неправильно расположенных пакетов. На самом деле, хотя у протокола UDP и есть порты, он даже не обладает соединениями! Хост просто отправляет сообщение от одного из своих портов порту на сервере, а сервер отправляет что-либо обратно, если желает. Тем не менее в протоколе UDP все же присутствует выявление ошибок данных внутри пакета. Хост может установить, что пакет поврежден, но он ничего не должен делать в связи с этим.

Если протокол TCP похож на телефонный разговор, то протокол UDP напоминает отправку письма, телеграммы или мгновенного сообщения (за исключением того, что мгновенные сообщения более надежны). Приложения, которые используют протокол UDP, часто заинтересованы в скорости: отправить сообщение настолько быстро, насколько возможно. Им не нужны дополнительные данные, как в протоколе TCP, поскольку они предполагают, что сетевое соединение между двумя хостами достаточно устойчивое. Им не нужна, как в TCP-протоколе, коррекция ошибок, так как они располагают собственными системами обнаружения ошибок или же просто не обращают на них внимания.

Одним из примеров приложения, которое использует протокол UDP, является *протокол NTP* (Network Time Protocol, протокол сетевого времени). Клиент отправляет короткий и простой запрос серверу, чтобы получить текущее время, ответ сервера такой же краткий. Поскольку ответ необходим клиенту по возможности быстро, приложению годится протокол UDP; если ответ сервера затеряется где-либо в сети, клиент может просто направить повторный запрос или прекратить

попытки. Другим примером является видеочат: в этом случае изображения пересылаются с помощью протокола UDP. Если некоторые фрагменты будут утрачены в пути, клиент на принимающей стороне сделает все возможное для их компенсации.

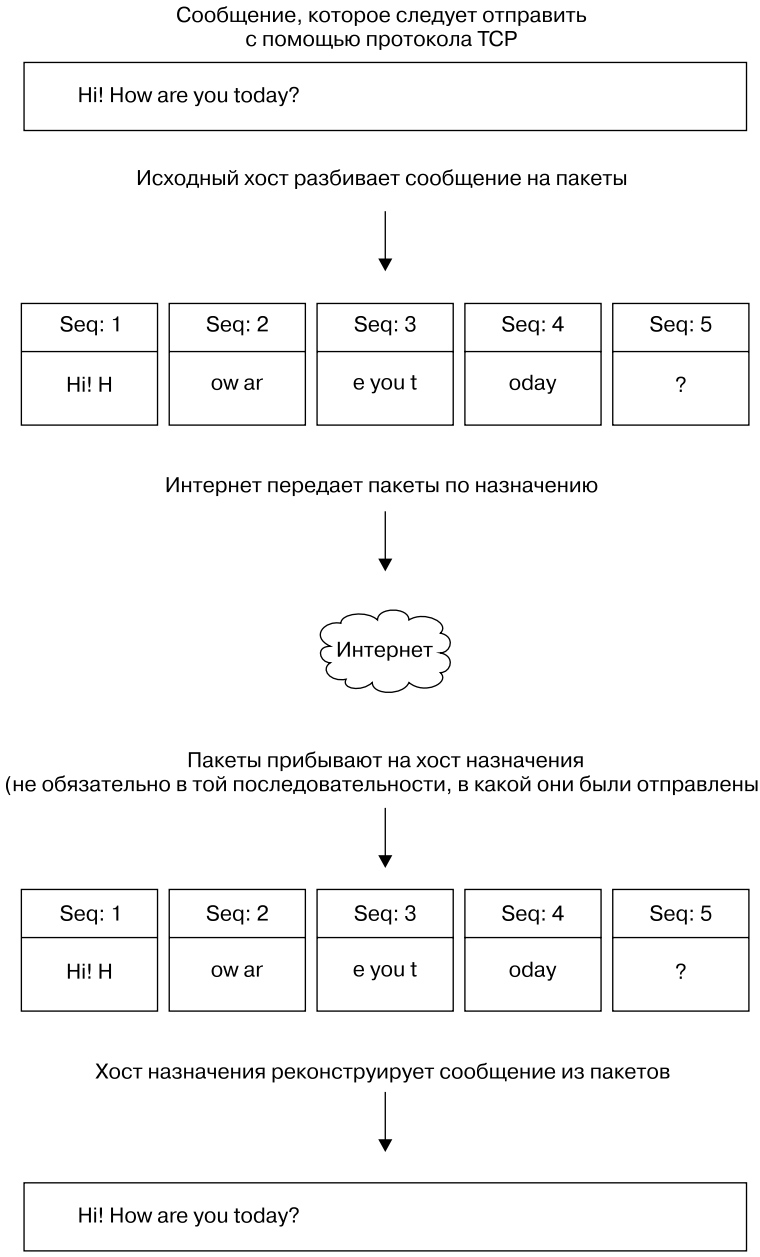


**Рис. 9.3.** Отправка сообщения с помощью протокола TCP

ПРИМЕЧАНИЕ

Оставшаяся часть этой главы посвящена боле сложным темам, таким как сетевая фильтрация и маршрутизаторы, поскольку они относятся к более низким сетевым уровням по сравнению с рассмотренными: физическим, сетевым и транспортным. Если желаете, можете спокойно приступать к следующей главе, чтобы узнать о прикладном уровне, на котором все объединяется в пространстве пользователя. Вы увидите процессы, использующие сеть, а не просто перекидывающие наборы адресов и пакетов.

# 9.15. Возвращаемся к простой локальной сети

Сейчас мы рассмотрим дополнительные компоненты простой локальной сети, о которой шла речь в разделе 9.3. Вспомните, что эта сеть состоит из одной местной сети в качестве подсети и маршрутизатора, который соединяет эту подсеть с остальной частью Интернета. Вы узнаете следующее:

- каким образом хост в этой подсети автоматически получает свою сетевую конфигурацию;
- как настроить маршрутизацию;
- что такое маршрутизатор на деле;
- как узнать, какой IP-адрес применить для подсети;
- как настроить брандмауэры, чтобы фильтровать нежелательный интернет-трафик.

Начнем с изучения того, каким образом хост в подсети автоматически получает свою сетевую конфигурацию.

# 9.16. Понятие о протоколе DHCP

Когда вы настраиваете сетевой хост на автоматическое получение конфигурации из сети, вы указываете ему, чтобы он использовал протокол DHCP (Dynamic Host Configuration Protocol, протокол динамического конфигурирования хоста) для получения IP-адреса, маски подсети, шлюза по умолчанию и серверов DNS. Помимо того что не приходится вводить эти параметры вручную, протокол DHCP обладает другими преимуществами для сетевого администратора, такими как предотвращение конфликтов IP-адресов и минимизация последствий при изменении сети. Нечасто можно встретить современную сеть, которая не использует протокол DHCP.

Чтобы хост получал свою конфигурацию с помощью протокола DHCP, он должен быть способен отправлять сообщения DHCP-серверу той сети, к которой он подключен. Следовательно, каждая физическая сеть должна обладать собственным DHCP-сервером, а в простой сети (подобной той, которая описана в разделе 9.3) его роль обычно выполняет маршрутизатор.

ПРИМЕЧАНИЕ

При выполнении первичного DHCP-запроса хост не знает даже адреса DHCP-сервера, поэтому он рассылает свой запрос всем хостам (как правило, всем хостам своей физической сети).

Когда компьютер запрашивает IP-адрес у сервера DHCP, на самом деле он просит об *аренде* этого адреса на некоторое время. Когда аренда заканчивается, клиент может запросить обновление аренды.

## 9.16.1. Клиент DHCP в Linux

Хотя и существует множество разных типов систем управления сетью, почти все они используют команду `dhclient` (которая придерживается стандартов ISC (Internet Software Consortium, Консорциум по разработке ПО для сети Интернет)) для выполнения реальной работы. Можно проверить работу команды `dhclient` вручную из командной строки, но сначала вы *обязаны* удалить маршрут шлюза по умолчанию. Чтобы выполнить тест, просто укажите имя сетевого интерфейса (в данном примере это `eth0`):

```
# dhclient eth0
```

Во время запуска команда `dhclient` сохраняет идентификатор своего процесса в файле `/var/run/dhclient.pid`, а информацию об аренде — в файле `/var/state/dhclient.leases`.

## 9.16.2. Серверы DHCP в Linux

Вы можете поручить компьютеру с Linux задачу по поддержанию сервера DHCP, чтобы обеспечить достаточную степень контроля над адресами, которые он раздает. Однако если вы не администрируете большую сеть с многими подсетями, то, вероятно, лучше будет использовать специальные аппаратные средства маршрутизации, в которые встроены серверы DHCP.

Возможно, о DHCP-серверах важнее всего знать следующее: необходимо применять только один такой сервер внутри подсети, чтобы избежать конфликтующих IP-адресов или неправильной конфигурации.

# 9.17. Настройка Linux в качестве маршрутизатора

По существу, маршрутизаторы являются всего лишь компьютерами, которые обладают несколькими интерфейсами физической сети. Можно с легкостью настроить компьютер с Linux как маршрутизатор.

Допустим, у вас есть две локальные подсети — 10.23.2.0/24 и 192.168.45.0/24. Чтобы их соединить, у вас имеется компьютер-маршрутизатор с Linux, в котором присутствуют три сетевых интерфейса: два для локальных подсетей и один — для связи с Интернетом, как показано на рис. 9.4. Как видите, это не сильно отличается от примера с простой сетью, который мы использовали в начале этой главы.

IP-адреса маршрутизатора для локальных подсетей таковы: 10.23.2.1 и 192.168.45.1. При настройке этих адресов таблица маршрутизации выглядит подобным образом (имена интерфейсов в действительности могут быть другими; интернет-ссылку пока проигнорируем):

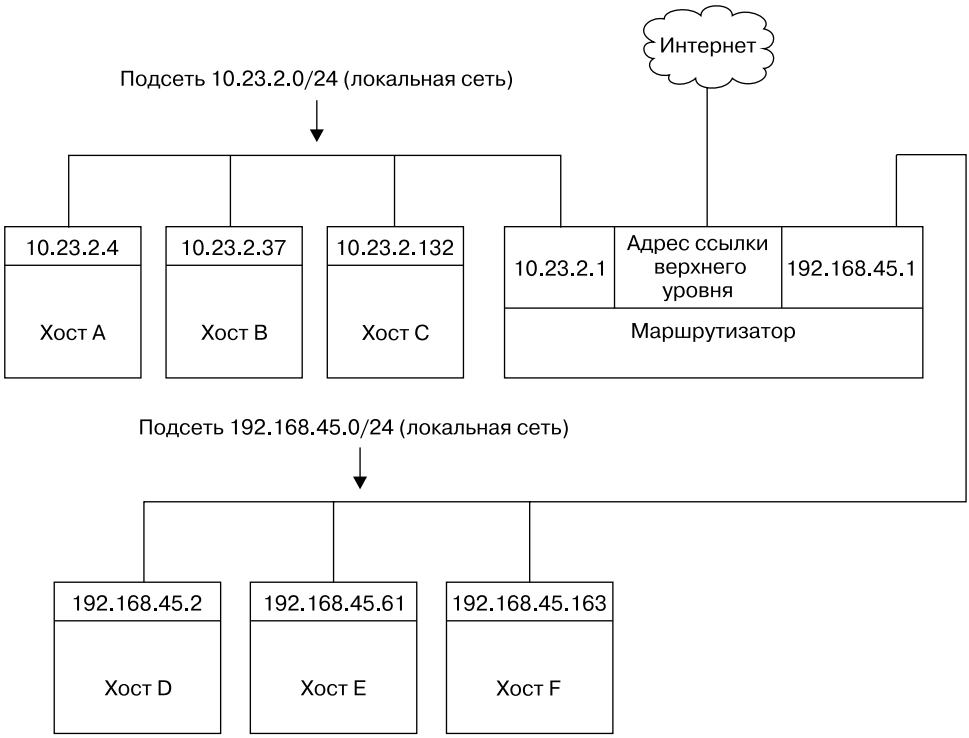


**Рис. 9.4.** Две подсети, соединенные с помощью маршрутизатора

```
Destination     Gateway      Genmask         Flags Metric Ref    Use Iface
10.23.2.0       0.0.0.0      255.255.255.0   U     0      0        0 eth0
192.168.45.0    0.0.0.0      255.255.255.0   U     0      0        0 eth1
```

Допустим, что для хостов в каждой из подсетей маршрутизатор указан в качестве шлюза по умолчанию (10.23.2.1 для сети 10.23.2.0/24 и 192.168.45.1 для сети 192.168.45.0/24). Если сеть 10.23.2.4 желает отправить пакет куда-либо за пределы сети 10.23.2.0/24, она передает такой пакет по адресу 10.23.2.1. Например, чтобы отправить пакет от адреса 10.23.2.4 (хост A) по адресу 192.168.45.61 (хост E), пакет приходит на адрес 10.23.2.1 (маршрутизатор) через интерфейс `eth0`, а затем уходит с него через интерфейс маршрутизатора `eth1`.

Однако по умолчанию ядро Linux не перемещает автоматически пакеты из одной подсети в другую. Чтобы задействовать эту основную функцию маршрутизации, необходимо включить *IP-перенаправление* в ядре маршрутизатора с помощью такой команды:

```
# sysctl -w net.ipv4.ip_forward
```

Как только вы введете эту команду, компьютер должен начать маршрутизацию пакетов между двумя подсетями при условии, что хосты этих подсетей знают о том, что пакеты следует отправлять только что созданному вами маршрутизатору.

Чтобы это изменение сохранилось после перезагрузки, можно добавить его в файл `/etc/sysctl.conf`. В зависимости от версии ОС может быть также вариант размещения в файле `/etc/sysctl.d`, чтобы ваши изменения не были перезаписаны при обновлениях системы.

**Интернет-ссылки верхнего уровня.** Когда в маршрутизаторе присутствует также третий сетевой интерфейс с интернет-ссылкой верхнего уровня, такая же настройка разрешает доступ в Интернет всем хостам, так как они сконфигурированы на использование этого маршрутизатора в качестве шлюза по умолчанию. Однако здесь все сложнее. Проблема в том, что конкретные IP-адреса, такие как 10.23.2.4, в действительности не видны всему Интернету; они находятся в так называемых *частных сетях*. Чтобы обеспечить их подключением к Интернету, следует настроить в маршрутизаторе функцию под названием *NAT* (Network Address Translation, *преобразование сетевых адресов*). Программное обеспечение почти у всех специализированных маршрутизаторов выполняет эту задачу, в ней нет ничего необычного, но рассмотрим более детально вопрос о частных подсетях.

# 9.18. Частные сети

Предположим, вы решили создать собственную сеть. Вы уже подготовили компьютеры, маршрутизатор и аппаратные средства сети. С теми знаниями о простой сети, которые у вас уже есть, возникает следующий вопрос: «Какие IP-адреса для подсети следует использовать?»

Если вам нужна группа интернет-адресов, которые может видеть любой хост в Интернете, можете приобрести их у своего поставщика интернет-услуг. Однако, поскольку диапазон адресов версии IPv4 весьма ограничен, это стоит довольно дорого и годится лишь для того, чтобы запустить сервер, который может видеть остальная часть Интернета. Большинству пользователей на самом деле не нужен такой тип услуг, поскольку они подключаются к Интернету в качестве клиентов.

Простая и недорогая альтернатива заключается в выборе адресов частной подсети на основе документов RFC 1918/6761, содержащих интернет-стандарты, показанные в табл. 9.2.

**Таблица 9.2.** Частные подсети, определяемые в документах RFC 1918 и 6761

| Сеть | Маска подсети | Форма CIDR |
|---|---|---|
| 10.0.0.0 | 255.0.0.0 | 10.0.0.0/8 |
| 192.168.0.0 | 255.255.0.0 | 192.168.0.0/16 |
| 172.16.0.0 | 255.240.0.0 | 172.16.0.0/12 |

Вы можете настраивать частные подсети как вам угодно. Если вы не планируете иметь более 254 хостов внутри одной сети, выберите небольшую подсеть вроде 10.23.2.0/24, которую мы рассматриваем в этой главе. Сети с такой маской иногда называют подсетями *класса C*. Хотя этот термин является технически устаревшим, он по-прежнему применяется.

Что же в итоге? Хосты в реальном Интернете ничего не знают о частных подсетях и не станут отправлять им пакеты, поэтому без дополнительной помощи хосты

частных подсетей не могут общаться с внешним миром. Маршрутизатору (с неприватным адресом), подключенному к Интернету, необходимо какое-либо средство, чтобы заполнить разрыв между данным соединением и хостами частной сети.

## 9.19. Преобразование сетевых адресов (маскировка IP-адреса)

Функция NAT — широко применяемый способ совместного использования единственного IP-адреса для частной сети. Он почти универсален в домашних сетях и сетях небольших офисов. В Linux вариант функции NAT, который использует большинство людей, известен как *маскировка IP-адреса*.

Суть функции NAT заключается в том, что маршрутизатор не просто передает пакеты от одной подсети в другую, но и трансформирует их во время передачи. Интернет-хосты знают, как подключиться к маршрутизатору, однако ничего не знают о расположенной за ним частной сети. Хостам частной сети не требуется специальная конфигурация; маршрутизатор является для них шлюзом по умолчанию.

Эта система в общих чертах работает так.

1. Хост внутренней частной сети намерен установить соединение с внешним миром, поэтому он отправляет пакеты своего запроса на соединение через маршрутизатор.
2. Маршрутизатор перехватывает эти пакеты, вместо того чтобы отправить их в Интернет (в котором они пропали бы, поскольку общественный Интернет ничего не знает о частных сетях).
3. Маршрутизатор определяет пункт назначения для пакета-запроса и открывает собственное соединение с этим пунктом назначения.
4. Когда маршрутизатор установит соединение, он отправляет фальсифицированное сообщение «соединение установлено» обратно, исходному внутреннему хосту.
5. Теперь маршрутизатор становится посредником между внутренним хостом и пунктом назначения. Пункт назначения ничего не знает о внутреннем хосте; для удаленного хоста соединение выглядит как исходящее от маршрутизатора.

Однако все не настолько просто, как выглядит. Нормальная IP-маршрутизация знает лишь IP-адреса источника и пункта назначения в интернет-уровне. Однако, если бы маршрутизатор имел дело только с интернет-уровнем, каждый хост внутренней сети смог бы установить только одно соединение с единственным пунктом назначения в данный момент времени (есть и другие ограничения), поскольку та часть пакета, которая относится к интернет-уровню, не содержит информации о том, как отличить повторные запросы от одного хоста к тому же пункту назначения. Следовательно, функция NAT должна выйти за пределы интернет-уровня и «вскрыть» пакеты, чтобы извлечь дополнительную идентифицирующую информацию, в частности номера портов UDP и TCP, из транспортных уровней. Протокол UDP достаточно прост, поскольку здесь есть порты, но нет соединений; сложнее дело с транспортным уровнем на основе протокола TCP.

Чтобы компьютер с Linux играл роль NAT-маршрутизатора, в конфигурации ядра должны быть активизированы следующие составляющие: фильтрация сетевых пакетов («поддержка брандмауэра»), отслеживание подключений, поддержка IP-таблиц, полная функция NAT и поддержка маскировки (MASQUERADE). В большинстве версий ядра эта поддержка присутствует.

Далее необходимо запустить несколько сложных на вид команд `iptables`, чтобы маршрутизатор использовал функцию NAT для частной подсети. Вот пример, который применяется к внутренней сети Ethernet с интерфейсом `eth1`, совместно использующей внешнее подключение к интерфейсу `eth0` (о синтаксисе команды `iptables` вы узнаете подробнее из раздела 9.21):

```
# sysctl -w net.ipv4.ip_forward
# iptables -P FORWARD DROP
# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
# iptables -A FORWARD -i eth0 -o eth1 -m state --state ESTABLISHED,RELATED -j ACCEPT
# iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
```

ПРИМЕЧАНИЕ

Хотя на практике функция NAT работает превосходно, помните о том, что по существу она является обходным маневром, который используется для продления времени жизни пространства адресов IPv4. В совершенном мире мы без каких-либо затруднений пользовались бы адресами IPv6 (Интернет следующего поколения), более длинными и усовершенствованными.

Вам, вероятно, никогда не потребуется использовать команды, приведенные выше, если вы не заняты разработкой собственного ПО. К тому же сейчас доступно так много специализированных маршрутизаторов. Однако роль Linux в сети на этом не заканчивается.

# 9.20. Маршрутизаторы и Linux

В ранние годы широкополосного Интернета пользователи с не столь высокими требованиями просто подключали свой компьютер напрямую к Интернету. Однако прошло совсем немного времени, и многим захотелось совместно использовать единственное широкополосное подключение для своих сетей, а пользователи Linux, в частности, настраивали дополнительный компьютер в качестве маршрутизатора с функцией NAT.

Производители откликнулись на этот новый рынок, предложив специализированные аппаратные средства маршрутизации, которые содержат эффективный процессор, некоторый объем флеш-памяти и несколько сетевых портов — этой функциональности достаточно для управления типичной простой сетью, запуска такого необходимого ПО, как сервер DHCP, а также для использования функции NAT. Когда дело дошло до программного обеспечения, многие производители прибегли к помощи Linux, чтобы привести в действие свои маршрутизаторы. Были добавлены необходимые функции ядра, упрощено ПО пространства пользователя и созданы графические интерфейсы администрирования.

Почти сразу же с появлением первых таких маршрутизаторов многие пользователи заинтересовались детальным устройством аппаратных средств. У одного из

производителей, компании Linksys, потребовали выпустить исходный код ПО на условиях лицензии для одного из его компонентов, и вскоре для маршрутизаторов стали появляться специальные версии Linux, такие как OpenWRT. Символы WRT возникли из названия моделей оборудования Linksys.

Помимо любознательности, есть веские причины для использования таких версий: они зачастую более стабильны, чем заводская прошивка, в особенности для старых маршрутизаторов, и, как правило, предлагают дополнительные функции. Например, чтобы создать мост между сетью и беспроводным подключением, многие производители требуют покупки подходящего оборудования. Однако если установить OpenWRT, то тогда ни производитель, ни возраст оборудования не будут иметь значения. Это связано с тем, что вы используете в маршрутизаторе по-настоящему открытую операционную систему, для которой неважно, что за аппаратные средства вы применяете, если только они поддерживаются.

Вы можете применить большую часть сведений из этой книги, чтобы исследовать внутренние части специализированной прошивки Linux, хотя вам и встретятся отличия, в особенности при входе в систему. Как и во многих внедренных системах, прошивки с открытым кодом стремятся использовать утилиты BusyBox, чтобы предоставить большинство функций оболочки. Утилиты BusyBox являются единым исполняемым приложением, которое предлагает ограниченную функциональность для многих команд Unix, таких как `shell`, `ls`, `grep`, `cat` и `more`. Так экономится существенный объем памяти. Кроме того, команда `init` для загрузки системы оказывается очень простой во внедренных системах. Тем не менее подобные ограничения не создадут проблем для вас, так как пользовательские прошивки Linux обычно содержат веб-интерфейс для администрирования, который напоминает интерфейс, предлагаемый производителем.

## 9.21. Брандмауэры

Маршрутизаторы должны всегда содержать какой-либо брандмауэр для запрещения нежелательного трафика в сети. *Брандмауэр* — это программное обеспечение и/или конфигурация аппаратных средств, которые обычно размещаются в маршрутизаторе между Интернетом и малой сетью, пытаясь не допустить того, чтобы что-либо «плохое» навредило малой сети. Можно также настроить функции брандмауэра для каждого компьютера, и тогда компьютер станет фильтровать все входящие и выходящие данные на уровне пакетов (в противоположность прикладному уровню, на котором серверные программы обычно пытаются самостоятельно обеспечить некоторый контроль доступа). Использование брандмауэра на отдельных компьютерах иногда называется *IP-фильтрацией*.

Система может фильтровать пакеты, когда она:

- получает пакет;
- отправляет пакет;
- перенаправляет пакет другому хосту или шлюзу.

Если брандмауэра нет, система просто обрабатывает пакеты и отправляет их по назначению. Брандмауэры помещают проверочные пункты для пакетов в указанных

выше точках передачи данных. На этих проверочных пунктах пакеты удаляются, отклоняются или принимаются, как правило, на основе таких критериев:

- IP-адрес или подсеть источника или пункта назначения;
- порт источника или пункта назначения (в сведениях транспортного уровня);
- сетевой интерфейс брандмауэра.

Брандмауэры обеспечивают возможность работы с подсистемой ядра Linux, которая обрабатывает IP-пакеты. Рассмотрим ее сейчас.

## 9.21.1. Брандмауэр в Linux: основные понятия

В Linux правила для брандмауэра создаются в виде последовательности, известной как *цепочка*. Набор цепочек формирует *таблицу*. Когда пакет проходит через различные части сетевой подсистемы Linux, ядро применяет правила из определенных цепочек к пакетам. Например, после получения нового пакета от физического уровня ядро активизирует правила в цепочках, соответствующих вводу.

Все эти структуры данных обслуживаются ядром. Такая система в целом называется таблицами `iptables`, а команда из пространства пользователя `iptables` создает правила и управляет ими.

ПРИМЕЧАНИЕ

Есть также более современная система, nftables, которая призвана заменить таблицы iptables. Однако на момент написания книги таблицы iptables являются преобладающими для брандмауэров.

Поскольку таблиц может быть несколько — каждая со своим набором цепочек, содержащих множество правил, — то перемещение пакетов становится довольно сложным. Тем не менее обычно работают с единственной таблицей, которая называется *«фильтр»* и контролирует основной поток пакетов. В таблице фильтра есть три главные цепочки: INPUT для входящих пакетов, OUTPUT для исходящих пакетов и FORWARD для перенаправляемых пакетов.

На рис. 9.5 и 9.6 приведены упрощенные схемы того, где к пакетам применяются правила из таблицы фильтра. Схем две, поскольку пакеты могут либо поступать в систему из сетевого интерфейса (см. рис. 9.5), либо генерироваться локальным процессом (см. рис. 9.6). Как видите, входящий пакет из сети может быть поглощен пользовательским процессом и не достичь цепочки FORWARD или OUTPUT. Пакеты, создаваемые пользовательскими процессами, не достигают цепочек INPUT или FORWARD.

В действительности все сложнее, поскольку есть и другие этапы, помимо этих трех цепочек. Например, пакеты проходят обработку в цепочках PREROUTING и POSTROUTING, а сама обработка может происходить на любом из трех нижних сетевых уровней. Чтобы увидеть полную схему, поищите в онлайн-источниках что-либо под названием «Поток пакетов в сетевом фильтре Linux», но имейте в виду, что подобные схемы пытаются учесть все возможные сценарии поступления и перемещения пакетов. Часто может помочь разбиение схем на основе источников пакетов, как на рис. 9.5 и 9.6.

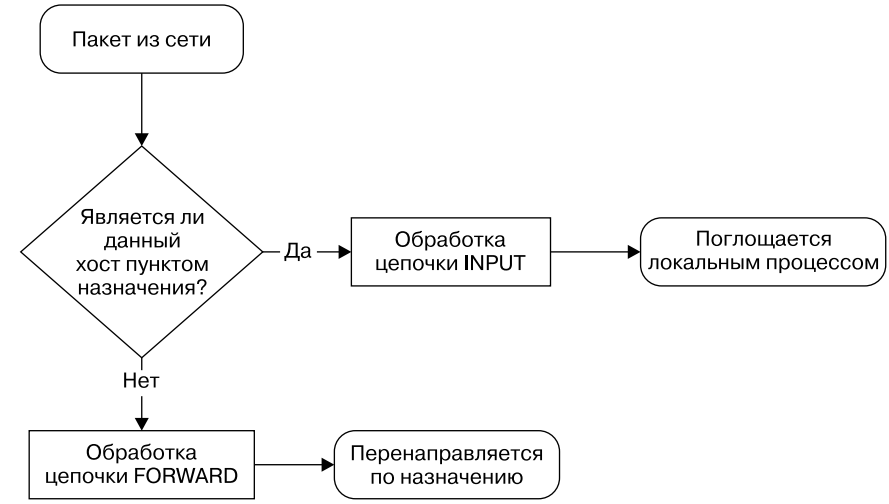


**Рис. 9.5.** Последовательность обработки цепочек для входящих пакетов сети

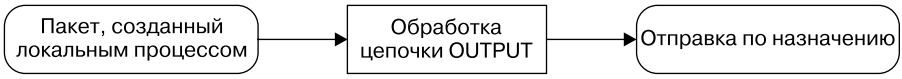


**Рис. 9.6.** Последовательность обработки цепочек для входящих пакетов от локального процесса

## 9.21.2. Определение правил для брандмауэра

Посмотрим, как работает на практике система IP-таблиц. Начнем с просмотра текущей конфигурации, использовав команду:

```
# iptables -L
```

Вывод обычно представляет собой пустой набор цепочек:

```
Chain INPUT (policy ACCEPT)
target     prot opt source        destination

Chain FORWARD (policy ACCEPT)
target     prot opt source        destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source        destination
```

У каждой цепочки брандмауэра есть *политика* по умолчанию, которая определяет, что делать с пакетом, если он не удовлетворяет ни одному из правил. Для всех трех цепочек в данном примере установлена политика ACCEPT. Это означает, что ядро разрешает пакету пройти через систему фильтрации. Политика DROP говорит ядру о том, что пакет надо отбросить. Чтобы указать политику цепочки, используйте команду `iptables -P`, например, так:

```
# iptables -P FORWARD DROP
```

ВНИМАНИЕ

Не торопитесь с изменением политик на своем компьютере, пока не дочитаете данный раздел до конца.

Допустим, кто-либо с IP-адресом 192.168.34.63 надоедает вам. Чтобы избавиться от его запросов к вашему компьютеру, запустите следующую команду:

```
# iptables -A INPUT -s 192.168.34.63 -j DROP
```

Здесь параметр `-A INPUT` присоединяет правило к цепочке `INPUT`. Фрагмент `-s 192.168.34.63` указывает IP-адрес источника в этом правиле, а часть `-j DROP` говорит ядру о том, чтобы оно отбрасывало любые пакеты, удовлетворяющие этому правилу. Следовательно, ваш компьютер будет отвергать любые пакеты, приходящие с адреса 192.168.34.63.

Чтобы увидеть это правило на своем месте, запустите команду `iptables -L`:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  -- 192.168.34.63         anywhere
```

Но вот беда: ваш приятель с адресом 192.168.34.63 сообщил всем в своей подсети, чтобы к вашему компьютеру подключались через порт SMTP (TCP-порт 25). Чтобы избавиться также и от этого трафика, запустите такую команду:

```
# iptables -A INPUT -s 192.168.34.0/24 -p tcp --destination-port 25 -j DROP
```

В этом примере добавлен спецификатор маски сети для адреса источника, а также флаг `-p tcp`, чтобы учитывать только пакеты TCP. Следующее ограничение, `--destination-port 25`, говорит о том, что данное правило должно применяться только к трафику порта 25. IP-таблица для цепочки INPUT теперь выглядит так:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  192.168.34.63        anywhere
DROP       tcp  --  192.168.34.0/24      anywhere         tcp dpt:smtp
```

Все идет замечательно, пока кто-то из ваших знакомых с адресом 192.168.34.37 не говорит вам, что он не может отправить вам почту, поскольку вы заблокировали его компьютер. Думая о том, что это легко исправить, вы запускаете следующую команду:

```
# iptables -A INPUT -s 192.168.34.37 -j ACCEPT
```

Однако это не срабатывает. Чтобы понять почему, взгляните на новую цепочку:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  192.168.34.63        anywhere
DROP       tcp  --  192.168.34.0/24      anywhere        tcp dpt:smtp
ACCEPT     all  --  192.168.34.37        anywhere
```

Ядро считывает цепочки сверху вниз, применяя первое правило, которое подходит.

Первое правило не подходит для адреса 192.168.34.37, а второе подходит, так как оно применяется ко всем хостам в диапазоне адресов от 192.168.34.1 до 192.168.34.254 и говорит о том, что пакеты должны отвергаться. Когда правило подходит, ядро приступает к действиям и уже не смотрит остальные цепочки. Вы, наверное, заметили, что с адреса 192.168.34.37 можно отправлять пакеты на любой порт вашего компьютера, *кроме* 25, так как второе правило применяется *только* для него.

Решение заключается в перемещении третьего правила вверх. Сначала удалите третье правило с помощью такой команды:

```
# iptables -D INPUT 3
```

Затем *вставьте* это правило в верхней части цепочки с помощью команды `iptables -I`:

```
# iptables -I INPUT -s 192.168.34.37 -j ACCEPT
```

Чтобы вставить правило где-либо внутри цепочки, укажите номер правила после имени цепочки (например, `iptables -I INPUT 4 ...`).

## 9.21.3. Стратегии для брандмауэров

Хотя приведенные выше указания объяснили вам, как вставлять правила и как ядро обрабатывает IP-цепочки, мы еще не видели стратегий для брандмауэра, которые реально работают. Теперь поговорим о них.

Существуют два основных типа сценариев для брандмауэров: один для защиты отдельных компьютеров (здесь можно указывать правила в цепочке INPUT каждого компьютера), а второй — для защиты компьютеров в составе сети (здесь правила указываются в цепочке FORWARD маршрутизатора). В обоих случаях вы не можете достичь серьезной защиты, если вы применяете по умолчанию политику ACCEPT, а затем постоянно добавляете правила, которые отбрасывают пакеты от источников, начинающих рассылать нежелательный контент. Вы должны разрешать только те пакеты, которым доверяете, и отвергать все остальные.

Допустим, что у вашего компьютера имеется SSH-сервер, обслуживающий TCP-порт 22. Нет никаких оснований для того, чтобы какой-либо хост инициировал соединение с другими портами вашего компьютера, и вы не должны давать такой возможности никаким хостам. Чтобы это организовать, сначала установите для цепочки INPUT политику DROP:

```
# iptables -P INPUT DROP
```

Чтобы включить ICMP-трафик (для команды `ping` и других утилит), используйте такую строку:

```
# iptables -A INPUT -p icmp -j ACCEPT
```

Убедитесь в том, что вы можете получать пакеты, которые вы отправляете как на IP-адреса внутри собственной сети, так и на адрес 127.0.0.1 (локальный хост). При условии, что *my_addr* — это IP-адрес вашего хоста, выполните следующее:

```
# iptables -A INPUT -s 127.0.0.1 -j ACCEPT
# iptables -A INPUT -s my_addr -j ACCEPT
```

Если вы управляете подсетью в целом (и доверяете в ней всему), можно заменить адрес *my_addr* на адрес вашей подсети и ее маску, например, на `10.23.2.0/24`.

Теперь, хотя вам по-прежнему требуется отклонять входящие TCP-соединения, необходимо убедиться в том, что ваш хост способен устанавливать TCP-соединения с внешним миром. Поскольку все TCP-соединения начинаются с пакета SYN (запрос соединения), то можно позволить поступление всех TCP-пакетов, которые не являются SYN-пакетами, и тогда все будет в порядке:

```
# iptables -A INPUT -p tcp '!' --syn -j ACCEPT
```

Далее, если вы используете удаленный сервер DNS с протоколом UDP, вы должны принимать трафик от вашего сервера имен, чтобы компьютер мог отыскивать имена с помощью службы DNS. Выполните это для всех серверов DNS, указанных в файле `/etc/resolv.conf`, используя следующую команду (здесь *ns_addr* означает адрес сервера имен):

```
# iptables -A INPUT -p udp --source-port 53 -s ns_addr -j ACCEPT
```

И наконец, разрешите SSH-соединения от кого угодно:

```
# iptables -A INPUT -p tcp --destination-port 22 -j ACCEPT
```

Приведенные настройки IP-таблиц подходят для многих ситуаций, включая любое прямое соединение (в особенности широкополосное), когда злоумышленник будет, скорее всего, сканировать порты вашего компьютера. Можно было бы также адаптировать эти настройки для маршрутизатора с функцией брандмауэра, использовав цепочку FORWARD вместо INPUT и указав подсети источника и назначения, где это допустимо. Для более сложных конфигураций может оказаться полезным такой инструмент, как надстройка Shorewall.

Наш рассказ лишь слегка затронул политику безопасности. Помните о ключевой идее: разрешать только то, что вы считаете приемлемым, и не пытаться выискивать и плохое содержимое. Более того, IP-фильтрация является лишь фрагментом в картине безопасности. Дополнительные сведения вы получите в следующей главе.

## 9.22. Сеть Ethernet, протоколы IP и ARP

В реализации протокола IP для сетей Ethernet есть одна интересная деталь, о которой мы еще не упоминали. Вспомните о том, что хост должен помещать IP-пакет в кадр Ethernet, чтобы переместить пакет на физическом уровне к другому хосту. Вспомните также, что сами кадры не содержат информацию об IP-адресе, они используют адреса MAC (аппаратные). Вопрос таков: каким образом при создании кадра Ethernet для IP-пакета хост определяет, какой MAC-адрес соответствует IP-адресу пункта назначения?

Обычно мы не раздумываем над этим вопросом, поскольку сетевое ПО содержит автоматическую систему поиска MAC-адресов, которая называется *протоколом ARP* (Address Resolution Protocol, протокол разрешения адресов). Хост, который использует сеть Ethernet как физический уровень и протокол IP как сетевой уро-

вень, содержит небольшую таблицу, называемую *кэшем ARP*, которая сопоставляет IP-адреса адресам MAC. В Linux кэш ARP расположен в ядре. Чтобы просмотреть кэш ARP на компьютере, используйте команду `arp`. Как и для многих других сетевых команд, параметр `-n` здесь отключает обратный поиск DNS.

```
$ arp -n
Address          Hwtype   Hwaddr              Flags Mask       Iface
10.1.2.141       ether    00:11:32:0d:ca:82   C                eth0
10.1.2.1         ether    00:24:a5:b5:a0:11   C                eth0
10.1.2.50        ether    00:0c:41:f6:1c:99   C                eth0
```

При загрузке компьютера кэш ARP пуст. Как же тогда MAC-адреса попадают в этот кэш? Все начинается тогда, когда компьютер желает отправить пакет другому хосту. Если целевой IP-адрес отсутствует в кэше ARP, выполняются следующие действия.

1. Хост-источник создает специальный кадр Ethernet, содержащий пакет запроса у кэша ARP тех адресов, которые соответствуют целевому IP-адресу.
2. Хост-источник передает этот кадр по всей физической сети в целевой подсети.
3. Если один из других хостов этой подсети знает правильный MAC-адрес, он создает ответный пакет и кадр, содержащий данный адрес, а затем отправляет его источнику. Зачастую отвечающий хост является целевым и просто отправляет в ответ собственный MAC-адрес.
4. Хост-источник добавляет пару адресов IP-MAC в кэш ARP и готов продолжить работу.

ПРИМЕЧАНИЕ

Помните о том, что кэш ARP применяется только для компьютеров локальных подсетей (загляните в раздел 9.4, чтобы определить ваши локальные подсети). Чтобы добраться до пунктов назначения за пределами подсети, хост отправляет пакет маршрутизатору, и эта задача в итоге переходит к кому-то другому. Конечно же, ваш хост должен знать MAC-адрес маршрутизатора и может использовать кэш ARP, чтобы выяснить адрес.

Единственная настоящая проблема с кэшем ARP может возникнуть, когда он становится устаревшим, если вы присвоили IP-адрес от одной карты сетевого интерфейса другой карте (например, при тестировании компьютера), поскольку у этих карт различные MAC-адреса. Система Unix делает недействительными записи в кэше ARP, если они неактивны в течение некоторого времени, поэтому здесь возникнет лишь небольшая задержка, вызванная недействующими данными. Можно немедленно удалить запись в кэше ARP с помощью такой команды:

```
# arp -d host
```

Можно также просмотреть кэш ARP для одного сетевого интерфейса с помощью команды:

```
$ arp -i interface
```

Страница руководства arp(8) содержит объяснение того, как вручную настроить записи в кэше ARP, но вам это вряд ли потребуется.

ПРИМЕЧАНИЕ

Не смешивайте кэш ARP с протоколом RARP (Reverse Address Resolution Protocol, протокол определения адреса по местоположению узла). Протокол RARP преобразует MAC-адрес обратно в имя хоста или в IP-адрес. До того как стал популярен протокол DHCP, некоторые бездисковые рабочие станции и другие устройства использовали протокол RARP для получения своей конфигурации, но сегодня он применяется редко.

# 9.23. Беспроводная сеть Ethernet

Беспроводные сети Ethernet (сети Wi-Fi) незначительно отличаются от проводных сетей. Подобно любым проводным аппаратным средствам, они обладают MAC-адресами и применяют кадры Ethernet, чтобы передавать и получать данные, в результате чего ядро Linux способно «общаться» с беспроводным сетевым интерфейсом во многом так же, как если бы это был проводной интерфейс. Все, что находится на сетевом уровне и выше, точно такое же; главные отличия — это дополнительные компоненты на физическом уровне, такие как частоты, идентификаторы сети, безопасность и т. д.

В отличие от проводных сетевых аппаратных средств, которые очень хороши при автоматической подстройке без лишней суеты под нюансы физической составляющей, конфигурация беспроводной сети допускает намного больше свободы. Чтобы беспроводной интерфейс работал корректно, в Linux необходимы дополнительные инструменты конфигурирования.

Вкратце рассмотрим дополнительные компоненты беспроводных сетей.

- **Подробности передачи.** Они содержат физические характеристики, такие как радиочастота.
- **Идентификация сети.** Поскольку одну и ту же базовую среду могут совместно использовать несколько беспроводных сетей, должна быть возможность их различения. Параметр SSID (Service Set Identifier, идентификатор сервисного набора, известный также как «имя сети») является идентификатором беспроводной сети.
- **Управление.** Хотя возможно настроить беспроводную сеть так, чтобы хосты взаимодействовали друг с другом напрямую, большинство беспроводных сетей управляется с помощью одной или нескольких *точек доступа*, через которые проходит весь трафик. Точки доступа часто выполняют функцию моста между беспроводной и проводной сетями, в результате чего обе они выглядят единой сетью.
- **Аутентификация.** Вам может понадобиться ограничение доступа к беспроводной сети. Чтобы это выполнить, можно настроить точки доступа таким образом, чтобы они запрашивали пароль или какой-либо аутентификационный ключ, прежде чем начать взаимодействие с клиентом.
- **Шифрование.** В дополнение к ограничению начального доступа к беспроводной сети, как правило, необходимо шифровать весь трафик, который передается с помощью радиоволн.

Конфигурация Linux, а также утилиты, которые работают с этими компонентами, расположены в нескольких областях системы. Некоторые находятся в ядре: Linux располагает набором расширений для работы с беспроводными сетями, стандартизирующими доступ к аппаратным средствам из пространства пользователя. По мере разрастания пространства пользователя беспроводная конфигурация может усложниться, поэтому большинство пользователей предпочитает использовать внешние графические интерфейсы, такие как апплет рабочего стола для менеджера NetworkManager, чтобы привести все в действие. Опять-таки иногда стоит посмотреть, что происходит за кулисами.

## 9.23.1. Утилита iw

Можно просмотреть и изменить конфигурацию устройств и сети в пространстве ядра с помощью утилиты `iw`. Чтобы ее использовать, обычно необходимо знать имя сетевого интерфейса для устройства, например `wlan0`. Вот пример, в котором выводится перечень доступных беспроводных сетей. Если вы находитесь в городе, будьте готовы к появлению длинного списка.

