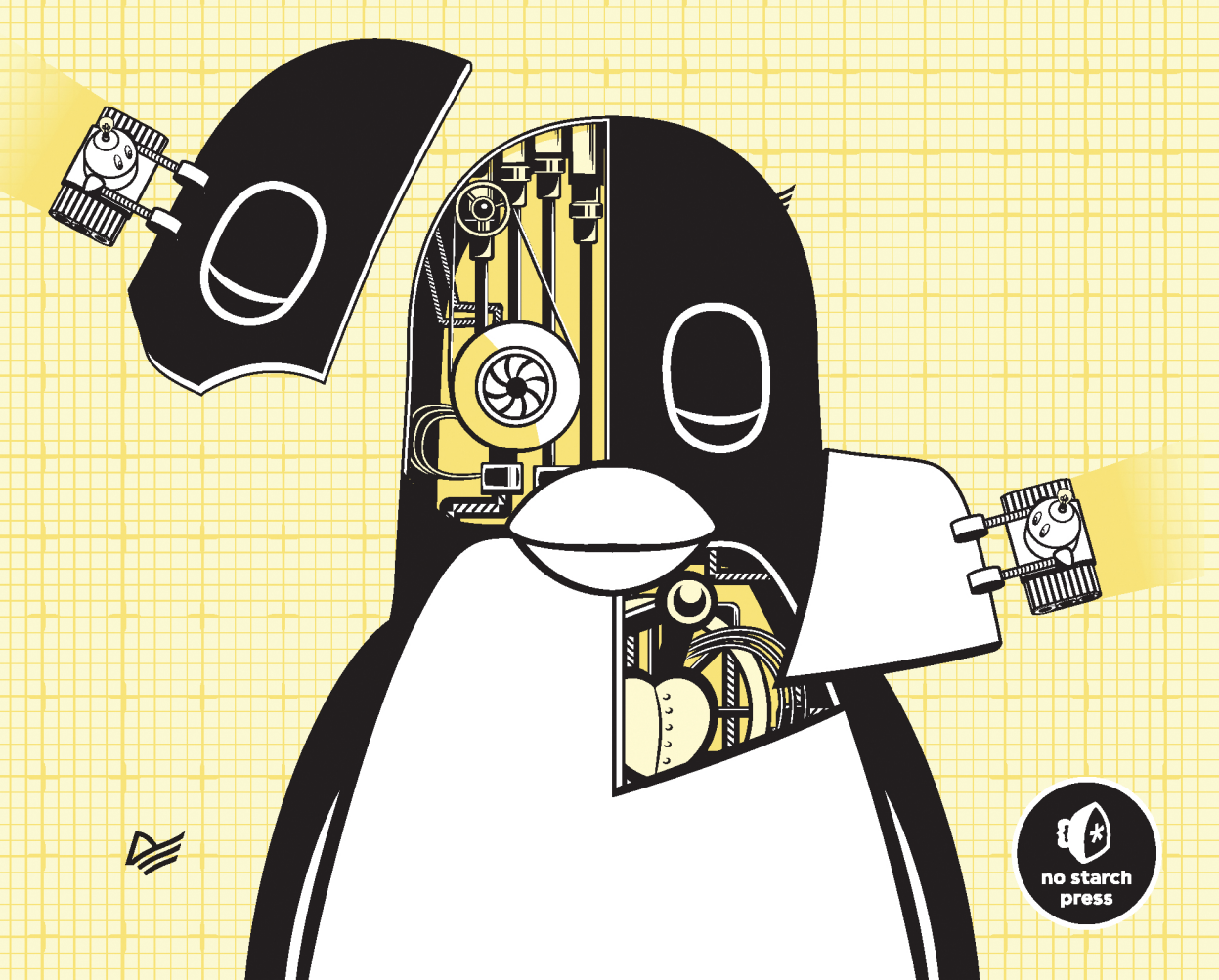
3-Е ИЗДАНИЕ

ВНУТРЕННЕЕ УСТРОЙСТВО LINUX

БРАЙАН УОРД



no starch press

HOW LINUX WORKS

3rd Edition

What Every Superuser Should Know

by Brian Ward

no starch press

San Francisco

ВНУТРЕННЕЕ УСТРОЙСТВО LINUX

3-Е ИЗДАНИЕ

БРАЙАН УОРД

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018.2
УДК 004.451
У64

Уорд Б.