```
# iw dev wlan0 scan
```

ПРИМЕЧАНИЕ

Для того чтобы эта команда сработала, необходимо, чтобы сетевой интерфейс функционировал (если это не так, запустите команду ifconfig wlan0 up), но вам не потребуется настраивать какие-либо параметры сетевого уровня, например IP-адрес.

Если сетевой интерфейс подключился к беспроводной сети, можно просмотреть сведения о сети следующим образом:

```
# iw dev wlan0 link
```

Адрес MAC, показанный в отчете этой команды, является адресом точки доступа, к которой вы в данный момент подключены.

ПРИМЕЧАНИЕ

Утилита iw делает различие между именами физических устройств, такими как phy0, и именами сетевых интерфейсов, например wlan0, и позволяет вам изменить различные параметры в каждом случае. Можно даже создать несколько сетевых интерфейсов для единственного физического устройства. Тем не менее почти во всех основных случаях вы будете использовать имя сетевого интерфейса.

Для подключения сетевого интерфейса к незащищенной беспроводной сети воспользуйтесь такой командой:

```
# iw wlan0 connect network_name
```

Подключение к защищенным сетям — это совсем другая история. Для довольно ненадежной системы WEP (Wired Equivalent Privacy, протокол шифрования в беспроводной связи) можно использовать в команде `iw` параметр `keys`. Однако, если вы серьезно относитесь к защите, не следует применять протокол WEP.

## 9.23.2. Безопасность беспроводных сетей

Для большинства настроек беспроводной сети Linux опирается на демон `wpa_supplicant`, который управляет аутентификацией и шифрованием для интерфейса беспроводной сети. Этот демон может работать со схемами аутентификации WPA (Wi-Fi Protected Access, защищенный доступ к беспроводной сети) и WPA2, а также почти с любым типом шифрования, используемым в беспроводных сетях. При своем первом запуске этот демон читает конфигурационный файл (по умолчанию `/etc/wpa_supplicant.conf`) и пытается идентифицировать себя с точкой доступа, а затем установить связь на основе предоставленного имени сети. Система хорошо документирована; в частности, страницы руководства wpa_supplicant(1) и wpa_supplicant.conf(5) содержат множество подробностей.

Ручной запуск демона каждый раз, когда вам необходимо установить соединение, является слишком трудоемким. На самом деле уже само создание файла конфигурации довольно утомительно вследствие большого количества возможных вариантов. Чтобы усугубить ситуацию, все работы по запуску утилиты `iw` и демона `wpa_supplicant` всего лишь позволяют вашей системе физически подключиться к беспроводной сети, при этом не выполняется даже настройка сетевого уровня. Именно здесь такие менеджеры автоматической сетевой конфигурации, как NetworkManager, избавляют от большей части неприятных моментов. Хотя они не выполняют никакой работы самостоятельно, им известна правильная последовательность действий и необходимая конфигурация для каждого шага на пути к получению работающей беспроводной сети.

# 9.24. Резюме

Понимание места и роли различных сетевых уровней очень важно для усвоения того, как происходит работа с сетью в Linux и каким образом выполняется конфигурирование сети. Хотя мы рассмотрели только основы, более сложные темы, связанные с физическим, сетевым и транспортным уровнями, подобны тому, что вы увидели. Сами уровни зачастую делятся на более мелкие части из-за различных составляющих физического слоя беспроводной сети.

Значительная часть действий, которые вы наблюдали в этой главе, происходит в ядре с добавлением некоторых основных управляющих утилит из пространства пользователя, работающих с внутренними структурами данных ядра (например, с таблицами маршрутизации). Это традиционный способ работы с сетями. Однако некоторые задачи не подходят для ядра из-за своей сложности и требуемой от них гибкости. Тогда к делу подключаются утилиты из пространства пользователя. В частности, менеджер NetworkManager контролирует ядро и выполняет запросы, а затем управляет конфигурацией ядра. Еще один пример — поддержка протоколов динамической маршрутизации, таких как протокол BGP (Border Gateway Protocol, пограничный шлюзовый протокол), который используется в больших интернет-маршрутизаторах.

Возможно, к этому моменту вам немного наскучило конфигурирование сети. Перейдем к ее использованию на прикладном уровне.

# 10 Сетевые приложения и службы

В этой главе рассмотрены основы сетевых приложений — клиентов и серверов, работающих в пространстве пользователя, которое располагается на прикладном уровне. Так как этот уровень находится на самом верху стека, ближе всего к конечным пользователям, материал данной главы более доступен, по сравнению с главой 9. Действительно, вы взаимодействуете с такими клиентскими сетевыми приложениями, как браузеры и почтовые клиенты, каждый день.

Для выполнения своей работы сетевые клиенты подключаются к соответствующим сетевым серверам. Сетевые серверы Unix включаются в дело различными способами. Серверная команда может либо самостоятельно прослушивать порт, либо через вторичный сервер. В дополнение к этому у серверов нет общей базы данных конфигурации и широкого набора функций. Большинство серверов использует файл конфигурации для контроля своего поведения (хотя для такого файла и нет установленного формата), а многие применяют также системную службу syslog для записи уведомлений. Мы рассмотрим некоторые распространенные серверы, а также инструменты, которые помогут вам понять и отладить работу сервера.

Сетевые клиенты используют протоколы и интерфейсы транспортного уровня операционной системы, поэтому важно понимать основы транспортных уровней TCP и UDP. Начнем рассмотрение сетевых приложений, поэкспериментировав с сетевым клиентом, который использует протокол TCP.

## 10.1. Основные понятия о службах

Службы TCP являются одними из самых простых для понимания, поскольку они построены на несложных, непрерывных двухсторонних потоках данных. Вероятно, лучший способ увидеть, как они работают, — «пообщаться» с веб-сервером напрямую через TCP-порт 80 и получить представление о том, как данные перемещаются через это соединение. Запустите, например, такую команду для подключения к веб-серверу:

```
$ telnet www.wikipedia.org 80
```

Вы должны увидеть в ответ нечто подобное:

```
Trying some address...
Connected to www.wikipedia.org.
Escape character is '^]'.
```

Теперь введите:

```
GET / HTTP/1.0
```

Нажмите клавишу Enter дважды. Сервер должен отправить в виде ответа некоторое количество HTML-текста, а затем разорвать соединение.

Это упражнение говорит нам о том, что:

- на удаленном хосте есть процесс веб-сервера, прослушивающий TCP-порт 80;
- клиентом, который инициировал соединение, являлась команда `telnet`.

ПРИМЕЧАНИЕ

Команда telnet изначально была предназначена для осуществления входа на удаленные хосты. Хотя вход на удаленный сервер с помощью команды telnet без использования технологии Kerberos совершенно не защищен (как вы увидите далее), клиент telnet может быть полезен для отладки удаленных служб. Команда telnet не работает с протоколом UDP или любым транспортным уровнем, отличным от TCP. Если вы ищете сетевой клиент общего назначения, попробуйте команду netcat, описанную в подразделе 10.5.3.

В приведенном выше примере вы вручную выполнили взаимодействие с веб-сервером в сети с помощью команды `telnet`, использовав протокол HTTP (Hypertext Transfer Protocol, протокол передачи гипертекста) прикладного уровня. Хотя в обычных условиях вы воспользовались бы браузером для установления подобного соединения, немного отойдем от команды `telnet` и применим команду, которая знает, как «говорить» с прикладным уровнем HTTP. Мы используем утилиту `curl` со специальным параметром, чтобы записать подробности ее взаимодействия:

```
$ curl --trace-ascii trace_file http://www.wikipedia.org/
```

ПРИМЕЧАНИЕ

В вашей версии ОС может не оказаться встроенной утилиты curl, но если она понадобится, ее установка не должна вызвать трудностей.

Вы получите обширный отчет в формате HTML. Проигнорируйте его (или перенаправьте в устройство `/dev/null`) и вместо этого посмотрите только что созданный файл `trace_file`. При условии, что соединение оказалось успешным, первая часть этого файла, в том месте, где команда `curl` пытается установить TCP-соединение с сервером, должна выглядеть так:

```
== Info: About to connect() to www.wikipedia.org port 80 (#0)
== Info:   Trying 10.80.154.224... == Info: connected
```

Все, что вы видели до сих пор, происходит на транспортном уровне или под ним. Однако, если это соединение оказывается успешным, команда `curl` пытается отправить запрос («заголовок»); именно в этот момент в дело вступает прикладной уровень:

```
=> Send header, 167 bytes (0xa7)
0000: GET / HTTP/1.1
0010: User-Agent: curl/7.22.0 (i686-pc-linux-gnu) libcurl/7.22.0 OpenS
0050: SL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp/2.3
007f: Host: www.wikipedia.org
```

```
0098: Accept: */*
00a5:
```

Здесь первая строка представляет отладочный вывод команды curl, сообщающий о дальнейших действиях команды. Остальные строки показывают, что́ именно команда curl отправляет серверу. Выделенный жирным шрифтом текст соответствует тому, что приходит на сервер; шестнадцатеричные числа в начале строк являются лишь отладочными смещениями команды curl, которые могут помочь вам отследить, какое количество данных было отправлено или получено.

Видно, что команда curl начинает работу с отправки запроса GET серверу (как вы это делали с помощью команды telnet), за которым следует дополнительная информация для сервера и пустая строка. Далее сервер отправляет ответ, первый с собственным заголовком, который выделен здесь жирным шрифтом:

```
<= Recv header, 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header, 16 bytes (0x10)
0000: Server: Apache
<= Recv header, 42 bytes (0x2a)
0000: X-Powered-By: PHP/5.3.10-1ubuntu3.9+wmf1
--snip--
```

Во многом подобно предыдущему выводу, здесь строки <= являются отладочными, а числа 0000:, с которых они начинаются, сообщают вам смещения.

Заголовок в ответе сервера может оказаться достаточно длинным, но в определенный момент сервер переходит от передачи заголовков к отправке запрашиваемого документа, например, так:

```
<= Recv header, 55 bytes (0x37)
0000: X-Cache: cp1055 hit (16), cp1054 frontend hit (22384)
<= Recv header, 2 bytes (0x2)
0000:
<= Recv data, 877 bytes (0x36d)
0000: 008000
0008: <!DOCTYPE html>.<html lang="mul" dir="ltr">.<head>.<!-- Sysops:
--snip--
```

Этот вывод иллюстрирует также важное свойство прикладного уровня. Даже если отладочный вывод содержит Recv header и Recv data, подразумевая за ними два различных типа сообщений от сервера, нет никаких различий ни в том, как команда curl общается с операционной системой для извлечения этих сообщений, ни в том, как операционная система обращается с ними, ни в том, как сеть обрабатывает лежащие в их основе пакеты. Различие содержится полностью внутри приложения curl в пространстве пользователя. Команда curl знает о том, что она получает заголовки, пока ей не встретится пустая строка (двухбайтный фрагмент в середине), которая сигнализирует об окончании HTTP-заголовков, тогда команда интерпретирует все, что последует далее, как запрашиваемый документ.

Это же верно и для сервера, отправляющего данные. При отправке ответа сервер не делает различий между заголовком и данными документа, отправленными

операционной системе; различия появляются внутри серверной программы в пространстве пользователя.

# 10.2. Сетевые серверы

Большинство сетевых серверов подобно другим демонам системы, таким как `cron`, за исключением того, что они взаимодействуют с сетевыми портами. В самом деле, вспомните демон `syslogd`, описанный в главе 7: он принимает пакеты UDP в сетевом порте 514, когда запущен с параметром `-r`.

Есть несколько других распространенных сетевых серверов, которые вы можете найти в своей системе:

- `httpd`, `apache`, `apache2` — веб-серверы;
- `sshd` — демон защищенной оболочки (см. раздел 10.3);
- `postfix`, `qmail`, `sendmail` — почтовые серверы;
- `cupsd` — сервер печати;
- `nfsd`, `mountd` — демоны сетевой файловой системы (для совместного использования файлов);
- `smbd`, `nmbd` — демоны совместного использования файлов Windows (см. главу 12);
- `rpcbind` — демон удаленного вызова процедуры (RPC, Remote Procedure Call) для службы зеркала портов.

Общим свойством большинства сетевых серверов является то, что они обычно действуют в виде нескольких процессов. Хотя бы один из процессов прослушивает сетевой порт, и когда поступает новое входящее соединение, прослушивающий процесс использует команду `fork()`, чтобы создать новый дочерний процесс, который становится ответственным за новое соединение. Процесс-потомок, или *исполнитель*, завершает работу при закрытии соединения. Тем временем исходный процесс продолжает прослушивание сетевого порта. Этот процесс позволяет серверу с легкостью справляться с множеством подключений, не создавая сложностей.

Однако имеются некоторые исключения из этой модели. Вызов команды `fork()` добавляет системе дополнительную работу. Для сравнения: такие высокопроизводительные TCP-серверы, как веб-сервер Apache, могут во время запуска создать несколько процессов-исполнителей, чтобы они всегда были наготове для обработки соединений. Серверы, которые принимают UDP-пакеты, просто получают данные и реагируют на них. У них нет соединений, которые надо прослушивать.

# 10.3. Защищенная оболочка (SSH)

Каждый сервер работает по-своему. Подробно рассмотрим автономный сервер SSH. Одним из самых распространенных сетевых сервисных приложений является защищенная оболочка (SSH) — стандарт де-факто для удаленного доступа к компьютерам с Unix. Настроенная оболочка SSH дает возможность защищенного входа в оболочку, удаленного исполнения команд, простого совместного использования

файлов, а также позволяет заменить старые, незащищенные системы удаленного доступа `telnet` и `rlogin` на криптографические системы с открытым ключом для аутентификации и упрощенными шифрами для сеансовых данных. Большинство поставщиков интернет-услуг и облачных сервисов требуют наличия оболочки SSH для доступа к своим сервисам, а многие сетевые устройства на основе Linux (например, устройства сетевого хранения данных) также обеспечивают доступ с помощью оболочки SSH. Оболочка OpenSSH (http://www.openssh.com/) является популярной бесплатной реализацией SSH для Unix, и она присутствует практически во всех версиях Linux. Клиент оболочки OpenSSH называется `ssh`, а сервер — `sshd`. Существуют две основные версии протокола SSH: 1 и 2. Оболочка OpenSSH поддерживает обе версии, однако первая применяется редко.

Из многих полезных возможностей и функций оболочки SSH можно упомянуть следующие:

- шифрование пароля и других сеансовых данных для защиты от шпионов;
- туннелирование других сетевых соединений, включая те, которые исходят от клиентов системы X Window (подробнее об этом рассказано в главе 14);
- наличие клиентов почти для любой операционной системы;
- использование ключей для аутентификации хоста.

ПРИМЕЧАНИЕ

Туннелирование — это процесс упаковки и передачи одного сетевого подключения с помощью другого. Преимущества использования оболочки SSH для туннелирования подключений системы X Window заключаются в том, что оболочка SSH настраивает среду отображения за вас и шифрует данные внутри туннеля.

Однако у оболочки SSH есть и свои недостатки. Для начала, чтобы установить SSH-соединение, вам необходим открытый ключ удаленного хоста, а он появляется у вас не обязательно с помощью защищенного способа (хотя можно проверить его вручную, чтобы убедиться в том, что вы не подверглись взлому). Чтобы получить представление о работе различных криптографических методов, обратитесь к книге Брюса Шнайера (Bruce Schneier) *Applied Cryptography: Protocols, Algorithms, and Source Code in C* («Прикладная криптография: протоколы, алгоритмы и программный код на языке C»), 2-е издание (Wiley, 1996). Более подробно об оболочке SSH и работе с ней рассказано в книге Майкла У. Лукаса (Michael W. Lucas) *OpenSSH, PuTTY, Tunnels and Keys* («Оболочка OpenSSH, клиент PuTTY, туннели и ключи»), а также в книге Дэниела Дж. Барретта (Daniel J. Barrett), Ричарда Е. Сильвермана (Richard E. Silverman) и Роберта Дж. Бернса (Robert G. Byrnes) *SSH, The Secure Shell* («SSH — защищенная оболочка»), 2-е издание (O'Reilly, 2005).

## 10.3.1. Сервер SSHD

Для запуска сервера `sshd` необходим файл конфигурации, а также ключи хоста. В большинстве версий ОС файл конфигурации находится в каталоге `/etc/ssh`, и если вы установили пакет `sshd`, то вся конфигурация будет выполнена корректно за вас. Имя файла конфигурации `sshd_config` легко спутать с файлом установщика клиента `ssh_config`, поэтому будьте внимательны.

В файле sshd_config вам не придется что-либо менять, но никогда не помешает проверить его. Этот файл состоит из пар «ключевое слово — значение», как показано в приведенном фрагменте:

```
Port 22
#Protocol 2,1
#ListenAddress 0.0.0.0
#ListenAddress ::
HostKey /etc/ssh/ssh_host_key
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
```

Строки, которые начинаются с символа #, являются комментариями, и многие из них в файле sshd_config могут указывать на значения по умолчанию. Страница руководства sshd_config(5) содержит описание всех возможных значений, наиболее важными из которых являются:

- HostKey *file* — использует файл *file* в качестве ключа хоста (о ключах хоста скоро пойдет речь);
- LogLevel *level* — заносит сообщения с помощью уровня *level* системного журнала;
- PermitRootLogin *value* — позволяет пользователю superuser войти в защищенную оболочку, если значение *value* равно yes. Чтобы предотвратить вход, установите значение no;
- SyslogFacility *name* — заносит сообщения с помощью устройства *name* системного журнала;
- X11Forwarding *value* — включает туннелирование клиента системы X Window, если значение *value* равно yes;
- XAuthLocation *path* — обеспечивает путь для команды xauth. Туннелирование системы X11 не будет работать без этого пути. Если команда xauth расположена не в каталоге /usr/bin, укажите для значения *path* полный путь к команде xauth.

## Ключи хоста

У оболочки OpenSSH есть три набора ключей хоста: один для протокола версии 1 и два для протокола 2-й версии. В каждом наборе присутствует *открытый ключ* (файл с расширением .pub) и *секретный ключ* (файл без расширения). Никому не показывайте секретный ключ, даже в собственной системе, поскольку при этом вы подвергаетесь риску вторжения злоумышленников.

В оболочке SSH версии 1 есть только ключи RSA, а в версии 2 есть ключи RSA и DSA. RSA и DSA являются алгоритмами шифрования открытого ключа. Имена файлов ключей приведены в табл. 10.1.

**Таблица 10.1.** Файлы ключей оболочки OpenSSH

| Имя файла | Тип ключа |
|---|---|
| ssh_host_rsa_key | Секретный ключ RSA (версия 2) |
| ssh_host_rsa_key.pub | Открытый ключ RSA (версия 2) |

| Имя файла | Тип ключа |
|---|---|
| ssh_host_dsa_key | Секретный ключ DSA (версия 2) |
| ssh_host_dsa_key.pub | Открытый ключ DSA (версия 2) |
| ssh_host_key | Секретный ключ RSA (версия 1) |
| ssh_host_key.pub | Открытый ключ RSA (версия 1) |

Обычно вам не понадобится создавать ключи, поскольку за вас это выполнит команда из оболочки OpenSSH или из вашей версии ОС, но вам все же следует знать о том, как создавать ключи, если вы планируете использовать команды, подобные `ssh-agent`. Чтобы создать ключи для протокола SSH версии 2, используйте команду `ssh-keygen`, которая включена в оболочку OpenSSH:

```
# ssh-keygen -t rsa -N '' -f /etc/ssh/ssh_host_rsa_key
# ssh-keygen -t dsa -N '' -f /etc/ssh/ssh_host_dsa_key
```

Для версии 1 воспользуйтесь таким вариантом:

```
# ssh-keygen -t rsa1 -N '' -f /etc/ssh/ssh_host_key
```

Сервер SSH и клиенты применяют также файл ключей `ssh_known_hosts`, который содержит открытые ключи от других хостов. Если вы намерены использовать аутентификацию на основе хостов, файл `ssh_known_hosts` на сервере должен содержать открытые ключи хостов для всех надежных клиентов. Знание о файлах ключей пригодится, когда вы приступите к замене компьютера. При настройке нового компьютера с нуля можно импортировать файлы ключей со старого компьютера, чтобы у пользователей не возникло несоответствие ключей при подключении к новому компьютеру.

## Запуск сервера SSH

Хотя в большинстве версий ОС присутствует оболочка SSH, сервер `sshd` обычно не запускается по умолчанию. В Ubuntu и Debian при установке пакета SSH-сервера создаются ключи, запускается сервер и заносится информация о запуске в конфигурацию загрузки системы. В Fedora сервер `sshd` установлен по умолчанию, но отключен. Чтобы запустить сервер `sshd` при загрузке системы, воспользуйтесь командой `chkconfig` таким образом (при этом сервер не будет запущен сразу же; для его запуска используйте команду `service sshd start`):

```
# chkconfig sshd on
```

В Fedora при первом запуске сервера `sshd` обычно создаются все отсутствующие файлы хост-ключей.

Если у вас еще не установлена поддержка системы init, то при запуске команды `sshd` с корневыми правами запускается сервер и во время запуска сервер `sshd` записывает свой идентификатор PID в файл `/var/run/sshd.pid`.

Можно также запустить сервер `sshd` в качестве модуля сокета в версии systemd или с помощью команды `inetd`, но это, как правило, не очень корректно, поскольку серверу иногда требуется создавать файлы ключей, а на это требуется довольно много времени.

## 10.3.2. Клиент SSH

Чтобы подключиться к удаленному хосту, запустите команду:

```
$ ssh remote_username@host
```

Можно опустить параметр *remote_username*@, если ваше локальное имя пользователя такое же, как и для хоста. Команду ssh можно также встраивать в «конвейер», как показано в приведенном ниже примере, в котором каталог dir копируется на другой хост:

```
$ tar zcvf - dir | ssh remote_host tar zxvf -
```

Файл ssh_config глобальной конфигурации клиента SSH должен располагаться в каталоге /etc/ssh вместе с вашим файлом sshd_config. Как и в файле конфигурации сервера, файл конфигурации клиента содержит пары «ключ — значение», но вам не следует их изменять.

Наиболее часто проблемы при использовании клиентов SSH возникают, если открытый ключ в локальных файлах ssh_known_hosts или .ssh/known_hosts не совпадает с ключом на удаленном хосте. Неправильные ключи могут вызвать ошибку или появление подобного предупреждения:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
38:c2:f6:0d:0d:49:d4:05:55:68:54:2a:2f:83:06:11.
Please contact your system administrator.
Add correct host key in /home/user/.ssh/known_hosts to get rid of this
message.
Offending key in /home/user/.ssh/known_hosts:12❶
RSA host key for host has changed and you have requested
strict checking.
Host key verification failed.
```

Обычно это означает лишь, что администратор удаленного хоста изменил ключи (такое часто происходит при замене аппаратных средств), но никогда не повредит спросить об этом у самого администратора, если вы не уверены. В любом случае предыдущее сообщение говорит вам о том, что неправильный ключ находится в строке 12 пользовательского файла known_hosts (отмечено маркером ❶).

При отсутствии каких-либо подозрений просто удалите ошибочную строку или замените ее правильным открытым ключом.

### Клиенты передачи файлов в оболочке SSH

Оболочка OpenSSH содержит команды для передачи файлов, scp и sftp, которые призваны заменить старые, незащищенные команды rcp и ftp.

Команду scp можно использовать для передачи файлов между удаленным компьютером и вашим или от одного хоста к другому. Она работает подобно команде cp. Приведем несколько примеров:

```
$ scp user@host:file .
$ scp file user@host:dir
$ scp user1@host1:file user2@host2:dir
```

Команда sftp работает подобно клиенту ftp из командной строки, используя команды get и put. На удаленном хосте должна быть установлена команда sftp-server, наличие которой можно ожидать, если удаленный хост также применяет оболочку OpenSSH.

ПРИМЕЧАНИЕ

Если вам необходимо больше функций и гибкости, чем предлагают команды scp и sftp (например, если вы часто передаете большие числа или файлы), обратите внимание на команду rsync, о которой рассказано в главе 12.

### Клиенты SSH для платформ, отличных от Unix

Существуют клиенты SSH для всех популярных операционных систем, как указано на веб-странице проекта OpenSSH (http://www.openssh.com/). Какой из них следует выбрать? Подойдет клиент PuTTY — базовый клиент для Windows, который содержит команду защищенного копирования файлов. Клиент MacSSH хорошо работает в Mac OS 9.x и более ранних версиях. Система Mac OS X основана на Unix и уже содержит клиент OpenSSH.

## 10.4. Демоны inetd и xinetd

Реализация автономных серверов для каждой службы была бы неэффективной. Каждый сервер должен быть отдельно настроен на прослушивание порта, контроль доступа и конфигурирование порта. Эти действия выполняются одинаково для большинства служб, различия возникают только в способе обработки связи, когда сервер принимает соединение.

Один из традиционных способов упрощения использования серверов — демон inetd, своеобразный *суперсервер*, предназначенный для стандартизации доступа к сетевым портам и интерфейсов между командами сервера и сетевыми портами. После запуска демон inetd читает свой файл конфигурации, а затем прослушивает сетевые порты, указанные в этом файле. При возникновении новых сетевых соединений демон inetd подключает вновь стартовавший процесс к соединению.

Новая версия демона inetd под названием xinetd обеспечивает упрощенную конфигурацию и лучший контроль доступа, однако роль демона xinetd сокращается в пользу применения варианта systemd, который может обеспечить такую же функциональность с помощью модулей сокетов, как описано в подразделе 6.4.7.

Хотя демон inetd уже не используется широко, его конфигурация демонстрирует все необходимое для настройки службы. Оказывается, демон sshd может быть также вызван с помощью демона inetd, а не как автономный север, что видно из файла конфигурации /etc/inetd.conf:

```
ident        stream    tcp    nowait    root    /usr/sbin/sshd    sshd -i
```

Семь полей, присутствующих здесь, таковы:

- **имя службы** — имя службы из файла /etc/services (см. подраздел 9.14.3);
- **тип сокета** — обычно это stream для протокола TCP и dgram для протокола UDP;
- **протокол** — транспортный протокол, обычно tcp или udp;
- **поведение сервера дейтаграмм** — для протокола UDP это wait или nowait. Службы, которые применяют другой транспортный протокол, должны использовать вариант nowait;
- **пользователь** — имя пользователя, который запускает службу. Добавьте .group, чтобы указать группу пользователей;
- **исполняемый файл** — команда, которую демон inetd должен подключить к службе;
- **аргументы** — аргументы для исполняемого файла. Первый аргумент должен быть именем команды.

**Обертки TCP: tcpd, /etc/hosts.allow и /etc/hosts.deny.** До того как низкоуровневые брандмауэры стали популярны, многие администраторы использовали библиотеку *обертки TCP* и демон, чтобы контролировать хосты при работе с сетевыми службами. В таких реализациях демон inetd запускает команду tcpd, которая сначала отыскивает входящее соединение, а также списки контроля доступа в файлах /etc/hosts.allow и /etc/hosts.deny. Команда tcpd регистрирует соединение и, если она решает, что входящее соединение в порядке, передает его окончательной команде службы. Вам может встретиться система, которая по-прежнему использует обертку TCP, но мы не будем детально рассматривать ее, поскольку она выходит из употребления.

# 10.5. Инструменты диагностики

Рассмотрим диагностические инструменты, необходимые при исследовании прикладного уровня. Некоторые из них проникают в транспортный и сетевой уровни, поскольку все, что находится на прикладном уровне, в конечном итоге ведет на уровень ниже, к чему-либо, расположенному там.

Как отмечалось в главе 9, команда netstat является базовой сетевой службой отладки, которая может отобразить разнообразную статистику о транспортном и сетевом уровнях. В табл. 10.2 приведены некоторые полезные параметры для просмотра соединений.

**Таблица 10.2.** Полезные параметры команды netstat, относящиеся к отчетам о соединении

| Параметр | Описание |
|---|---|
| -t | Вывести информацию о порте TCP |
| -u | Вывести информацию о порте UDP |
| -l | Вывести прослушивающие порты |
| -a | Вывести все активные порты |
| -n | Отключить поиск имен (для ускорения работы; полезно также, если не работает служба DNS) |

## 10.5.1. Команда lsof

Из главы 8 вы узнали о том, что команда `lsof` способна отслеживать открытые файлы, но она может также выводить список команд, которые в данный момент используют или прослушивают порты. Чтобы увидеть полный перечень команд, применяющих или прослушивающих порты, запустите такую команду:

```
# lsof -i
```

Если ее запустить с правами обычного пользователя, она покажет только процессы этого пользователя. При запуске с корневыми правами отчет будет выглядеть подобно приведенному ниже, с различными процессами и пользователями:

```
COMMAND   PID    USER   FD  TYPE    DEVICE SIZE/OFF  NODE NAME
rpcbind   700    root   6u  IPv4     10492      0t0  UDP *:sunrpc
rpcbind   700    root   8u  IPv4     10508      0t0  TCP *:sunrpc (LISTEN)
avahi-dae 872    avahi  13u IPv4  21736375      0t0  UDP *:mdns
cupsd     1010   root   9u  IPv6  42321174      0t0  TCP ip6-localhost:ipp (LISTEN)
ssh       14366  juser  3u  IPv4  38995911      0t0  TCP thishost.local:55457->
    somehost.example.com:ssh (ESTABLISHED)
chromium- 26534  juser  8r  IPv4  42525253      0t0  TCP thishost.local:41551->
    anotherhost.example.com:https (ESTABLISHED)
```

Этот пример отчета показывает пользователей и идентификаторы процессов для команд сервера и клиента, начиная со «старомодных» служб RPC вверху и заканчивая многоадресной службой DNS, которую обеспечивает команда `avahi`, и даже сервером печати (`cupsd`), готовым к использованию протокола IPv6. Две последние записи показывают соединения клиента: SSH-соединение и защищенное веб-соединение, установленное браузером Chromium. Поскольку отчет может оказаться довольно обширным, лучше применить фильтр (как рассказано в следующем разделе).

Команда `lsof` похожа на команду `netstat` тем, что она пытается выполнить обратное разрешение каждого IP-адреса в имя хоста, и это замедляет вывод. Используйте параметр `-n`, чтобы отключить разрешение имен:

```
# lsof -n -i
```

Можно также установить флаг `-P`, чтобы отключить просмотр имен портов в файле `/etc/services`.

### Фильтрация по протоколу и порту

Если вы ищете какой-либо конкретный порт (допустим, вам известно, что какой-то процесс использует этот порт, и вы желаете узнать, что это за процесс), примените такую команду:

```
# lsof -i:port
```

Полный синтаксис такой:

```
# lsof -iprotocol@host:port
```

Параметры *protocol*, *@host* и *:port* являются необязательными и будут соответствующим образом фильтровать вывод команды `lsof`. Как и в большинстве сетевых утилит, параметры *host* и *port* могут быть либо именами, либо числами. Например, если вы желаете увидеть лишь соединения для TCP-порта 80 (это порт протокола HTTP), используйте команду:

```
# lsof -iTCP:80
```

### Фильтрация по статусу соединения

Чрезвычайно удобным фильтром команды `lsof` является статус соединения. Чтобы, например, отобразить только те процессы, которые прослушивают порты TCP, введите такую команду:

```
# lsof -iTCP -sTCP:LISTEN
```

Эта команда даст вам хороший обзор процессов сетевого сервера, запущенных в данный момент в системе. Однако, поскольку серверы UDP не выполняют прослушивание и не имеют соединений, вам придется использовать параметр `-iUDP`, чтобы увидеть запущенные клиенты наряду с серверами. Как правило, это не вызовет затруднений, так как в вашей системе будет, вероятно, немного серверов UDP.

## 10.5.2. Команда tcpdump

Если вам необходимо в точности узнать, что проходит через вашу сеть, команда `tcpdump` переводит карту сетевого интерфейса в *неизбирательный режим* и докладывает о каждом пакете, который перемещается по проводам. Если ввести команду `tcpdump` без аргументов, то в результате появится отчет, подобный приведенному, содержащий запрос ARP и веб-соединение:

```
# tcpdump
tcpdump: listening on eth0
20:36:25.771304 arp who-has mikado.example.com tell duplex.example.com
20:36:25.774729 arp reply mikado.example.com is-at 0:2:2d:b:ee:4e
20:36:25.774796 duplex.example.com.48455 > mikado.example.com.www: S
3200063165:3200063165(0) win 5840 <mss 1460,sackOK,timestamp 38815804[|tcp]>
(DF)
20:36:25.779283 mikado.example.com.www > duplex.example.com.48455: S
3494716463:3494716463(0) ack 3200063166 win 5792 <mss 1460,sackOK,timestamp
4620[|tcp]> (DF)
20:36:25.779409 duplex.example.com.48455 > mikado.example.com.www: . ack 1 win
```

```
5840 <nop,nop,timestamp 38815805 4620> (DF)
20:36:25.779787 duplex.example.com.48455 > mikado.example.com.www: P 1:427(426)
ack 1 win 5840 <nop,nop,timestamp 38815805 4620> (DF)
20:36:25.784012 mikado.example.com.www > duplex.example.com.48455: . ack 427
win 6432 <nop,nop,timestamp 4620 38815805> (DF)
20:36:25.845645 mikado.example.com.www > duplex.example.com.48455: P 1:773(772)
ack 427 win 6432 <nop,nop,timestamp 4626 38815805> (DF)
20:36:25.845732 duplex.example.com.48455 > mikado.example.com.www: . ack 773
win 6948 <nop,nop,timestamp 38815812 4626> (DF)

9 packets received by filter
0 packets dropped by kernel
```

Можно сделать этот отчет более конкретным, если добавить фильтры. Можно выполнить фильтрацию на основе хостов источника и назначения, сети, адресов Ethernet, протоколов и множества различных уровней в модели сети, а также многого другого. В число протоколов, которые распознает команда `tcpdump`, входят протоколы ARP, RARP, ICMP, TCP, UDP, IP, IPv6, AppleTalk и пакеты IPX. Чтобы, например, вывести с помощью команды `tcpdump` только пакеты TCP, запустите:

```
# tcpdump tcp
```

Чтобы увидеть веб-пакеты и пакеты UDP, введите такую команду:

```
# tcpdump udp or port 80
```

ПРИМЕЧАНИЕ

Если вам необходима тщательная проверка пакетов, попробуйте использовать альтернативу команде tcpdump с графическим интерфейсом — например, приложение Wireshark.

## Примитивы

В предыдущих примерах элементы `tcp`, `udp` и `port 80` называются *примитивами*. Самые важные примитивы приведены в табл. 10.3.

**Таблица 10.3.** Примитивы команды tcpdump

| Примитив | Спецификация пакета |
|---|---|
| tcp | Пакеты TCP |
| udp | Пакеты UDP |
| port port | Пакеты TCP и/или UDP к порту port или от него |
| host host | Пакеты к хосту host или от него |
| net network | Пакеты к сети network или от нее |

## Операторы

В предыдущем примере в качестве *оператора* применяется слово `or`. Команда `tcpdump` может использовать несколько операторов (таких как `and` и `!`), которые можно группировать с помощью скобок. Если вы планируете какую-либо серьезную работу с командой `tcpdump`, обязательно прочтите страницы руководства, в особенности раздел, который описывает примитивы.

### Когда не следует применять команду tcpdump

Будьте очень осторожны при использовании команды `tcpdump`. Отчет команды `tcpdump`, показанный выше в этом разделе, содержит только информацию заголовка пакетов TCP (транспортный уровень) и IP (интернет-уровень), но вы можете также вывести с помощью команды `tcpdump` все содержимое пакета. Хотя многие сетевые операторы позволяют с легкостью просматривать сетевые пакеты, не следует шпионить за сетями, если вы не являетесь их владельцем.

## 10.5.3. Команда netcat

Если вам необходимо больше гибкости, чем позволяет команда вроде `telnet` *`host port`* при соединении с удаленным хостом, используйте команду `netcat` (или `nc`). Эта команда может подключаться к удаленным портам TCP/UDP, определять локальный порт, прослушивать и сканировать порты, перенаправлять стандартный ввод/вывод к сетевым соединениям и от них, а также многое другое. Чтобы открыть TCP-соединение с портом с помощью команды `netcat`, запустите:

```
$ netcat host port
```

Команда `netcat` завершает работу, только если другая сторона соединения разрывает его. Это может привести к путанице, если вы перенаправляете стандартный ввод в команду `netcat`. Разорвать соединение в любой момент можно с помощью нажатия сочетания клавиш Ctrl+C. Если вы предпочитаете, чтобы команда завершала работу и разрывала соединение на основе стандартного потока ввода, попробуйте команду `sock`.

Для прослушивания конкретного порта запустите такую команду:

```
$ netcat -l -p port_number
```

## 10.5.4. Сканирование портов

Иногда неизвестно даже то, какие службы предлагают компьютеры вашей сети или какие IP-адреса используются. Утилита Network Mapper (Nmap) сканирует все порты компьютера или сети компьютеров в поисках открытых портов, а затем выводит список всех обнаруженных портов. Большинство версий ОС содержит пакет Nmap, который можно также получить на сайте http://www.insecure.org/. Ознакомьтесь со страницей руководства по утилите Nmap или онлайн-источниками, чтобы узнать о том, что она может делать.

При перечислении портов вашего компьютера удобно запустить сканирование по крайней мере с двух точек: на вашем компьютере и на каком-либо еще (возможно, за пределами вашей локальной сети). Так вы получите представление о том, что блокируется вашим брандмауэром.

ВНИМАНИЕ

Если кто-либо еще контролирует сеть, которую вы собираетесь просканировать с помощью команды Nmap, спросите разрешение. Администраторы сети следят за сканированием портов и, как правило, лишают доступа те компьютеры, которые пользуются этим.

Запустите команду `nmap` *host*, чтобы выполнить обобщенное сканирование портов. Например, так:

```
$ nmap 10.1.2.2
Starting Nmap 5.21 ( http://nmap.org ) at 2015-09-21 16:51 PST
Nmap scan report for 10.1.2.2
Host is up (0.00027s latency).
Not shown: 993 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
25/tcp   open  smtp
80/tcp   open  http
111/tcp  open  rpcbind
8800/tcp open  unknown
9000/tcp open  cslistener
9090/tcp open  zeus-admin

Nmap done: 1 IP address (1 host up) scanned in 0.12 seconds
```

Как видите, здесь открыто несколько служб, многие по умолчанию не включены в большинстве версий ОС. На самом деле единственным портом, который по умолчанию включен, является порт 111 (порт `rpcbind`).

# 10.6. Удаленный вызов процедур (RPC)

Что же это за служба `rpcbind`, которую вы только что видели в предыдущем разделе при сканировании? RPC означает *«удаленный вызов процедур»* (Remote Procedure Call). Это система, которая расположена в нижней части прикладного уровня. Она предназначена для того, чтобы программистам было легче получать доступ к сетевым приложениям, используя следующее обстоятельство: команды вызывают функции в удаленных командах (которые идентифицируются по номерам), а затем удаленные команды возвращают результат или сообщение.