У64 Внутреннее устройство Linux. 3-е изд. — СПб.: Питер, 2022. — 480 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-3946-0

Познакомьтесь со всеми тонкостями работы операционной системы Linux — от системного администрирования до глубинных механизмов, обеспечивающих низкоуровневый функционал Linux. Эта книга, сразу после выхода ставшая бестселлером Amazon, даст вам базовые знания о работе с ядром Linux и принципах правильной эксплуатации компьютерных сетей, о программировании сценариев оболочки и обращении с языком C. Вы изучите вопросы защиты информации, виртуализацию и многое другое. Книга необходима системным администраторам, программистам, специалистам по защите информации, а также всем, кто изучает или хочет изучить Linux максимально быстро и эффективно.

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

ББК 32.973.2-018.2
УДК 004.451

Права на издание получены по соглашению с No Starch Press. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1718500402 англ.

ISBN 978-5-4461-3946-0

© 2021 by Brian Ward. How Linux Works, 3rd Edition: What Every Superuser Should Know, ISBN 9781718500402, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103.
© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Для профессионалов», 2022

Краткое содержание

Оглавление

Если вы интересуетесь системой Linux, книга «Внутреннее устройство Linux» подойдет вам лучше всего.

LinuxInsider

В книге множество информации практически по всем аспектам архитектуры Linux.

Everyday Linux User

Вы получите полноценное представление о том, что происходит внутри системы, не увязая во множестве деталей. Это отличное дополнение к литературе по Linux.

Фил Булл, соавтор книги Ubuntu Made Easy и член команды по составлению документации для Ubuntu

Автор погружается в глубину операционных систем на базе Linux и рассказывает, как все части системы сочетаются друг с другом.

DistroWatch

Книга заслуживает места на полке суперпользователя Linux.

Журнал The MagPi

Об авторе

Брайан Уорд работает с операционной системой Linux с 1993 года. Кроме этой, он написал книги *The Linux Kernel HOWTO*, *The Book of VMware* и *The Linux Problem Solver*.

О научных редакторах

Жорди Гутьеррес Эрмосо — профессиональный разработчик и пользователь GNU/Linux с почти двадцатилетним опытом работы, внесший вклад в развитие GNU Octave и Mercurial. В разное время он работал с криптографией, медицинской визуализацией и в сфере экологии — везде на Linux. Когда Жорди не сидит за компьютером, то занимается плаванием, математикой и вязанием.

Петрос Кутупис в настоящее время старший инженер по производительности программного обеспечения в компании HPE (ранее Cray Inc.) в подразделении Lustre High Performance File System. Он создатель RapidDisk Project (www.rapiddisk.org) и занимается его сопровождением. Петрос более десяти лет работает в индустрии хранения данных и помог внедрить многие современные технологии.

Благодарности

Вклад в эту книгу внесли не только те, кто участвовал в процессе ее создания, но и те, без кого я бы ничего не узнал о Linux, а именно: Джеймс Дункан, Дуглас Н. Арнольд, Билл Феннер, Кен Хорнстайн, Скотт Диксон, Дэн Эрлих, Феликс Ли и Грегори П. Смит. За помощь в работе над предыдущими изданиями благодарю Кароля Хурадо, Лорел Чун, Серену Янг, Элисон Лоу, Райли Хоффмана, Скотта Шварца, Дэна Салли, Доминика Пулена, Дональда Кейрона и Джину Стил.

В подготовке третьего издания участвовали Барбара Куин, Рэйчел Монаган, Джилл Франклин, Ларри Уэйда, Хорди Гутьерреса Эрмосо и Петрос Кутуписа. Билл Поллок, основатель издательства No Starch Press, сыграл особую роль в создании этой книги с ее первого издания. И еще раз благодарю Синьцзю Сие за то, что вытерпел меня и в этот раз.

Предисловие

Операционная система не должна быть загадкой. Любому специалисту хочется использовать свое программное обеспечение по желанию, без магических заклинаний или ритуалов. Чтобы этого добиться, необходимо понимать, что делает программное обеспечение и как оно работает, и именно этому посвящена данная книга. Больше вам никогда не придется сражаться со своим компьютером.

Linux — отличная платформа для обучения, так как она ничего не пытается скрыть от пользователя. Большинство сведений о конфигурации системы можно найти в легкочитаемых текстовых файлах. Единственная сложность — выяснить, какие части за что отвечают и как они все сочетаются друг с другом.

Для кого эта книга

Интерес к устройству Linux затрагивает разные сферы жизни. Профессионалы в сфере DevOps и разработчики должны знать почти всю информацию, которая рассматривается в этой книге. Архитекторы и разработчики программного обеспечения Linux также должны знать это, чтобы пользоваться операционной системой наилучшим образом. Для исследователей и студентов, часто работающих в своих собственных системах Linux, будет полезно узнать, почему в системе все устроено именно так, а не иначе, что и рассказывается в книге. Кроме того, есть еще и любители — люди, которые просто проводят время за своими компьютерами ради развлечения, выгоды или и того и другого сразу.

Хотите знать, почему одни вещи работают, а другие нет? Вам интересно, что произойдет, если что-либо изменить? Если вы ответили «Да!», то вы, скорее всего, любитель и найдете ответы на свои вопросы в этой книге.

Требуемый уровень знаний

Вам необязательно быть программистом, чтобы прочитать эту книгу. Понадобятся только базовые навыки пользователя компьютера: вы должны ориентироваться в графическом интерфейсе (особенно в интерфейсе установки и настроек для дистрибутива Linux) и знать, что такое файлы и каталоги (папки). Кроме того, будьте готовы искать и проверять дополнительную документацию в вашей системе и в интернете. И самое главное — быть готовыми 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 трудом 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 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, в главе 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

Я сгруппировал главы книги в три основные части. Первая часть — вводная, дает представление о системе в целом, 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 запуска системы.

И наконец, вы познакомитесь с частями работающей системы, освоите необходимые навыки и рассмотрите инструменты, которые используют программисты.

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. Кроме того, я описал настольные и серверные установки. Значительный объем материала можно использовать во встроенных системах, таких как Android и OpenWRT, но поиск различий между ними предоставляется вам.

Новое в третьем издании

Второе издание вышло в переходный период для всех систем Linux. Часть традиционных компонентов постепенно изменялась, что затрудняло изучение, потому что читатели сталкивались с большим разнообразием настроек. Однако сейчас новые элементы (в частности, `systemd`) надежно заняли свои места в системах, поэтому я смог значительно упростить и сократить материал по этой теме.

Я все так же сохранил акцент на роли ядра в системе Linux. Именно эта информация была наиболее интересна читателям, и вы сами, скорее всего, взаимодействуете с ядром чаще, чем думаете.

В новом издании появилась глава о виртуализации. Несмотря на то что Linux часто устанавливают именно через виртуальные машины (например, через облачные серверы), подобный способ виртуализации выходит за рамки данной книги. Все потому, что система работает на виртуальной машине практически так же, как на «голом» железе. Таким образом, информация об этом отличается в основном лишь терминологией. Кроме того, со времени выхода второго издания контейнеризация стала более популярной, поэтому я добавил информацию и о ней, ведь контейнеры в основном состоят из множества функций Linux, которые описаны в книге. В контейнерах часто используются контрольные группы `cgroups`, о которых я также рассказываю в третьем издании. В книгу в том числе добавлены сведения о менеджере логических томов, системе ведения журнала `journald` и протоколе Ipv6 (в главе 9).

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 удалил кое-какие исторические детали, которые были в первом издании. Если вы интересуетесь системой Linux и ее отношением к истории системы Unix, обратитесь к книге Питера Салуса *The Daemon, the Gnu, and the Penguin* (Reed Media Services, 2008) — в ней рассказано о том, как развивалось используемое нами программное обеспечение.

Немного о терминологии

До сих пор ведутся споры о наименованиях некоторых элементов операционных систем. Они затрагивают 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 распространенные и по возможности наименее громоздкие названия.