В реализациях RPC использованы транспортные протоколы, такие как TCP и UDP, и для них необходима специальная посредническая служба, которая сопоставляет номера команд с портами TCP и UDP. Сервер под названием `rpcbind` должен быть запущен на каждом компьютере, который будет использовать службы RPC.

Чтобы узнать, какие службы RPC есть на вашем компьютере, запустите такую команду:

```
$ rpcinfo -p localhost
```

Протокол RPC является одним из тех, который просто не должен выйти из употребления. Сетевая файловая система (NFS, Network File System) и служба сетевого информирования (NIS, Network Information Service) используют протокол RPC, но они абсолютно излишни для автономных компьютеров. Однако, как только вы решите, что полностью избавились от необходимости в команде `rpcbind`, возникает что-либо еще, например поддержка монитора доступа к файлам (FAM, File Access Monitor) в среде GNOME.

# 10.7. Сетевая безопасность

Поскольку Linux является очень популярным вариантом Unix для персональных компьютеров, в особенности потому, что широко используется для веб-серверов, она притягивает внимание многих субъектов, которые пытаются проникнуть в компьютерные системы. Нами были рассмотрены брандмауэры, но на этом рассказ о безопасности не заканчивается.

Сетевая безопасность привлекает экстремистов: как тех, кому *действительно* по душе вторжение в чужие системы (ради развлечения или ради денег), так и тех, кто создает замысловатые схемы и кому *действительно* нравится бороться со взломщиками систем (это также может быть очень прибыльным делом). К счастью, вам не надо знать очень много, чтобы обезопасить свою систему. Вот несколько основных правил.

- **Запускайте как можно меньше служб.** Взломщики не смогут взломать службу, которой нет в вашей системе. Если вы знаете о какой-либо службе, что она не используется вами, не включайте ее только из соображений, что она может вам пригодиться «когда-нибудь потом».
- **Блокируйте с помощью брандмауэра настолько много, насколько возможно.** В системах Unix есть несколько внутренних служб, о которых вы можете не догадываться (например, TCP-порт 111 для сервера RPC), и о них *не следует* знать никакой другой системе в мире. Может оказаться очень трудно отслеживать службы и управлять ими, если различные команды прослушивают разные порты. Чтобы не позволить взломщикам определить внутренние службы вашей системы, используйте эффективные правила для брандмауэра, а также установите брандмауэр в вашем маршрутизаторе.
- **Отслеживайте службы, которые выходят в Интернет.** Если у вас запущен сервер SSH, Postfix или подобные им службы, обновляйте программное обеспечение и принимайте соответствующие меры защиты (см. подраздел 10.7.2).
- **Используйте для серверов дистрибутивы «с долгосрочной поддержкой».** Группы разработчиков систем безопасности обычно вплотную работают над стабильными и поддерживаемыми релизами ПО. Релизам для разработчиков и для тестирования, таким как Debian Unstable и Fedora Rawhide, уделяется гораздо меньше внимания.
- **Не создавайте в своей системе учетную запись для того, кому она не нужна.** Намного проще получить доступ с корневыми правами из локальной учетной записи, чем выполнить взлом удаленно. На самом деле вследствие огромного количества программ (в которых есть ошибки и недоработки), доступных в большинстве систем, можно легко получить доступ в систему с корневыми правами после получения приглашения от оболочки. Не рассчитывайте на то, что ваши друзья знают о том, как защищать свои пароли (или умеют выбирать хорошие пароли).
- **Не устанавливайте сомнительные двоичные пакеты.** Они могут содержать вирусы-трояны.

Такова практическая сторона самозащиты. Почему это важно? Есть три основных типа сетевых атак.

- **Полная компрометация.** Это означает получение корневого доступа к компьютеру (полный контроль). Взломщик может выполнить это, применив сервисную атаку, например за счет использования ошибки переполнения буфера или захватив плохо защищенную учетную запись пользователя, а затем взломав некачественно написанную команду `setuid`.
- **DoS-атака (Denial-of-Service, отказ в обслуживании).** В этом случае компьютеру мешают выполнять сетевые службы или принуждают к неправильной работе каким-либо другим способом, не используя какого-либо специального доступа. Такие атаки трудно предотвратить, но на них проще отреагировать.
- **Вредоносные программы.** Пользователи Linux в основном защищены от таких вредоносных программ, как почтовые черви и вирусы, просто потому, что почтовые клиенты не настолько глупы, чтобы запускать программы, которые они обнаруживают в приложенных файлах. Однако вредоносные программы для Linux все же существуют. Избегайте загрузки и установки двоичного ПО из незнакомых онлайн-ресурсов.

## 10.7.1. Типичные уязвимости

Есть два основных типа уязвимостей, о которых следует обеспокоиться: прямые атаки и перехват пароля в виде простого текста. Прямые атаки пытаются захватить компьютер не особо изящными способами. Чаще всего используется ошибка переполнения буфера, которая вызвана тем, что небрежный программист не проверил границы буферного массива. Атакующий создает стековый фрейм внутри большого фрагмента данных, скидывает его на удаленный сервер и надеется на то, что сервер перезапишет свои командные инструкции и в конечном итоге исполнит новый стековый фрейм. Несмотря на сложность такой атаки, ее легко повторить многократно.

В атаках второго типа перехватываются пароли, передающиеся по сети как простой текст. Как только взломщик добудет ваш пароль, игра окончена. С этого момента противник будет неуклонно стремиться получить локальный доступ с корневыми правами (что гораздо проще, чем выполнение удаленной атаки), использовать компьютер в качестве посредника для атак на другие хосты или для обоих вариантов.

ПРИМЕЧАНИЕ

Если у вас есть служба, не обладающая встроенной поддержкой шифрования, попробуйте утилиту Stunnel (http://www.stunnel.org/) — пакет шифрующей обертки, похожий на обертку TCP. Подобно команде tcpd, утилита Stunnel особенно хорошо действует для служб inetd.

Некоторые службы постоянно являются целями атак вследствие своей плохой реализации и разработки. Всегда следует отключать следующие службы (в большинстве систем они довольно редко активизируются).

- `ftpd` — по какой-то причине все FTP-серверы переполнены уязвимостями. Кроме того, большинство FTP-серверов использует пароли в виде простого текста.

Если вам необходимо передавать файлы с одного компьютера на другой, попробуйте применить решение на основе SSH или сервер `rsync`.

- `telnetd`, `rlogind`, `rexecd` — все эти службы передают данные удаленного сеанса (в том числе и пароли) в виде простого текста. Избегайте их, если у вас не установлена версия с применением технологии шифрования Kerberos.
- `fingerd` — взломщики могут получить списки пользователей и другую информацию с помощью службы сканера отпечатков пальцев.

## 10.7.2. Онлайн-ресурсы, посвященные безопасности

Вот несколько хороших сайтов, посвященных вопросам безопасности:

- http://www.sans.org/ — предлагает тренинги, сервисы, бесплатное еженедельное новостное письмо с перечислением важнейших уязвимостей, примеры политик безопасности и многое другое;
- http://www.cert.org/ — здесь можно узнать о самых серьезных проблемах;
- http://www.insecure.org/ — на этом сайте можно получить утилиту Nmap и другие ссылки на различные инструменты проверки сети на устойчивость к взломам. Это сайт намного более открыт и конкретен по сравнению с другими.

Если вас заинтересовала сетевая безопасность, следует узнать все о протоколе TLS (Transport Layer Security, защита (безопасности) транспортного уровня) и о его предшественнике — протоколе SSL (Secure Socket Layer, уровень защищенных сокетов). Сетевые уровни пространства пользователя обычно добавляются к сетевым клиентам и серверам для поддержки трансакций с помощью шифрования с открытым ключом и сертификатов. Хорошим руководством послужит книга Дейви (Davie) *Implementing SSL/TLS Using Cryptography and PKI* («Реализация протоколов SSL/TLS с применением криптографии и инфраструктуры открытых ключей», Wiley, 2011).

# 10.8. Заглядывая вперед

Если вам интересно попрактиковаться с какими-либо сложными сетевыми серверами, используйте веб-сервер Apache и почтовый сервер Postfix. В частности, сервер Apache легко установить, и большинство версий ОС содержит пакет поддержки. Если ваш компьютер расположен за брандмауэром или маршрутизатором с функцией NAT, можете экспериментировать с конфигурацией этого сервера сколько пожелаете, не беспокоясь о безопасности.

В последних главах книги мы плавно переходим из пространства ядра в пространство пользователя. Лишь немногие из утилит, рассмотренных в этой главе (например, `tcpdump`), взаимодействуют с ядром. В оставшейся части этой главы описано, каким образом сокеты заполняют разрыв между транспортным уровнем ядра и прикладным уровнем пространства пользователя. Этот материал более сложен и представляет особый интерес для программистов, поэтому можете спокойно его пропустить и, если желаете, перейти к следующей главе.

# 10.9. Сокеты: как процессы взаимодействуют с сетью

Сейчас мы «включим другую передачу» и посмотрим, как процессы выполняют работу по чтению данных из сети и записи данных в сеть. Достаточно просто выполняется чтение/запись для сетевых соединений, уже настроенных: для этого потребуются некоторые системные вызовы, о которых вы можете прочитать на страницах руководства recv(2) и send(2). С точки зрения процесса, возможно, самое важное — узнать, как ссылаться на сеть при использовании таких системных вызовов. В системах Unix процесс задействует *сокет*, чтобы идентифицировать, когда и как он «общается» с сетью. Сокеты являются интерфейсом, который процессы применяют для доступа к сети через ядро, они представляют собой границу между пространством пользователя и пространством ядра. Сокеты также часто используются для межпроцессного взаимодействия (IPC, Interprocess Communication).

Существуют разные типы сокетов, так как процессам необходимо получать доступ к сети по-разному. Например, TCP-соединения представлены сокетами потоков (SOCK_STREAM, с точки зрения программиста), а UDP-соединения — сокетами дейтаграмм (SOCK_DGRAM).

Настройка сетевого сокета может оказаться довольно сложной, так как вам необходимо учесть тип сокета, IP-адреса, порты и в некоторых случаях транспортный протокол. Однако, когда все начальные подробности приведены в порядок, серверы используют стандартные методы для работы с входящим сетевым трафиком.

Схема на рис. 10.1 показывает, сколько серверов обслуживают соединения от входящих сокетов потоков. Обратите внимание на то, что этот тип серверов затрагивает два типа сокетов: сокет прослушивания и сокет для чтения и записи. Основной процесс использует сокет прослушивания для поиска подключений от сети. Когда возникает новое подключение, основной процесс применяет системный вызов accept(), чтобы принять подключение, создав при этом сокет чтения/записи, предназначенный для этого соединения. После этого основной процесс использует команду fork(), чтобы создать новый дочерний процесс для работы с новым соединением. Наконец, исходный сокет остается в роли прослушивателя и продолжает поиск новых подключений от имени основного процесса.

Когда процесс настроит сокет определенного типа, он может взаимодействовать с ним подходящим для этого сокета способом. Именно это делает сокеты гибкими: если вам необходимо изменить лежащий в основе транспортный уровень, вам не потребуется переписывать все части кода, которые отвечают за отправку и получение данных; вам понадобится лишь изменить код инициализации.

Если вы программист и хотели бы изучить, как использовать интерфейс сокетов, то классическим руководством является 3-е издание книги У. Ричарда Стефенса (W. Richard Stephens), Билла Феннера (Bill Fenner) и Эндрю М. Рудоффа (Andrew M. Rudoff) *Unix Network Programming, Volume 1* («Программирование для сетей Unix. Том 1», Addison-Wesley Professional, 2003). Во втором томе рассмотрено также межпроцессное взаимодействие.

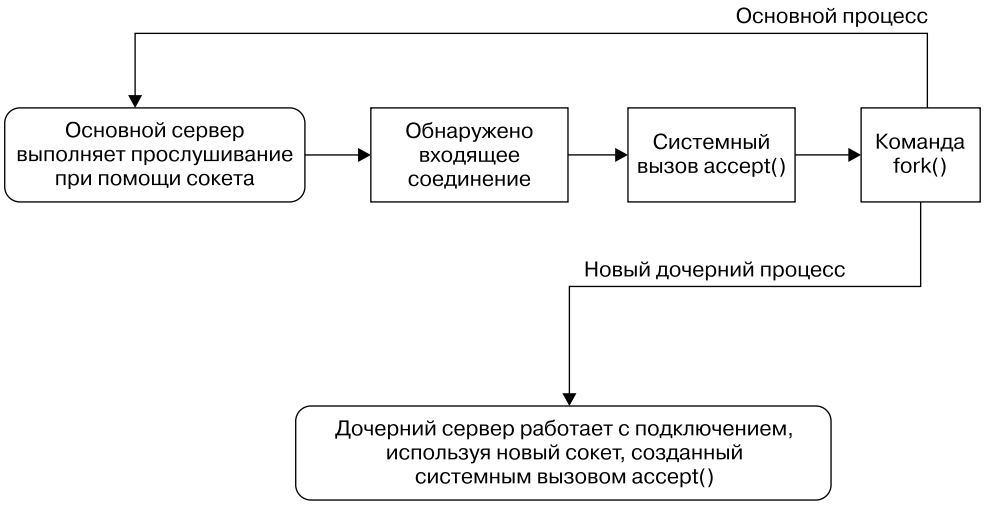


**Рис. 10.1.** Один из методов принятия и обработки входящих соединений

# 10.10. Сокеты домена Unix

Приложения, которые используют сетевые средства, не обязаны задействовать два отдельных хоста. Многие приложения разработаны как механизмы «клиент — сервер» или «узел — узел», в которых процессы, запущенные на одном и том же компьютере, используют межпроцессное взаимодействие (IPC), чтобы выяснить, какую работу необходимо выполнить и кто этим займется. Вспомните, например, о том, что демоны вроде systemd и NetworkManager используют шину D-Bus для отслеживания системных событий и реагирования на них.

Процессы могут использовать для взаимодействия обычную IP-связь через локальный хост сети (127.0.0.1), но вместо этого, как правило, применяется специальный тип сокета, о котором мы вкратце упомянули в главе 3, — *сокет домена Unix*. Когда процесс подключается к сокету домена Unix, он ведет себя почти так же, как и сетевой сокет: может прослушивать и принимать подключения к сокету, и вы можете даже выбирать между различными типами сокетов, чтобы он работал как сокет TCP или UDP.

ПРИМЕЧАНИЕ

Важно помнить о том, что сокет домена Unix не является сетевым сокетом и за ним не расположено никакой сети. Для его использования даже нет необходимости настраивать сеть. К тому же сокеты домена Unix не должны быть привязаны к файлам сокетов. Процесс может создать безымянный сокет домена Unix и сообщить его адрес другому процессу.

## 10.10.1. Преимущества для разработчиков

Разработчикам нравятся сокеты домена Unix для межпроцессного взаимодействия. На это есть две причины. Во-первых, эти сокеты дают возможность использовать специальные файлы сокетов в файловой системе для контроля доступа,

чтобы любой процесс, у которого нет доступа к файлу сокета, не смог применить сокет. Поскольку здесь отсутствует взаимодействие с сетью, такой вариант проще и дает меньше поводов для стандартных вторжений в сеть. Например, в каталоге `/var/run/dbus` обычно можно обнаружить такой файл сокета для шины D-Bus:

```
$ ls -l /var/run/dbus/system_bus_socket
srwxrwxrwx 1 root root 0 Nov 9 08:52 /var/run/dbus/system_bus_socket
```

Во-вторых, поскольку ядру Linux при работе с сокетами домена Unix не приходится проходить через множество уровней сетевой подсистемы, производительность при этом улучшается.

Написание программного кода для сокетов домена Unix не сильно отличается от поддержки обычных сетевых сокетов. Поскольку преимущества от этого могут быть существенными, некоторые сетевые серверы осуществляют взаимодействие как по сети, так и через сокеты домена Unix. Например, сервер `mysqld` базы данных MySQL способен принимать клиентские соединения от удаленных хостов, и он же обычно предлагает сокет домена Unix в файле `/var/run/mysqld/mysqld.sock`.

## 10.10.2. Просмотр списка сокетов домена Unix

Можно просмотреть список сокетов домена Unix, которые в данный момент использованы в системе, с помощью команды `lsof -U`:

```
# lsof -U
COMMAND     PID    USER  FD TYPE     DEVICE  SIZE/OFF     NODE     NAME
mysqld    19701   mysql 12u unix  0xe4defcc0  0t0      35201227     /var/run/mysqld/
                                                                     mysqld.sock
chromium- 26534   juser  5u unix  0xeeac9b00  0t0      42445141     socket
tlsmgr    30480 postfix  5u unix  0xc3384240  0t0      17009106     socket
tlsmgr    30480 postfix  6u unix  0xe20161c0  0t0         10965     private/tlsmgr
--snip--
```

Этот перечень будет довольно длинным, так как многие современные приложения широко используют безымянные сокеты. Их можно узнать по идентификатору `socket` в столбце `NAME` такого отчета.

# 11 Введение в сценарии оболочки

Если вы можете вводить команды в оболочке, то это означает, что вы можете создавать сценарии оболочки (известные также как сценарии оболочки Bourne shell). *Сценарий оболочки* — это набор команд, записанных в файл. Оболочка считывает эти команды из файла так, словно вы вводите их в терминале.

## 11.1. Основы сценариев оболочки

Сценарии оболочки Bourne shell обычно начинаются с приведенной ниже строки, которая указывает на то, что инструкции в файле сценария должна выполнять команда `/bin/sh`. Убедитесь в том, что в начале файла сценария нет пробелов.

```
#!/bin/sh
```

Фрагмент `#!` (в англоязычных источниках он называется *shebang*) будет часто встречаться в других сценариях этой книги. Можно перечислить любые команды, исполнение которых вы желаете поручить оболочке, указав их после строки `#!/bin/sh`. Например, так:

```
#!/bin/sh
#
# Print something, then run ls

echo About to run the ls command.
ls
```

ПРИМЕЧАНИЕ

Символ # в начале строки указывает на то, что данная строка является комментарием, то есть оболочка проигнорирует все, что расположено в строке после этого символа. Используйте комментарии, чтобы объяснить трудные для понимания части ваших сценариев.

После создания сценария оболочки и настройки прав доступа можно запустить его, поместив файл сценария в один из каталогов вашего командного пути, а затем набрав имя сценария в командной строке. Можно также запустить команду *./script*, если этот сценарий расположен в вашем текущем рабочем каталоге, или же использовать полный путь.

Как и для других команд Unix, необходимо установить исполняемый бит для файла сценария оболочки, но при этом следует также установить бит чтения, чтобы оболочка могла читать этот файл. Проще всего это выполнить следующим образом:

```
$ chmod +rx script
```

Команда `chmod` позволяет другим пользователям читать и исполнять сценарий. Если вы желаете запретить это, используйте абсолютный режим файла 700 (и обратитесь заодно к разделу 2.17, чтобы освежить свои знания о правах доступа).

Преодолев основы, рассмотрим некоторые ограничения сценариев оболочки.

**Ограничения сценариев оболочки.** Оболочка Bourne shell сравнительно легко обращается с командами и файлами. Из раздела 2.14 вы узнали способ, с помощью которого оболочка может перенаправлять вывод, это один из важнейших элементов программирования сценариев оболочки. Однако сценарии оболочки — это лишь один из инструментов программирования в Unix, и хотя сценарии обладают некоторой мощью, у них также есть и ограничения.

Одной из сильнейших сторон сценариев оболочки является возможность упрощения и автоматизации задач, которые в противном случае вам пришлось бы выполнять из строки приглашения оболочки (например, групповая работа с файлами). Однако если вы анализируете строки, выполняете повторяющиеся арифметические вычисления, осуществляете доступ к сложным базам данных или же вам необходимы функции и управляющие структуры, лучше использовать язык сценариев типа Python, Perl или awk, или даже более сложный язык, вроде C. Это важно, и мы постоянно будем напоминать об этом в данной главе.

Наконец, следите за размером файлов ваших сценариев оболочки. Старайтесь делать сценарии короткими. Сценарии оболочки Bourne shell не должны быть огромными (хотя вам обязательно повстречаются некоторые монстры).

## 11.2. Кавычки и литералы

Одним из самых запутанных моментов при работе с оболочкой и сценариями является использование *кавычек* и других знаков пунктуации, а также причины, по которым необходимо их применять. Допустим, вы желаете напечатать строку `$100` и для этого набираете следующее:

```
$ echo $100
00
```

Почему в результате появляется строка `00`? Потому что оболочка увидела фрагмент `$1`, который является переменной оболочки (об этом вскоре пойдет речь). Вы могли бы решить, что, если поместить текст в кавычки, оболочка не заметит фрагмент `$1`. Но это также не срабатывает:

```
$ echo "$100"
00
```

Тогда вы спрашиваете об этом у приятеля, который отвечает, что необходимо вместо двойных кавычек использовать одинарные:

```
$ echo '$100'
$100
```

Почему же сработало это волшебное слово?

## 11.2.1. Литералы

Часто, когда вы используете кавычки, вы пытаетесь создать *литерал* — строку, которую оболочка должна в неизмененном виде передать в командную строку. Помимо символа $ (его вы видели в примере), сходные обстоятельства возникают при передаче символа * такой команде, как grep, когда вам необходимо, чтобы оболочка не развертывала его, а также тогда, когда вы желаете использовать в какой-либо команде точку с запятой (;).

При написании сценариев и работе в командной строке помните о том, что происходит, когда оболочка запускает команду.

1. Перед запуском команды оболочка выполняет поиск переменных, шаблонов и других подстановок, а затем выполняет подстановки, если они есть.
2. Оболочка передает команде результаты подстановок.

Проблемы, вызванные литералами, могут быть неуловимыми. Допустим, вы ищете все записи в файле /etc/passwd, соответствующие регулярному выражению r.*t (то есть такие строки, которые содержат символ r и чуть далее символ t; это могло бы позволить вам отыскать такие имена пользователей, как root, ruth и robot). Можно запустить такую команду:

```
$ grep r.*t /etc/passwd
```

В большинстве случаев она будет срабатывать, но иногда по непонятной причине давать сбой. Почему? Ответ заключен, вероятно, в вашем корневом каталоге. Если этот каталог содержит файлы с такими именами, как r.input и r.output, то тогда оболочка развернет выражение r.*t в r.input r.output и создаст такую команду:

```
$ grep r.input r.output /etc/passwd
```

Ключом к обходу подобных проблем служит, во-первых, распознавание символов, которые могут вызвать неприятности, а затем — применение правильного типа кавычек, чтобы защитить символы.

## 11.2.2. Одинарные кавычки

Простейший способ создать литерал и сделать так, чтобы оболочка его не трогала, — поместить всю строку в одинарные кавычки, как в следующем примере с командой grep и символом *:

```
$ grep 'r.*t' /etc/passwd
```

Поскольку дело касается оболочки, все символы между двумя одинарными кавычками, включая пробелы, образуют единый параметр. Следовательно, приво-

димая ниже команда не будет работать, поскольку она просит команду grep выполнить поиск строки r.*t /etc/passwd в стандартном вводе (так как у команды grep здесь лишь один параметр):

```
$ grep 'r.*t /etc/passwd'
```

Когда вам необходимо использовать литерал, в первую очередь следует обратиться к одинарным кавычкам, так как при этом вы будете уверены в том, что оболочка не станет пытаться выполнить какие-либо подстановки. В результате синтаксис будет довольно ясным. Тем не менее иногда может потребоваться дополнительная гибкость, и тогда вам пригодятся двойные кавычки.

## 11.2.3. Двойные кавычки

Двойные кавычки (") действуют подобно одинарным, за исключением того, что оболочка развертывает все переменные, которые появляются внутри двойных кавычек. Можно увидеть это отличие, если запустить следующую команду, а затем заменить в ней двойные кавычки одинарными и выполнить команду повторно:

```
$ echo "There is no * in my path: $PATH"
```

При запуске этой команды обратите внимание на то, что оболочка выполняет подстановку для переменной $PATH, но не заменяет символ *.

ПРИМЕЧАНИЕ

Если вы используете двойные кавычки при выводе больших объемов текста, попробуйте использовать синтаксис heredoc, как описано в разделе 11.9.

## 11.2.4. Передача одинарной кавычки в литерале

Хитрый момент при использовании литералов в оболочке Bourne shell возникает тогда, когда необходимо передать команде одинарную кавычку как литерал. Один из способов это выполнить — поместить символ обратной косой черты перед знаком одинарной кавычки:

```
$ echo I don\'t like contractions inside shell scripts.
```

Обратная косая черта и кавычка *должны* располагаться вне любой другой пары одинарных кавычек, поэтому строка наподобие 'don\'t вызовет синтаксическую ошибку. Как ни странно, но можно помещать одинарную кавычку внутри пары двойных кавычек, как показано в следующем примере (результат работы этой команды такой же, как и в предыдущем примере):

```
$ echo "I don't like contractions inside shell scripts."
```

Если вы в затруднении и вам необходимо общее правило для помещения всей строки в кавычки без подстановок, воспользуйтесь такой процедурой.

1. Замените все экземпляры ' (одинарная кавычка) на '\'' (одинарная кавычка, обратная косая черта, одинарная кавычка, одинарная кавычка).
2. Заключите всю строку в одинарные кавычки.

Следовательно, такую неуклюжую строку, как `this isn't a forward slash: \`, можно поместить в кавычки следующим образом:

```
$ echo 'this isn'\''t a forward slash: \'
```

ПРИМЕЧАНИЕ

Стоит еще раз упомянуть о том, что при помещении строки в кавычки оболочка расценивает все, что находится внутри них, как единый параметр. Следовательно, символы a b c представляют три параметра, а символы a "b c" — только два.

# 11.3. Специальные переменные

Большинство сценариев оболочки понимает параметры командной строки и взаимодействует с запускаемыми командами. Чтобы перевести ваши сценарии с уровня простого перечня команд на уровень более гибких приложений для оболочки, вам необходимо знать о том, как использовать специальные переменные оболочки Bourne shell. Эти специальные переменные подобны любым другим переменным оболочки, как рассказано в разделе 2.8, за исключением того, что значения некоторых из них нельзя изменить.

ПРИМЕЧАНИЕ

После прочтения следующих разделов вы поймете, почему в сценариях оболочки многие специальные символы присутствуют в том виде, как они написаны. Если вы пытаетесь разобраться в какомлибо сценарии оболочки и вам встречается строка, которая выглядит совершенно необъяснимо, рассмотрите ее, разбив на фрагменты.

## 11.3.1. Индивидуальные аргументы: $1, $2...

Переменные `$1`, `$2`, а также все переменные, названные с помощью положительных ненулевых целых чисел, содержат значения параметров сценария или аргументы. Допустим, например, что файл следующего сценария называется `pshow`:

```
#!/bin/sh
echo First argument: $1
echo Third argument: $3
```

Попробуйте запустить этот сценарий, как показано ниже, чтобы увидеть выводимые им аргументы:

```
$ ./pshow one two three
First argument: one
Third argument: three
```

Встроенная в оболочку команда `shift` может быть использована с переменными-аргументами, чтобы удалить первый аргумент (`$1`) и сдвинуть все оставшиеся. Конкретнее, аргумент `$2` превратится в `$1`, `$3` — в `$2` и т. д. Предположим, что файл следующего сценария называется `shiftex`:

```
#!/bin/sh
echo Argument: $1
```

```
shift
echo Argument: $1
shift
echo Argument: $1
```

Запустите его следующим образом, чтобы понять, как он работает:

```
$ ./shiftex one two three
Argument: one
Argument: two
Argument: three
```

Как видите, сценарий `shiftex` выводит все три аргумента: начинает с первого, сдвигает оставшиеся и повторяет вывод.

## 11.3.2. Количество аргументов: $#

Переменная `$#` хранит количество аргументов, переданных в сценарий, и особенно важна при циклическом запуске команды `shift` для выбора аргументов. Если значение `$#` равно `0`, аргументов не остается, поэтому переменная `$1` пустая (см. раздел 11.6, который содержит описание циклических структур).

## 11.3.3. Все аргументы: $@

Переменная $@ представляет все аргументы сценария и весьма полезна для передачи их команде внутри сценария. Например, команды Ghostscript (`gs`) обычно длинные и сложные. Допустим вам необходимо создать шаблон команды для растрирования файла PostScript с разрешением 150 dpi, используя стандартный поток вывода, но оставив при этом также возможность для передачи других параметров в команду `gs`. Для этих целей можно было бы написать сценарий, подобный приведенному ниже:

```
#!/bin/sh
gs -q -dBATCH -dNOPAUSE -dSAFER -sOutputFile=- -sDEVICE=pnmraw $@
```

ПРИМЕЧАНИЕ

Если какая-либо строка в сценарии оболочки становится слишком длинной для текстового редактора, можете разбить ее с помощью символа \. Например, предыдущий сценарий можно записать таким образом:

```
#!/bin/sh
gs -q -dBATCH -dNOPAUSE -dSAFER \
  -sOutputFile=- -sDEVICE=pnmraw $@
```

## 11.3.4. Имя сценария: $0

Переменная `$0` хранит имя сценария, и она полезна при создании диагностических сообщений. Допустим, ваш сценарий должен сообщить о неправильном аргументе, который хранится в переменной `$BADPARM`. Можно вывести диагностическое

сообщение с помощью такой строки, при этом в сообщении об ошибке будет указано имя сценария:

```
echo $0: bad option $BADPARM
```

Все диагностические сообщения об ошибках должны следовать в стандартную ошибку. Вспомните из подраздела 2.14.1 о том, что синтаксис `2>&1` перенаправляет стандартную ошибку в стандартный вывод. Для записи в стандартную ошибку можно обратить этот процесс с помощью синтаксиса `1>&2`. Чтобы использовать его в предыдущем примере, примените такую строку:

```
echo $0: bad option $BADPARM 1>&2
```

### 11.3.5. Идентификатор процесса: $$

Переменная `$$` хранит идентификатор процесса оболочки.

### 11.3.6. Код выхода: $?

Переменная `$?` хранит код выхода последней команды, которую выполнила оболочка. Коды выхода, играющие важную роль в освоении сценариев оболочки, рассмотрены далее.

## 11.4. Коды выхода

Когда команда Unix завершает работу, она оставляет для родительского процесса, который запустил эту команду, *код выхода*. Код выхода является числом, иногда его называют *кодом ошибки* или *значением выхода*. Когда код выхода равен нулю, это обычно означает, что команда отработала без ошибок. Если же в команде произошла ошибка, то она обычно завершает работу с числом, отличным от 0 (но не всегда, как вы увидите далее).

Оболочка хранит код выхода последней команды в специальной переменной `$?`, поэтому его можно узнать из командной строки:

```
$ ls / > /dev/null
$ echo $?
0
$ ls /asdfasdf > /dev/null
ls: /asdfasdf: No such file or directory
$ echo $?
1
```

Вы видите, что успешно завершившая работу команда вернула значение 0, а команда с ошибкой вернула значение 1 (при условии того, что в вашей системе нет каталога `/asdfasdf`).

Если вы намерены использовать код выхода команды, вы *должны* применить или сохранить его сразу же по окончании работы этой команды. Если, например, вы запустите команду `echo $?` два раза подряд, то результатом второй команды всегда будет 0, так как первая команда `echo` завершилась успешно.

При написании кода для аварийного завершения работы сценария используйте что-нибудь типа `exit 1`, чтобы передать код выхода 1 родительскому процессу, который запустил этот сценарий. Можно применять разные числа для различных условий.

Следует отметить, что некоторые команды, подобные `diff` и `grep`, используют ненулевые коды выхода, чтобы сообщить о нормальных условиях. Например, команда `grep` возвращает значение `0`, если она находит что-либо, совпадающее с шаблоном, и `1`, если не находит. Для таких команд код выхода 1 не свидетельствует об ошибке; для настоящих проблем команды `grep` и `diff` применяют код выхода 2. Если вы подозреваете, что какая-либо команда использует ненулевой код выхода, чтобы сообщить об успешном завершении, прочитайте страницу руководства по этой команде. Коды выхода обычно разъясняются в разделах EXIT VALUE (Код выхода) или DIAGNOSTICS (Диагностика).

# 11.5. Условные операторы

В оболочке Bourne shell есть специальные конструкции для условных операторов, таких как `if/then/else` и `case`. Например, следующий простой сценарий с условным оператором `if` проверяет, является ли строка `hi` значением первого аргумента сценария:

```
#!/bin/sh
if [ $1 = hi ]; then
   echo 'The first argument was "hi"'
else
   echo -n 'The first argument was not "hi" -- '
   echo It was '"'$1'"'
fi
```

Слова `if`, `then`, `else` и `fi` в этом сценарии являются ключевыми словами оболочки; все остальное — команды. Такое разграничение чрезвычайно важно, так как в одной из команд, `[ $1 = "hi" ]`, используется символ `[`, который представляет реальную команду Unix, а *не* специальный синтаксис оболочки. На самом деле это не вполне верно, но пока рассматривайте ее как отдельную команду. Во всех системах Unix есть команда `[`, которая выполняет проверку условных операторов сценария оболочки. Эта команда известна также как `test`. При тщательном исследовании команд `[` и `test` выясняется, что они совместно используют один и тот же дескриптор inode, то есть одна из них является символической ссылкой на другую.

Понимание кодов выхода, описанных в разделе 11.4, существенно важно, поскольку весь процесс протекает следующим образом.

1. Оболочка запускает команду, которая расположена за ключевым словом `if`, и получает код выхода этой команды.
2. Если код выхода равен 0, оболочка выполняет команды, которые следуют после ключевого слова `then`, и останавливается, когда доходит до ключевого слова `else` или `fi`.

3. Если код выхода не равен 0 и существует условие `else`, оболочка выполняет команды, расположенные после ключевого слова `else`.
4. Условный оператор заканчивается ключевым словом `fi`.

## 11.5.1. Немного о пустом списке параметров

Есть небольшая проблема в условном операторе из предыдущего примера, вызванная весьма распространенной ошибкой: переменная `$1` может оказаться пустой, так как пользователь может не ввести параметр. При отсутствии параметра команда тестирования прочитает `[ = hi ]`, и тогда команда `[` прервет исполнение, выдав ошибку. Этого можно избежать, если заключить параметр в кавычки одним из приведенных ниже способов (оба являются общепринятыми):

```
if [ "$1" = hi ]; then
if [ x"$1" = x"hi" ]; then
```

## 11.5.2. Использование других команд для проверки условий

Все, что следует после ключевого слова `if`, является командой. Следовательно, если вы желаете поместить в этой же строке ключевое слово `then`, необходимо применить точку с запятой (`;`) после команды проверки. Если вы опустите точку с запятой, оболочка передаст слово `then` команде проверки в качестве параметра. Если вам не нравится использование точки с запятой, можно поместить ключевое слово `then` в отдельной строке.

Существует множество возможностей применять другие команды вместо `[`. Вот пример, в котором использована команда `grep`:

```
#!/bin/sh
if grep -q daemon /etc/passwd; then
    echo The daemon user is in the passwd file.
else
    echo There is a big problem. daemon is not in the passwd file.
fi
```

## 11.5.3. Ключевое слово elif

Ключевое слово `elif` позволяет вам связать вместе условные операторы `if`, как показано ниже. Однако не слишком увлекайтесь использованием `elif`, так как конструкция `case`, о которой вы узнаете в подразделе 11.5.6, часто оказывается более подходящей.

```
#!/bin/sh
if [ "$1" = "hi" ]; then
   echo 'The first argument was "hi"'
elif [ "$2" = "bye" ]; then
   echo 'The second argument was "bye"'
else
```

```
    echo -n 'The first argument was not "hi" and the second was not "bye"-- '
    echo They were '"'$1'"' and '"'$2'"'
fi
```

## 11.5.4. Логические конструкции && и ||

Существуют две простые однострочные конструкции, которые могут вам время от времени встречаться: `&&` («и») и `||` («или»). Конструкция `&&` устроена следующим образом:

```
command1 && command2
```

Здесь оболочка запускает команду *command1*, и если ее код выхода равен 0, оболочка запускает также и команду *command2*. Конструкция `||` подобна первой: если команда, расположенная перед символами `||`, возвращает ненулевой код выхода, оболочка запускает вторую команду.

Конструкции `&&` и `||` часто находят себе применение в проверках `if`, и в обоих случаях код выхода последней запущенной команды определяет то, как оболочка обработает условный оператор. В случае с конструкцией `&&`, если первая команда завершается неудачно, оболочка использует ее код выхода для инструкции `if`, но если первая команда завершена успешно, оболочка применяет для условного оператора код выхода второй команды. В случае с конструкцией `||` оболочка использует код выхода первой команды при ее успешном выполнении или код выхода второй команды, если первая завершила работу неудачно.

Например:

```
#!/bin/sh
if [ "$1" = hi ] || [ "$1" = bye ]; then
    echo 'The first argument was "'$1'"'
fi
```

Если ваши условные операторы содержат команду проверки (`[`), как показано здесь, можно использовать символы `-a` и `-o` вместо конструкций `&&` и `||`, о чем рассказано в следующем разделе.

## 11.5.5. Проверка условий

Вы увидели, как работает команда `[`:, код выхода равен 0, если условие проверки истинно, и не равен 0, если проверка завершена неудачно. Вы знаете также, как проверить равенство строк с помощью команды `[ str1 = str2 ]`. Помните о том, что сценарии оболочки хорошо приспособлены для операций с целыми файлами, поскольку наиболее полезные проверки с помощью команды `[` затрагивают свойства файлов. Например, следующая строка проверяет, является ли файл *file* обычным файлом (а не каталогом или специальным файлом):

```
[ -f file ]
```

В сценариях можно увидеть проверку `-f`, помещенную в цикл, подобный приведенному ниже. Такой цикл проверяет все элементы, находящиеся в текущем рабочем каталоге (вскоре вы узнаете более подробно о циклах):

```
for filename in *; do
    if [ -f $filename ]; then
        ls -l $filename
        file $filename
    else
       echo $filename is not a regular file.
    fi
done
```

Можно выполнить инверсию условия, поместив оператор ! перед аргументами команды проверки. Например, условие `[ ! -f file ]` возвращает значение `true`, если файл *file* не является обычным файлом. Кроме того, флаги `-a` и `-o` являются логическими операторами «и» и «или» (например, `[ -f file1 -a file2 ]`).

ПРИМЕЧАНИЕ

Поскольку команда test так широко применяется в сценариях, во многих версиях оболочки Bourne shell (включая версию bash) эта команда является встроенной. Это может ускорить выполнение сценариев, так как оболочке не приходится для каждой проверки запускать отдельную команду.

Существуют десятки операторов проверки, и все они попадают в одну из трех основных категорий: проверка файлов, проверка строк и арифметическая проверка. Интерактивное руководство info содержит всю необходимую документацию, однако страница руководства test(1) позволит быстрее навести справки. В следующих разделах приведены общие сведения об основных видах проверок.

## Проверка файлов

Большинство проверок файлов, вроде `-f`, называется *унарными* операциями, поскольку им необходим только один аргумент: файл, который следует проверить. Вот две важные проверки файлов:

- `-e` — возвращает значение `true`, если файл существует;
- `-s` — возвращает значение `true`, если файл непустой.

Многие операции отслеживают тип файла, это значит, что они способны определить, является ли что-либо обычным файлом, каталогом или специальным устройством, как перечислено в табл. 11.1. Есть также несколько унарных операций, которые проверяют права доступа к файлу, как указано в табл. 11.2 (см. также обзор прав доступа в разделе 2.17).

**Таблица 11.1.** Операторы проверки типа файла

| Оператор | Условие проверки |
|---|---|
| -f | Обычный файл |
| -d | Каталог |
| -h | Символическая ссылка |
| -b | Блочное устройство |
| -c | Символьное устройство |

| Оператор | Условие проверки |
|---|---|
| -p | Именованный канал |
| -s | Сокет |

ПРИМЕЧАНИЕ

Команда test отслеживает символические ссылки (кроме варианта -h), то есть если ссылка link является символической ссылкой на обычный файл, проверка [ -f link ] возвратит код выхода 0 (true).

**Таблица 11.2.** Операторы проверки прав доступа к файлу

| Оператор | Оператор |
|---|---|
| -r | Для чтения |
| -w | Для записи |
| -x | Исполняемый |
| -u | Setuid |
| -g | Setgid |
| -k | «Закрепленный» |

Наконец, три *бинарных* оператора (это проверки, которым необходимы два файла в качестве аргументов) используются при проверке файлов, но такие проверки не слишком распространены. Посмотрите на такую команду, которая содержит оператор `-nt` («более поздний, чем»):

```
[ file1 -nt file2 ]
```

Результатом будет значение `true`, если у файла *file1* дата изменения более поздняя по сравнению с файлом *file2*. Оператор `-ot` («более ранний, чем») выполняет противоположную проверку. Если же вам необходимо установить идентичность жестких ссылок, оператор `-ef` позволяет сравнить два файла и выдать результат `true`, если такие файлы совместно используют одинаковые номера дескрипторов inode и устройства.