От издательства

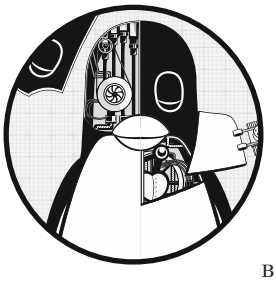
Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

Общая картина



На первый взгляд такая современная операционная система, как Linux, очень сложна и состоит из невероятного количества инструментов, которые работают и взаимодействуют друг с другом одновременно. Например, веб-сервер может взаимодействовать с сервером базы данных, который, в свою очередь, может применять разделяемую библиотеку, используемую многими другими программами. Как же все это работает и какой во всем этом смысл?

Самый эффективный способ понять, как работает операционная система, — абстрагироваться от большинства деталей, составляющих часть, которую вы изучаете, и сосредоточиться на ее основной цели и функциональности. Например, когда вы едете в автомобиле, вам не нужно думать о таких деталях, как крепежные болты, которые удерживают двигатель внутри него, или о людях, которые строят и ремонтируют дорогу, по которой он едет. Все, что вам действительно нужно знать, — это цель автомобиля (перевезти вас куда-то) и основные знания о том, как им пользоваться (например, как открывать дверь и пристегивать ремень безопасности).

Подобный уровень абстрагирования сработает, если вы просто пассажир в машине. Но если вы управляете автомобилем, вам нужно копнуть глубже и разбить свою абстракцию на несколько частей. Вы должны расширить свои знания в трех областях: сам автомобиль (например, его размер и возможности), манипуляции его частями (рулевое колесо, педаль акселератора и т. д.) и особенности дороги.

Абстрагирование от деталей может помочь при попытке найти и устранить проблемы. Предположим, что вы ведете машину и поездка дается тяжело. Вы можете

быстро оценить три основные упомянутые абстракции, связанные с автомобилем, чтобы определить источник проблемы. Если с первыми двумя абстракциями все в порядке (ваш автомобиль и способ вождения) и ни одна из них не является проблемой, можете сузить проблему до самой дороги. В таком случае выясняется, что дорога ухабистая. Теперь, если нужно, вы можете копнуть глубже в абстракцию дороги и подумать, почему ее состояние ухудшилось или, если она новая, почему строители так паршиво сделали свою работу.

Разработчики программного обеспечения используют абстракцию в качестве инструмента при создании операционной системы и ее приложений. В программном обеспечении существует множество терминов для абстрактного разделения, включая *«подсистему»*, *«модуль»* и *«пакет»*, но в этой главе мы будем применять термин *«компонент»*, потому что так проще. При создании программного компонента разработчики обычно не задумываются о внутренней структуре других компонентов, но рассматривают те из них, которые могут задействовать (чтобы не приходилось писать дополнительное ненужное программное обеспечение), и размышляют о том, как их использовать.

В этой главе мы подробно рассмотрим компоненты, составляющие систему Linux. Хотя каждый из них имеет огромное количество технических деталей, сейчас не будем брать их в расчет и сосредоточимся на том, что именно компоненты делают по отношению ко всей системе. А детали рассмотрим в последующих главах.

1.1. Уровни абстракции в системе Linux

Использование абстракции для разделения вычислительных систем на компоненты облегчает понимание, но не работает без организации процессов. Мы объединяем компоненты в слои или уровни классификации (или группы) в соответствии с тем, где они находятся между пользователем и оборудованием. Веб-браузеры, игры и т. п. находятся на верхнем уровне, на нижнем располагается память в компьютерном оборудовании — нули и единицы. Операционная система занимает множество промежуточных уровней.

Система Linux имеет три основных уровня. На рис. 1.1 показаны эти уровни и компоненты внутри каждого из них. Аппаратное оборудование компьютера (hardware) находится на базовом уровне. Оно включает в себя память и один или несколько процессоров (CPU) для выполнения вычислений, а также для чтения из памяти и записи в нее. Такие устройства, как диски и сетевые интерфейсы, — тоже часть аппаратного оборудования.

Следующий уровень — это *ядро (kernel)*, основа операционной системы. Ядро — это программное обеспечение, содержащееся в памяти. Оно сообщает процессору, где находится его следующая задача. Выступая в качестве посредника, ядро управляет оборудованием (особенно оперативной памятью) и является основным интерфейсом между оборудованием и любой запущенной программой.