## Проверка строк

Вы уже видели бинарный строковый оператор `=`, который возвращает значение `true`, если его операнды равны. Оператор `!=` возвращает значение `true`, если его операнды не равны. Вот еще два унарных строковых оператора:

- `-z` — возвращает значение `true`, если его аргумент пустой (условие `[ -z "" ]` возвратит значение `0`);
- `-n` — возвращает значение `true`, если его аргумент непустой (условие `[ -n "" ]` возвратит значение `1`).

## Арифметическая проверка

Важно осознавать, что знак равенства (`=`) проверяет равенство *строк*, а *не чисел*. Следовательно, проверка `[ 1 = 1 ]` вернет результат `0` (`true`), однако проверка `[ 01 = 1 ]` возвратит `false`. При работе с числами используйте оператор `-eq` вместо знака равенства: проверка `[ 01 -eq 1 ]` вернет значение `true`. В табл. 11.3 приведен полный список операторов численного сравнения.

**Таблица 11.3.** Арифметические операции сравнения

| Оператор | Возвращает значение true, если первый аргумент ... |
|---|---|
| -eq | ...равен второму |
| -ne | ...не равен второму |
| -lt | ...меньше второго |
| -gt | ...больше второго |
| -le | ...меньше второго или равен ему |
| -ge | ...больше второго или равен ему |

## 11.5.6. Сопоставление строк с помощью конструкции case

Ключевое слово `case` формирует еще одну условную конструкцию, которая чрезвычайно полезна при сопоставлении строк. Условный оператор `case` не выполняет никаких команд проверки и, следовательно, не выдает никаких кодов выхода. Тем не менее он может проверять соответствие шаблону. Приводимый ниже пример должен пояснить основную часть сказанного:

```
#!/bin/sh
case $1 in
    bye)
        echo Fine, bye.
        ;;
    hi|hello)
        echo Nice to see you.
        ;;
    what*)
        echo Whatever.
        ;;
    *)
        echo 'Huh?'
        ;;
esac
```

Оболочка выполняет это следующим образом.

1. Сценарий сопоставляет значение переменной `$1` с каждым из вариантов, который отделен с помощью символа `)`.
2. Если какое-либо из значений совпадает со значением переменной `$1`, оболочка выполняет команды, расположенные под этим вариантом, пока не встретит символы `;;`, после которых она пропускает все остальное до ключевого слова `esac`.
3. Условный оператор завершается словом `esac`.

Для каждого варианта значений можно сопоставить единственную строку (подобно строке `bye` в приведенном примере) или же несколько строк, использовав оператор | (условие `hi|hello` возвращает значение `true`, если значение переменной `$1` равно `hi` или `hello`). Можно также применять шаблоны * или ? (`what*`).

Чтобы определить условие по умолчанию, которое охватывает все возможные значения, отличающиеся от указанных, применяйте единственный символ *, как показано в последнем условии приведенного примера.

ПРИМЕЧАНИЕ

Каждое условие должно завершаться двойной точкой с запятой (;;), чтобы не возникло синтаксической ошибки.

# 11.6. Циклы

В оболочке Bourne shell существуют два типа циклов: цикл `for` и цикл `while`.

## 11.6.1. Цикл for

Цикл `for` (который является циклом «для каждого») самый распространенный. Вот пример:

```
#!/bin/sh
for str in one two three four; do
    echo $str
done
```

В этом листинге слова `for`, `in`, `do` и `done` — ключевые слова оболочки. Оболочка выполняет следующее.

1. Присваивает переменной `str` первое (`one`) из четырех значений, следующих после слова `in` и разделенных символами пробела.
2. Запускает команду `echo`, расположенную между словами `do` и `done`.
3. Возвращается к строке `for`, присваивает переменной `str` следующее значение (`two`), выполняет команды между словами `do` и `done`, а затем повторяет процесс до тех пор, пока не закончатся значения, следующие после ключевого слова `in`.

Результат работы этого сценария выглядит так:

```
one
two
three
four
```

## 11.6.2. Цикл while

Цикл `while` в оболочке Bourne shell использует коды выхода, подобно условному оператору `if`. Например, такой сценарий выполняет десять итераций:

```
#!/bin/sh
FILE=/tmp/whiletest.$$;
echo firstline > $FILE
while tail -10 $FILE | grep -q firstline; do
    # add lines to $FILE until tail -10 $FILE no longer prints "firstline"
```

```
    echo -n Number of lines in $FILE:' '
    wc -l $FILE | awk '{print $1}'
    echo newline >> $FILE
done

rm -f $FILE
```

Здесь проверяется код выхода команды `grep -q firstline`. Как только код выхода становится ненулевым (в данном случае, когда строка `firstline` не будет появляться в десяти последних строках файла `$FILE`), цикл завершается.

Можно выйти из цикла `while` с помощью инструкции `break`. В оболочке Bourne shell есть также цикл `until`, который действует подобно циклу `while`, за исключением того, что он завершается, когда встречает нулевой код выхода, а не код, не равный 0. Однако не следует использовать слишком часто циклы `while` и `until`. В действительности, если вам необходимо применить цикл `while`, возможно, лучше воспользоваться языком awk или Python.

# 11.7. Подстановка команд

Оболочка Bourne shell может перенаправлять стандартный вывод какой-либо команды обратно в командную строку оболочки. То есть можно использовать вывод команды в качестве аргумента для другой команды или же сохранить вывод команды в переменной оболочки, поместив команду внутрь выражения `$()`.

Следующий пример сохраняет команду внутри переменной `FLAGS`. Жирным шрифтом во второй строке выделена подстановка команды.

```
#!/bin/sh
FLAGS=$(grep ^flags /proc/cpuinfo | sed 's/.*://' | head -1)
echo Your processor supports:
for f in $FLAGS; do
    case $f in
        fpu)   MSG="floating point unit"
               ;;
        3dnow) MSG="3DNOW graphics extensions"
               ;;
        mtrr)  MSG="memory type range register"
               ;;
        *)     MSG="unknown"
               ;;
    esac
    echo $f: $MSG
done
```

Данный пример достаточно сложен, поскольку он показывает возможность использования как одинарных кавычек, так и каналов внутри подстановки. Результат команды `grep` отправляется в команду `sed` (подробности см. в подразделе 11.10.3), которая удаляет все, что соответствует выражению `.*:`, а затем результат команды `sed` передается команде `head`.

Используя подстановку команд, очень легко сделать лишнее. Например, не применяйте в сценариях выражение `$(ls)`, поскольку оболочка выполняет развертывание символа * быстрее. Кроме того, если вы желаете применить команду к нескольким именам файлов, которые вы получаете в результате работы команды `find`, попробуйте использовать канал для команды `xargs`, а не подстановку или же параметр `-exec` (см. подраздел 11.10.4).

ПРИМЕЧАНИЕ

Традиционный синтаксис для подстановки команды: размещение команды внутри «обратных чертоточек» (``). Такой вариант вы встретите во многих сценариях. Синтаксис $() является новой формой, которая следует стандарту POSIX, легче записывается и читается.

# 11.8. Управление временным файлом

Иногда бывает необходимо создать временный файл, чтобы собрать вывод для его использования в следующей команде. При создании такого файла убедитесь в том, что имя этого файла достаточно индивидуально, чтобы другие команды случайно не выполнили запись в него.

Приведу пример того, как использовать команду `mktemp` для создания имен временных файлов. Данный сценарий показывает аппаратные прерывания, которые возникли за последние две секунды.

```
#!/bin/sh
TMPFILE1=$(mktemp /tmp/im1.XXXXXX)
TMPFILE2=$(mktemp /tmp/im2.XXXXXX)

cat /proc/interrupts > $TMPFILE1
sleep 2
cat /proc/interrupts > $TMPFILE2
diff $TMPFILE1 $TMPFILE2
rm -f $TMPFILE1 $TMPFILE2
```

Аргумент команды `mktemp` является шаблоном. Команда `mktemp` превращает `XXXXXX` в уникальный набор символов и создает пустой файл с таким именем. Обратите внимание на то, что этот сценарий использует имена переменных для хранения имен файлов, поэтому, чтобы изменить имя файла, необходимо изменить всего одну строку.

ПРИМЕЧАНИЕ

Не все варианты Unux содержат команду mktemp. Если у вас возникнут проблемы с переносимостью, лучше установить GNU-пакет coreutils для вашей операционной системы.

Еще одна проблема сценариев, которые используют временные файлы, такова: если выполнение сценария будет прервано, временные файлы могут остаться в системе. В предыдущем примере, если нажать сочетание клавиш Ctrl+C до начала второй команды, то в каталоге `/tmp` останется временный файл. Избегайте этого по возможности. Старайтесь использовать команду `trap` для создания обработчика сигнала, который будет перехватывать сигнал от нажатия клавиш Ctrl+C и удалять временные файлы, как в этом примере:

```
#!/bin/sh
TMPFILE1=$(mktemp /tmp/im1.XXXXXX)
TMPFILE2=$(mktemp /tmp/im2.XXXXXX)
trap "rm -f $TMPFILE1 $TMPFILE2; exit 1" INT
--snip--
```

Вы должны использовать команду `exit` в таком обработчике, чтобы явным образом завершить выполнение сценария, а иначе оболочка продолжит его обычную работу после выполнения обработчика сигнала.

ПРИМЕЧАНИЕ

Не обязательно передавать аргументы команде mktemp. Если их нет, то шаблон будет начинаться с префикса /tmp/tmp.

# 11.9. Синтаксис heredoc

Допустим, вам необходимо вывести большой фрагмент текста или передать его другой команде. Чтобы не использовать несколько команд `echo`, можно применить *синтаксис* `heredoc`, как показано в следующем сценарии:

```
#!/bin/sh
DATE=$(date)
cat <<EOF
Date: $DATE

The output above is from the Unix date command.
It's not a very interesting command.
EOF
```

Элементы, выделенные жирным шрифтом, управляют синтаксисом `heredoc`. Маркер `<<EOF` дает оболочке указание перенаправить все строки, которые последуют дальше, в стандартный ввод команды, предшествующей маркеру `<<EOF` (в данном случае это команда `cat`). Перенаправление прекращается, как только в какой-либо строке появляется единственный маркер `EOF`. На самом деле этот маркер может быть любым, просто помните о том, что в начале и в конце фрагмента с синтаксисом `heredoc` следует указать одинаковые маркеры. По принятому соглашению название маркера должно быть набрано прописными буквами.

Обратите внимание на переменную `$DATE` в приведенном примере. Оболочка раскрывает переменные оболочки внутри документов с синтаксисом `heredoc`, это особенно полезно, когда вы выводите отчеты, которые содержат много переменных.

# 11.10. Основные утилиты в сценариях оболочки

Некоторые команды чрезвычайно полезно применять в сценариях оболочки. Такие утилиты, как `basename`, пригодны лишь при использовании с другими командами

и, следовательно, нечасто встречаются за пределами сценариев. Тем не менее другие команды, например awk, могут быть полезными и в командной строке.

## 11.10.1. Команда basename

Если вам необходимо удалить расширение из имени файла или изъять названия каталогов из полного пути, воспользуйтесь командой basename. Попробуйте ввести в командную строку следующие примеры, чтобы понять, как работает эта команда:

```
$ basename example.html .html
$ basename /usr/local/bin/example
```

В обоих случаях команда basename возвращает результат example. Первая команда удаляет суффикс .html из имени файла example.html, а вторая — названия каталогов из полного пути.

Следующий пример демонстрирует, как применить команду basename в сценарии, который конвертирует файлы изображений из формата GIF в формат PNG:

```
#!/bin/sh
for file in *.gif; do
    # exit if there are no files
    if [ ! -f $file ]; then
        exit
    fi
    b=$(basename $file .gif)
    echo Converting $b.gif to $b.png...
    giftopnm $b.gif | pnmtopng > $b.png
done
```

## 11.10.2. Команда awk

Команда awk не является простой командой с единственным способом применения; на самом деле это мощный язык программирования. К сожалению, искусство применения языка awk сейчас практически утрачено, поскольку его заменили более развитые языки, такие как Python.

Языку awk посвящены целые книги, например *The AWK Programming Language* («Язык программирования AWK») Альфреда В. Эйхо (Alfred V. Aho), Брайана Кернигана (Brian W. Kernighan) и Питера Дж. Вайнбергера (Peter J. Weinberger) (Addison-Wesley, 1988). Очень многие пользователи используют команду awk с единственной целью: чтобы выбрать отдельное поле из потока ввода, как здесь:

```
$ ls -l | awk '{print $5}'
```

Эта команда выводит пятое поле из отчета команды ls (размер файла). В результате получится список, содержащий размеры файлов.

## 11.10.3. Команда sed

Команда sed (сокращение от *stream editor* — «редактор потока») является автоматическим текстовым редактором, который принимает входящий поток (файл или

стандартный ввод), изменяет его в соответствии с некоторым выражением и выводит результат в стандартный вывод. Во многих отношениях команда sed подобна команде ed, первичному текстовому редактору Unix. Она обладает множеством операций, инструментами подстановки и возможностями работы с адресацией. Как и для команды awk, есть книги и о команде sed, среди которых краткий справочник по обеим командам: *sed & awk Pocket Reference* («Карманный справочник по командам sed и awk») Арнольда Роббинса (Arnold Robbins), 2-е издание (O'Reilly, 2002).

Хотя команда sed является довольно большой и ее детальное рассмотрение выходит за рамки этой книги, легко понять, как она устроена. В общих чертах, команда sed воспринимает адрес и операцию как один аргумент. Адрес является набором строк, и команда решает, что делать с этими строками.

Очень распространенная задача для команды sed: заменить какое-либо регулярное выражение текстом (см. подраздел 2.5.1), например, так:

```
$ sed 's/exp/text/'
```

Так, если вы желаете заменить первое двоеточие в файле /etc/passwd на символ %, а затем отправить результат в стандартный вывод, следует выполнить следующую команду:

```
$ sed 's/:/%/' /etc/passwd
```

Чтобы заменить *все* двоеточия в файле /etc/passwd, добавьте спецификатор g в конце операции, как здесь:

```
$ sed 's/:/%/g' /etc/passwd
```

Приведу команду, которая работает построчно; она считывает файл /etc/passwd и удаляет строки с третьей по шестую, а затем отправляет результат в стандартный вывод:

```
$ sed 3,6d /etc/passwd
```

В этом примере число 3,6 является адресом (диапазоном строк), а флаг d — операцией (удаление). Если адрес опустить, команда sed будет работать со всеми строками входного потока. Двумя самыми распространенными операциями команды sed являются s (найти и заменить) и d.

В качестве адреса можно также использовать регулярное выражение. Эта команда удаляет любую строку, которая соответствует регулярному выражению *exp*:

```
$ sed '/exp/d'
```

## 11.10.4. Команда xargs

Когда вам приходится запускать одну команду для большого количества файлов, эта команда или оболочка может ответить, что она не способна вместить все аргументы в свой буфер. Чтобы справиться с этой проблемой, используйте команду xargs, запуская ее в стандартном потоке ввода для каждого имени файла.

Многие применяют команду xargs вместе с командой find. Например, следующий сценарий может помочь проверить, что в текущем каталоге каждый файл

с расширением .gif действительно является изображением в формате GIF (Graphic Interchange Format, формат обмена графическими данными):

```
$ find . -name '*.gif' -print | xargs file
```

В приведенном примере команда xargs запускает команду file. Однако такой вызов может привести к ошибкам или подвергнуть вашу систему рискам, связанным с безопасностью, поскольку имена файлов могут содержать пробелы и символы перевода строки. При написании сценариев используйте приводимую ниже форму, которая изменяет выходной разделитель команды find и разделитель аргументов команды xargs — вместо символа перевода строки применяется символ NULL:

```
$ find . -name '*.gif' -print0 | xargs -0 file
```

Команда xargs запускает множество процессов, поэтому не ожидайте высокой производительности, если вы работаете с большим количеством файлов.

Может потребоваться добавить два дефиса (--) в конце команды xargs, если есть вероятность того, что название какого-либо целевого файла начинается с дефиса (-). Двойной дефис (--) можно применять, чтобы сообщить какой-либо команде, что аргументы, которые за ним следуют, являются именами файлов, а не параметрами. При этом помните, что не все команды поддерживают использование двойного дефиса.

Есть альтернатива команде xargs при применении команды find: параметр -exec. Однако ее синтаксис довольно мудреный, так как вам необходимо использовать символы {} для подстановки имени файла и литерал ;, чтобы указать окончание команды. Вот как выполняется предыдущая задача с помощью одной лишь команды find:

```
$ find . -name '*.gif' -exec file {} \;
```

## 11.10.5. Команда expr

Если вам необходимо использовать арифметические операторы в сценариях оболочки, может прийти на помощь команда expr (которая выполняет даже некоторые операции со строками). Например, команда expr 1 + 2 выводит результат 3. Запустите команду expr -help, чтобы получить полный перечень операций.

Применение команды expr — это неуклюжий и медленный способ выполнения математических вычислений. Если вам часто приходится заниматься ими, то, вероятно, лучше использовать что-либо вроде языка Python вместо сценария оболочки.

## 11.10.6. Команда exec

Команда exec является встроенной в оболочку функцией, которая заменяет текущий процесс оболочки той командой, которую вы укажете после команды exec. Она осуществляет системный вызов exec(), о котором вы узнали из главы 1. Эта функция предназначена для сохранения системных ресурсов, однако помните о ее необратимости: если запустить команду exec в сценарии оболочки, то этот сценарий и сама оболочка прекратят работу, уступив место новой команде.

Чтобы проверить это в окне оболочки, попробуйте запустить команду `exec cat`. После нажатия сочетания клавиш Ctrl+D или Ctrl+C для завершения команды `cat` окно оболочки должно исчезнуть, поскольку его дочерний процесс больше не существует.

# 11.11. Подоболочки

Предположим, вам необходимо немного изменить среду в оболочке, но это изменение не должно стать постоянным. Можно изменить, а потом вернуть в исходное состояние часть среды (например, путь или рабочий каталог), используя переменные оболочки, однако такой способ довольно груб. Простой вариант выполнения подобных задач заключается в применении *подоболочки*, совершенно нового процесса оболочки, который можно создать только для того, чтобы выполнить одну-две команды. Новая оболочка обладает копией среды исходной оболочки, и когда новая оболочка существует, любые изменения, которые будут сделаны в ее среде, не отразятся на обычной работе исходной оболочки.

Чтобы использовать подоболочку, поместите в скобки те команды, которые она должна выполнить. Например, следующая строка выполняет команду `uglyprogram` в каталоге `uglydir`, оставляя исходную оболочку без изменений:

```
$ (cd uglydir; uglyprogram)
```

Следующий пример показывает, как добавить компонент пути, который может вызвать проблемы, если сделать это изменение постоянным:

```
$ (PATH=/usr/confusing:$PATH; uglyprogram)
```

Использование подоболочки для выполнения одноразового изменения среды является настолько распространенным, что существует даже встроенный синтаксис, который не прибегает к подоболочке:

```
$ PATH=/usr/confusing:$PATH uglyprogram
```

Каналы и фоновые процессы также работают в подоболочках. Следующий пример использует команду `tar` для архивирования всего дерева каталогов внутри каталога `orig`, а затем распаковывает этот архив в новый каталог `target`, дублируя тем самым файлы и папки каталога `orig` (это оправданно, поскольку при этом сохраняются сведения о владельцах и правах доступа, и это обычно выполняется быстрее, чем команда типа `cp -r`):

```
$ tar cf - orig | (cd target; tar xvf -)
```

ВНИМАНИЕ

Тщательно проверяйте подобные команды перед их запуском, чтобы убедиться в том, что каталог target существует и он абсолютно отличается от каталога orig.

# 11.12. Включение других файлов в сценарии

Если вам необходимо включить другой файл в сценарий оболочки, используйте оператор «точка» (`.`). Например, такая строка выполняет команды из файла `config.sh`:

```
. config.sh
```

Подобный синтаксис «включения» не запускает подоболочку и может быть полезен для группы сценариев, которым необходимо использовать единственный файл конфигурации.

## 11.13. Чтение пользовательского ввода

Команда `read` считывает строку текста из стандартного ввода и сохраняет ее текст в переменной. Например, следующая команда сохраняет ввод в переменной `$var`:

```
$ read var
```

Эта встроенная в оболочку команда может оказаться полезной в сочетании с другими функциями оболочки, не упомянутыми в данной книге.

## 11.14. Когда (не) использовать сценарии оболочки

Оболочка настолько богата функциями, что трудно рассказать обо всех ее важнейших элементах в одной главе. Если вы заинтересовались, на что еще способна оболочка, загляните в одну из книг, посвященных программированию в оболочке (например, *Unix Shell Programming* («Программирование в оболочке Unix») Стефена Дж. Коучена (Stephen G. Kochan) (3-е издание, SAMS Publishing, 2003)) или рассказывающих о сценариях оболочки (*The UNIX Programming Environment* («Среда программирования Unix») Брэна У. Кернигана (Bran W. Kernighan) и Роба Пайка (Rob Pike) (Prentice Hall, 1984)).

Но, несмотря на это, в некоторый момент (особенно если вы начинаете пользоваться встроенной командой `read`), вы должны задать себе вопрос: применяете ли вы по-прежнему верный инструмент для работы? Вспомните, с чем лучше всего справляются сценарии оболочки: это работа с простыми файлами и командами. Как уже было сказано выше, если вам приходится писать что-либо замысловатое, а в особенности задействовать сложные строковые или арифметические операции, вероятно, лучше обратиться к таким языкам сценариев, как Python, Perl или awk.

# 12 Передача файлов по сети

В этой главе рассмотрены возможности передачи и совместного использования файлов компьютерами одной сети. Мы начнем со способов копирования, которые отличаются от утилит `scp` и `sftp`, уже известных вам. Затем мы кратко рассмотрим подлинное совместное применение файлов, при котором какой-либо каталог компьютера включается в число каталогов другого компьютера.

Эта глава описывает некоторые альтернативные способы передачи файлов, поскольку не все задачи переноса файлов одинаковы. Иногда необходимо обеспечить быстрый временный доступ к компьютерам, о которых вы знаете совсем немного, или, например, эффективно поддерживать копии больших структур каталогов. Временами требуется более продолжительный доступ.

## 12.1. Быстрое копирование

Допустим, вы желаете скопировать файл (или файлы) с вашего компьютера на другой компьютер вашей сети и при этом вам не обязательно возвращать его обратно или предпринимать какие-либо особые действия. Вы просто хотите выполнить это быстро. Существует удобный способ осуществить желаемое с помощью модуля Python. Просто перейдите в каталог с необходимыми файлами и запустите такую команду:

```
$ python -m SimpleHTTPServer
```

При этом запускается базовый веб-сервер, который делает данный каталог доступным для любого браузера в сети. Обычно он работает с портом 8000, поэтому, если адрес компьютера, на котором вы запустили эту команду, равен 10.1.2.4, наберите адрес `http://10.1.2.4:8000` на компьютере назначения, и у вас появится возможность забрать необходимые файлы.

## 12.2. Команда rsync

Если вы желаете переместить всю структуру каталогов, это можно выполнить с помощью команды `scp -r` или, если необходима дополнительная функциональность, с помощью «конвейерной» команды `tar`:

```
$ tar cBvf - directory | ssh remote_host tar xBvpf -
```

Эти методы делают свое дело, но они не слишком гибкие. В частности, по окончании передачи на удаленном хосте может оказаться не вполне точная копия каталога. Если на удаленном компьютере такой *каталог* уже существует и содержит какие-либо дополнительные файлы, такие файлы останутся после передачи.

Если вам необходимо выполнять подобные задачи регулярно (в особенности если вы планируете автоматизировать этот процесс), используйте выделенную систему синхронизации. В Linux команда `rsync` является стандартным синхронизатором, который обладает хорошей производительностью и многими полезными функциями для выполнения передачи. Мы рассмотрим несколько основных режимов работы команды `rsync`, а также отметим их особенности.

## 12.2.1. Основы команды rsync

Чтобы команда `rsync` работала между двумя хостами, она должна быть установлена как на хосте-источнике, так и на хосте-назначении и у вас должен быть способ доступа к одному компьютеру с другого компьютера. Проще всего передавать файлы с использованием удаленной учетной записи оболочки, и мы будем предполагать, что вам необходимо передавать файлы с помощью SSH-доступа. Тем не менее помните также о том, что команда `rsync` может принести пользу даже при копировании файлов и каталогов в пределах одного компьютера, например из одной файловой системы в другую.

На первый взгляд команда `rsync` незначительно отличается от команды `scp`. На самом деле команду `rsync` можно запускать с теми же аргументами. Например, чтобы скопировать группу файлов в домашний каталог вашего хоста, введите такую команду:

```
$ rsync file1 file2 ... host:
```

В любой современной системе команда `rsync` подразумевает, что вы используете протокол SSH для соединения с удаленным хостом.

Опасайтесь такого сообщения об ошибке:

```
rsync not found
rsync: connection unexpectedly closed (0 bytes read so far)
rsync error: error in rsync protocol data stream (code 12) at io.c(165)
```

Оно говорит о том, что удаленная оболочка не может отыскать команду `rsync` в своей системе. Если команды `rsync` нет в удаленном пути, но в системе она есть, используйте команду `--rsync-path=path`, чтобы вручную указать ее местоположение.

Если на удаленном хосте у вас другое имя пользователя, добавьте к имени хоста параметр *user@*, в котором *user* является вашим именем пользователя для этого хоста:

```
$ rsync file1 file2 ... user@host:
```

Если вы не укажете дополнительные параметры, команда `rsync` скопирует только файлы. В действительности, если вы укажете только те параметры, которые

были описаны к настоящему моменту, и попытаетесь указать параметр *dir* как аргумент, вы получите такое сообщение:

```
skipping directory dir
```

Чтобы передать всю иерархию каталога — со всеми символическими ссылками, правами доступа, режимами файлов и устройствами, — используйте параметр `-a`. Более того, если вы желаете выполнить копирование в какое-либо место, не являющееся вашим домашним каталогом на удаленном хосте, укажите это назначение после названия удаленного хоста:

```
$ rsync -a dir host:destination_dir
```

Копирование каталогов может оказаться довольно хитрым занятием, поэтому, если вы не вполне уверены в том, что в итоге произойдет после переноса файлов, используйте комбинацию параметров `-nv`. Параметр `-n` дает команде `rsync` указание о работе в режиме «пробного прогона», то есть выполнить проверку, не копируя в действительности никаких файлов. Параметр `-v` означает подробный режим, при котором отображаются детали передачи всех вовлеченных файлов:

```
$ rsync -nva dir host:destination_dir
```

Отчет будет выглядеть так:

```
building file list ... done
ml/nftrans/nftrans.html
[more files]
wrote 2183 bytes read 24 bytes 401.27 bytes/sec
```

## 12.2.2. Создание точной копии структуры каталога

По умолчанию команда `rsync` копирует файлы и каталоги, не принимая во внимание уже имеющееся содержимое каталога назначения. Например, если вы передали каталог `d`, который содержит файлы `a` и `b`, компьютеру, на котором уже есть файл `d/c`, то после выполнения команды `rsync` этот компьютер будет содержать файлы `d/a`, `d/b` и `d/c`.

Чтобы создать точную копию каталога источника, необходимо удалить те файлы в каталоге назначения, которых нет в исходном каталоге (вроде файла `d/c` в нашем примере). Чтобы выполнить это, используйте параметр `--delete`:

```
$ rsync -a --delete dir host:destination_dir
```

ВНИМАНИЕ

Это может оказаться опасным, так как обычно следует проверить каталог назначения на наличие файлов, которые вы можете ненароком удалить. Помните: если вы не уверены в результатах переноса, используйте параметр -n, чтобы выполнить пробный прогон и точно узнать о том, когда команде rsync потребуется удалить файлы.

## 12.2.3. Использование завершающей косой черты

Будьте особо внимательны при указании кого-либо каталога в качестве источника для команды rsync. Рассмотрим основную команду, с которой мы до сих пор работали:

```
$ rsync -a dir host:dest_dir
```

По ее выполнении у вас на хосте будет каталог dir внутри каталога dest_dir. На рис. 12.1 приведен пример того, как команда rsync обычно обрабатывает каталог с файлами a и b. Однако при добавлении косой черты (/) ее поведение радикально меняется:

```
$ rsync -a dir/ host:dest_dir
```

В этом случае команда rsync копирует все, что находится внутри каталога dir, в каталог dest_dir хоста, не создавая в действительности каталог dir на хосте назначения. Следовательно, можно рассматривать передачу каталога dir/ как операцию, подобную операции cp dir/* dest_dir в локальной файловой системе.

Допустим, что у вас есть каталог dir, который содержит файлы a и b (dir/a и dir/b). Вы запускаете версию команды с косой чертой, чтобы передать их в каталог dest_dir на хосте:

```
$ rsync -a dir/ host:dest_dir
```

По окончании передачи каталог dest_dir будет содержать копии файлов a и b, но *не* каталога dir. Если же вы опустите конечную косую черту после dir, в каталоге dest_dir окажется копия каталога dir с файлами a и b внутри него. Итак, в результате передачи на удаленном хосте появились бы файлы и каталоги с именами dest_dir/dir/a и dest_dir/dir/b. На рис. 12.2 показано, как команда rsync обрабатывает структуру каталога, приведенную на рис. 12.1, если используется завершающая косая черта.

При передаче файлов и каталогов на удаленный хост случайное добавление символа / в конце пути вызвало бы лишь некоторое неудобство; вам пришлось бы перейти на удаленный *хост*, добавить каталог dir и поместить в него все переданные файлы. К сожалению, следует быть осторожными, чтобы избежать катастрофы, если соединить завершающую косую черту с параметром --delete, поскольку так можно легко удалить не относящиеся к делу файлы.

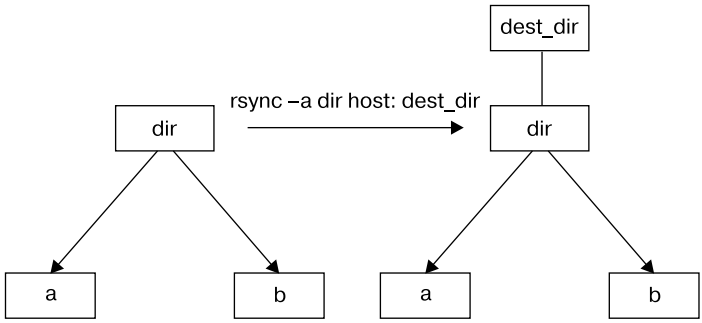


**Рис. 12.1.** Нормальное копирование с помощью команды rsync

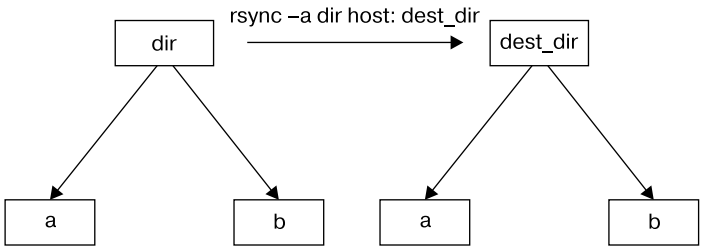


**Рис. 12.2.** Результат использования завершающего слеша в команде rsync

ПРИМЕЧАНИЕ

Будьте осторожны при использовании в оболочке функции автоматического завершения имени файла. Библиотека readline графического интерфейса пользователя и многие другие функции завершения преобразуют завершающие слеши в полные названия каталогов.

## 12.2.4. Исключение файлов и каталогов

Одной из важнейших функций команды `rsync` является возможность исключения файлов и каталогов из операции переноса. Допустим, что вы желаете переместить локальный каталог `src` на хост *host*, исключив при этом все, что называется `.git`. Это можно выполнить так:

```
$ rsync -a --exclude=.git src host:
```

Обратите внимание на то, что эта команда исключает все файлы и каталоги с именем `.git`, поскольку параметр `--exclude` учитывает шаблон, а не абсолютное имя файла. Чтобы исключить конкретный элемент, укажите абсолютный путь, который начинается с символа `/`, как показано здесь:

```
$ rsync -a --exclude=/src/.git src host:
```

ПРИМЕЧАНИЕ

Первый символ / в записи /src/.git данной команды является не корневым каталогом системы, а базовым каталогом для переноса.

Приведу еще несколько советов по использованию шаблонов исключения.

- Можно использовать столько параметров `--exclude`, сколько необходимо.
- Если вы применяете одни и те же шаблоны регулярно, поместите их в обычный текстовый файл (по одному шаблону в строке) и используйте параметр `--exclude-from=`*file*.
- Чтобы исключить каталоги с именем *item*, но включить расположенные в них файлы, применяйте завершающий слеш: `--exclude=`*item*`/`.
- Шаблон исключения основан на полном имени файла или каталога и может содержать простые джокерные символы. Например, шаблону `t*s` соответствует имя `this`, но не имя `ethers`.
- Если вы исключили каталог или файл, а потом обнаружили, что ваш шаблон является слишком строгим, используйте параметр `--include`, чтобы специальным образом включить еще один файл или каталог.

## 12.2.5. Целостность переноса, меры предосторожности и подробные режимы

Для ускорения работы команда `rsync` выполняет быструю проверку того, нет ли в пункте назначения каких-либо файлов источника. Эта проверка использует комбинацию из размера файла и даты его последнего изменения. Когда вы в первый раз передаете полную иерархию каталогов на удаленный хост, команда `rsync` видит, что в пункте назначения нет ни одного из передаваемых файлов, поэтому она переносит все. Убедиться в этом позволит вам команда `rsync -n`.

После первого выполнения команды `rsync` запустите ее еще раз в таком варианте: `rsync -v`. Теперь вы должны увидеть, что список файлов для переноса пуст, поскольку данный набор файлов присутствует на обоих компьютерах и с теми же датами изменения.

Когда файлы на стороне источника не идентичны файлам на стороне назначения, команда `rsync` переносит файлы источника и перезаписывает все существующие файлы на удаленной стороне. Однако такое поведение может оказаться неадекватным, поскольку необходимо дополнительное подтверждение того, что эти файлы действительно такие же, прежде чем пропускать их при переносе.

Вам могут также понадобиться некоторые дополнительные меры предосторожности. Вот несколько параметров, которые бывают удобны.

- `--checksum` (сокращенно `-c`) — вычисляет контрольные суммы (обычно это уникальные сигнатуры) файлов, чтобы убедиться в их одинаковости. Для этого во время переноса потребуются дополнительные ресурсы ввода/вывода и процессора, но, если вы работаете с важными данными или ваши файлы часто обладают одинаковым размером, этот параметр обязателен.
- `--ignore-existing` — не затирает файлы, которые уже присутствуют на целевой стороне.
- `--backup` (сокращенно `-b`) — не затирает файлы, которые уже есть на целевой стороне, а переименовывает их, добавляя к именам суффикс ~ до начала передачи новых файлов.
- `--suffix=s` — изменяет суффикс, использованный в параметре `--backup`, с ~ на `s`.
- `--update` (сокращенно `-u`) — не затирает на целевой стороне никаких файлов, которые обладают более поздней датой изменения по сравнению с соответствующими файлами источника.

Если не указывать специальные параметры, команда `rsync` работает, не привлекая внимания, и выводит сообщения только в случае возникновения проблем. Тем не менее можно использовать команду `rsync -v` для подробного режима или `rsync -vv`, чтобы выводилось еще больше подробностей. Можно указать столько флагов `v`, сколько пожелаете, но двух, вероятно, будет более чем достаточно.

Чтобы получить всеобъемлющий отчет после переноса файлов, используйте команду `rsync --stats`.

## 12.2.6. Сжатие

Многим пользователям нравится применять параметр `-z` в сочетании с параметром `-a`, чтобы выполнить сжатие данных перед передачей:

```
$ rsync -az dir host:destination_dir
```

Сжатие может улучшить производительность в некоторых ситуациях, например при выгрузке большого объема данных через медленное соединение (такое как поток, противоположный основному трафику, или многие DSL-соединения) или когда время ожидания между двумя хостами велико. Тем не менее в быстрых локальных сетях передача между двумя компьютерами может сдерживаться временем, которое процессор затрачивает на сжатие и распаковку данных, поэтому несжатая передача может оказаться быстрее.

## 12.2.7. Ограничение ширины полосы пропускания

Очень легко «засорить» интернет-подключение, выгружая на удаленный хост большие объемы данных. Даже если вы не станете использовать (обычно большую) скорость скачивания во время такой передачи, ваше соединение все же станет довольно медленным, если вы позволите команде `rsync` работать максимально быстро, поскольку исходящие TCP-пакеты, вроде запросов HTTP, будут соперничать с вашими передачами за ширину полосы.

Чтобы разобраться с этим, применяйте параметр `--bwlimit`, который даст вашей выгрузке больше пространства для маневра. Чтобы, например, ограничить ширину пропускания значением 10 000 Кбит в секунду, следует выполнить такую команду:

```
$ rsync --bwlimit=10000 -a dir host:destination_dir
```

## 12.2.8. Перенос файлов на ваш компьютер

Команда `rsync` предназначена не только для копирования файлов с локального компьютера на удаленный хост. Можно также переносить файлы с удаленного компьютера на локальный хост, указав удаленный хост и путь к удаленному источнику в качестве первого аргумента в командной строке. Следовательно, чтобы перенести каталог *src_dir* удаленного хоста в каталог *dest_dir* локального хоста, запустите следующую команду:

```
$ rsync -a host:src_dir dest_dir
```

ПРИМЕЧАНИЕ

Как отмечалось выше, можно использовать команду rsync для дублирования каталогов на локальном компьютере, если полностью опустить параметр host:.

## 12.2.9. Дальнейшие темы, относящиеся к команде rsync

Когда вам необходимо скопировать большое количество файлов, команда `rsync` должна прийти на ум одной из первых. Запуск этой команды в пакетном режиме

чрезвычайно удобен, вы обнаружите множество параметров, позволяющих использовать вспомогательные файлы, связанные с параметрами команды, отчетами и состоянием передачи. В частности, файлы состояния позволяют ускорить продолжительные передачи и легко возобновить их в случае прерывания.

Команда `rsync` окажется также полезной для создания резервных копий. Можно, например, подключить к Linux интернет-хранилище вроде S3 от компании Amazon, а затем использовать команду `rsync --delete`, чтобы периодически синхронизировать файлы с этим хранилищем, организовав тем самым весьма эффективную систему резервного копирования.

Существует намного больше параметров командной строки, чем упомянуто здесь. Чтобы получить краткий обзор, запустите команду `rsync --help`. Более подробную информацию можно найти на странице руководства rsync(1), а также на домашней странице, посвященной команде `rsync` (http://rsync.samba.org/).

## 12.3. Введение в совместное использование файлов

Ваш компьютер с Linux, вероятно, не один в вашей сети, а когда в сети несколько компьютеров, практически всегда возникают поводы для совместного использования их файлов. Оставшаяся часть главы будет посвящена главным образом совместному использованию файлов на компьютерах с Windows и Mac OS X, поскольку будет интересно увидеть, каким образом Linux приспосабливается к совершенно другому окружению. Для совместного применения файлов на компьютерах с Linux или для доступа к файлам на устройстве сетевого хранения данных (NAS, Network Area Storage) нужно знать об использовании сетевой файловой системы (NFS, Network File System) в качестве клиента.

## 12.4. Совместное использование файлов с помощью пакета Samba

Если у вас есть компьютеры с Windows, то, вероятно, потребуется разрешить доступ к файлам и принтерам системы Linux с таких компьютеров, применяя стандартный сетевой протокол Windows — SMB (Server Message Block, блок сообщений сервера). Совместное использование файлов с помощью протокола SMB поддерживается также в Mac OS X.

Стандартный пакет ПО для совместного использования файлов в Linux называется Samba. Он не только позволяет Windows-компьютерам вашей сети получать доступ к системе Linux, но также работает в обратном направлении: можно получать доступ к файлам на серверах Windows с Linux-компьютера с помощью клиента Samba.

Чтобы настроить сервер Samba, выполните следующие действия.

1. Создайте файл `smb.conf`.
2. Добавьте в него разделы о совместном использовании файлов.

3. Добавьте в него разделы о совместном использовании принтеров.
4. Запустите демоны Samba: `nmbd` и `smbd`.

При установке сервера Samba из пакета дистрибутива ваша система должна выполнить перечисленные выше шаги, применив для сервера некоторые разумные параметры по умолчанию. Тем не менее ей, по-видимому, не удастся определить, какие именно ресурсы вашего Linux-компьютера вы предоставляете клиентам.

ПРИМЕЧАНИЕ

Обсуждение сервера Samba в этой главе краткое и сводится к тому, как добиться того, чтобы Windows-компьютеры внутри одной подсети видели автономный Linux-компьютер с помощью браузера Windows Network Places. Конфигурировать сервер Samba можно бесчисленным количеством способов, поскольку существует множество возможностей контроля доступа и разная топология сети. Чтобы узнать подробности о конфигурировании широкомасштабного сервера, обратитесь к книге Using Samba («Применение Samba», 3-е издание, O'Reilly, 2007), которая является намного более полным руководством, а также посетите сайт проекта Samba по адресу http://www.samba.org/.