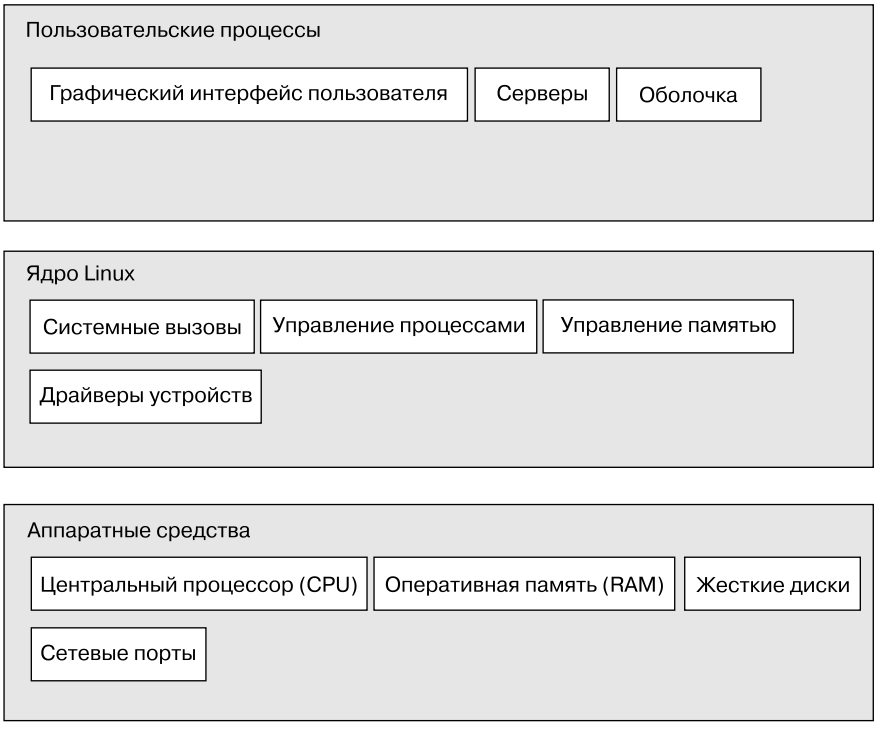


Рис. 1.1. Основные компоненты системы Linux

Процессы (processes) — запущенные программы, которыми управляет ядро, — в совокупности составляют верхний уровень системы, называемый *пользовательским пространством (user space)*. (Более конкретный термин для процесса — *«пользовательский процесс» (user process)*, независимо от того, взаимодействует ли пользователь непосредственно с процессом. Например, все веб-серверы работают как пользовательские процессы.)

Существует огромная разница между тем, как работает ядро и пользовательские процессы: ядро работает в *режиме ядра (kernel mode)*, а пользовательские процессы — *в пользовательском режиме*. Код, работающий в режиме ядра, имеет неограниченный доступ к процессору и оперативной памяти. Такая мощная, но опасная привилегия позволяет легко повредить ядро и вывести из строя всю систему. Область памяти, доступ к которой может получить только ядро, называется *пространством ядра (kernel space)*.

Пользовательский же режим ограничен доступом к подмножеству памяти (обычно довольно небольшому) и безопасным операциям процессора. *Пользовательское пространство* — это части основной памяти, к которым могут получить доступ пользовательские процессы. Если процесс совершает ошибку и выходит из строя,

последствия локальны и устраняются с помощью ядра. Это означает, что если ваш веб-браузер выйдет из строя, он не будет продолжать научные вычисления, которые выполнялись в фоновом режиме в течение нескольких дней.

Теоретически, пользовательский процесс, вышедший из строя, не может нанести серьезного ущерба остальной части системы. На самом деле это зависит от того, что вы считаете серьезным ущербом, а также от особых привилегий процесса, ведь некоторым процессам разрешено делать больше, чем другим. Например, может ли пользовательский процесс полностью уничтожить данные на диске? Если имеет необходимые привилегии — да, и это довольно опасно. Однако существуют меры предосторожности, и большинству процессов просто не разрешается сеять хаос в системе.