## 12.4.1. Конфигурирование сервера Samba

Главным файлом конфигурации Samba является файл `smb.conf`, который в большинстве версий ОС помещается внутри каталога `etc`, например в каталоге `/etc/samba`. Тем не менее вам может потребоваться поискать его, поскольку он может также располагаться внутри каталога `lib`, например в каталоге `/usr/local/samba/lib`.

Формат файла `smb.conf` подобен стилю XDG, который вы уже встречали всюду (например, в конфигурации службы systemd): файл разбит на несколько секций, отмеченных с помощью квадратных скобок (например, так: `[global]` и `[printers]`). Секция `[global]` содержит общие параметры, которые применяются к серверу в целом и ко всем совместно используемым ресурсам. Эти параметры относятся главным образом к сетевой конфигурации и к контролю доступа. Приведенный ниже пример секции `[global]` демонстрирует, как настроить имя сервера, описание и рабочую группу:

```
[global]
# server name
netbios name = name
# server description
server string = My server via Samba
# workgroup
workgroup = MYNETWORK
```

Эти параметры означают следующее:

- `netbios name` — имя сервера. Если опустить этот параметр, сервер Samba использует имя хоста Unix;
- `server string` — краткое описание сервера. По умолчанию это номер версии пакета Samba;
- `workgroup` — имя рабочей группы SMB. Если вы находитесь в домене Windows, укажите для этого параметра имя вашего домена.

## 12.4.2. Контроль доступа к серверу

Можно добавить в файл smb.conf параметры, позволяющие ограничить круг компьютеров и пользователей, которые имеют доступ к серверу Samba. Следующий перечень содержит многие из параметров, которые можно указать в секции [global], а также в секциях, контролирующих отдельные ресурсы (как объяснено далее в этой главе).

- interfaces. Настройте этот параметр, чтобы сервер Samba прослушивал только указанные сети или интерфейсы. Например:

```
interfaces = 10.23.2.0/255.255.255.0
interfaces = eth0
```

- bind interfaces only. Установите значение yes, если используете параметр interfaces, чтобы ограничить доступ к компьютерам только этими интерфейсами.
- valid users. Укажите только тех пользователей, которым разрешается доступ. Например:

```
valid users = jruser, bill
```

- guest ok. Установите для этого параметра значение true, чтобы сделать совместно используемые ресурсы доступными для анонимных пользователей сети.
- guest only. Установите для этого параметра значение true, чтобы разрешить только анонимный доступ.
- browseable. Установите этот параметр, чтобы совместно используемые ресурсы можно было видеть с помощью сетевых браузеров. Если для этого параметра установить значение no для всех совместных ресурсов, то у вас по-прежнему останется возможность доступа к ним на сервере Samba, но для этого вам потребуется точно знать их имена.

## 12.4.3. Пароли

Следует позволять доступ к серверу Samba только с использованием аутентификации с помощью пароля. К сожалению, базовая система паролей в Unix отличается от такой системы в Windows, поэтому, если вы не укажете сетевые пароли в виде простого текста или пароли аутентификации на сервере Windows, вам придется устанавливать альтернативную систему паролей. Данный раздел рассказывает о том, как настроить такую систему с помощью серверного приложения Trivial Database (TDB), которое подходит для небольших сетей.

Для начала используйте такие записи в секции [global] файла smb.conf, чтобы определить характеристики базы данных паролей Samba:

```
# use the tdb for Samba to enable encrypted passwords
security = user
passdb backend = tdbsam
obey pam restrictions = yes
smb passwd file = /etc/samba/passwd_smb
```

Они позволяют работать с этой базой данных с помощью команды smbpasswd. Параметр obey pam restrictions дает гарантию того, что каждый пользователь, который будет менять свой пароль с помощью команды smbpasswd, должен будет соблюсти все правила, предусмотренные модулем PAM (Pluggable Authentication Module, подключаемый модуль аутентификации) при обычной смене пароля. Для параметра passdb backend можно добавить после двоеточия необязательное имя пути к файлу TDB, например так: tdbsam:/etc/samba/private/passwd.tdb.

ПРИМЕЧАНИЕ

Если у вас есть доступ к домену Windows, можно установить параметр security = domain, чтобы сервер Samba использовал доменные имена пользователей и не нуждался в базе данных паролей. Однако, чтобы пользователи домена имели доступ к компьютеру с сервером Samba, у каждого пользователя домена должна быть локальная учетная запись с таким же именем пользователя и на компьютере с сервером Samba.

## Добавление и удаление пользователей

Первое, что вам потребуется для предоставления доступа Windows-пользователю к вашему серверу Samba, — добавить этого пользователя в базу данных паролей с помощью команды smbpasswd -a:

```
# smbpasswd -a username
```

Параметр *username* в команде smbpasswd должен быть действительным именем пользователя в вашей системе Linux.

Подобно обычной системной команде passwd, команда smbpasswd дважды запрашивает у вас ввод нового пароля пользователя. Если пароль проходит все необходимые проверки надежности, команда smbpasswd подтверждает создание нового пользователя.

Чтобы удалить пользователя, используйте параметр -x в команде smbpasswd:

```
# smbpasswd -x username
```

Чтобы временно деактивизировать пользователя, применяйте параметр -d; параметр -e заново активизирует такого пользователя:

```
# smbpasswd -d username
# smbpasswd -e username
```

## Изменение паролей

Можно изменить пароль Samba, если с правами пользователя superuser запустить команду smbpasswd без каких-либо параметров или ключевых слов, кроме имени пользователя:

```
# smbpasswd username
```

Однако, если сервер Samba запущен, любой пользователь может изменить собственный пароль Samba, введя в командной строке команду smbpasswd.

Наконец, следует предупредить вас об одном месте в конфигурации, которого следует остерегаться. Если вы увидите в файле smb.conf строку, подобную приведенной, будьте осторожны:

```
unix password sync = yes
```

Эта строка приводит к тому, что команда smbpasswd изменяет обычный пароль пользователя *вместе с* паролем Samba. Результат может внести путаницу, особенно тогда, когда пользователь меняет свой пароль Samba на что-либо, не являющееся паролем Linux, а потом обнаруживает, что не может войти в систему. В некоторых версиях ОС этот параметр по умолчанию установлен в пакете сервера Samba!

## 12.4.4. Запуск сервера

Вам может потребоваться запустить сервер, если вы не устанавливали сервер Samba из пакета дистрибутива. Чтобы это выполнить, запустите команды nmbd и smbd со следующими аргументами (здесь параметр *smb_config_file* определяет полный путь для вашего файла smb.conf):

```
# nmbd -D -s smb_config_file
# smbd -D -s smb_config_file
```

Демон nmbd является сервером имен NetBIOS, а демон smbd выполняет реальную обработку запросов на совместное использование. Параметр -D определяет режим демона. Если вы измените файл smb.conf во время работы демона smbd, вы сможете уведомить демон об изменениях с помощью сигнала HUP или использовать команду перезапуска службы (такую как systemctl или initctl).

## 12.4.5. Диагностические файлы и журналы

Если что-либо происходит не так при запуске одного из серверов Samba, в командной строке появляется сообщение об ошибке. В то же время диагностические сообщения времени исполнения записываются в файлы журналов log.nmbd и log.smbd, которые обычно находятся в подкаталоге /var/log, таком как /var/log/samba. Здесь вы найдете также другие файлы журналов, например отдельные журналы для каждого клиента.

## 12.4.6. Конфигурирование совместного использования файлов

Чтобы экспортировать каталог клиентам SMB (то есть чтобы совместно с клиентом использовать какой-либо каталог), добавьте в файл smb.conf секцию, подобную приведенной ниже (здесь параметр *label* задает название для совместно используемого ресурса, а параметр path — полный путь к этому каталогу):

```
[label]
path = path
comment = share description
guest ok = no
writable = yes
printable = no
```

При совместном использовании каталогов полезны следующие параметры.

- guest ok. Разрешает гостевой доступ к совместному ресурсу. Синонимом является параметр public.

- writable. Значение yes или true делает ресурс доступным для чтения и записи. Не разрешайте гостевой доступ к совместному ресурсу, позволяющему чтение и запись.
- printable. Определяет выводимый на печать совместный ресурс. Этот параметр должен быть равен no или false для совместно используемого каталога.
- veto files. Предотвращает экспорт всех файлов, которые соответствуют указанным шаблонам. Каждый шаблон следует поместить между слешами (чтобы он выглядел как */pattern/*). Следующий пример запрещает экспорт объектных файлов, а также любых файлов или каталогов с именем bin:

```
veto files = /*.o/bin/
```

## 12.4.7. Домашние каталоги

Можно добавить в файл smb.conf секцию [homes], если вы желаете экспортировать домашние каталоги пользователям. Такая секция должна выглядеть следующим образом:

```
[homes]
comment = home directories
browseable = no
writable = yes
```

По умолчанию сервер Samba считывает из файла /etc/passwd запись, которая относится к вошедшему в систему пользователю, чтобы определить его домашний каталог для секции [homes]. Однако, если вы не хотите, чтобы сервер Samba так поступал (то есть вам необходимо хранить домашние каталоги Windows в месте, которое отличается от обычных домашних каталогов Linux), можно использовать подстановку %S в параметре path. Вот как, например, можно переключить для пользователя каталог [homes] на каталог /u/user:

```
path = /u/%S
```

Сервер Samba подставит текущее имя пользователя вместо %S.

## 12.4.8. Совместное использование принтеров

Можно экспортировать все принтеры клиентам Windows, добавив секцию [printers] в файл smb.conf. Вот как выглядит эта секция, если вы применяете стандартную для Unix систему печати CUPS:

```
[printers]
comment = Printers
browseable = yes
printing = CUPS
path = cups
printable = yes
writable = no
```

Чтобы использовать параметр `printing = CUPS`, ваша версия сервера Samba должна быть настроена на применение библиотеки CUPS и связана с ней.

ПРИМЕЧАНИЕ

В зависимости от конфигурации вам может также потребоваться разрешить гостевой доступ к принтерам с помощью параметра guest ok = yes, вместо того чтобы предоставлять пароль или учетную запись каждому, кому необходим доступ к принтерам. Например, легко ограничить доступ к принтерам в пределах одной подсети, использовав правила брандмауэра.

## 12.4.9. Использование клиента Samba

Клиент Samba — команда `smbclient` — может осуществлять вывод и на совместно используемые ресурсы Windows или пользоваться ими. Эта команда удобна, когда вы оказываетесь в среде, в которой необходимо взаимодействовать с серверами Windows, не предлагающими дружественные к Unix средства коммуникации.

Чтобы начать работу с командой `smbclient`, применяйте параметр `-L` для получения списка совместно используемых ресурсов на удаленном сервере с именем *SERVER*:

```
$ smbclient -L -U username SERVER
```

Не обязательно указывать параметр `-U` *username*, если ваше имя пользователя в Linux совпадает с именем пользователя на сервере *SERVER*.

После запуска команды `smbclient` она запросит у вас пароль. Чтобы попытаться получить гостевой доступ к ресурсу, нажмите клавишу Enter; в противном случае введите ваш пароль для сервера *SERVER*. При успешном подключении вы должны получить перечень совместно используемых ресурсов, подобный приведенному:

```
Sharename   Type       Comment
---------   ----       -------
Software    Disk       Software distribution
Scratch     Disk       Scratch space
IPC$        IPC        IPC Service
ADMIN$      IPC        IPC Service
Printer1    Printer    Printer in room 231A
Printer2    Printer    Printer in basement
```

Поле `Type` поможет вам определить назначение каждого из ресурсов. Обращайте внимание только на ресурсы с типами `Disk` и `Printer` (ресурсы `IPC` предназначены для удаленного управления). В данном списке есть два дисковых ресурса и два принтера. Используйте имя из столбца `Sharename`, чтобы получить доступ к каждому ресурсу.

## 12.4.10. Доступ к файлам в качестве клиента

Если вам необходим лишь нерегулярный доступ к файлам на совместно используемом дисковом ресурсе, используйте следующую команду. Опять-таки можно

опустить параметр *-U username*, если ваше имя пользователя в Linux совпадает с именем пользователя на сервере.

```
$ smbclient -U username '\\SERVER\sharename'
```

При успешном подключении появится приглашение, подобное приводимому ниже, которое говорит о том, что теперь вы можете перемещать файлы:

```
smb: \>
```

В таком режиме передачи файлов команда `smbclient` подобна команде `ftp` Unix, и вы можете запускать такие команды:

- `get file` — копирует файл *file* с удаленного сервера в текущий локальный каталог;
- `put file` — копирует файл *file* с локального компьютера на удаленный сервер;
- `cd dir` — переходит в каталог *dir* на удаленном сервере;
- `lcd localdir` — переходит в каталог *localdir* на локальном компьютере;
- `pwd` — выводит текущий каталог на удаленном сервере, включая имя сервера и названия совместно используемых ресурсов;
- `!command` — запускает команду *command* на локальном хосте. Очень удобны следующие две команды: `!pwd` и `!ls`, которые позволяют определить статус каталога и файла на локальной стороне;
- `help` — показывает полный список команд.

### Использование файловой системы CIFS

Если вам необходим регулярный и частый доступ к файлам на сервере Windows, можно напрямую подключить совместно используемый ресурс с помощью монтирования. Синтаксис команды приведен ниже. Обратите внимание на применение формата *SERVER:sharename* вместо обычного *\\SERVER\sharename*.

```
# mount -t cifs SERVER:sharename mountpoint -o user=username,pass=password
```

Чтобы использовать команду `mount` подобным образом, у вас на сервере Samba должны быть доступны утилиты CIFS (Common Internet File System, общая межсетевая файловая система). В большинстве дистрибутивов они предлагаются в виде отдельного пакета.

## 12.5. Клиенты NFS

Стандартной системой для совместного использования файлов в системах Unix является NFS (Network File System, сетевая файловая система); существует много разных версий NFS для различных вариантов действий. Можно применять систему NFS в протоколах TCP и UDP с большим количеством способов аутентификации и шифрования. Поскольку параметров так много, рассмотрение системы NFS может вылиться в большую тему, поэтому мы затронем лишь основы клиентов NFS.

Чтобы смонтировать удаленный каталог на сервере с помощью системы NFS, используйте такой же базовый синтаксис, как и для монтирования каталога CIFS:

```
# mount -t nfs server:directory mountpoint
```

Технически параметр `-t nfs` не нужен, так как команда `mount` определит его сама, но вам может потребоваться изучение параметров, описанных на странице руководства `nfs(5)`. Вы найдете несколько различных вариантов для настройки безопасности с помощью параметра `sec`. Многие администраторы небольших закрытых сетей используют контроль доступа на основе хоста. Однако более сложные методы, такие как аутентификация на основе технологии Kerberos, требуют дополнительной конфигурации других частей вашей системы.

Когда вы обнаружите, что используете файловые системы в сети довольно интенсивно, настройте автоматическое монтирование, чтобы ваша система монтировала файловые системы лишь тогда, когда вы действительно пытаетесь применять их. Это позволит предотвратить проблемы с зависимостями при загрузке системы. Традиционный инструмент для автоматического монтирования называется `automount`, его новая версия — `amd`, но сейчас оба они вытесняются модулем с типом `automount` в команде `systemd`.

## 12.6. Добавочные параметры и ограничения сетевой файловой системы

Настройка сервера NFS для совместного использования файлов другими Linux-компьютерами является более сложной, чем использование простого клиента NFS. Необходимо запускать серверные демоны (`mountd` и `nfsd`) и настраивать файл `/etc/exports`, чтобы отразить в нем каталоги, которые применяются совместно. Тем не менее мы не будем рассматривать серверы NFS, в основном потому, что создать совместно используемое сетевое хранилище зачастую намного удобнее, если приобрести устройство NAS, которое справится с задачей за вас. Многие из таких устройств основаны на Linux, поэтому в них изначально присутствует поддержка сервера NFS. Производители устройств NAS делают их еще более полезными, предоставляя собственные инструменты администрирования, которые избавляют от таких утомительных задач, как настройка конфигурации RAID или облачного резервного копирования.

Раз речь зашла об облачном резервном копировании, следует сказать о другом варианте сетевой файловой службы — облачном хранилище. Оно может быть удобно, когда вам необходимо дополнительное пространство для автоматических резервных копий и вас не сильно огорчит снижение быстродействия. Такое хранилище особенно полезно, если этот сервис не требуется вам в течение продолжительного времени или слишком часто. Обычно можно смонтировать и интернет-хранилище, во многом подобное системе NFS.

Хотя система NFS и другие варианты совместного использования файлов работают хорошо при нерегулярном применении, не рассчитывайте на отменную

производительность. Доступ к большим файлам в режиме «только чтение» должен работать хорошо, например, при передаче аудио или видео, поскольку данные считываются в виде больших предсказуемых фрагментов, для которых не требуется интенсивное взаимодействие между файловым сервером и его клиентом. Если сеть достаточно быстрая, а у клиента достаточно памяти, сервер может поставлять данные, как требуется.

Локальное хранилище намного быстрее для задач, вовлекающих множество маленьких файлов, например при компиляции пакетов ПО или при запуске среды рабочего стола. Картина усложняется для большой сети, в которой множество пользователей имеют доступ ко многим различным компьютерам, поскольку в этом случае необходим компромисс между удобством, производительностью и простотой администрирования.

# 13 Пользовательское окружение

Основной акцент этой книги сделан на системе Linux, которая обычно лежит в основе серверных процессов и интерактивных сеансов пользователя. В конечном итоге система и пользователь должны где-либо встретиться. Файлы запуска системы играют в этом важную роль, поскольку они задают параметры по умолчанию для оболочки и для других интерактивных команд. Они определяют, как поведет себя система при входе пользователя.

Большинство пользователей не уделяют пристального внимания файлам запуска, затрагивая их только тогда, когда необходимо добавить что-либо в целях удобства, например псевдоним. Со временем эти файлы засоряются лишними переменными окружения и проверками, это может привести к раздражающим ошибкам или (довольно серьезным) проблемам.

Если вы уже пользуетесь некоторое время Linux, то могли заметить, что ваш домашний каталог постепенно накапливает невообразимо большое количество файлов запуска. Они иногда называются *файлами с точкой*, поскольку их имена практически всегда начинаются с точки ( . ). Многие из них создаются автоматически, когда вы впервые запускаете какую-либо программу, и вам никогда не понадобится менять их. В данной главе рассматриваются главным образом файлы запуска оболочки, которые вам, скорее всего, придется изменять или создавать с нуля. Посмотрим, насколько аккуратно следует работать с такими файлами.

## 13.1. Рекомендации по созданию файлов запуска

При создании файлов запуска думайте о пользователе. Если вы являетесь единственным пользователем компьютера, вам не о чем излишне беспокоиться, поскольку ошибки будут касаться только вас и они достаточно просты в устранении. Однако, если вы создаете файлы запуска, которые будут применяться по умолчанию для всех новых пользователей компьютера или сети, или же вы рассчитываете на то, что кто-либо скопирует такие файлы для использования на другом компьютере, ваша задача становится существенно сложнее. Если вы сделаете ошибку в файле запуска, предназначенном для десяти пользователей, вам придется исправлять ее десять раз.

При создании файлов запуска для других пользователей придерживайтесь следующих принципов.

- **Простота.** Старайтесь, чтобы количество файлов запуска было небольшим, сами файлы были бы маленькими, насколько это возможно, простыми для изменения и надежными. Каждый элемент в файле запуска является всего лишь еще одним компонентом, который может выйти из строя.
- **Читаемость.** Создавайте обширные комментарии в файлах, чтобы пользователи получили полное представление о том, что выполняет каждая часть файла.

# 13.2. Когда изменять файлы запуска

Прежде чем выполнить изменение в файле запуска, задайте себе вопрос, действительно ли это изменение необходимо. Вот несколько веских оснований для изменения файлов запуска.

- Необходимо изменить приглашение по умолчанию.
- Необходимо приспособить какое-либо ПО, установленное локально. Попробуйте, однако, для начала использовать сценарии обертки.
- Существующие файлы запуска неисправны.

Если в вашей системе все работает нормально, будьте осторожны. Иногда файлы запуска, используемые по умолчанию, взаимодействуют с другими файлами из каталога `/etc`.

И все же вы, вероятно, не стали бы читать эту главу, если вам неинтересно изменять настройки по умолчанию. Давайте разберемся, что же важно.

# 13.3. Элементы файла запуска оболочки

Что входит в состав файла запуска оболочки? Некоторые элементы могут показаться очевидными, например настройки пути и приглашения. Но что именно *должно* быть указано в качестве пути и как выглядит приемлемое приглашение? Какое количество элементов будет излишним для размещения в файле запуска?

В нескольких следующих разделах рассмотрены основы файла запуска оболочки — начиная с командного пути, приглашения, псевдонимов и заканчивая маской прав доступа.

## 13.3.1. Командный путь

Самой важной частью любого файла запуска является командный путь. Этот путь должен охватывать каталоги, которые содержат приложения, представляющие интерес для обычного пользователя. По меньшей мере этот путь должен содержать следующие элементы в указанном порядке:

```
/usr/local/bin
/usr/bin
/bin
```

Такой порядок гарантирует, что вы сможете переопределить стандартные команды по умолчанию с помощью локальных вариантов, расположенных в каталоге /usr/local.

В большинстве версий Linux исполняемые файлы практически для всех пакетов ПО помещаются в каталог /usr/bin. Иногда бывают исключения, например при размещении игр в каталоге /usr/games, а графических редакторов — в отдельном каталоге, поэтому сначала проверьте настройки вашей системы по умолчанию. Убедитесь также в том, что каждая из системных команд общего пользования доступна в каком-либо из перечисленных выше каталогов. Если это не так, то ваша система, вероятно, вышла из-под контроля. Не меняйте путь по умолчанию для вашей среды пользователя, чтобы подстроиться под каталог установки нового ПО. Простой способ учесть отдельные каталоги установки — использование символических ссылок в каталоге /usr/local/bin.

Многие пользователи применяют каталог bin для хранения собственных сценариев оболочки и команд, поэтому может потребоваться добавить его в самое начало пути:

```
$HOME/bin
```

ПРИМЕЧАНИЕ

По новому соглашению двоичные файлы помещают в каталог $HOME/.local/bin.

Если вам интересны системные утилиты (такие как traceroute, ping и lsmod), добавьте в путь каталоги sbin:

```
/usr/local/sbin
/usr/sbin
/sbin
```

### Добавление точки (.) в путь

Есть один небольшой, но противоречивый компонент командного пути — точка. Если поместить точку (.) в пути, то это позволит запускать команды в текущем каталоге, не используя символы ./ перед именем команды. Это может оказаться удобным при написании сценариев или при компиляции программ, однако такой способ плох по двум причинам.

- Могут появиться проблемы с безопасностью. *Никогда* не следует помещать точку *в начале* пути. Вот что при этом может произойти: взломщик может поместить вирус-троян с именем ls в архив, распространяемый через Интернет. Даже если точка окажется в конце пути, вы по-прежнему будете уязвимы для таких опечаток, как sl или ks.
- Это непоследовательно и может привести к путанице. Точка в пути может означать, что поведение команды будет изменяться в соответствии с текущим каталогом.

## 13.3.2. Путь к страницам руководства

Традиционный путь к страницам руководства определяется с помощью переменной окружения MANPATH, однако не стоит его изменять, поскольку при этом будут перезаписаны системные настройки по умолчанию из файла /etc/manpath.config.

## 13.3.3. Приглашение

Опытные пользователи избегают длинных, усложненных и бесполезных приглашений. Для сравнения: многие администраторы и версии системы стремятся уместить все в приглашении по умолчанию. Ваш выбор должен отражать потребности пользователей; укажите в приглашении текущий рабочий каталог, имя хоста, а также имя пользователя, если это действительно необходимо.

Кроме того, избегайте использовать символы, которые означают что-либо важное для оболочки, например такие:

```
{ } = & < >
```

ПРИМЕЧАНИЕ

Особенно внимательно относитесь к символу >, который может привести к появлению пустых ошибочных файлов в текущем каталоге, если вы случайно скопируете и вставите часть окна оболочки (вспомните о том, что команда > перенаправляет вывод в файл).

Даже приглашение, используемое оболочкой по умолчанию, неидеально. Например, приглашение оболочки `bash` содержит название оболочки и номер версии.

В следующей простой настройке приглашение для оболочки `bash` заканчивается символом `$` (традиционное приглашение оболочки `csh` заканчивается символом `%`):

```
PS1='\u\$ '
```

Вместо текущего имени пользователя используется подстановка `\u` (см. раздел PROMPTING («ПРИГЛАШЕНИЕ» на странице руководства bash(1))). Следующие популярные подстановки содержат:

- `\h` — имя хоста (в короткой форме, без имени домена);
- `\!` — номер в истории;
- `\w` — текущий каталог. Поскольку он может оказаться длинным, можно ограничить отображение только последним компонентом, указав параметр `\W`;
- `\$` — при запуске с учетной записью обычного пользователя применяется `$`, для корневого пользователя — #.

## 13.3.4. Псевдонимы

Одной из характерных черт современной среды пользователя являются *псевдонимы* — это функция оболочки, которая заменяет одну строку другой перед выполнением команды. Псевдонимы могут послужить как сокращения, избавляющие от набора команд. Однако у них есть недостатки:

- передавать аргументы может оказаться затруднительно;
- они приводят к путанице. Встроенная в оболочку команда `which` может сказать, псевдоним ли перед вами, но она не сообщит вам, где он определен;
- они не одобряются в подоболочках и в неинтерактивных оболочках, а также не работают в других оболочках.

Учитывая эти неудобства, вам следует избегать псевдонимов, где это возможно, поскольку проще написать функцию оболочки или полностью новый сценарий.

Современные компьютеры способны запускать и исполнять команды оболочки настолько быстро, что различие между псевдонимом и абсолютно новой командой не будет значимым для вас.

И все же псевдонимы оказываются удобными, когда необходимо изменить часть среды оболочки. Нельзя изменить переменную окружения с помощью сценария оболочки, так как сценарии запускаются в качестве подоболочек. Для выполнения этой задачи можно также определить функции оболочки.

### 13.3.5. Маска прав доступа

Встроенная в оболочку функция `umask` (маска прав доступа) устанавливает права доступа по умолчанию. Следует указать запуск команды `umask` в одном из файлов запуска, чтобы гарантировать то, что каждая выполняемая команда создает файлы с необходимыми вам правами доступа. Два разумных варианта таковы.

- `077`. Эта маска является самой сдерживающей маской прав доступа, поскольку она не разрешает никаким другим пользователям доступ к новым файлам и каталогам. Часто это подходит для многопользовательских систем, в которых нежелательно, чтобы другие пользователи видели какие-либо ваши файлы. Тем не менее, если установить ее по умолчанию, это может привести к проблемам, если пользователи желают использовать файлы совместно, но не умеют правильно настраивать права доступа. Неопытные пользователи стремятся назначить файлам режим доступа «доступен для записи всем».
- `022`. Эта маска дает другим пользователям право чтения новых файлов и каталогов, что может быть важно в однопользовательской системе, так как многие демоны, которые работают как псевдопользователи, не могут видеть файлы и каталоги, созданные с помощью более строгой маски `077`.

ПРИМЕЧАНИЕ

Некоторые приложения (в особенности почтовые программы) переопределяют права доступа, указанные командой umask, изменяя маску на 077, поскольку они считают, что их файлы являются собственностью лишь их владельца и никого более.

## 13.4. Порядок следования файлов запуска. Примеры

Теперь, когда вы знаете, что́ поместить в файлы запуска оболочки, самое время посмотреть на некоторые конкретные примеры. Удивительно то, что одним из самых трудных и запутанных моментов при создании файлов запуска является определение того, какой из нескольких файлов запуска использовать. В следующих разделах рассказано о двух самых популярных оболочках Unix: `bash` и `tcsh`.

### 13.4.1. Оболочка bash

В оболочке `bash` можно выбирать среди файлов запуска `.bash_profile`, `.profile`, `.bash_login` и `.bashrc`. Какой из них соответствует командному пути, пути к страницам

руководства, приглашению, псевдонимам и маске прав доступа? Ответ такой: файл `.bashrc` должен сопровождаться символической ссылкой `.bash_profile`, указывающей на файл `.bashrc`, поскольку существует несколько различных типов экземпляров оболочки.

Два главных типа экземпляров оболочки — интерактивный и неинтерактивный, но из них только интерактивные оболочки представляют интерес, поскольку неинтерактивные оболочки (например, те, которые запускают сценарии оболочки) обычно не читают никаких файлов запуска. Интерактивными оболочками являются те, которые вы применяете для запуска команд из терминала, вроде тех, что вы видите в этой книге, и они могут быть разделены на оболочки *для входа в систему* и *не для входа в систему*.

## Оболочки для входа в систему

Традиционно оболочка для входа в систему возникает, когда вы в первый раз входите в систему с помощью терминала, используя такую команду, как `/bin/login`. Удаленный вход в систему по протоколу SSH также выдает вам оболочку для входа в систему. Основная идея: оболочка для входа в систему является начальной оболочкой. Можно определить, является ли оболочка оболочкой для входа в систему, запустив команду `echo $0`; если первый символ ответа дефис (`-`), то эта оболочка является оболочкой для входа в систему.

Когда оболочка `bash` запущена для входа в систему, она выполняет файл `/etc/profile`. После этого она ищет пользовательские файлы `.bash_profile`, `.bash_login` и `.profile`, запуская только первый файл, который обнаружит.

Как бы странно это ни звучало, но возможно запустить неинтерактивную оболочку в качестве оболочки входа в систему, чтобы принудительно исполнить ее файлы запуска. Для этого запустите такую оболочку с параметром `-l` или `--login`.

## Оболочки не для входа в систему

Оболочка не для входа в систему — дополнительная оболочка, которую вы запускаете после входа в систему. Это всего лишь любая интерактивная оболочка, которая не является оболочкой входа. Терминальные команды системы управления окнами (xterm, GNOME Terminal и т. д.) запускают оболочки не для входа в систему, если только вы не запросите специально оболочку для входа.

После запуска в качестве оболочки не для входа в систему оболочка `bash` выполняет файл `/etc/bash.bashrc`, а затем запускает пользовательский файл `.bashrc`.

## Последствия наличия двух типов оболочек

Причины, по которым существуют два различных типа файлов запуска, заключаются в том, что в давние времена пользователи входили в систему через традиционный терминал с оболочкой для входа в систему, после чего запускали подоболочки не для входа с системами управления окнами или экранной программой. По поводу подоболочек не для входа в систему было решено, что повторяющаяся настройка среды пользователя и запуск набора команд, которые уже запущены, являются расточительством. В оболочках для входа в систему можно было бы запускать замысловатые команды запуска из таких файлов, как `.bash_profile`, оставив только псевдонимы и другие «легковесные» вещи файлу `.bashrc`.

В наши дни большинство пользователей входит в систему через графические менеджеры дисплея (из следующей главы вы узнаете об этом подробнее). Многие из них начинают работу с одной неинтерактивной оболочки для входа в систему, чтобы сохранить описанную выше модель «для входа — не для входа». Если они этого не делают, вам придется полностью настроить окружение (путь, путь к страницам руководства и т. п.) в файле .bashrc, а иначе вы не увидите ничего из вашего окружения в терминальном окне оболочек. Кроме того, вам потребуется также файл .bash_profile, если вы желаете войти в консоль или удаленно, поскольку такие оболочки для входа даже не беспокоятся о файле .bashrc.

## Пример файла .bashrc

Чтобы удовлетворить оба типа оболочек, каким образом создавать файл .bashrc, который можно было бы использовать и в качестве файла .bash_profile? Приведу один весьма элементарный (и в то же время совершенно достаточный) пример:

```
# Command path.
PATH=/usr/local/bin:/usr/bin:/bin:/usr/games
PATH=$HOME/bin:$PATH

# PS1 is the regular prompt.
# Substitutions include:
# \u username \h hostname \w current directory
# \! history number \s shell name \$ $ if regular user
PS1='\u\$ '

# EDITOR and VISUAL determine the editor that programs such as less
# and mail clients invoke when asked to edit a file.
EDITOR=vi
VISUAL=vi

# PAGER is the default text file viewer for programs such as man.
PAGER=less

# These are some handy options for less.
# A different style is LESS=FRX
# (F=quit at end, R=show raw characters, X=don't use alt screen)
LESS=meiX

# You must export environment variables.
export PATH EDITOR VISUAL PAGER LESS

# By default, give other users read-only access to most new files.
umask 022
```

Как описано ранее, можно использовать такой файл .bashrc в качестве файла .bash_profile с помощью символической ссылки. Можно также сделать эту связь еще четче, создав файл .bash_profile, как в этой однострочной команде:

```
. $HOME/.bashrc
```

### Проверка того, является ли оболочка интерактивной

Если файл .bashrc соответствует файлу .bash_profile, то обычно не приходится запускать дополнительные команды для оболочек входа в систему. Однако, если вы желаете назначить различные действия для разных типов оболочек, можно добавить следующую проверку в файл .bashrc, которая отслеживает наличие символа i в переменной оболочки $-:

```
case $- in
 *i*) # interactive commands go here
    command
    --snip--
    ;;
 *)   # non-interactive commands go here
    command
    --snip-
    ;;
esac
```

## 13.4.2. Оболочка tcsh

Стандартным вариантом оболочки csh практически во всех системах Linux является оболочка tsch, улучшенная C-оболочка, которая сделала популярными такие функции, как редактирование командной строки и многорежимное завершение имен файлов и команд. Даже если вы не используете оболочку tsch как оболочку по умолчанию для нового пользователя (мы рекомендуем применять оболочку bash), наличие файлов запуска для нее все же необходимо на тот случай, если пользователь натолкнется на нее.

В оболочке tsch вам не надо беспокоиться о различиях между оболочками для входа/не для входа в систему. Во время запуска оболочка tsch ищет файл .tcshrc. Если это не удается, она ищет файл запуска .cshrc оболочки csh. Причина такого порядка действий в том, что файл .tcshrc можно использовать для таких расширений оболочки tcsh, которые не работают в оболочке csh. Вам, возможно, стоит придерживаться обычного файла .cshrc вместо .tcshrc; весьма маловероятно, что кому-либо понадобится применить ваши файлы запуска для оболочки csh. Если какой-либо пользователь действительно встретит оболочку csh в другой системе, ваш файл .cshrc будет работать.

### Пример файла.cshrc

Приведу пример файла .cshrc:

```
# Command path.
setenv PATH /usr/local/bin:/usr/bin:/bin:$HOME/bin

# EDITOR and VISUAL determine the editor that programs such as less
# and mail clients invoke when asked to edit a file.
setenv EDITOR vi
setenv VISUAL vi

# PAGER is the default text file viewer for programs such as man.
```

```
setenv PAGER less

# These are some handy options for less.
setenv LESS meiX

# By default, give other users read-only access to most new files.
umask 022

# Customize the prompt.
# Substitutions include:
# %n username %m hostname %/ current directory
# %h history number %l current terminal %% %
set prompt="%m%% "
```

# 13.5. Пользовательские настройки по умолчанию

Лучший способ написать файлы запуска и выбрать параметры по умолчанию для новых пользователей — поэкспериментировать в системе с новым «тестовым» пользователем. Создайте такого пользователя с пустым домашним каталогом и воздержитесь от копирования ваших файлов запуска в каталог этого пользователя. Напишите новые файлы запуска с нуля.

Когда вы решите, что рабочие настройки готовы, зайдите в систему как новый тестовый пользователь всеми возможными способами (через консоль, удаленно и т. д.). Убедитесь в том, что вы проверили все, что только можно, включая систему управления окнами и страницы руководства. Когда вас устроят параметры тестового пользователя, создайте второго тестового пользователя, скопировав файлы запуска от первого. Если все работает так же хорошо, то у вас теперь есть набор файлов запуска, который можно предоставлять новым пользователям.

В следующих разделах приведены разумные параметры по умолчанию для новых пользователей.

## 13.5.1. Параметры по умолчанию для оболочки

Оболочкой по умолчанию для всех новых пользователей системы Linux следует сделать `bash`, так как:

- пользователи будут взаимодействовать с той же оболочкой, в какой они привыкли создавать сценарии (к примеру, оболочка `csh` является весьма плохим инструментом для создания сценариев — даже и не думайте о ней);
- оболочка `bash` является стандартом в системах Linux;
- оболочка `bash` использует библиотеку GNU readline, и, следовательно, ее интерфейс идентичен интерфейсу многих других инструментов;
- оболочка `bash` обеспечивает вас прекрасным и понятным контролем над перенаправлением ввода/вывода и работой с файлами.

Тем не менее многие опытные кудесники Unix пользуются такими оболочками, как csh и tcsh, только потому, что они не любят менять привычки. Конечно же, вы можете выбрать любую оболочку по своему вкусу, но, если у вас нет никаких предпочтений, выбирайте bash, а также применяйте ее как оболочку по умолчанию для любого нового пользователя системы. Пользователь может изменить оболочку с помощью команды chsh, чтобы она соответствовала его индивидуальным вкусам.

ПРИМЕЧАНИЕ

Есть множество других оболочек (rc, ksh, zsh, es и т. п.). Некоторые из них не подходят в качестве оболочки для новичка, однако иногда новые пользователи, которые ищут альтернативные оболочки, применяют оболочки zsh и fish.

## 13.5.2. Редактор

В традиционной системе редактором по умолчанию является vi или emacs. Это единственные редакторы, наличие которых практически гарантировано для любой системы Unix, это значит, что они доставят меньше всего неудобств для нового пользователя в процессе работы. Однако часто в системах Linux в качестве редактора по умолчанию настраивается nano, поскольку для начинающих он проще в применении.

Что касается файлов запуска оболочки, избегайте больших файлов запуска для редактора по умолчанию. Небольшая команда set showmatch в файле запуска .exrc еще никому не повредила: она избавляет редактор от всего, что существенно меняет его поведение или внешний вид (например, функция showmode, автоматическая расстановка отступов и перенос текста на следующую строку).

## 13.5.3. Переменная PAGER

Абсолютно разумно указать для переменной окружения PAGER значение по умолчанию less.

# 13.6. Подводные камни в файлах запуска

Придерживайтесь следующих правил в файлах запуска.

- Не помещайте никаких графических команд в файл запуска оболочки.
- Не определяйте переменную окружения DISPLAY в файле запуска оболочки.
- Не определяйте тип терминала в файле запуска оболочки.
- Не скупитесь на подробные комментарии в файлах запуска по умолчанию.
- Не запускайте в файле запуска команды, которые выполняют печать в стандартный вывод.
- Никогда не определяйте переменную LD_LIBRARY_PATH в файле запуска оболочки (см. подраздел 15.1.4).

## 13.7. Дальнейшие вопросы, связанные с запуском

Поскольку в этой книге рассматриваются только уровни, лежащие в основе Linux, я не буду рассказывать о файлах запуска оконной среды. Это действительно объемная тема, поскольку менеджер дисплея, который выполняет ваш вход в современной системе Linux, обладает собственным набором файлов запуска, таких как `.xsession`, `.xinitrc`, а также бесконечным набором комбинаций из элементов, относящихся к средам GNOME и KDE.

Варианты работы с окнами могут показаться сбивающими с толку, поскольку не существует общего способа запустить оконную среду в Linux. В следующей главе описаны некоторые из множества возможностей. Тем не менее, когда вы определяете, что должна делать ваша система, вас могут немного увести в сторону файлы, относящиеся к графическому окружению. Это нормально, но не взваливайте эту работу на новых пользователей. Принцип, согласно которому файлы запуска оболочки следует делать как можно проще, творит также чудеса и для файлов запуска графического интерфейса пользователя. На самом деле вам, наверное, вовсе не требуется изменять эти файлы запуска.

# 14 Краткий обзор рабочего стола Linux

Эта глава является кратким описанием компонентов, которые расположены на рабочем столе типичной системы Linux. Среди различных типов программного обеспечения, которое можно найти в Linux, область рабочего стола является одной из самых необузданных и красочных, поскольку для выбора существует очень много сред и приложений, и в большинстве версий ОС сравнительно легко их опробовать.

В отличие от других частей системы Linux, таких как хранилища и сеть, при создании структуры рабочего стола не сильно задействована иерархия слоев. Вместо этого каждый компонент выполняет определенную задачу, взаимодействуя по мере необходимости с другими компонентами. Некоторые компоненты все же используют совместно общие строительные блоки (в частности, библиотеки для графического инструментария), которые можно представлять себе как простые уровни абстракции, но не более того.

В этой главе в общих чертах затронуты высокоуровневые компоненты рабочего стола, однако мы более подробно рассмотрим два из них: систему X Window, являющуюся основной инфраструктурой для большинства рабочих столов, и шину D-Bus, службу межпроцессного взаимодействия, которая использована во многих частях системы. Мы сведем практические вопросы и примеры к рассмотрению нескольких диагностических утилит, которые, хоть и не слишком полезны в повседневной работе (чтобы взаимодействовать с большинством графических интерфейсов пользователя, нет необходимости вводить команды оболочки), но помогут вам понять лежащие в их основе механизмы системы и, возможно, попутно немного развлекут вас. Мы также вкратце рассмотрим печать.

## 14.1. Компоненты рабочего стола

Конфигурация рабочего стола Linux является очень гибкой. Большая часть того, с чем приходится иметь дело пользователю Linux («впечатления и ощущения» от использования рабочего стола), исходит от приложений или их строительных блоков. Если вам не нравится какое-либо приложение, то, как правило, можно найти ему замену. Если то, что вы ищете, пока еще не существует, вы можете создать это самостоятельно. Разработчики Linux стремятся к тому, чтобы у рабочего стола

было большое разнообразие возможных вариантов настройки, это приводит к большему выбору.

Для совместной работы всем приложениям необходимо иметь что-либо общее, и в сердцевине почти всего в большинстве рабочих столов Linux находится X-сервер (сервер системы X Window). Рассматривайте его как своего рода «ядро» рабочего стола, которое управляет всем, начиная с рендеринга окон и конфигурирования дисплеев и завершая обработкой ввода от таких устройств, как клавиатура и мышь. X-сервер является также тем компонентом, которому вы с трудом сможете подыскать замену (см. раздел 14.4).

X-сервер — это всего лишь сервер, и он не навязывает способ действия или отображения чего-либо. Вместо него с пользовательским интерфейсом работают команды X-клиента. Базовые приложения X-клиента, например окна терминала и браузеры, подключаются к X-серверу и просят его нарисовать окна. В ответ на это X-сервер выясняет, где разместить эти окна, и выполняет рендеринг. Когда требуется, X-сервер отправляет также ввод обратно клиенту.

## 14.1.1. Менеджеры окон

X-клиенты не должны действовать подобно приложениям, реализованным с помощью окон; они могут действовать как службы для других клиентов или обеспечивать другие функции интерфейса. *Менеджер окна* является, вероятно, самым важным приложением службы клиента, поскольку он вычисляет, как организовать окна на экране, и снабжает их интерактивными «украшениями» вроде заголовочной строки, которая позволяет пользователю перемещать и минимизировать окна. Эти возможности являются центральными для работы пользователя.

Существует масса реализаций менеджера окон. Такие варианты, как Mutter/GNOME Shell и Compiz, предназначены для автономной работы, в то время как другие являются встроенными в среду, например Xfce. Многие из менеджеров окон, входящие в стандартные версии Linux, стремятся предоставить пользователю максимальное удобство работы, но некоторые снабжены специфичными визуальными эффектами или используют минималистский подход. Вряд ли когда-либо появится стандартный менеджер окон Linux, поскольку вкусы и потребности пользователей различны и постоянно изменяются, в результате чего все время появляются новые менеджеры окон.

## 14.1.2. Инструментарий

Приложения рабочего стола содержат некоторые общие элементы, например кнопки и меню, которые называют *виджетами*. Для ускорения разработки и придания единства оформлению программисты используют графический *инструментарий* для реализации этих элементов. В таких операционных системах, как Windows или Mac OS X, их производители обеспечивают единый инструментарий, который использует большинство программистов. В Linux чаще всего применяется инструментарий GTK+, но вам также будут встречаться виджеты, созданные с помощью фреймворка Qt и других.

Инструментарии обычно состоят из совместно используемых библиотек и файлов поддержки, таких как изображения и информация о теме.

### 14.1.3. Окружение рабочего стола

Несмотря на то что инструментарий обеспечивает унифицированное оформление, для некоторых деталей рабочего стола требуется определенная степень кооперации между различными приложениями. Например, одному из приложений может потребоваться совместно использовать данные с другим приложением или обновить общую строку уведомлений на рабочем столе. Чтобы удовлетворить эти потребности, инструментарии и другие библиотеки объединяются в обширные пакеты, называемые *окружениями рабочего стола*. Распространенными окружениями рабочего стола Linux являются GNOME, KDE, Unity и Xfce.

Инструментарии лежат в основе большинства окружений рабочего стола, но для создания унифицированного рабочего стола окружение должно также включать большое количество файлов поддержки, таких как значки и конфигурации, которые образуют темы. Все это объединяется воедино с помощью документации, описывающей соглашения о дизайне, такие как внешний вид меню и заголовков приложений, а также то, как приложения должны реагировать на определенные системные события.

### 14.1.4. Приложения

На вершине рабочего стола находятся приложения, например браузеры и окно терминала. X-приложения могут быть как довольно «грубыми» (вроде древней команды `xclock`), так и достаточно сложными (например, браузер Chrome и пакет LibreOffice). Эти приложения, как правило, работают автономно, но они часто используют межпроцессное взаимодействие, чтобы быть в курсе происходящих событий. Например, какое-либо приложение может проявить интерес к тому, что вы подключили новое устройство хранения или получили новое электронное письмо или мгновенное сообщение. Такое взаимодействие обычно происходит по шине D-Bus, описанной в разделе 14.5.

## 14.2. Подробнее о системе X Window

Система X Window (http://www.x.org/) исторически сложилась очень большой, с основным дистрибутивом, включающим X-сервер, библиотеки поддержки клиентов и самих клиентов. Вследствие появления таких сред рабочего стола, как GNOME и KDE, роль пакета X со временем изменилась, и теперь акцент сделан на основном сервере, который управляет рендерингом и устройствами ввода, а также на упрощении библиотеки клиентов.

X-сервер легко обнаружить в системе. Он называется `X`. Поищите его в списке процессов; обычно вы сможете обнаружить, что он запущен с некоторым количеством параметров, например, так:

```
/usr/bin/X :0 -auth /var/run/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
```

Параметр :0, показанный здесь, называется *дисплеем*. Это идентификатор, представляющий один или несколько мониторов, к которым вы получаете доступ с помощью клавиатуры и/или мыши. Обычно дисплей соответствует единственному монитору, который подключен к вашему компьютеру, но вы можете поместить несколько мониторов за одним и тем же дисплеем. При использовании X-сеанса для переменной окружения DISPLAY установлено значение идентификатора дисплея.

ПРИМЕЧАНИЕ

Дисплеи можно далее подразделить на экраны, такие как :0.0 и :0.1, но это используется все реже, поскольку такие расширения, как RandR, способны объединить несколько мониторов в один общий виртуальный экран.

В Linux X-сервер запускается в виртуальном терминале. В данном примере аргумент vt7 говорит нам о том, что сервер был запущен в терминале /dev/tty7 (обычно сервер запускается в первом доступном виртуальном терминале). Можно запустить более одного X-сервера в данный момент времени, предоставив каждому из них отдельный виртуальный терминал, но в таком случае для каждого сервера потребуется уникальный идентификатор дисплея. Между серверами можно переключаться с помощью сочетания клавиш Ctrl+Alt+Fn или команды chvt.

## 14.2.1. Менеджеры дисплея

Обычно X-сервер не запускают с помощью командной строки, поскольку запуск этого сервера не определяет никаких клиентов, предназначенных для работы с ним. Если вы запустите сервер сам по себе, вы просто получите пустой экран. Вместо этого самым распространенным способом запуска X-сервера является использование *менеджера дисплея* — утилиты, которая запускает сервер и помещает на экран окно входа в систему. Когда вы выполните вход, менеджер дисплея запускает ряд клиентов, таких как менеджер окон и менеджер файлов, чтобы вы смогли начать использование компьютера.

Существует много разных менеджеров дисплея, например gdm (для среды GNOME) и kdm (для среды KDE). Аргумент lightdm, присутствующий в приведенном выше вызове X-сервера, является кросс-платформенным менеджером дисплея, предназначенным для запуска в сеансах GNOME или KDE.

Чтобы запустить X-сеанс из виртуальной консоли, а не с помощью менеджера дисплея, можно выполнить команду startx или xinit. Однако сессия, которую вы в итоге получите, будет, вероятно, очень простой и совершенно отличающейся от той, которая создается менеджером дисплея, поскольку их функции и стартовые файлы различны.

## 14.2.2. Прозрачность сети

Одним из свойств X-сервера является прозрачность сети. Поскольку клиенты общаются с сервером с помощью протокола, то возникает возможность непосредственного запуска клиентов по сети для сервера, расположенного на другом

компьютере, при этом X-сервер прослушивает TCP-соединения порта 6000. Клиенты, подключающиеся к этому серверу, могут пройти аутентификацию, а затем отправлять окна на сервер.

К сожалению, этот метод не предлагает никакого шифрования и, как следствие, не является защищенным. Чтобы закрыть эту брешь, сейчас в большинстве версий ОС для X-сервера отключен прослушиватель сети (с помощью параметра `-nolisten tcp`, как видно из приведенного примера). Тем не менее X-клиенты все же возможно запустить с удаленного компьютера с помощью SSH-туннелирования, как рассказано в главе 10, подключив сокет домена Unix X-сервера к сокету удаленного компьютера.

# 14.3. Исследование X-клиентов

Хотя обычно никому не приходит в голову работать с графическим интерфейсом пользователя из командной строки, есть несколько утилит, которые позволяют вам исследовать части системы X Window. В частности, можно инспектировать клиенты во время их работы.

Одним из простейших является инструмент `xwininfo`. Если запустить эту команду без аргументов, она попросит вас щелкнуть кнопкой мыши на окне.

```
$ xwininfo
xwininfo: Please select the window about which you
          would like information by clicking the
          mouse in that window.
```

После щелчка команда выводит об этом окне такую информацию, как его положение и размер:

```
xwininfo: Window id: 0x5400024 "xterm"

  Absolute upper-left X: 1075
  Absolute upper-left Y: 594
--snip--
```

Обратите здесь внимание на идентификатор окна — X-сервер и менеджеры окон используют его, чтобы отслеживать окна. Чтобы получить перечень всех идентификаторов окон и клиентов, используйте команду `xlsclients -l`.

ПРИМЕЧАНИЕ

Есть специальное окно, которое называется корневым окном, — это фон дисплея. Однако, возможно, вы его никогда не видели (см. пункт «Фон рабочего стола» подраздела 14.3.2).

## 14.3.1. X-события

X-клиенты получают данные ввода и другую информацию о состоянии сервера через систему событий. X-события устроены подобно любому другому асинхронному межпроцессному взаимодействию, такому как события менеджера udev

и шины D-Bus: X-сервер получает информацию от источника (например, устройства ввода), а затем предоставляет эти данные как событие для любого заинтересованного X-клиента.

Поэкспериментировать с событиями можно с помощью команды `xev`. При ее запуске открывается новое окно, в котором вы можете перемещать указатель мыши, щелкать кнопкой мыши и набирать текст. Когда вы делаете это, команда `xev` генерирует вывод, описывающий X-события, которые она получает от сервера. Вот, например, фрагмент вывода, относящегося к перемещениям мыши:

```
$ xev
--snip--
MotionNotify event, serial 36, synthetic NO, window 0x6800001,
    root 0xbb, subw 0x0, time 43937883, (47,174), root:(1692,486),
    state 0x0, is_hint 0, same_screen YES

MotionNotify event, serial 36, synthetic NO, window 0x6800001,
    root 0xbb, subw 0x0, time 43937891, (43,177), root:(1688,489),
    state 0x0, is_hint 0, same_screen YES
```

Обратите внимание на координаты в скобках. Первая пара представляет координаты `x` и y для указателя мыши внутри окна, а вторая (`root:`) определяет положение указателя на всем дисплее.

В число других низкоуровневых событий входят нажатия на клавиши и кнопки мыши, а более сложные учитывают, был ли перемещен указатель в окно или из окна или получило ли окно фокус из менеджера окон. Вот, например, соответствующие события выхода за пределы окна и утраты фокуса:

```
LeaveNotify event, serial 36, synthetic NO, window 0x6800001,
    root 0xbb, subw 0x0, time 44348653, (55,185), root:(1679,420),
    mode NotifyNormal, detail NotifyNonlinear, same_screen YES,
    focus YES, state 0

FocusOut event, serial 36, synthetic NO, window 0x6800001,
    mode NotifyNormal, detail NotifyNonlinear
```

Одним распространенным способом применения команды `xev` является извлечение кодов клавиш и символов для различных клавиатур, когда выполняется настройка раскладки клавиатуры. Вот результат нажатия клавиши L; код клавиши при этом равен 46:

```
KeyPress event, serial 32, synthetic NO, window 0x4c00001,
    root 0xbb, subw 0x0, time 2084270084, (131,120), root:(197,172),
    state 0x0, keycode 46 (keysym 0x6c, l), same_screen YES,
    XLookupString gives 1 bytes: (6c) "l"
    XmbLookupString gives 1 bytes: (6c) "l"
    XFilterEvent returns: False
```

Можно также прикрепить команду `xev` к существующему идентификатору окна с помощью параметра `-id` *id* (используйте для параметра *id* тот идентификатор, который получен с помощью команды `xwininfo`) или отслеживать корневое окно с помощью параметра `-root`.

## 14.3.2. Понятие о X-вводе и настройка предпочтений

Одной из потенциально вводящих в тупик характеристик X-системы является то, что зачастую есть несколько способов настройки предпочтений, и некоторые из них могут не работать. Например, одним из распространенных клавиатурных предпочтений в Linux является переназначение клавиши Caps Lock на клавишу Control. Это можно выполнить несколькими способами, начиная с небольших регулировок с помощью старой команды `xmodmap` и заканчивая созданием совершенно новой раскладки клавиатуры с помощью утилиты `setxkbmap`. Как понять, какой из них следует (и следует ли) применить? Все упирается в знание того, какие части системы отвечают за это, но выяснить бывает сложно. Помните о том, что окружение рабочего стола может обладать собственными настройками и переопределениями.

С учетом сказанного опишу некоторые моменты, относящиеся к основной инфраструктуре.

### Устройства ввода (в целом)

X-сервер использует расширение *X Input Extension*, чтобы управлять вводом от различных устройств. Есть два основных типа устройств ввода — клавиатура и указатель (мышь), — и можно подключить столько устройств, сколько пожелаете. Чтобы одновременно использовать несколько устройств одного типа, расширение X Input Extension создает «виртуальное устройство ядра», которое направляет ввод от устройства на X-сервер. Устройство ядра называется ведущим; а физические устройства, подключаемые к компьютеру, — подчиненными.

Чтобы увидеть конфигурацию устройств на вашем компьютере, попробуйте запустить команду `xinput --list`:

```
$ xinput --list
  Virtual core pointer                    id=2    [master pointer  (3)]
      Virtual core XTEST pointer          id=4    [slave pointer   (2)]
      Logitech Unifying Device            id=8    [slave pointer   (2)]
  Virtual core keyboard                   id=3    [master keyboard (2)
      Virtual core XTEST keyboard         id=5    [slave keyboard  (3)]
      Power Button                        id=6    [slave keyboard  (3)]
      Power Button                        id=7    [slave keyboard  (3)]
      Cypress USB Keyboard                id=9    [slave keyboard  (3)]
```

У каждого устройства есть связанный с ним идентификатор, который можно использовать в команде `xinput` и в других командах. В данном выводе идентификаторы 2 и 3 соответствуют устройствам ядра, а идентификаторы 8 и 9 — реальным устройствам. Обратите внимание на то, что кнопки включения компьютера также рассматриваются как устройства ввода.

Большинство X-клиентов выполняют прослушивание ввода от устройств ядра, поскольку им нет повода беспокоиться о том, какое именно устройство вызвало событие ввода. В действительности большинство клиентов ничего не знает о расширении X Input Extension. Тем не менее клиент может использовать это расширение, чтобы избрать какое-либо конкретное устройство.

Каждое устройство обладает набором связанных с ним *свойств*. Чтобы просмотреть эти свойства, используйте команду `xinput` с номером устройства, как в этом примере:

```
$ xinput --list-props 8
Device 'Logitech Unifying Device. Wireless PID:4026':
        Device Enabled (126): 1
        Coordinate Transformation Matrix (128): 1.000000, 0.000000, 0.000000,
0.000000, 1.000000, 0.000000, 0.000000, 0.000000, 1.000000
        Device Accel Profile (256): 0
        Device Accel Constant Deceleration (257): 1.000000
        Device Accel Adaptive Deceleration (258): 1.000000
        Device Accel Velocity Scaling (259): 10.000000
--snip--
```

Как видите, есть несколько весьма интересных свойств, которые можно изменить с помощью параметра `--set-prop`. Дополнительную информацию можно получить на странице руководства xinput(1).

## Мышь

Можно управлять параметрами, относящимися к устройству, с помощью команды `xinput`, наиболее полезные из них относятся к мыши (указателю). Можно изменить многие параметры непосредственно как свойства, но обычно проще выполнить это с помощью специальных параметров `--set-ptr-feedback` и `--set-button-map` команды `xinput`. Например, если к устройству *dev* подключена мышь с тремя кнопками и вы желаете поменять на ней порядок следования кнопок (для удобства работы левши), попробуйте такую команду:

```
$ xinput --set-button-map dev 3 2 1
```

## Клавиатура

Множество различных вариантов раскладки клавиатуры, доступных в разных странах, представляет особые сложности для интеграции в любую оконную систему. В системе X всегда присутствовала возможность настройки клавиатуры с помощью протокола ядра, которым можно управлять, используя команду `xmodmap`, однако в любой достаточно современной системе применяется расширение XKB (X-клавиатура), позволяющее добиться более точной настройки.

Расширение XKB является настолько сложным, что многие пользователи до сих пор применяют команду `xmodmap`, когда им необходимо быстро внести изменения.

Основная идея, заложенная в расширение XKB, такова: можно определить раскладку клавиатуры и скомпилировать ее с помощью команды `xkbcomp`, а затем загрузить и активизировать эту раскладку на X-сервере с помощью команды `setxkbmap`. У этой системы есть две чрезвычайно интересные особенности.

- Можно определять частичные раскладки, чтобы дополнить уже существующие. Это особенно удобно для таких задач, как превращение клавиши Caps Lock в клавишу Control, и используется многими утилитами настройки клавиатуры в окружении рабочего стола.
- Можно определить индивидуальные раскладки для каждой подключенной клавиатуры.

### Фон рабочего стола

Старая команда `xsetroot` системы X позволяет вам указать цвет фона и другие характеристики корневого окна, но это никак не проявляется на большинстве компьютеров, поскольку корневого окна никогда не видно. Вместо него во многих окружениях рабочего стола позади всех окон помещается большое окно, позволяющее задействовать такие функции, как «активные обои» и просмотр файлов с рабочего стола. Существуют способы изменить фон из командной строки (например, с помощью команды `gsettings` в некоторых версиях среды GNOME), но если вам это действительно необходимо, то у вас, вероятно, очень много свободного времени.

### Команда xset

Самой старой командой для выполнения настройки является команда `xset`. Она больше не применяется, но вы можете запустить короткую команду `xset q`, чтобы получить отчет о состоянии некоторых функций. Возможно, самыми полезными из них будут параметры хранителя экрана и DPMS (Display Power Management Signaling, сигналы управления энергопотреблением дисплеев).

## 14.4. Будущее системы X Window

При чтении изложенного выше у вас могло сложиться впечатление, что система X является довольно старой и для того, чтобы она смогла научиться выполнять новые задачи, пришлось дать ей изрядное количество «пинков». Вы будете не очень далеки от истины. Первый вариант системы X Window появился в 80-е годы. Несмотря на то что за прошедшие годы она существенно эволюционировала (гибкость являлась важной частью исходного замысла), даже сейчас можно проследить в ней исходную архитектуру.

Одним из признаков эпохи системы X Window является то, что сам ее сервер поддерживает исключительно огромное количество библиотек, многие из них — в целях обратной совместимости. Но, вероятно, более важным является следующее: идея о том, чтобы сервер мог управлять клиентами, их окнами, а также выступать в роли посредника для оконной памяти, сильно повлияла на быстродействие. Если позволить приложениям выполнять рендеринг содержимого их окон непосредственно в дисплейной памяти, то это происходило бы гораздо быстрее. Для этого применяется облегченный менеджер окон, который называется *композитным менеджером окон* и выполняет минимальное управление дисплейной памятью.

Новый стандарт, основанный на этой идее (Wayland), начинает набирать силу. Наиболее важной частью стандарта Wayland является протокол, который определяет, как клиенты взаимодействуют с композитным менеджером окна. Другими частями являются управление устройством ввода и система совместимости со стандартом X. Как протокол, Wayland также поддерживает идею сетевой прозрачности. Многие среды рабочего стола Linux, например GNOME и KDE, поддерживают теперь стандарт Wayland.

Однако стандарт Wayland является не единственной альтернативой системе X. На момент написания этой книги известно, что другой проект, Mir, имеет похожие

цели, хотя в его архитектуре использован немного другой подход. Когда-нибудь произойдет повсеместное принятие хотя бы одной новой системы, которая может оказаться или не оказаться какой-либо из упомянутых.

Эти новые разработки важны, так как они не будут ограничены лишь рабочим столом Linux. Вследствие своего малого быстродействия и гигантского объема используемого дискового пространства система X Window непригодна для таких сред, как планшеты и смартфоны, поэтому производителям до сих пор приходилось использовать альтернативные системы, чтобы привести в действие встроенные дисплеи Linux. Тем не менее стандартизированный прямой рендеринг может содействовать появлению менее затратных способов поддержки таких дисплеев.

## 14.5. Шина D-Bus

Одной из самых важных разработок, которая возникла из рабочего стола Linux, является *шина рабочего стола* (D-Bus, Desktop Bus) — система передачи сообщений. Шина D-Bus важна, поскольку она служит механизмом межпроцессного взаимодействия, который позволяет приложениям рабочего стола «общаться» друг с другом. Она важна еще и потому, что большинство систем Linux применяет ее для уведомления процессов о системных событиях, таких как вставка USB-накопителя.

Эта шина состоит из библиотеки, которая стандартизирует межпроцессное взаимодействие с помощью протокола и функций поддержки, позволяющих любым двум процессам взаимодействовать друг с другом. Сама по себе эта библиотека представляет не что иное, как специальную версию обычных функций IPC, таких как сокеты домена Unix. Шину D-Bus делает полезной наличие центрального «концентратора», который называется `dbus-daemon`. Процессы, которым необходимо реагировать на события, могут подключаться к этому демону и регистрироваться для получения различных типов событий. Процессы могут также создавать события. Например, процесс `udisks-daemon` прослушивает службу `ubus` на наличие дисковых событий, а затем отправляет их демону `dbus-daemon`, который передает их приложениям, заинтересованным в дисковых событиях.

### 14.5.1. Системный и сеансовый экземпляры

Шина D-Bus стала неотъемлемой частью Linux, и сейчас она выходит за рамки рабочего стола. Например, в системах systemd и Upstart есть каналы коммуникации шины D-Bus. Тем не менее добавление в ядро системы зависимостей от инструментов рабочего стола идет вразрез с основным замыслом Linux.

Чтобы решить эту проблему, есть два типа экземпляров (процессов) `dbus-daemon`, которые можно запустить. Первый является системным экземпляром, запускаемым системой init во время загрузки системы с помощью параметра `--system`. Системный экземпляр обычно работает как пользователь D-Bus, и его файлом конфигурации является `/etc/dbus-1/system.conf` (хотя, возможно, вам не придется его менять). Процессы могут подключаться к системному экземпляру через сокет домена Unix `/var/run/dbus/system_bus_socket`.

Независимо от системного экземпляра шины D-Bus существует необязательный сеансовый экземпляр, который работает только тогда, когда вы запускаете сеанс рабочего стола. Приложения рабочего стола подключаются к этому экземпляру.

## 14.5.2. Отслеживание сообщений шины D-Bus

Одним из лучших способов увидеть различия между системным и сеансовым экземплярами демона `dbus-daemon` является отслеживание событий, которые проходят по шине. Попробуйте применить утилиту `dbus-monitor` в системном режиме следующим образом:

```
$ dbus-monitor --system
signal sender=org.freedesktop.DBus -> dest=:1.952 serial=2 path=/org/
freedesktop/DBus; interface=org.freedesktop.DBus; member=NameAcquired
   string ":1.952"
```

Стартовое сообщение говорит о том, что монитор подключен и получил имя. При подобном запуске вы не должны увидеть большую активность, поскольку системный экземпляр обычно не очень занят. Чтобы увидеть какие-либо события, попробуйте подключить USB-накопитель.

По сравнению с этим сеансовому экземпляру приходится выполнять многое. При условии, что вы вошли в сеанс рабочего стола, попробуйте ввести такую команду:

```
$ dbus-monitor --session
```

Теперь проведите указателем мыши над разными окнами; если ваш рабочий стол знает о шине D-Bus, вы должны получить шквал сообщений об активизированных окнах.

# 14.6. Печать

Печать документа в Linux является многоэтапным процессом. Он протекает следующим образом.

1. Программа, выполняющая печать, обычно конвертирует документ в формат PostScript. Этот шаг необязателен.
2. Программа отправляет документ на сервер печати.
3. Сервер печати получает документ и помещает его в очередь печати.
4. Когда доходит очередь до этого документа, сервер печати отправляет документ в фильтр печати.
5. Если документ не в формате PostScript, фильтр печати может выполнить его конвертацию.
6. Если целевой принтер не понимает язык PostScript, драйвер принтера конвертирует документ в формат, пригодный для принтера.

7. Драйвер принтера добавляет к документу дополнительные инструкции, такие как параметры лотка бумаги и двухсторонней печати.
8. Сервер печати использует прикладную часть, чтобы отправить документ на принтер.

Больше всего в этом процессе смущает постоянное обращение к формату PostScript. В действительности это язык программирования, поэтому, когда вы печатаете файл с его использованием, вы отправляете на принтер программу. Формат PostScript играет роль стандарта при печати в системах вроде Unix, подобно тому как формат `.tar` выступает в качестве стандарта при архивировании. Некоторые приложения теперь используют стандарт вывода PDF, но его сравнительно легко конвертировать.

Чуть позже мы поговорим о формате печати больше, но сначала посмотрим на систему работы с очередью печати.

## 14.6.1. Система CUPS

Стандартной системой печати в Linux является CUPS (http://www.cups.org/). Эта же система использована в Mac OS X. Демон сервера CUPS называется `cupsd`, и можно применять в качестве простого клиента команду `lpr` для отправки файлов этому демону.

Одной важной чертой системы CUPS является реализация *протокола IPP* (Internet Print Protocol, протокол печати через Интернет). Это система, которая позволяет трансакции, подобные трансакциям по протоколу HTTP, между клиентами и серверами через TCP-порт 631. На самом деле, если в вашей системе работает система CUPS, вы, вероятно, сможете подключиться к порту http://localhost:631/, чтобы увидеть текущую конфигурацию и проверить задания принтера. Большинство сетевых принтеров и серверов печати поддерживает протокол IPP. Его поддерживает и Windows, с помощью которой довольно просто настроить удаленные принтеры.

Вероятно, вы не сможете администрировать эту систему с помощью веб-интерфейса, поскольку его настройка по умолчанию не слишком защищена. Вместо этого в вашей версии ОС наверняка есть графический интерфейс настройки, позволяющий добавлять и изменять принтеры. Эти инструменты работают с файлами конфигурации, которые обычно находятся в каталоге `/etc/cups`. Как правило, лучше всего позволить этим инструментам выполнить работу за вас, поскольку конфигурация может оказаться сложной. Даже если вы встретитесь с проблемами и вам понадобится ручная настройка, лучше создать принтер с помощью графических инструментов, чтобы вам было с чего начать.

## 14.6.2. Преобразование формата и фильтры печати

Многие принтеры, в число которых входят почти все недорогие модели, не понимают язык PostScript или формат PDF. Чтобы система Linux могла поддерживать такие принтеры, следует конвертировать документы в формат, пригодный для принтера. Система CUPS отправляет документ в процессор RIP (Raster Image Processor, процессор растровых изображений), чтобы создать растровое изображение. Процессор

RIP почти всегда использует команду Ghostscript (`gs`) для выполнения большей части работы, но это довольно сложно, поскольку растровое изображение должно соответствовать формату принтера. Тогда драйверы принтера, которые использует система CUPS, «консультируются» с файлом PPD (PostScript Printer Definition, описание принтера на языке PostScript), чтобы узнать такие параметры, как разрешение печати и размер бумаги.

## 14.7. Другие темы, относящиеся к рабочему столу

Одним интересным свойством окружения рабочего стола Linux является то, что вы, как правило, можете выбрать элементы, которые желаете использовать, и прекратить применять те, которые вам не нравятся. Чтобы получить обзор многих проектов, относящихся к рабочему столу, посмотрите список рассылки и ссылки на проекты на сайте http://www.freedesktop.org/. Можно найти и другие проекты, такие как Ayatana, Unity и Mir.

Еще одним усовершенствованием рабочего стола Linux является проект с открытым кодом Chromium OS и его эквивалент, Google Chrome OS, который можно найти в компьютерах Chromebook. Это система Linux, использующая многие из технологий, описанных в данной главе, но с главным акцентом на браузеры Chromium/Chrome. Многое из того, что есть на традиционном рабочем столе, урезано в версии Chrome OS.

# 15 Инструменты разработчика

Операционные системы Linux и Unix очень популярны среди программистов не только благодаря впечатляющему набору инструментов и доступных сред, но также потому, что система является исключительно хорошо документированной и прозрачной. При работе на компьютере с Linux вам не обязательно быть программистом, чтобы использовать преимущества инструментов разработки, однако вам следует знать об инструментах программирования, поскольку они играют более важную роль в управлении системами Unix, если сравнивать с другими операционными системами. В конечном итоге вы должны уметь идентифицировать утилиты для разработки, а также иметь некоторое представление о том, как их запускать.

В данной главе большой объем информации собран в небольшом пространстве, но вам не обязательно осваивать все, что здесь упоминается. Можно поверхностно ознакомиться с ним и вернуться к нему чуть позже. Вопрос, касающийся совместно используемых библиотек, вероятно, самое важное из того, что вам необходимо знать. Однако, чтобы понять, откуда берутся эти библиотеки, сначала следует получить основные сведения о том, как создаются программы.

## 15.1. Компилятор C

Знание того, как запустить компилятор языка программирования C, поможет вам получить серьезное представление о происхождении тех команд, которые вы встречаете в Linux. Исходный программный код большинства утилит Linux, а также многих приложений для этой системы написан на языке C или C++. В основном мы будем использовать в этой главе примеры на языке C, но вы сможете применить эту информацию и для языка C++.

Программы на языке C создаются традиционным для разработки способом: вы пишете программу, компилируете ее, а затем запускаете. То есть, когда вы пишете программу на языке C и желаете ее запустить, вы должны *скомпилировать* исходный код, превратив его в низкоуровневое двоичное представление, которое понимает компьютер. Можете сопоставить это с языками сценариев (о них мы поговорим чуть позже), где вам не придется ничего компилировать.

ПРИМЕЧАНИЕ

По умолчанию в большинстве версий системы нет инструментов, необходимых для компилирования кода на языке C, поскольку такие инструменты занимают довольно много пространства. Если вам не удастся обнаружить некоторые из описанных здесь инструментов, можно попробовать установить необходимый для конкретного релиза Debian/Ubuntu пакет или применить групповую установку для Fedora/CentOS с помощью менеджера yum. Если это не завершится успехом, попробуйте поискать пакет по запросу «C compiler».

Исполняемый файл компилятора C в большинстве версий систем Unix является компилятором GNU C, `gcc`, хотя новый компилятор `clang`, разработанный группой LLVM, набирает популярность. Файлы программного кода на языке C имеют расширение `.c`. Взгляните на одиночный модульный файл `hello.c` с кодом на языке C, который можно найти в книге Брайана У. Кернигана (Brian W. Kernighan) и Денниса М. Ритчи (Dennis M. Ritchie) *The C Programming Language* («Язык программирования C»), 2-е издание (Prentice Hall, 1988):

```
#include <stdio.h>

main() {
    printf("Hello, World.\n");
}
```

Поместите этот код в файл с названием `hello.c`, а затем запустите такую команду:

```
$ cc hello.c
```

В результате появится исполняемый файл с именем `a.out`, который можно запустить подобно любому другому исполняемому файлу системы. Однако следует присвоить этому исполняемому файлу другое имя (например, `hello`). Чтобы это сделать, используйте параметр компилятора `-o`:

```
$ cc -o hello hello.c
```

Для небольших программ компилировать больше нечего. Может понадобиться добавить каталог включаемых файлов или библиотеку (см. подразделы 15.1.2 и 15.1.3), но прежде, чем переходить к этим темам, посмотрим на программы, которые немного больше по объему.

## 15.1.1. Исходный код в виде нескольких файлов

Большинство программ на языке C слишком велики, чтобы уместиться в пределах единственного файла с исходным кодом. Исполинские файлы становятся неуправляемыми для программиста, а компиляторы иногда даже испытывают сложности при синтаксическом анализе больших файлов. По этой причине разработчики группируют компоненты исходного кода вместе, предоставляя каждому фрагменту отдельный файл.

При компиляции большинства файлов `.c` исполняемый файл создается не сразу. Вместо этого сначала используется параметр компилятора `-c` для каждого файла, чтобы создать *объектные файлы*. Чтобы понять, как это устроено, предположим,

что у вас есть два файла, `main.c` и `aux.c`. Следующие две команды для компилятора выполняют основную часть работы по созданию программы:

```
$ cc -c main.c
$ cc -c aux.c
```

Эти две команды компилируют два файла источника в два объектных файла: `main.o` и `aux.o`.

Объектный файл является двоичным файлом, который процессор уже почти готов понять, если учесть еще несколько моментов. Во-первых, операционная система не знает, как запускать объектные файлы, а во-вторых, вам, вероятно, потребуется скомбинировать несколько объектных файлов и системных библиотек, чтобы создать завершенную программу.

Чтобы создать полностью функционирующий исполняемый файл из одного или нескольких объектных файлов, следует запустить компоновщик, команду `ld` в Unix. Программисты редко используют эту команду в командной строке, поскольку компилятор C знает, как запускать компоновщик. Для создания исполняемого файла с названием `myprog` из двух приведенных выше объектных файлов запустите такую команду:

```
$ cc -o myprog main.o aux.o
```

Хотя и возможно скомпилировать несколько исходных файлов вручную, как показано в этом примере, трудно отслеживать их во время компиляции, если число таких файлов велико. Утилита `make`, описанная в разделе 15.2, является стандартом Unix для управления компиляцией. Эта утилита особенно важна при управлении файлами, описанными в следующих двух разделах.

## 15.1.2. Заголовочные файлы (Include) и каталоги

*Заголовочные файлы* C являются дополнительными файлами с исходным кодом, который обычно содержит объявления типов и библиотечных функций. Например, файл `stdio.h` является заголовочным (см. простую программу в разделе 15.1).

К сожалению, с заголовочными файлами связано большое число проблем при компиляции. Большинство глюков возникает, когда компилятор не может отыскать заголовочные файлы и библиотеки. Бывают даже случаи, когда программист забывает подключить необходимый заголовочный файл, это приводит к тому, что исходный код не компилируется.

### Исправление проблем, вызванных включаемыми файлами

Отследить правильный включаемый файл не всегда легко. Иногда несколько включаемых файлов с одинаковыми именами расположены в разных каталогах и неясно, какой из них правильный. Когда компилятор не может обнаружить включаемый файл, сообщение об ошибке выглядит так:

```
badinclude.c:1:22: fatal error: notfound.h: No such file or directory
```

Это сообщение говорит о том, что компилятор не может найти заголовочный файл notfound.h, на который ссылается файл badinclude.c. Эта ошибка является прямым следствием такой директивы в первой строке файла badinclude.c:

```
#include <notfound.h>
```

По умолчанию в Unix каталогом для включаемых файлов является /usr/include; компилятор всегда просматривает его, если вы явно не укажете ему не выполнять этого. Тем не менее можно настроить компилятор так, чтобы он просматривал другие каталоги (большинство каталогов с заголовочными файлами содержит слово include где-либо в своем имени).

ПРИМЕЧАНИЕ

Из главы 16 вы узнаете о том, как отыскать отсутствующие включаемые файлы.

Предположим, вы обнаружили файл notfound.h в каталоге /usr/junk/include. Можно сделать так, чтобы компилятор видел этот каталог с помощью параметра -I:

```
$ cc -c -I/usr/junk/include badinclude.c
```

Теперь компилятор не должен спотыкаться на строке кода в файле badinclude.c, которая ссылается на заголовочный файл.

Следует также опасаться включаемых файлов, использующих двойные кавычки (" ") вместо угловых скобок (< >), например так:

```
#include "myheader.h"
```

Двойные кавычки означают, что заголовочный файл не располагается в системном каталоге для включаемых файлов и компилятору следует поискать его путь. Часто это говорит о том, что включаемый файл находится в том же каталоге, что и файл с исходным кодом. Если вам встретится проблема с двойными кавычками, то, вероятно, вы пытаетесь скомпилировать неполный исходный код.

## Что такое препроцессор C (cpp)?

Оказывается, компилятор C не выполняет работу по отыскиванию всех этих включаемых файлов. Эта задача приходится на долю *препроцессора* C — команды, которую компилятор применяет к исходному коду, прежде чем выполнить синтаксический анализ реальной программы. Препроцессор перезаписывает исходный код в такой форме, которую способен понять компилятор; это инструмент, делающий исходный код более легким для чтения (и снабжающий его обходными маневрами).

Команды препроцессора в исходном коде называются *директивами*, они начинаются с символа #. Существуют три основных типа директив.

- **Включаемые файлы.** Директива #include дает препроцессору указание о том, чтобы он включил весь файл. Обратите внимание на то, что флаг компилятора -I является в действительности параметром, который вынуждает препроцессор искать включаемые файлы в указанном каталоге, как вы видели в предыдущем разделе.

- **Макроопределения.** Строка, подобная `#define BLAH something`, говорит препроцессору о том, чтобы он выполнил замену всех вхождений элемента `BLAH` на элемент `something` в исходном коде. По соглашению названия макроопределений даются прописными буквами, но не следует удивляться тому, что программисты иногда используют макроопределения, имена которых похожи на функции и переменные. Сплошь и рядом это причиняет массу неприятностей. Многие программисты превращают в спорт неправильное использование препроцессора.

ПРИМЕЧАНИЕ

Вместо того чтобы приводить макроопределения в исходном коде, можно также передавать параметры в компилятор: команда -DBLAH=something будет работать так же, как приведенная выше директива.

- **Условные операторы.** Можно пометить отдельные фрагменты кода с помощью слов `#ifdef`, `#if` и `#endif`. Директива `#ifdef` *`MACRO`* проверяет, определено ли макроопределение *`MACRO`* для препроцессора, а директива `#if` *`condition`* проверяет, является ли результат условия *`condition`* ненулевым. Для обеих директив в том случае, когда условие, следующее за директивой `if`, является ложным, препроцессор не передает компилятору текст программы, который расположен между директивами `#if` и `#endif`. Следует привыкнуть к этому, если вы собираетесь исследовать какой-либо код на языке C.

Приведу далее пример условной директивы. Когда препроцессор встречает такой код, он проверяет, есть ли макроопределение `DEBUG`, и если оно определено, передает компилятору строку, содержащую команду `fprintf()`. В противном случае препроцессор пропускает эту строку и продолжает обработку файла после директивы `#endif`:

```
#ifdef DEBUG
  fprintf(stderr, "This is a debugging message.\n");
#endif
```

ПРИМЕЧАНИЕ

Препроцессор C ничего не знает о синтаксисе языка C, переменных, функциях и других элементах. Он понимает только свои собственные макроопределения и директивы.

В Unix препроцессор C называется `cpp`, но можно также запускать его с помощью команды `gcc -E`. Однако вам нечасто понадобится запускать препроцессор как таковой.

## 15.1.3. Связывание с библиотеками

Компилятор C знает о вашей системе недостаточно для того, чтобы самостоятельно создать пригодную программу. Для построения завершенных программ вам необходимы *библиотеки*. Библиотека C является набором распространенных, заранее скомпилированных функций, которые можно встраивать в программу. Например, многие исполняемые файлы используют библиотеку math, поскольку она обеспечивает работу с тригонометрическими и другими функциями.

Библиотеки вступают в игру главным образом во время компоновки, когда программа-компоновщик создает исполняемый файл из объектных файлов. Например, если у вас есть программа, которая использует библиотеку gobject, но вы забыли указать компилятору о связывании с этой библиотекой, то появятся ошибки компоновщика, подобные этой:

```
badobject.o(.text+0x28): undefined reference to 'g_object_new'
```

Наиболее важные части этого сообщения выделены жирным шрифтом. Когда компоновщик проверял объектный файл `badobject.o`, ему не удалось найти функцию, которая выделена жирным шрифтом, и, как следствие, не удалось создать исполняемый файл. В данном частном случае можно предположить, что вы забыли о библиотеке gobject, поскольку отсутствующая функция называется `g_object_new()`.

ПРИМЕЧАНИЕ

Неопределенные ссылки не всегда означают, что вы упустили библиотеку. Один из объектных файлов команды может отсутствовать в команде компоновки. Но обычно достаточно легко понять, что отсутствует — библиотечные функции или объектные файлы.

Чтобы исправить эту ошибку, сначала вы должны отыскать библиотеку gobject, а затем использовать параметр компилятора `-l`, чтобы установить связь с библиотекой. Так же как и включаемые файлы, библиотеки разбросаны по всей системе (по умолчанию используется каталог `/usr/lib`), хотя большинство из них расположено в подкаталоге `lib`. В предыдущем примере основным файлом библиотеки gobject является `libgobject.a`, поэтому имя библиотеки — `gobject`. Объединяя все это, можно выполнить компоновку команды следующим образом:

```
$ cc -o badobject badobject.o -lgobject
```

Вы должны сообщать компоновщику о нестандартном расположении библиотеки; для этого применяется параметр `-L`. Допустим, что команде `badobject` необходим файл `libcrud.a` в каталоге `/usr/junk/lib`. Чтобы выполнить компиляцию и создать исполняемый файл, используйте команду, подобную этой:

```
$ cc -o badobject badobject.o -lgobject -L/usr/junk/lib -lcrud
```

ПРИМЕЧАНИЕ

Если вам необходимо отыскать в библиотеке некоторую функцию, применяйте команду nm. Будьте готовы к обширному отчету. Попробуйте, например, такую команду: nm libgobject.a. Вам может понадобиться команда locate, чтобы отыскать файл libgobject.a; многие версии системы теперь помещают библиотеки в подкаталоги, зависящие от архитектуры, внутри каталога /usr/lib.

## 15.1.4. Совместно используемые библиотеки

Библиотека, имя файла которой оканчивается на `.a` (например, `libgobject.a`), называется *статической библиотекой*. Когда вы связываете команду со статической библиотекой, компоновщик копирует машинный код из библиотеки в исполняемый файл. Следовательно, для работы окончательного исполняемого файла не требуется наличие исходного файла библиотеки и, более того, поведение исполняемого файла никогда не меняется.

Однако размеры библиотек постоянно возрастают, равно как и количество используемых библиотек; это делает статические библиотеки неэкономными с точки зрения использования дискового пространства и памяти. Кроме того, если впоследствии обнаружится, что статическая библиотека реализована неадекватно или является незащищенной, то нет никакого способа исправить связанный с ней исполняемый файл, кроме повторной компиляции.

*Совместно используемые библиотеки* избавляют от таких проблем. Когда вы запускаете команду, связанную с такой библиотекой, система загружает код библиотеки в область памяти процесса только тогда, когда это требуется. Несколько процессов могут совместно использовать один и тот же код библиотеки в памяти. Если вам потребуется немного изменить код библиотеки, то обычно это можно выполнить, не компилируя заново другие команды.

У совместно используемых библиотек есть свои издержки: трудность управления и довольно сложная процедура связывания. Тем не менее можно взять такие библиотеки под контроль, если вы будет знать четыре вещи.

- Как получить список совместно используемых библиотек, необходимых исполняемому файлу.
- Как исполняемый файл отыскивает совместно используемые библиотеки.
- Как связать команду с совместно используемой библиотекой.
- Подводные камни при использовании таких библиотек.

В следующих разделах рассказано о том, как применять и обслуживать совместно используемые библиотеки в вашей системе. Если вам интересно, как устроены эти библиотеки, или вы желаете узнать о компоновщиках в целом, обратитесь к книге Джона Р. Ливайна (John R. Levine) *Linkers and Loaders* («Компоновщики и загрузчики», Morgan Kaufmann, 1999) или к статье Дэвида М. Бизли (David M. Beazley), Брайана Д. Уарда (Brian D. Ward) и Йена Р. Кука (Ian R. Cooke) The Inside Story on Shared Libraries and Dynamic Loading («Внутренняя история совместно используемых библиотек и динамической загрузки», журнал Computing in Science & Engineering, сентябрь/октябрь 2001), а также к таким онлайн-ресурсам, как Program Library HOWTO (http://dwheeler.com/program-library/). Стоит также прочитать страницу руководства ld.so(8).

## Вывод зависимостей совместно используемой библиотеки

Файлы совместно используемой библиотеки обычно размещаются там же, где и статические библиотеки. Двумя стандартными каталогами для библиотек в системе Linux являются `/lib` и `/usr/lib`. Каталог `/lib` не должен содержать статических библиотек.

Имя файла совместно используемой библиотеки содержит суффикс `.so` (shared object — «совместно используемый объект»), как, например, у файлов `libc-2.15.so` и `libc.so.6`. Чтобы увидеть, какие совместно используемые библиотеки применяет команда, запустите команду `ldd` *prog* (параметр *prog* — это имя исполняемого файла). Вот пример для команды оболочки:

```
$ ldd /bin/bash
    linux-gate.so.1 => (0xb7799000)
```

```
libtinfo.so.5 => /lib/i386-linux-gnu/libtinfo.so.5 (0xb7765000)
libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb7760000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75b5000)
/lib/ld-linux.so.2 (0xb779a000)
```

В интересах оптимального быстродействия и гибкости исполняемые файлы обычно не знают о расположении своих совместно используемых библиотек; они знают лишь их названия и, возможно, немного о том, где их искать. Небольшая команда `ld.so` (динамический компоновщик/загрузчик времени исполнения) отыскивает и загружает совместно используемые библиотеки для команды во время исполнения. В приведенном выше отчете команды `ldd` имена библиотек показаны слева — именно они известны исполняемому файлу. Правая часть показывает, где команда `ld.so` ищет данную библиотеку.

Последняя строка приведенного отчета дает актуальное местоположение команды `ld.so`: `ld-linux.so.2`.

## Как команда ld.so отыскивает совместно используемые библиотеки

Одной распространенной проблемой совместно используемых библиотек является то, что динамический компоновщик не может отыскать библиотеку. Первое местоположение, в котором компоновщику обычно *следует* искать совместно используемые библиотеки, — это заранее сконфигурированный *путь поиска библиотеки времени исполнения (rpath)* для исполняемого файла, если такой путь существует. О том, как его определить, вы вскоре узнаете.

Далее динамический компоновщик смотрит в системный кэш `/etc/ld.so.cache`, чтобы понять, находится ли библиотека в стандартном месте расположения. Это быстрый кэш имен файлов библиотек, найденных в каталогах, которые перечислены в файле конфигурации `/etc/ld.so.conf`.

ПРИМЕЧАНИЕ

Типичным для файла ld.so.conf, как и для многих файлов конфигурации Linux, которые вы уже видели, является то, что он может включать некоторые файлы из такого каталога, как /etc/ld.so.conf.d.

Каждая строка файла `ld.so.conf` является каталогом, который вы можете включить в кэш. Перечень каталогов обычно короткий и содержит нечто вроде этого:

```
/lib/i686-linux-gnu
/usr/lib/i686-linux-gnu
```

Каталоги стандартных библиотек `/lib` и `/usr/lib` являются неявными, это означает, что их не нужно включать в файл `/etc/ld.so.conf`.

Если изменить файл `ld.so.conf` или сделать изменения в одном из каталогов совместно используемых библиотек, необходимо перестроить файл `/etc/ld.so.cache` вручную с помощью следующей команды:

```
# ldconfig -v
```

Параметр `-v` сообщает детальную информацию о библиотеках (которую команда `ldconfig` добавляет в кэш), а также информацию о любых обнаруженных изменениях.

Есть еще одно место, где команда ld.so ищет совместно используемые библиотеки: переменная окружения LD_LIBRARY_PATH. Вскоре мы поговорим о ней.

Не добавляйте что-либо в файл /etc/ld.so.conf. Вы должны знать, какие совместно используемые библиотеки есть в системном кэше, а если помещать в кэш каждую непонятную маленькую библиотеку, могут возникнуть конфликты и система станет крайне неорганизованной. Когда выполняется компиляция программы, которой необходим нестандартный путь к библиотеке, передайте исполняемому файлу встроенный путь поиска библиотеки времени исполнения. Посмотрим, как это делается.

## Связывание программ с совместно используемыми библиотеками

Допустим, у вас есть совместно используемая библиотека с именем libweird.so.1 в каталоге /opt/obscure/lib, с которой вам необходимо связать программу myprog. Выполните это следующим образом:

```
$ cc -o myprog myprog.o -Wl,-rpath=/opt/obscure/lib -L/opt/obscure/lib -lweird
```

Параметр -Wl,-rpath сообщает компоновщику о том, чтобы он включил следующий каталог в путь поиска библиотеки времени исполнения для исполняемого файла. Тем не менее, даже если вы используете параметр -Wl,-rpath, флаг -L по-прежнему необходим.

Если у вас уже есть готовый двоичный файл, можно применять команду patchelf, чтобы вставить в него другой путь поиска библиотеки времени исполнения, но обычно лучше делать это во время компиляции.

## Проблемы, вызванные совместно используемыми библиотеками

Совместно используемые библиотеки обеспечивают замечательную гибкость, не говоря уже о некоторых действительно превосходных решениях, но в то же время их легко применить неправильно до такой степени, что ваша система превращается в полную кашу. Могут возникнуть три чрезвычайно неприятные ситуации:

- отсутствие библиотек;
- ужасное быстродействие;
- несоответствие библиотек.

Первой причиной всех проблем с совместно используемыми библиотеками является переменная окружения LD_LIBRARY_PATH. Если в этой переменной указать перечень имен каталогов, разделенных с помощью двоеточия, то тогда команда ld.so выполнит поиск в указанных каталогах *прежде* поиска совместно используемых библиотек где-либо еще. Это легкий способ заставить ваши программы работать, если вы переместили библиотеку и у вас нет исходного кода программы или же вы не можете использовать команду patchelf, а возможно, просто ленитесь заново компилировать исполняемые файлы. К сожалению, вы получаете то, за что платите.

*Никогда* не определяйте переменную LD_LIBRARY_PATH в файлах запуска оболочки или при компиляции программ. Когда динамический компоновщик встречает эту

переменную, ему зачастую приходится просматривать содержимое каждого указанного каталога большее число раз, чем вы могли себе представить. Это сильно сказывается на быстродействии, но, что более важно, могут возникнуть конфликты и несоответствия библиотек, поскольку компоновщик времени исполнения просматривает эти каталоги для *каждой* программы.

Если вы *должны* использовать переменную `LD_LIBRARY_PATH`, чтобы запустить какую-либо программу, для которой у вас нет исходного кода (или приложение, которое вы предпочли бы не компилировать, вроде Mozilla или каких-либо других), применяйте сценарий обертки. Допустим, исполняемому файлу `/opt/crummy/bin/crummy.bin` необходимы совместно используемые библиотеки из каталога `/opt/crummy/lib`. Напишите сценарий обертки с именем `crummy` следующим образом:

```
#!/bin/sh
LD_LIBRARY_PATH=/opt/crummy/lib
export LD_LIBRARY_PATH
exec /opt/crummy/bin/crummy.bin $@
```

Если избегать переменной `LD_LIBRARY_PATH`, то можно предотвратить большинство проблем с совместно используемыми библиотеками. Иногда возникает еще одна существенная проблема для разработчиков: интерфейс прикладного программирования (API) для какой-либо библиотеки может немного измениться при переходе от одной младшей версии к другой, это нарушит работу установленных программ. Лучшие решения проблемы являются профилактическими: либо пользуйтесь последовательной методологией, чтобы установить совместно используемые библиотеки с помощью команды `-Wl,-rpath` для создания ссылки на путь времени исполнения, либо просто применяйте статические версии непонятных библиотек.

## 15.2. Утилита make

Программа, у которой есть несколько файлов исходного кода или для которой необходимы необычные параметры компиляции, слишком неудобна для компиляции вручную. Эта проблема возникает на протяжении многих лет, и традиционным средством Unix для управления компиляцией является утилита `make`. Вам следует узнать немного об этой утилите, если вы работаете в системе Unix, поскольку системные утилиты иногда опираются на нее в своей работе. Однако данная глава является лишь верхушкой айсберга. Утилите `make` посвящены целые книги, например *Managing Projects with GNU Make* («Управление проектами с помощью утилиты GNU Make») Роберта Мекленбурга (Robert Mecklenburg) (O'Reilly, 2004). Кроме того, большинство пакетов Linux собрано с использованием дополнительного уровня, основанного на утилите `make` или подобном средстве. Есть множество систем для сборки; одну из них с названием Autotools мы рассмотрим в главе 16. Утилита `make` является большой системой, но получить представление о том, как она работает, совсем не трудно. Когда вы увидите файл с именем `Makefile` или `makefile`, знайте, что вы имеете дело с утилитой `make`. Попробуйте запустить команду `make`, чтобы понять, можно ли что-нибудь собрать.

Главной идеей утилиты make является *цель* — то, чего вы желаете достичь. Целью может быть файл (файл .o, исполняемый файл и т. д.) или ярлык. Кроме того, некоторые цели зависят от других целей; например, вам необходимо создать полный набор файлов .o, прежде чем вы сможете скомпоновать исполняемый файл. Эти требования называются *зависимостями*.

Чтобы собрать цель, утилита make следует какому-либо правилу, например, определяющему, как перейти от исходного файла .c к объектному файлу .o. Утилите make уже известны некоторые правила, но вы можете изменить их, а также создать собственные.

## 15.2.1. Пример файл Makefile

Следующий очень простой файл Makefile собирает программу myprog из файлов aux.c и main.c:

```
# object files
OBJS=aux.o main.o

all: myprog

myprog: $(OBJS)
        $(CC) -o myprog $(OBJS)
```

Символ # в первой строке этого файла означает комментарий.

Следующая строка является всего лишь макроопределением; она задает для переменной OBJS два имени объектных файлов. Это будет важно в дальнейшем. Сейчас обратите внимание на то, как записывается макроопределение и как на него ссылаются далее ( ($(OBJS)) ).

Следующий элемент файла Makefile содержит первую цель, all. Первая цель всегда является целью по умолчанию, утилита make будет собирать ее, если вы запустите команду make в командной строке саму по себе.

Правило сборки цели следует после двоеточия. Для цели all в этом файле Makefile сказано, что вам необходимо удовлетворить чему-то по имени myprog. Это первая зависимость в данном файле; цель all зависит от myprog. Заметьте, что myprog может быть реальным файлом или целью другого правила. В данном случае оно является и тем и другим (правилом для цели all и целью для OBJS).

Чтобы собрать программу myprog, этот файл Makefile использует макроопределение $(OBJS) в зависимостях. Макроопределение развертывается в имена aux.o и main.o, поэтому программа myprog зависит от этих двух файлов (они должны быть реальными файлами, поскольку нигде в файле Makefile нет целей с такими именами).

Данный файл Makefile предполагает, что у вас есть два файла с исходным кодом на языке C: aux.c и main.c в одном каталоге. Если запустить утилиту make для файла Makefile, то в результате будут показаны команды, выполняемые утилитой:

```
$ make
cc        -c -o aux.o aux.c
cc        -c -o main.o main.c
cc -o myprog aux.o main.o
```

Схема зависимостей приведена на рис. 15.1.

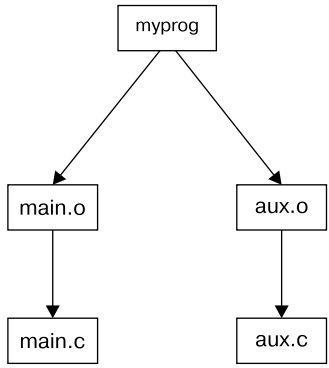


**Рис. 15.1.** Зависимости в файле Makefile

## 15.2.2. Встроенные правила

Каким же образом утилита `make` узнает о том, как перейти от файла `aux.c` к файлу `aux.o`? Ведь файла `aux.c` нет внутри файла `Makefile`. Ответ такой: утилита `make` следует встроенным правилам. Она знает о том, что следует искать файл `.c`, если вам необходим файл `.o`, и, более того, она знает, как запустить команду `cc -c` для этого файла `.c`, чтобы добиться цели — создать файл `.o`.

## 15.2.3. Окончательная сборка программы

Последний шаг при получении программы `myprog` довольно хитрый, но идея достаточно ясная. Когда у вас появятся два объектных файла в макроопределении `$(OBJS)`, можно запустить компилятор C в соответствии со следующей строкой (здесь переменная `$(CC)` развертывается в имя компилятора):

```
$(CC) -o myprog $(OBJS)
```

Отступ перед переменной `$(CC)` является табуляцией. Вы *обязаны* вставлять табуляцию перед каждой реальной командой в той строке, где она находится.

Остерегайтесь следующего сообщения:

```
Makefile:7: *** missing separator. Stop.
```

Подобная ошибка означает, что файл `Makefile` не в порядке. Табуляция является разделителем, и, если она отсутствует или есть какая-либо другая помеха, вы увидите такую ошибку.

## 15.2.4. Поддержание актуальных версий файлов

Одним из последних фундаментальных свойств утилиты `make` является то, что ее цели должны соответствовать по времени зависимостям. Если вы дважды вызове-

те утилиту make из предыдущего примера, первая из них соберет программу myprog, но вторая выдаст такое сообщение:

```
make: Nothing to be done for 'all'.
```

Во второй раз утилита make посмотрела свои правила и обнаружила, что программа myprog уже существует, поэтому она не стала повторно собирать ее, поскольку ни одна из зависимостей не изменилась с тех пор, как была собрана программа. Чтобы поэкспериментировать, выполните следующее.

1. Запустите команду touch aux.c.
2. Запустите утилиту make еще раз. На этот раз она определит, что файл aux.c является более «свежим», чем файл aux.o, уже находящийся в каталоге, поэтому она скомпилирует файл aux.o заново.
3. Программа myprog зависит от файла aux.o, и теперь файл aux.o является более новым, чем уже существующая программа myprog, поэтому утилита make должна создать программу myprog заново.

   «Цепные реакции» такого типа весьма характерны.

## 15.2.5. Аргументы и параметры командной строки

Вы можете извлечь существенную пользу от утилиты make, если будете знать, как работают ее аргументы и параметры командной строки.

Один из самых полезных параметров позволяет указать единичную цель в командной строке. Для предыдущего файла Makefile можно запустить команду make aux.o, если вам необходим только файл aux.o.

Можно также создать макроопределение в командной строке. Чтобы, например, использовать компилятор clang, попробуйте такую команду:

```
$ make CC=clang
```

Здесь утилита make использует ваше определение переменной CC вместо собственного компилятора по умолчанию cc. Макроопределения из командной строки становятся полезны при тестировании определений и библиотек препроцессора, в особенности с макроопределениями CFLAGS и LDFLAGS, о которых мы вкратце поговорим.

На самом деле вам даже не требуется файл Makefile для запуска утилиты make. Если встроенные в утилиту правила соответствуют цели, можно просто попросить утилиту создать цель. Например, если у вас есть исходный код для очень простой программы blah.c, попробуйте запустить команду make blah. Работа утилиты будет выглядеть так:

```
$ make blah
cc   blah.o -o blah
```

Подобный вариант использования утилиты подходит только для очень простых программ. Если же вашей программе необходимы библиотека или специальный каталог для включаемых файлов, то, вероятно, потребуется написать файл Makefile. Запуск утилиты без файла Makefile в действительности наиболее полезен

в том случае, когда вы имеете дело с чем-либо вроде языка Fortran, Lex или Yacc и не знаете, как работает компилятор или утилита. Почему бы не дать возможность утилите make выяснить это за вас? Даже если ей не удастся создать цель, она, вероятно, снабдит вас очень хорошей подсказкой о том, как применять данный инструмент.

Среди ряда параметров утилиты make особо выделяются следующие:

- -n — выводит список команд, которые необходимы для сборки, но удерживает утилиту make от запуска каких-либо команд;
- -f *file* — дает утилите make указание на чтение из файла *file* вместо файлов Makefile или makefile.

## 15.2.6. Стандартные макроопределения и переменные

У команды make есть много специальных макроопределений и переменных. Сложно уловить различия между макроопределением и переменной, поэтому мы будем использовать термин *«макроопределение»* для обозначения чего-либо, что обычно не изменяется, когда утилита приступает к сборке целей.

Как вы видели ранее, можно назначить макроопределения в начале файла Makefile. Вот самые распространенные из них.

- CFLAGS. Параметры компилятора C. При создании объектного кода из файла .c утилита make передает этот параметр компилятору в качестве аргумента.
- LDFLAGS. Подобны параметрам CFLAGS, но они предназначены для компоновщика при создании исполняемого файла из объектного кода.
- LDLIBS. Если вы используете параметры LDFLAGS, но не желаете комбинировать параметры имени библиотеки с путем поиска, поместите параметры имени библиотеки в этот файл.
- CC. Компилятор C. По умолчанию это команда cc.
- CPPFLAGS. Параметры *препроцессора* C. Когда утилита make каким-либо образом запускает препроцессор, она передает ему в качестве аргумента это макроопределение.
- CXXFLAGS. Утилита GNU make использует эти параметры в качестве флагов для компилятора C++.

Переменная утилиты make изменяется, когда происходит сборка целей. Поскольку вы никогда не определяете переменные утилиты make вручную, следующий перечень содержит символ $:

- $@ — внутри правила эта переменная развертывается в имя текущей цели;
- $* — развертывается в *базовое имя* текущей цели. Например, если вы собираете файл blah.o, эта переменная будет развернута в blah.

Наиболее полный перечень переменных Linux можно найти в info-руководстве к утилите make.

ПРИМЕЧАНИЕ

Помните о том, что утилита GNU make обладает множеством таких расширений, встроенных правил и функций, каких нет у других вариантов. Это замечательно, пока вы работаете в Linux, но, если вы окажетесь за компьютером с Solaris или BSD и будете рассчитывать, что все станет работать так же, вас может поджидать сюрприз. Однако с этой проблемой можно разобраться, если применить такую многоплатформенную систему сборки, как GNU autotools.

## 15.2.7. Обычные цели

Большинство файлов `Makefiles` содержит некоторые стандартные цели, которые выполняют вспомогательные задачи, относящиеся к компиляции.

- `clean`. Цель `clean` является повсеместной; команда `make clean` обычно дает утилите `make` указание удалить все объектные и исполняемые файлы, чтобы можно было начать все заново или подготовить пакет ПО. Вот пример правила для файла `Makefile` программы `myprog`:

  ```
  clean:

          rm -f $(OBJS) myprog
  ```

- `distclean`. В файле `Makefile`, созданном с помощью системы GNU autotools, всегда есть цель `distclean`, чтобы удалить все, что не является частью исходного дистрибутива, включая и сам файл `Makefile`. Подробнее об этом вы узнаете из главы 16. В очень редких случаях вы обнаружите, что разработчик предпочитает удалять исполняемые файлы не с помощью этой цели, а чем-либо вроде цели `realclean`.
- `install`. Копирует файлы и скомпилированные программы в местоположение системы, которое файл `Makefile` считает надлежащим. Это может быть опасным, поэтому всегда сначала запустите команду `make -n install`, чтобы увидеть, что произойдет, не выполняя в действительности никаких команд.
- `test` или `check`. Некоторые разработчики предусматривают цели `test` или `check`, чтобы убедиться в том, что все работает после выполнения сборки.
- `depend`. Создает зависимости, вызывая компилятор с параметром `-M` для проверки исходного кода. Эта цель выглядит необычно, поскольку часто она изменяет сам файл `Makefile`. Это больше не является общепринятой практикой, но, если вам встретятся инструкции, в которых говорится об использовании этого правила, следуйте им.
- `all`. Часто является первой целью в файле `Makefile`. Вам много раз будут попадаться ссылки на эту цель, а не на исполняемый файл.

## 15.2.8. Устройство файла Makefile

Несмотря на то что существуют различные стили файла `Makefile`, большинство программистов придерживается некоторых основных принципов его написания. Для начала в первой части файла `Makefile` (внутри макроопределений) следует указать библиотеки и включаемые файлы, сгруппированные в соответствии с пакетами:

```
MYPACKAGE_INCLUDES=-I/usr/local/include/mypackage
MYPACKAGE_LIB=-L/usr/local/lib/mypackage -lmypackage
PNG_INCLUDES=-I/usr/local/include
PNG_LIB=-L/usr/local/lib -lpng
```

Каждый тип флагов компилятора и компоновщика часто оформляется в виде таких макроопределений:

```
CFLAGS=$(CFLAGS) $(MYPACKAGE_INCLUDES) $(PNG_INCLUDES)
LDFLAGS=$(LDFLAGS) $(MYPACKAGE_LIB) $(PNG_LIB)
```

Объектные файлы обычно группируются в соответствии с исполняемыми файлами. Допустим, например, что у вас есть пакет, который создает исполняемые файлы `boring` и `trite`. У каждого из них есть файл `.c` с исходным кодом и необходимый код в файле `util.c`. Вы можете увидеть нечто вроде следующего:

```
UTIL_OBJS=util.o

BORING_OBJS=$(UTIL_OBJS) boring.o
TRITE_OBJS=$(UTIL_OBJS) trite.o

PROGS=boring trite
```

Остальная часть файла `Makefile` могла бы выглядеть так:

```
all: $(PROGS)

boring: $(BORING_OBJS)
        $(CC) -o $@ $(BORING_OBJS) $(LDFLAGS)
trite: $(TRITE_OBJS)
       $(CC) -o $@ $(TRITE_OBJS) $(LDFLAGS)
```

Вы могли бы скомбинировать две цели для исполняемых файлов в виде одного правила, но обычно так поступать не следует, поскольку вам было бы непросто перенести правило в другой файл `Makefile` и удалить исполняемый файл или группу исполняемых файлов в отдельности. Более того, зависимости стали бы некорректными: если бы у вас было лишь одно правило для файлов `boring` и `trite`, файл `trite` зависел бы от файла `boring.c`, файл `boring` — от файла `trite.c`, и утилита `make` всегда пыталась бы собрать заново обе программы, если вы изменили один из файлов с исходным кодом.

ПРИМЕЧАНИЕ

Если вам необходимо определить специальное правило для объектного файла, поместите такое правило непосредственно над правилом, которое задает сборку исполняемого файла. Если несколько исполняемых файлов используют один и тот же объектный файл, поместите правило для объектного файла над всеми правилами для исполняемых файлов.

# 15.3. Отладчики

Стандартным отладчиком в системах Linux является `gdb`; доступны также системы с дружественным к пользователю интерфейсом, например Eclipse IDE и Emacs. Чтобы включить полную отладку ваших программ, запустите компилятор с пара-

метром -g для записи таблицы имен и другой отладочной информации в исполняемый файл. Чтобы запустить отладчик gdb для исполняемого файла *program*, выполните такую команду:

```
$ gdb program
```

Вы должны получить приглашение (gdb). Чтобы запустить программу *program* с параметром командной строки *options*, введите следующую команду после приглашения отладчика:

```
(gdb) run options
```

Если программа в порядке, она должна запускаться, работать и завершать выполнение нормально. Однако если возникает проблема, отладчик gdb останавливается, выводит ошибочный исходный код и возвращает вас в строку приглашения (gdb). Поскольку фрагмент исходного кода часто содержит подсказку о причине проблемы, вам может потребоваться вывести значение какой-либо переменной, с которой связана ошибка. Команда print работает также для массивов и структур языка C.

```
(gdb) print variable
```

Чтобы отладчик остановил программу в указанном месте исходного кода, используйте контрольные точки. В следующей команде файл *file* является файлом с исходным кодом, а параметр *line_num* — это номер строки этого файла, в которой отладчик должен остановиться:

```
(gdb) break file:line_num
```

Для продолжения отладки выполните такую команду:

```
(gdb) continue
```

Чтобы удалить контрольную точку, введите команду:

```
(gdb) clear file:line_num
```

Этот раздел содержит только краткое введение в отладчик gdb в надежде на то, что вы изучите более полное руководство, онлайн или в печатном виде, например 10-е издание книги Ричарда М. Столлмана (Richard M. Stallman) и др. *Debugging with GDB* («Отладка с помощью GDB», GNU Press, 2011). Еще одним руководством по отладке является книга Нормана Матлофа (Norman Matloff) и Питера Джея Зальцмана (Peter Jay Salzman) *The Art of Debugging* («Искусство отладки», No Starch Press, 2008).

ПРИМЕЧАНИЕ

Если вам интересно выявление проблем в памяти и запуск профильных тестов, посетите сайт проекта Valgrind (http://valgrind.org/).

# 15.4. Инструменты Lex и Yacc

Инструменты Lex и Yacc могли встретиться вам при компиляции программ, которые читают файлы конфигурации или команды. Эти инструменты являются строительными блоками для языков программирования.

- Lex — это *разметчик (tokenizer)*, который переводит текст в пронумерованные теги с ярлыками. Версия GNU/Linux для этого инструмента называется `flex`. Для его совместной работы с компоновщиком могут потребоваться флаги `-ll` или `-lfl`.
- Yacc — это *синтаксический анализатор*, который пытается считывать метки в соответствии с *грамматикой*. Анализатор GNU называется `bison`; для его совместимости с Yacc запустите команду `bison -y`. Может потребоваться флаг компоновщика `-ly`.

# 15.5. Языки сценариев

В давние времена обычному системному администратору Unix не приходилось особенно беспокоиться насчет других языков сценариев, кроме Bourne shell и awk. Сценарии оболочки (рассмотренные в главе 11) по-прежнему остаются важной частью системы Unix, но язык awk понемногу сходит со сценарной арены. В то же время появились его мощные наследники, и теперь многие системные команды созданы не на языке C, а на языках сценариев (например, практичная версия команды `whois`). Рассмотрим некоторые основы сценариев.

Для начала вам необходимо знать о любом языке сценариев следующее: первая строка сценария выглядит так же, как и в сценарии оболочки Bourne shell. Например, сценарий на языке Python начинается так:

```
#!/usr/bin/python
```

Или так:

```
#!/usr/bin/env python
```

В Linux любой исполняемый текстовый файл, начинающийся символами `#!`, является сценарием. Путь, который следует за этим префиксом, представляет исполняемый файл интерпретатора языка сценариев. Когда Unix пытается запустить исполняемый файл, который начинается с символов #!, она выполняет следующую за ним команду, используя оставшуюся часть файла как стандартный ввод. Следовательно, даже такой код является сценарием:

```
#!/usr/bin/tail -2
This program won't print this line,
but it will print this line...
and this line, too.
```

Первая строка сценария оболочки часто содержит одну из самых распространенных проблем со сценариями: неверный путь к интерпретатору языка сценариев. Допустим, вы назвали предыдущий сценарий `myscript`. Что будет, если команда `tail` на самом деле находится в вашей системе в каталоге `/bin` вместо `/usr/bin`? В этом случае запуск сценария `myscript` вызвал бы такую ошибку:

```
bash: ./myscript: /usr/bin/tail: bad interpreter: No such file or directory
```

Не рассчитывайте на то, что в первой строке сценария будет работать несколько аргументов. То есть аргумент `-2` в приведенном примере мог бы сработать, но

если вы добавите еще один аргумент, то система могла бы расценивать -2 и этот новый аргумент как один большой аргумент с пробелами, и все. Это может быть различным для разных систем, поэтому не испытывайте свое терпение на таких малозначащих вещах, как эта.

Теперь рассмотрим некоторые языки.

## 15.5.1. Python

Python — это язык сценариев с хорошим сопровождением и набором мощных функций, таких как обработка текста, доступ к базам данных, работа с сетью и многопоточный режим. Он обладает производительным интерактивным режимом и хорошо организованной объектной моделью.

Его исполняемый файл называется `python` и обычно помещается в каталоге `/usr/bin`. Тем не менее язык Python применяется не только для создания сценариев командной строки. Его можно встретить также в качестве инструмента для создания сайтов. Замечательным справочником, который содержит в начале небольшое руководство, может стать 4-е издание книги Дэвида М. Бизли (David M. Beazley) *Python Essential Reference* («Основной справочник по языку Python», Addison-Wesley, 2009).

## 15.5.2. Perl

Одним из старейших языков сценариев Unix, разработанным независимо, является Perl. Это действительно «швейцарский армейский нож» среди инструментов программирования. Хотя язык Perl в последние годы уступил часть позиций языку Python, он превосходен, в частности, при обработке текста, конвертации файлов и работе с ними. Вы можете встретить много инструментов, созданных с его помощью. Введением, изложенным в стиле учебника, может послужить 6-е издание книги Рэндала Л. Шварца (Randal L. Schwartz), Брайана Д. Фойя (Brian D. Foy) и Тома Феникса (Tom Phoenix) *Learning Perl* («Осваиваем язык Perl», O'Reilly, 2011). Более полным справочником является книга *Modern Perl* («Современный язык Perl») группы Chromatic (Onyx Neon Press, 2014).

## 15.5.3. Другие языки сценариев

Вы можете встретить также следующие языки сценариев.

- **PHP.** Этот язык обработки гипертекста часто можно увидеть в динамических веб-сценариях. Некоторые пользователя применяют язык PHP для автономных сценариев. Сайт проекта PHP — http://www.php.net/.
- **Ruby.** Приверженцы объектно-ориентированного подхода и веб-разработчики обожают программировать на этом языке (http://www.ruby-lang.org/).
- **JavaScript.** Этот язык применяется в браузерах в основном для работы с динамическим содержимым. Наиболее искушенные программисты остерегаются использовать его как средство для создания автономных сценариев из-за его многочисленных недостатков, но этого практически невозможно избежать

в веб-программировании. В своей системе вы сможете отыскать его реализацию с именем Node.js и исполняемым файлом `node`.

- **Emacs Lisp.** Разновидность языка программирования Lisp, которая используется в текстовом редакторе Emacs.
- **Matlab, Octave.** Matlab является платным языком программирования и библиотекой для матричных и математических вычислений. Существует очень похожий на него бесплатный проект Octave.
- **R.** Популярный бесплатный язык для статистического анализа. Дополнительную информацию можно получить на сайте http://www.r-project.org/ и в книге Нормана Матлофа (Norman Matloff) *The Art of R Programming* («Искусство программирования на языке R», No Starch Press, 2011).
- **Mathematica.** Еще один коммерческий математический язык программирования с библиотеками.
- **m4.** Это язык обработки макроопределений, который обычно находится только в утилите GNU autotools.
- **Tcl** (Tool command language, инструментальный командный язык). Это простой язык сценариев, обычно ассоциируемый с инструментарием графического интерфейса пользователя Tk и утилитой автоматизации Expect. Хотя язык Tcl уже не столь широко распространен, как раньше, не пренебрегайте его возможностями. Многие бывалые разработчики предпочитают использовать язык Tk, в особенности за его внедренные возможности. Подробности можно узнать на сайте http://www.tcl.tk/.

## 15.6. Java

Язык Java является транслируемым, подобно языку C, с более простым синтаксисом и мощной поддержкой объектно-ориентированного программирования. В системах Unix у него несколько специальных применений. Так, он часто используется как среда для веб-приложений, а также популярен для специальных программ. Например, приложения для платформы Android обычно написаны на языке Java. Хотя он и не часто встречается в типичном рабочем столе Linux, вы должны знать, как он работает, по меньшей мере в автономных приложениях.

Есть два вида компиляторов Java: собственные компиляторы, создающие машинный код для вашей системы (подобно компилятору C), и компиляторы байт-кода для его использования в интерпретаторе байт-кода (иногда называемом *виртуальной машиной*, которая отличается от виртуальной машины, предлагаемой гипервизором, как описано в главе 17). В Linux вы практически всегда будете встречать байт-код.

Имена файлов байт-кода Java заканчиваются на `.class`. Среда времени исполнения Java (JRE, Java runtime environment) содержит все команды, которые необходимы вам для запуска байт-кода. Чтобы выполнить байт-код, используйте такую команду:

```
$ java file.class
```

Можно также встретить файлы байт-кода, которые оканчиваются на `.jar`; это подборки заархивированных файлов `.class`. Чтобы запустить файл `.jar`, используйте следующий синтаксис:

```
$ java -jar file.jar
```

Иногда вам потребуется указать в переменной окружения `JAVA_HOME` путь к среде Java. Если вам сильно не повезет, то придется использовать переменную `CLASSPATH`, чтобы включить все каталоги, которые содержат ожидаемые командой классы. Этот список каталогов приводится с разделителем-двоеточием, подобно обычной переменной `PATH` для исполняемых файлов.

Если вам необходимо скомпилировать файл `.java` в байт-код, потребуется набор для Java-разработки (JDK, Java Development Kit). Запустить компилятор `javac` из набора JDK для создания файлов `.class` можно так:

```
$ javac file.java
```

В набор JDK входит также команда `jar`, которая позволяет собирать файлы `.jar`. Она работает подобно команде `tar`.

## 15.7. Заглядывая вперед: компиляция программных пакетов

Мир компиляторов и языков сценариев велик и постоянно расширяется. Пока пишутся эти строки, набирают популярность новые транслируемые языки, такие как Go (golang) и Swift.

Инфраструктура компиляции LLVM (http://llvm.org/) существенно облегчила разработку компиляторов. Если вам интересно узнать о разработке и реализации компиляторов, в этом вам помогут две хорошие книги: *Compilers: Principles, Techniques and Tools* («Компиляторы: принципы, методы и инструменты») Альфреда В. Эхоу (Alfred V. Aho) (2-е издание, Addison-Wesley, 2006) и *Modern Compiler Design* («Разработка современного компилятора») Дика Грюна (Dick Grune) и др. (2-е издание, Springer, 2012). О разработке с помощью языков сценариев лучше узнавать из онлайн-ресурсов, поскольку реализации весьма различаются.

Теперь, когда вы знакомы с основами инструментов программирования в системе, вы готовы узнать, что они могут делать. Следующая глава полностью посвящена тому, как в Linux создавать пакеты программного обеспечения на основе исходного кода.

# 16 Введение в программное обеспечение для компиляции кода на языке C

Большинство общедоступных пакетов сторонних разработчиков ПО для Unix поставляется в виде исходного кода, который можно скомпоновать и установить. Одной из причин для этого является наличие такого числа различных версий и архитектур Unix (и самой Linux), что было бы затруднительно создать двоичные пакеты для всех возможных комбинаций платформ. Еще одна важная причина состоит в том, что широкое распространение исходного кода в Unix-сообществе воодушевляет пользователей на исправление ошибок в ПО и внесение новых функций, наполняя смыслом термин *«открытый исходный код»*.

Практически все, что вы видите в системе Linux, можно получить как исходный код: начиная с ядра и библиотеки C и заканчивая браузерами. Возможно даже обновить и дополнить систему в целом, (пере)установив части системы из исходного кода. Однако вам, вероятно, *не следует* обновлять свой компьютер, устанавливая *все* из исходного кода, если только вы не получаете удовольствие от этого процесса или у вас нет какой-либо другой причины.

Linux обычно обеспечивает простые способы обновления важнейших частей системы, таких как команды в каталоге `/bin`, а одним чрезвычайно важным свойством систем является то, что они обычно очень быстро устраняют проблемы в защите. Однако не ожидайте, что ваша версия обеспечит вас всем необходимым без вашего участия. Вот несколько причин, по которым может потребоваться самостоятельно установить определенные пакеты:

- чтобы контролировать параметры конфигурации;
- чтобы установить ПО туда, куда вам необходимо. Вы можете даже установить несколько разных версий одного пакета;
- чтобы управлять версией, которую вы устанавливаете. В дистрибутивах системы не всегда присутствует самая последняя версия всех пакетов, в особенности относящихся к дополнительному ПО (такому как библиотеки Python);
- чтобы лучше понимать, как работает пакет.

# 16.1. Системы для сборки программного обеспечения

В Linux есть различные среды программирования, начиная от традиционного языка C и заканчивая такими интерпретируемыми языками сценариев, как Python. У каждой из них есть по меньшей мере одна собственная система для сборки и установки пакетов в дополнение к тем инструментам, которые предлагает система Linux.

В этой главе мы рассмотрим компиляцию и установку исходного кода на языке C с помощью лишь одной их таких систем — сценариев конфигурирования, создаваемых пакетом GNU Autotools. Эта система считается стабильной, и многие основные утилиты Linux используют ее. Поскольку она основана на таких существующих инструментах, как команда `make`, вы сможете применить свои знания для других систем сборки, увидев ее в действии.

Установка пакета из исходного кода на языке C обычно включает следующие шаги.

1. Распаковку архива с исходным кодом.
2. Конфигурирование пакета.
3. Запуск команды `make` для сборки команд.
4. Запуск команды `make install` или специфичной для данной версии ОС команды, которая устанавливает пакет.

ПРИМЕЧАНИЕ

Вы должны понимать основы, изложенные в главе 15, прежде чем продолжать чтение этой главы.

# 16.2. Распаковка архива с исходным кодом на языке C

Исходный код какого-либо пакета обычно предоставляется в виде файла `.tar.gz`, `.tar.bz2` или `.tar.xz`, и вам следует распаковать этот файл, как описано в разделе 2.18. Однако перед распаковкой проверьте содержимое архива с помощью команд `tar tvf` или `tar ztvf`, поскольку некоторые пакеты не создают собственные подкаталоги в том каталоге, где вы распаковываете архив.

Отчет, подобный приводимому ниже, свидетельствует о том, что пакет в порядке и готов для распаковки:

```
package-1.23/Makefile.in
package-1.23/README
package-1.23/main.c
package-1.23/bar.c
--snip--
```

Тем не менее вы можете обнаружить, что не все файлы находятся внутри одного каталога (вроде каталога `package-1.23` из приведенного примера):

```
Makefile
README
```

```
main.c
--snip--
```

Распаковка такого архива может создать большую путаницу в вашем текущем каталоге. Чтобы избежать этого, создайте новый каталог и перейдите в него перед извлечением содержимого архива. Наконец, остерегайтесь пакетов, которые содержат файлы с абсолютными путями, вроде таких:

```
/etc/passwd
/etc/inetd.conf
```

Возможно, вам не встретится ничего подобного, но если все-таки встретится — удалите такой архив из системы. Возможно, он содержит вирус-троян или какой-либо вредоносный код.

**С чего начать.** После извлечения содержимого архива с исходным кодом и появления множества файлов перед вами попытайтесь получить представление о пакете. В частности, поищите файлы `README` и `INSTALL`. Обязательно загляните в каждый файл `README`, поскольку они часто содержат описание пакета, краткое руководство, советы по установке и другую полезную информацию. Многие пакеты сопровождаются также файлами `INSTALL`, содержащими инструкции по компиляции и установке пакета. Особое внимание уделите специальным параметрам компилятора и определениям.

В дополнение к файлам `README` и `INSTALL` вы найдете другие файлы пакета, которые можно грубо распределить по трем категориям.

- Файлы, относящиеся к команде `make`, такие как `Makefile`, `Makefile.in`, `configure` и `CMakeLists.txt`. Очень старые пакеты содержат файл `Makefile`, который вам может понадобиться изменить, но большинство пакетов использует утилиту конфигурирования, например GNU Autoconf или CMake. Они поставляются с файлом сценария или конфигурации (такими как `configure` или `CMakeLists.txt`), чтобы помочь создать файл `Makefile` из файла `Makefile.in`, основываясь на настройках вашей системы и параметрах конфигурации.
- Файлы исходного кода, имена которых оканчиваются на `.c`, `.h` или `.cc`. Файлы исходного кода на языке C могут появиться практически всюду в каталоге пакета. У файлов исходного кода на языке C++ обычно есть суффиксы `.cc`, `.C` или `.cxx`.
- Объектные файлы, имена которых оканчиваются на `.o`, или двоичные файлы. Обычно объектные файлы отсутствуют в пакетах с исходным кодом, но вы можете встретить их в редких случаях, когда поставщику пакета не разрешено распространение исходного кода и вам необходимо предпринимать что-либо особое, чтобы использовать такие объектные файлы. В большинстве случаев наличие объектных (или двоичных исполняемых) файлов в пакете означает, что он был составлен не очень хорошо и вам следует запустить команду `make clean`, чтобы убедиться в пригодности кода для компиляции.

# 16.3. Утилита GNU Autoconf

Хотя исходный код на языке C обычно довольно хорошо портируется, различия между платформами делают невозможной компиляцию большинства пакетов с по-

мощью единственной утилиты Makefile. Раньше для решения этой проблемы предлагались отдельные утилиты Makefile для каждой операционной системы или же утилита, которую легко изменять. При таком подходе приходилось задействовать сценарии, которые генерируют файлы Makefiles на основе анализа системы, использованной для сборки пакета.

Утилита GNU Autoconf является популярным средством для автоматического создания файла Makefile. Пакеты, которые используют эту систему, содержат файлы с именами configure, Makefile.in и config.h.in. Файлы с суффиксом .in являются шаблонами; идея состоит в том, чтобы запускать сценарий конфигурирования, который выявляет характеристики вашей системы, а затем выполняет подстановки в файлах .in для создания реальных файлов сборки. Для конечного пользователя это просто; чтобы создать файл Makefile из файла Makefile.in, запустите команду configure:

```
$ ./configure
```

Вы должны получить пространный диагностический вывод, пока сценарий проверяет вашу систему на соответствие необходимым условиям. Если все завершается удачно, команда configure создает один или несколько файлов Makefile и файл config.h, а также файл кэша (config.cache), чтобы ей не приходилось выполнять некоторые проверки заново.

Теперь можно запустить команду make для компиляции пакета. Успешное прохождение этапа с командой configure не обязательно означает, что этап make тоже будет пройден, но шансы на это весьма велики (см. раздел 16.6, чтобы разобраться с неудачными результатами конфигурирования и компиляции).

Получим некоторый опыт выполнения такого процесса.

ПРИМЕЧАНИЕ

Для этого вам требуется иметь все необходимые инструменты сборки, которые доступны в вашей системе. Для Debian и Ubuntu простейшим способом является установка пакета, существенного для сборки; для систем, подобных Fedora, воспользуйтесь утилитой groupinstall.

## 16.3.1. Пример работы утилиты Autoconf

Прежде чем обсуждать то, как вы можете изменить поведение утилиты Autoconf, посмотрим на простой пример, чтобы вы знали, чего ожидать. Вы будете устанавливать пакет GNU coreutils в ваш домашний каталог (для гарантии того, что он не смешается с вашей системой). Скачайте пакет с веб-страницы http://ftp.gnu.org/gnu/coreutils/ (обычно самой лучшей является последняя версия), распакуйте его, перейдите в распакованный каталог и выполните конфигурирование таким образом:

```
$ ./configure --prefix=$HOME/mycoreutils
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
--snip--
config.status: executing po-directories commands
config.status: creating po/POTFILES
config.status: creating po/Makefile
```

Теперь запустите команду `make`:

```
$ make
  GEN        lib/alloca.h
  GEN        lib/c++defs.h
--snip--
make[2]: Leaving directory '/home/juser/coreutils-8.22/gnulib-tests'
make[1]: Leaving directory '/home/juser/coreutils-8.22'
```

Далее попробуйте запустить один из исполняемых файлов, только что созданных вами, например `./src/ls`, и попытайтесь выполнить команду `make check`, чтобы осуществить ряд проверок пакета. На это может потребоваться некоторое время, но наблюдать за проверкой интересно.

Наконец вы готовы к установке пакета. Выполните сначала пробный прогон с помощью команды `make -n`, чтобы увидеть то, что выполняет команда `make install`, но не устанавливая сам пакет:

```
$ make -n install
```

Просмотрите результат работы и, если не заметите никаких странностей (таких как установка в каталог, отличающийся от каталога `mycoreutils`), выполните реальную установку:

```
$ make install
```

Теперь в вашем домашнем каталоге должен появиться подкаталог с именем `mycoreutils`, содержащий подкаталоги `bin`, `share` и др. Попробуйте запустить некоторые команды из каталога `bin` (вы только что выполнили сборку многих основных инструментов, о которых узнали из главы 2). Наконец, поскольку вы настроили каталог `mycoreutils` так, чтобы он не зависел от остальной части системы, можно полностью удалить его, не опасаясь причинить вред системе.

## 16.3.2. Установка с помощью инструментов для создания пакетов

В большинстве версий ОС возможно устанавливать новое ПО как пакет, который в дальнейшем вы можете обслуживать с помощью инструментов для работы с пакетами. Версии системы на основе Debian, например Ubuntu, вероятно, наиболее просты: вместо запуска обычной команды `make install` можно выполнить установку с помощью утилиты `checkinstall` следующим образом:

```
# checkinstall make install
```

Используйте параметр `--pkgname=`*`name`*, чтобы указать имя для нового пакета.

Создание пакета RPM выполняется немного сложнее, поскольку сначала вы должны создать дерево каталогов для ваших пакетов. Это можно сделать с помощью команды `rpmdev-setuptree`; по ее выполнении можно применить утилиту `rpmbuild`, чтобы осуществить оставшиеся шаги. В этом процессе лучше всего следовать интерактивному руководству.

## 16.3.3. Параметры сценария configure

Вы только что увидели один из самых полезных параметров сценария `configure`: использование `--prefix` для указания каталога установки. По умолчанию цель команды `install` из созданного утилитой Autoconf файла `Makefile` использует *префикс* `/usr/local` — то есть двоичные команды отправляются в каталог `/usr/local/bin`, библиотеки в каталог `/usr/local/lib` и т. д. Вам часто потребуется изменить этот префикс подобным образом:

```
$ ./configure --prefix=new_prefix
```

В большинстве версий сценария `configure` есть параметр `--help`, который позволяет вывести список других параметров конфигурации. К сожалению, этот перечень обычно настолько длинный, что иногда бывает трудно понять, что может оказаться полезным, поэтому приведу несколько важнейших параметров.

- `--bindir=`*`directory`*. Устанавливает исполняемые файлы в каталог *`directory`*.
- `--sbindir=`*`directory`*. Устанавливает системные исполняемые файлы в каталог *`directory`*.
- `--libdir=`*`directory`*. Устанавливает библиотеки в каталог *`directory`*.
- `--disable-shared`. Предотвращает создание совместно используемых библиотек для пакета. В зависимости от библиотеки в дальнейшем это может помочь избежать неприятностей (см. подраздел 15.1.4).
- `--with-package=`*`directory`*. Говорит сценарию `configure` о том, что пакет *`package`* находится в каталоге *`directory`*. Это удобно, когда необходимая библиотека расположена в нестандартном месте. К сожалению, не все сценарии конфигурирования распознают этот тип параметра, а выяснить точный синтаксис для них бывает затруднительно.

### Использование отдельных каталогов сборки

Можно создать отдельные каталоги для сборки, если вы желаете поэкспериментировать с некоторыми из параметров. Чтобы это сделать, создайте новый каталог где-либо в системе и из этого каталога запустите сценарий `configure` в каталоге исходного кода пакета. Вы обнаружите, что при этом сценарий `configure` создаст ферму символических ссылок в вашем новом каталоге, где все ссылки указывают на дерево источника в исходном каталоге пакета. Некоторые разработчики предпочитают, чтобы вы собирали пакеты именно так, поскольку при этом не изменяется дерево источника. Это полезно также, если вы желаете собрать с помощью одного исходного пакета несколько версий для разных платформ или параметров конфигурации.

## 16.3.4. Переменные окружения

Можно повлиять на сценарий `configure` с помощью переменных окружения, которые сценарий `configure` помещает в переменные утилиты `make`. Самыми важными являются переменные `CPPFLAGS`, `CFLAGS` и `LDFLAGS`. Но будьте осторожны: сценарий

configure может оказаться весьма требовательным к переменным окружения. Например, обычно вам следует использовать переменную CPPFLAGS вместо переменной CFLAGS для каталогов с заголовочными файлами, поскольку сценарий configure часто запускает препроцессор независимо от компилятора.

В оболочке bash простейший способ отправить переменную окружения сценарию configure — это поместить назначение переменной перед фрагментом ./configure в командной строке. Например, чтобы создать макроопределение DEBUG для препроцессора, используйте следующую команду:

```
$ CPPFLAGS=-DDEBUG ./configure
```

ПРИМЕЧАНИЕ

Можно также передать переменную как параметр для сценария configure, например так:

```
$ ./configure CPPFLAGS=-DDEBUG
```

Переменные окружения особенно удобны, когда сценарий configure не знает, где искать включаемые файлы и библиотеки сторонних разработчиков. Чтобы, например, препроцессор выполнил поиск в каталоге *include_dir*, запустите такую команду:

```
$ CPPFLAGS=-Iinclude_dir ./configure
```

Как показано в подразделе 15.2.6, чтобы компоновщик заглянул в каталог lib_dir, используйте следующую команду:

```
$ LDFLAGS=-Llib_dir ./configure
```

Если в каталоге lib_dir есть совместно используемые библиотеки (см. подраздел 15.1.4), то приведенная команда, вероятно, не будет определять путь для динамической компоновки времени исполнения. В таком случае используйте параметр компоновщика -rpath в дополнение к флагу -L:

```
$ LDFLAGS="-Llib_dir -Wl,-rpath=lib_dir" ./configure
```

Будьте внимательны при назначении переменных. Небольшая ошибка может сбить с толку компилятор и завершить компиляцию неудачей. Допустим, вы забыли дефис во флаге -I, как показано здесь:

```
$ CPPFLAGS=Iinclude_dir ./configure
```

Это приведет к следующей ошибке:

```
configure: error: C compiler cannot create executables
See 'config.log' for more details
```

Если покопаться в журнале config.log, созданном при этой неудачной попытке, то можно обнаружить следующее:

```
configure:5037: checking whether the C compiler works
configure:5059: gcc Iinclude_dir conftest.c >&5
gcc: error: Iinclude_dir: No such file or directory
configure:5063: $? = 1
configure:5101: result: no
```

## 16.3.5. Цели утилиты Autoconf

Когда сценарий `configure` заработает, вы обнаружите, что в созданном с его помощью файле `Makefile` есть несколько других полезных целей, помимо стандартных `all` и `install`.

- `make clean`. Удаляются все объектные файлы, исполняемые файлы и библиотеки.
- `make distclean`. Эта цель подобна цели `make clean`, но при этом удаляются все автоматически созданные файлы, включая `Makefiles`, `config.h`, `config.log` и т. п. Идея в том, чтобы дерево источника выглядело как только что распакованный дистрибутив после выполнения команды `make distclean`.
- `make check`. Некоторые пакеты поставляются с обоймой тестов для проверки правильности работы скомпилированных программ; команда `make check` запускает эти тесты.
- `make install-strip`. Подобна `make install`, но при установке из исполняемых файлов и библиотек удаляется таблица имен и другая отладочная информация. Для урезанных двоичных файлов требуется намного меньше места.

## 16.3.6. Файлы журналов утилиты Autoconf

Если что-либо идет не так во время процесса конфигурирования, а причина этого не ясна, можно изучить файл `config.log`, чтобы отыскать проблему. К сожалению, этот файл зачастую очень велик, что затрудняет определение точного источника проблем.

Общий подход к поиску проблем заключается в переходе к самому концу файла `config.log` (нажатием, например, клавиши G в команде `less`) и дальнейшей прокрутке назад, пока не обнаружится проблема. Однако при этом для проверки по-прежнему остается довольно много информации, поскольку сценарий `configure` помещает сюда все окружение, включая переменные вывода, переменные кэша и другие определения. По этой причине вместо того, чтобы переходить в конец и прокручивать результат вверх, перейдите в конец вывода и выполните поиск в обратном направлении, указав строку `for more details` или какую-либо другую часть недалеко от конца ошибочного вывода сценария `configure`. Напомню, что поиск в обратном направлении можно запустить в команде `less` с помощью команды `?`. Весьма велики шансы на то, что ошибка окажется как раз над той строкой, которая была найдена при поиске.

## 16.3.7. Команда pkg-config

Существует настолько много библиотек сторонних разработчиков, что размещение их всех в общем расположении может привести к путанице. Однако и установка каждой из них с помощью отдельного префикса может вызвать проблемы, когда при сборке пакетов эти библиотеки понадобятся. Если, например, вы желаете скомпилировать пакет OpenSSH, вам необходима библиотека OpenSSL. Как сообщить процессу конфигурирования пакета OpenSSH о том, где расположены библиотеки OpenSSL и какие из них потребуются?

Многие пакеты используют теперь команду pkg-config не только для информирования о местоположении своих включаемых файлов и библиотек, но также и для указания точных флагов, которые вам необходимы для компиляции и сборки программ. Синтаксис выглядит так:

```
$ pkg-config options package1 package2 ...
```

Чтобы, например, отыскать библиотеки, необходимые для пакета OpenSSL, можно запустить следующую команду:

```
$ pkg-config --libs openssl
```

Результат будет вроде этого:

```
-lssl -lcrypto
```

Чтобы увидеть список всех библиотек, о которых знает команда pkg-config, запустите такую команду:

```
$ pkg-config --list-all
```

## Как работает команда pkg-config

Если заглянуть за кулисы, то можно обнаружить, что команда pkg-config отыскивает информацию о пакете, читая файл конфигурации, имя которого оканчивается на .pc. Вот пример того, так выглядит файл openssl.pc для библиотеки сокетов OpenSSL в системе Ubuntu (он расположен в каталоге /usr/lib/i386-linux-gnu/pkgconfig):

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib/i386-linux-gnu
includedir=${prefix}/include

Name: OpenSSL
Description: Secure Sockets Layer and cryptography libraries and tools
Version: 1.0.1
Requires:
Libs: -L${libdir} -lssl -lcrypto
Libs.private: -ldl -lz
Cflags: -I${includedir} exec_prefix=${prefix}
```

Можно изменить этот файл, добавив, например, флагам библиотеки параметр -Wl,-rpath=${libdir}, чтобы указать путь динамической компоновки времени исполнения. Однако на первом месте остается более серьезный вопрос: как команда pkg-config отыскивает файлы .pc? По умолчанию команда pkg-config смотрит каталог lib/pkgconfig в соответствии с префиксом установки. Например, команда pkg-config, установленная с префиксом /usr/local, будет смотреть каталог /usr/local/lib/pkgconfig.

## Установка файлов команды pkg-config в нестандартных размещениях

К сожалению, по умолчанию команда pkg-config не читает никаких файлов .pc за пределами своего каталога установки. Если файл .pc находится в нестандартном

месте, например в каталоге /opt/openssl/lib/pkgconfig/openssl.pc, то он будет недоступен для любой стандартно установленной команды pkg-config. Есть два основных способа сделать файлы .pc доступными за пределами каталога установки команды pkg-config.

- Создать символические ссылки (или копии) из реальных файлов .pc в основном каталоге pkgconfig.
- Определить переменную окружения PKG_CONFIG_PATH так, чтобы она содержала все дополнительные каталоги. Эта стратегия работает неудовлетворительно с точки зрения системы в целом.

# 16.4. Практика установки

Хорошо знать о том, как собирать и устанавливать программы, но еще полезнее знать о том, когда и где устанавливать собственные пакеты программ. В дистрибутивы Linux стараются уместить максимально возможное количество программ, и вам всегда следует проверять, будет ли лучше, если вы установите пакет ПО самостоятельно. Вот преимущества самостоятельной установки программ:

- можно настроить параметры по умолчанию;
- при установке пакета возникает более ясное представление о том, как его применять;
- вы управляете тем релизом, который запускаете;
- проще создать резервную копию пользовательского пакета;
- проще распространить по сети самоустанавливающиеся пакеты (до тех пор пока архитектура совместима и местоположение для установки изолировано).

Но есть и неудобства:

- на это требуется время;
- пользовательские пакеты не обновляются автоматически. Дистрибутивы обновляют большинство пакетов, не требуя больших трудов. В особенности это относится к пакетам, которые взаимодействуют с сетью, поскольку вам необходимо быть уверенными в том, что применены последние обновления защиты;
- если вы реально не используете пакет, то вы напрасно тратите время;
- есть вероятность неправильной конфигурации пакетов.

Нет особого смысла в установке таких пакетов, как, например, coreutils, который мы собирали в этой главе чуть выше (ls, cat и т. д.), если только вы не собираетесь создать особенную систему. С другой стороны, если вы проявляете живой интерес к сетевым серверам, таким как Apache, то лучший способ получить полный контроль над ним — установить его самостоятельно.

**Куда устанавливать.** Префикс по умолчанию в утилите GNU Autoconf и многих других пакетах — /usr/local, традиционный каталог для устанавливаемого локального ПО. Обновления операционной системы игнорируют каталог /usr/local, поэтому вы не потеряете ничего из установленного в нем во время обновления ОС. Для небольших локальных программ каталог /usr/local подходит. Единственная

проблема заключается в том, что большое количество установленных программ превращается в месиво. Тысячи странных маленьких файлов могут оказаться внутри каталога `/usr/local`, и у вас не будет ни малейшего понятия о том, откуда они.

Если все настолько вышло из-под контроля, следует создавать собственные пакеты, как рассказано в подразделе 16.3.2.

# 16.5. Применение исправлений

Большинство изменений в исходном коде программного обеспечения доступно в виде ветвей от исходного кода разработчика (например, репозиторий git). Тем не менее то и дело вам может потребоваться применить *исправление* к исходному коду, чтобы избавиться от ошибок или добавить новые функции. Вы можете также встретить термин *diff*, используемый как синоним, поскольку исправление выполняется с помощью команды `diff`.

Начало исправления может выглядеть таким образом:

```
--- src/file.c.orig     2015-07-17 14:29:12.000000000 +0100
+++ src/file.c   2015-09-18 10:22:17.000000000 +0100
@@ -2,16 +2,12 @@
```

Исправления обычно содержат изменения для нескольких файлов. Отыщите строку с тремя дефисами подряд (`---`), чтобы понять, какие файлы изменены, и всегда смотрите начало исправления, чтобы определить рекомендуемый рабочий каталог. Обратите внимание на то, что приведенный пример ссылается на файл `src/file.c`. Следовательно, до применения исправления вы должны перейти в каталог, который содержит каталог `src`, но *не* в сам каталог `src`.

Чтобы применить исправление, запустите команду `patch`:

```
$ patch -p0 < patch_file
```

Если все пройдет успешно, команда `patch` незаметно завершит свою работу, оставив вам обновленный набор файлов. Однако она может задать вам такой вопрос:

```
File to patch:
```

Обычно это означает, что вы находитесь не в том каталоге, но может свидетельствовать также о том, что ваш исходный код не соответствует исходному коду в исправлении. В таком случае вам, вероятно, не повезло: даже если вам удалось бы определить некоторые файлы, подлежащие исправлению, другие остались бы не обновленными, и тогда исходный код не удастся скомпилировать.

В некоторых случаях можно встретить исправление, которое ссылается на версию пакета, например, так:

```
--- package-3.42/src/file.c.orig     2015-07-17 14:29:12.000000000 +0100
+++ package-3.42/src/file.c   2015-09-18 10:22:17.000000000 +0100
```

Если номер вашей версии немного отличается (или вы просто переименовали каталог), можно дать указание команде `patch`, чтобы она обрезала начальные элементы пути. Допустим, например, что вы находитесь в каталоге, который содержит

каталог src (как выше). Чтобы команда patch игнорировала часть пути package-3.42/ (то есть отрезала один начальный фрагмент пути), используйте флаг -p1:

```
$ patch -p1 < patch_file
```

## 16.6. Устранение проблем при компиляции и установке

Если вы усвоили различия между ошибками и предупреждениями компилятора, ошибками компоновщика и проблемами, связанными с совместно используемыми библиотеками, как рассказано в главе 15, у вас не должно возникнуть много сложностей при исправлении большинства глюков, которые возникают при сборке программ. Этот раздел описывает некоторые распространенные проблемы. Хотя маловероятно, что при использовании утилиты Autoconf вам встретится что-либо подобное, никогда не повредит узнать о том, как выглядят такие проблемы.

Прежде чем переходить к деталям, убедитесь в том, что вы понимаете некоторые типы сообщений утилиты make. Важно знать различия между ошибкой и проигнорированной ошибкой. Вот реальная ошибка, с которой вам необходимо разобраться:

```
make: *** [target] Error 1
```

Однако некоторые файлы Makefiles подозревают, что может возникнуть условие для какой-либо ошибки, но они знают, что такие ошибки безвредны. Обычно можно не обращать внимания на такие сообщения:

```
make: *** [target] Error 1 (ignored)
```

Команда GNU make часто вызывает себя многократно для больших пакетов, и каждый ее экземпляр отмечен в сообщениях об ошибках с помощью символов [*N*], где *N* является числом. Зачастую можно быстро отыскать ошибку, посмотрев на то сообщение об ошибке команды make, которое идет *сразу* после ошибки компилятора. Например:

```
 [compiler error message involving file.c]
make[3]: *** [file.o] Error 1
make[3]: Leaving directory '/home/src/package-5.0/src'
make[2]: *** [all] Error 2
make[2]: Leaving directory '/home/src/package-5.0/src'
make[1]: *** [all-recursive] Error 1 make[1]: Leaving directory '/home/src/
package-5.0/'
make: *** [all] Error 2
```

Первые три строки практически выдают ее: проблема связана с файлом file.c, который расположен в каталоге /home/src/package-5.0/src. К сожалению, дополнительной информации так много, что бывает сложно найти важные детали. Выяснение того, как отфильтровать более поздние ошибки утилиты make, позволит отыскать настоящую причину.

**Особые ошибки.** Вот несколько распространенных ошибок сборки, которые могут вам встретиться.

**Ошибка**

Сообщение об ошибке компилятора:

```
src.c:22: conflicting types for 'item'
/usr/include/file.h:47: previous declaration of 'item'
```

**Объяснение и устранение**

Программист выполнил ошибочное повторное объявление элемента *item* в строке 22 файла `src.c`. Обычно это можно исправить, удалив ошибочную строку (с помощью комментария, директивы `#ifdef` или чего-либо подобного).

**Ошибка**

Сообщение об ошибке компилятора:

```
src.c:37: 'time_t' undeclared (first use this function)
--snip--
src.c:37: parse error before '...'
```

**Объяснение и устранение**

Программист забыл важный заголовочный файл. Страницы руководства лучше всего помогут в отыскании упущенного файла. Сначала посмотрите на ошибочную строку (в данном случае это строка 37 в файле `src.c`). Вероятно, она содержит объявление переменной вроде следующего:

```
time_t v1;
```

Отыщите в программе строку, в которой переменная `v1` использует вызов какой-либо функции. Например, так:

```
v1 = time(NULL);
```

Теперь запустите команды `man 2 time` или `man 3 time`, чтобы отыскать системные и библиотечные вызовы с именем `time()`. В данном случае подходит второй раздел страницы руководства:

```
SYNOPSIS
      #include <time.h>

      time_t time(time_t *t);
```

Это означает, что вызову `time()` необходим файл `time.h`. Поместите строку `#include <time.h>` в начало файла `src.c` и попробуйте еще раз.

**Ошибка**

Сообщение об ошибке компилятора (препроцессора):

```
src.c:4: pkg.h: No such file or directory
(long list of errors follows)
```

**Объяснение и устранение**

Компилятор запустил препроцессор C для файла `src.c`, но не смог найти включаемый файл `pkg.h`. Исходный код, вероятно, зависит от библиотеки, которую необходимо установить, или же необходимо указать для компилятора нестандартный путь с включаемыми файлами. Как правило, вам понадобится лишь добавить параметр `-I` к флагам препроцессора C (`CPPFLAGS`), чтобы добавить этот путь. Помни-

те о том, что может также понадобиться флаг компоновщика `-L`, чтобы использовать эти включаемые файлы.

Если ситуация не похожа на отсутствие библиотеки, есть незначительная вероятность того, что вы пытаетесь скомпилировать файл для операционной системы, которая не поддерживается данным исходным кодом. Ознакомьтесь с файлами `Makefile` и `README`, чтобы узнать подробности о платформах.

Если вы работаете с версией ОС на основе Debian, попробуйте применить команду `apt-file` к имени заголовочного файла:

```
$ apt-file search pkg.h
```

Это может выявить необходимый пакет разработки. Для версий, которые содержат утилиту `yum`, попробуйте такой вариант:

```
$ yum provides */pkg.h
```

**Ошибка**

Сообщение утилиты `make`:

```
make: prog: Command not found
```

**Объяснение и устранение**

Чтобы собрать пакет, в вашей системе должна быть программа *prog*. Если таковой является что-либо вроде `cc`, `gcc` или `ld`, то в системе не установлены утилиты для разработки. С другой стороны, если вы полагаете, что команда *prog* уже установлена в системе, попробуйте изменить файл `Makefile`, чтобы выяснить полный путь к команде *prog*.

В редких случаях команда `make` собирает программу *prog*, а затем немедленно ее использует, считая, что текущий каталог (`.`) расположен в командном пути. Если переменная `$PATH` не содержит текущий каталог, можно отредактировать файл `Makefile` и изменить *prog* на *./prog*. Как вариант, можно было бы на время добавить точку к имени пути.

# 16.7. Заглядывая вперед

Мы только затронули основы сборки программного обеспечения. Вот несколько дополнительных направлений, которые вы можете исследовать, когда займетесь собственными разработками.

- **Понимание того, как использовать другие системы компоновки, кроме утилиты Autoconf (например, CMake и SCons).**
- **Настройка сборок для вашего ПО.** Если вы разрабатываете собственные программы, вам необходимо выбрать систему сборки и научиться применять ее. При создании пакетов с помощью утилиты GNU Autoconf вам может пригодиться книга Джона Кэлкоута (John Calcote) *Autotools* («Утилиты Autotools») (No Starch Press, 2010).
- **Компиляция ядра системы Linux.** Система сборки ядра полностью отличается от других инструментов. У нее есть собственная система конфигурирования, предназначенная для точной подгонки под ваше ядро и модули. Однако сама

процедура проста. Если вы понимаете, как работает загрузчик системы, у вас не возникнет никаких затруднений. Тем не менее следует с осторожностью заниматься этим: всегда проверяйте, что у вас под рукой есть старое ядро на тот случай, когда вы не сможете загрузиться с помощью нового.

- **Специализированные пакеты исходного кода.** Дистрибутивы Linux поддерживают собственные версии исходного программного кода в качестве специальных пакетов исходного кода. Иногда могут оказаться полезными исправления, которые расширяют функциональность или исправляют ошибки, недоступные для исправления как-либо иначе. Системы для управления пакетами исходного кода содержат инструменты для автоматической сборки, например `debuild` в Debian и `mock` для систем на основе RPM.

Сборка программного обеспечения — это зачастую стартовая площадка для изучения языков программирования и методов разработки ПО. Инструменты, которые вы видели в последних двух главах, снимают завесу тайны с того, откуда берутся ваши системные программы. Совсем несложно выполнить дальнейшие шаги: заглянуть в исходный код, внести изменения и создать собственную программу.

# 17 Строим на фундаменте

В главах этой книги были рассмотрены фундаментальные компоненты системы Linux, начиная с низкоуровневой организации ядра и процессов, устройства сети и заканчивая некоторыми инструментами, предназначенными для сборки программного обеспечения. В итоге вы умеете делать довольно многое. Поскольку Linux поддерживает почти любой тип общедоступной среды программирования, приложения доступны в изобилии. Рассмотрим некоторые прикладные области, в которых преуспевает Linux, и поймем, как к ним относится то, что вы узнали из этой книги.

## 17.1. Веб-серверы и приложения

Linux является популярной системой для веб-серверов, а правящим монархом среди серверов приложений Linux является Apache HTTP Server (который обычно называют просто Apache). Другим веб-сервером, о котором вы часто будете слышать, является Tomcat (это также проект разработчиков Apache); он обеспечивает поддержку Java-приложений.

Сами по себе веб-серверы выполняют немногое: они могут хранить файлы, и на этом все. Конечной целью большинства веб-серверов, таких как Apache, является предоставление платформы для использования веб-приложений. Например, проект Wikipedia построен на пакете MediaWiki, который вы можете использовать, чтобы организовать собственный wiki-проект. Системы управления контентом вроде Wordpress и Drupal позволяют вам создавать собственные блоги и мультимедийные сайты. Все эти приложения основаны на языках программирования, которые особенно хорошо проявляют себя в Linux. Так, например, системы MediaWiki, Wordpress и Drupal написаны на языке PHP.

Строительные блоки, из которых составлены веб-приложения, являются в большой степени модульными, поэтому легко добавлять собственные расширения и создавать приложения с помощью таких фреймворков, как Django, Flask и Rails, которые предоставляют средства для распространенных веб-инфрастуктур и функций, например, использование шаблонов, управление многими пользователями и поддержка баз данных.

Нормально функционирующий веб-сервер зависит от прочного основания в виде операционной системы. В частности, чрезвычайно важным является материал, изложенный в главах 8–10.

Конфигурация вашей сети должна быть безупречной, и вы, что, пожалуй, более важно, должны уметь управлять ресурсами. Разумно распределенное дисковое пространство и память имеют решающее значение, особенно если вы планируете использовать базу данных в своем приложении.

## 17.2. Базы данных

Базы данных — это специальные службы для хранения и извлечения данных, и в Linux можно запускать многие различные серверы и системы баз данных. Две основные особенности делают базы данных привлекательными: они обеспечивают простой и унифицированный подход к управлению индивидуальными фрагментами и группами данных, а также превосходную скорость доступа.

Базы данных облегчают приложениям проверку и изменение данных, в особенности если сравнить их с анализом и изменением текстовых файлов. К примеру, могут возникнуть сложности при управлении файлами `/etc/passwd` и `/etc/shadow` в Linux в сети компьютеров. Вместо них можно настроить базу данных, которая предоставляет протокол LDAP (Lightweight Directory Access Protocol, облегченный протокол доступа к (сетевым) каталогам), чтобы снабдить необходимой информацией систему аутентификации Linux. Конфигурация на стороне клиента Linux простая; все, что потребуется сделать, — это отредактировать файл `/etc/nsswitch.conf` и добавить дополнительную конфигурацию.

Основная причина, по которой базы данных обычно обеспечивают превосходное быстродействие при извлечении данных, заключается в том, что они выполняют индексирование, чтобы отслеживать размещение данных. Допустим, у вас есть набор данных, которые представляют справочник, содержащий имена, фамилии и телефонные номера. Можно использовать базу данных, чтобы поместить индекс на любом из этих атрибутов, например на фамилии. Тогда при поиске кого-либо по фамилии база данных просто сверяется с индексом фамилий, а не просматривает весь справочник полностью.

**Типы баз данных.** Базы данных бывают двух основных типов: реляционные и нереляционные. *Реляционные базы данных* (также называемые *реляционными системами управления базами данных*, *РСУБД*; Relational Database Management Systems, RDBMS), такие как MySQL, PostgreSQL, Oracle и MariaDB, являются базами данных общего назначения, которые отличаются способностью объединять различные наборы данных. Допустим, у вас есть два набора данных: один с почтовыми индексами и именами, а другой — с почтовыми индексами и соответствующими населенными пунктами. Реляционная база данных позволила бы вам очень быстро извлечь список имен людей, проживающих в одном населенном пункте. Обычно с реляционными базами данных «разговаривают» с помощью языка программирования SQL (Structured Query Language, язык структурированных запросов).

Нереляционные базы данных, иногда называемые базами данных *NoSQL*, призваны решать частные задачи, с которыми нелегко справляются реляционные базы данных. Например, базы данных для хранения документов, такие как MongoDB, стремятся облегчить хранение и индексацию документа в целом. Базы данных «ключ — значение», например redis, делают акцент на производи-

тельности. Для доступа к базам данных NoSQL нет общего языка запросов, подобного SQL. Вместо этого с ними работают с помощью различных интерфейсов и команд.

Вопросы, связанные с производительностью дисков и памяти, рассмотренные в главе 8, чрезвычайно важны для большинства реализаций баз данных, поскольку существует компромисс между тем, сколько вы можете позволить хранить в оперативной памяти (это быстрее) и сколько — на диске. Более мощные системы баз данных задействуют также работу с сетью, так как они распределены по нескольким серверам. Самый распространенный вариант такого способа работы с сетью называется *репликацией*: одна база данных, по существу, копируется на несколько серверов базы данных, чтобы увеличить количество клиентов, которые могут подключиться к серверам.

## 17.3. Виртуализация

В большинстве крупных организаций неэффективно выделять аппаратное обеспечение для специальных серверных задач, поскольку установка операционной системы, привязанной к одной задаче на одном сервере, означает, что вы ограничены лишь этой задачей, пока не переустановите систему. Технология виртуальной машины позволяет одновременно установить одну или несколько операционных систем (часто называемых *гостевыми* ОС) на одном аппаратном средстве, а затем по мере надобности активизировать и деактивизировать эти системы. Можно даже перемещать и копировать виртуальные машины на другие компьютеры.

Для Linux есть много систем виртуализации, например KVM (виртуальная машина ядра) и Xen. Виртуальные машины особенно удобны для веб-серверов и серверов баз данных. Хотя возможно настроить единственный сервер Apache для обслуживания нескольких сайтов, за это придется поплатиться гибкостью и управляемостью. Если такие сайты поддерживаются различными пользователями, то вам придется управлять как серверами, так и пользователями сразу. Вместо этого обычно предпочтительнее настроить виртуальные машины на одном физическом сервере, каждую из которых поддерживает свой пользователь, чтобы они не препятствовали друг другу, а вы могли бы изменять и перемещать их по желанию.

Программа, которая работает с виртуальными машинами, называется *гипервизором*. Гипервизор взаимодействует с множеством частей системы Linux на низших уровнях, которые вы видели в этой книге, и в результате при установке гостевой Linux в виртуальной машине она должна вести себя точно так же, как и любая другая установленная система Linux.

## 17.4. Распределенные вычисления и вычисления по запросу

Чтобы облегчить управление локальными ресурсами, можно поместить замысловатые инструменты управления поверх технологии виртуальной машины. Термин «*облачные вычисления*» настолько всеобъемлющ, что часто используется для обозначения

этой сферы. Более точно, *инфраструктурой в качестве службы* (IaaS, Infrastructure as a Service), называют системы, которые позволяют вам обеспечивать и контролировать основные вычислительные ресурсы, такие как процессор, память, хранилище данных и сеть, на удаленном сервере. Проект OpenStack является одним из таких интерфейсов прикладного программирования и платформой, которая содержит систему IaaS.

Переместившись за «сырую» инфраструктуру, можно также выполнить обеспечение платформы ресурсами, такими как операционная система, серверы баз данных и веб-серверы. Систему, которая предоставляет ресурсы на таком уровне, часто называют *платформой в качестве службы* (PaaS, Platform as a Service).

Linux является центральной для многих из таких вычислительных служб, поскольку она часто лежит в их основе. Практически все элементы, которые вы видели в этой книге, начиная с ядра, находят отражение в этих системах.

## 17.5. Встроенные системы

*Встроенная система* — это что-либо, разработанное с определенной целью, например музыкальный проигрыватель, видеостример или термостат. Сравните это с персональной или серверной системой, которая может выполнять множество различных задач (но не может делать очень хорошо какую-либо конкретную).

Можно представлять встроенные системы как почти полную противоположность распределенным вычислениям: вместо расширения масштаба операционной системы встроенная система обычно (но не всегда) сужает его, часто до небольшого устройства. Система Android является сегодня, наверное, самой распространенной встроенной версией Linux.

Встроенные системы часто сочетают специализированные аппаратные средства и программное обеспечение. Например, можно настроить ПК на выполнение всего того, что делает беспроводной маршрутизатор, добавив достаточное количество аппаратных средств и корректно выполнив конфигурирование системы Linux. Однако обычно предпочтительнее купить специализированное устройство, состоящее из необходимых аппаратных средств и не содержащее ничего лишнего. Например, маршрутизатору необходимо большее количество сетевых портов по сравнению с ПК, но не нужна видеокарта или звуковая система. Поскольку аппаратные средства особые, необходимо подогнать под них и программное обеспечение, такое как внутреннее наполнение системы и пользовательский интерфейс. Система OpenWRT, о которой упоминалось в главе 9, является одной из таких специальных версий Linux.

Интерес к встроенным системам возрастает по мере того, как появляются более вместительные небольшие аппаратные средства, в частности однокристальные системы (SoC, System-on-a-chip), которые могут уместить на небольшом пространстве процессор, память и периферийные интерфейсы. Например, одноплатные компьютеры Raspberry Pi и BeagleBone основаны на такой схеме, и для них в качестве операционной системы можно выбрать один из вариантов Linux. Такие устройства обладают легкодоступным выводом и сенсорным вводом, который подключается к языковым интерфейсам вроде Python, делая их популярными для макетирования и небольших гаджетов.

Встроенные версии Linux различаются тем, как осуществляются многие функции из серверной/настольной версии. В небольших, очень ограниченных устройствах следует урезать все, кроме необходимого минимума, вследствие недостатка свободного пространства. Это часто означает, что даже утилиты оболочки и ядра существуют в виде единственного исполняемого файла BusyBox. Такие системы обычно демонстрируют наибольшее отличие от полнофункциональной версии Linux, и вы часто встретите в них более старые программы, вроде System V init.

Как правило, разработка ПО для встроенных систем ведется на обычном компьютере. Более мощные устройства, такие как Raspberry Pi, обладают роскошью в виде более объемного хранилища данных и вычислительной мощностью для запуска нового и более полного ПО, поэтому на них можно напрямую запускать многие инструменты для разработки.

Однако, несмотря на различия, встроенные системы все так же наследуют «гены» Linux, о которых рассказано в этой книге: вы обнаружите ядро, ряд устройств, сетевые интерфейсы и систему init, а также несколько пользовательских процессов. Встроенные ядра, как правило, близки (или идентичны) обычным, просто в них отключены многие функции. Но когда вы доберетесь до пространства пользователя, различия станут выражены более ярко.

## 17.6. Заключительные замечания

Каковы бы ни были ваши цели при достижении лучшего понимания систем Linux, я надеюсь, что эта книга оказалась полезной. Моя цель — привить вам уверенность, чтобы вы смогли изучить свою систему для выполнения изменений или чего-то нового. Теперь у вас должно появиться ощущение полного контроля над системой. Можете подступиться к ней вплотную и получать удовольствие.

*Брайан Уорд*

**Внутреннее устройство Linux**

*Перевел с английского М. Райтман*


| | |
|---|---|
| Заведующий редакцией | *О. Сивченко* |
| Ведущий редактор | *Н. Гринчик* |
| Литературный редактор | *О. Андриевич* |
| Художник | *В. Шимкевич* |
| Корректоры | *Т. Курьянович, Е. Павлович* |
| Верстка | *А. Барцевич* |

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 30.10.15. Формат 70×100/16. Бумага писчая. Усл. п. л. 30,960. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87