ПРИМЕЧАНИЕ

Ядро Linux может запускать потоки ядра (kernel threads), очень похожие на процессы, но имеющие доступ к пространству ядра, например kthreadd и kblockd.

1.2. Оборудование: оперативная память

Оперативная память — это, пожалуй, самый важный элемент оборудования компьютерной системы. В своей первоначальной форме оперативная память — это просто огромная область хранения для группы нулей и единиц. Каждый слот для нуля или единицы называется битом. Именно в оперативной памяти хранятся запущенные ядро и процессы — по сути, тоже большие наборы битов. Все входные и выходные данные с периферийных устройств проходят через оперативную память также в виде набора битов. Процессор — это оператор памяти, он считывает свои инструкции и данные из памяти и записывает данные обратно в память.

В отношении памяти, процессов, ядра и других частей компьютерной системы часто используется термин *«состояние» (state)*. Строго говоря, состояние — это особое расположение битов. Например, если у вас есть четыре бита в памяти, то биты 0110, 0001 и 1011 представляют три разных состояния системы.

Если учесть, что один процесс может легко состоять из миллионов битов в памяти, то когда речь идет о состояниях, проще использовать абстрактные термины. Вместо того чтобы описывать состояние с помощью битов, процесс описывают как завершенный или происходящий в данный момент. Например, вы можете сказать: «Процесс ожидает ввода» или «Процесс находится на втором этапе запуска».

ПРИМЕЧАНИЕ

Поскольку принято ссылаться на состояние в абстрактных терминах, а не на фактические биты, термин «образ» (image) относится к определенному физическому расположению битов.

1.3. Ядро

Почему мы говорим об оперативной памяти и состояниях? Почти все, что делает ядро, связано с оперативной памятью. Одной из задач ядра является разделение памяти на множество областей, и оно должно постоянно отслеживать определенную информацию об их состоянии. Каждый процесс получает некоторую часть памяти, и ядро должно обеспечивать необходимое количество памяти для каждого из процессов. Ядро отвечает за управление задачами в четырех основных областях системы.

- **Процессы.** Ядро определяет, каким процессам разрешено использовать процессор.
- **Память.** Ядро должно отслеживать распределение памяти: сколько в данный момент выделено конкретному процессу, сколько можно разделить между другими процессами и сколько свободно.
- **Драйверы устройств.** Ядро действует как интерфейс между оборудованием (например, диском) и процессами. Обычно оно управляет подключенным оборудованием.
- **Системные вызовы и поддержка.** Процессы обычно используют системные вызовы для связи с ядром.

Далее мы кратко изучим каждую из этих областей.

ПРИМЕЧАНИЕ

Если вы хотите изучить работу ядра более подробно, то я советую прочитать две отличные книги на эту тему: десятое издание книги Ави Зильбершаца Operating System Concepts (Wiley, 2018) и четвертое издание книги Эндрю Таненбаума Modern Operating Systems (Prentice Hall, 2014).

1.3.1. Управление процессами

Управление процессами описывает запуск, приостановку, возобновление, планирование и завершение процессов. Концепции, лежащие в основе запуска и завершения процессов, довольно просты, но описание того, как процессом используется процессор, немного сложнее.

В любой современной операционной системе многие процессы выполняются параллельно. Например, у вас могут быть открыты веб-браузер и электронная таблица одновременно. Однако все не так, как кажется: процессы, стоящие за этими приложениями, обычно не выполняются в одно и то же время.

Рассмотрим систему с одноядерным процессором. Многие процессы могут задействовать процессор, но только один из них фактически использует процессор

в любой момент времени. На практике процесс работает с процессором в течение небольшой доли секунды и делает паузу, затем другой процесс занимает процессор в течение еще одной небольшой доли секунды, потом в дело вступает еще один процесс и т. д. Акт передачи одним процессом управления процессором другому процессу называется *переключением контекста*.

Каждый отрезок времени, называемый *квантом времени*, позволяет процессу выполнить значительные вычисления (и действительно, процесс часто завершает свою задачу в течение одного кванта времени). Однако из-за того, что кванты очень малы, пользователи их не воспринимают, и возникает ощущение, что система выполняет несколько процессов одновременно (режим многозадачности).

Ядро отвечает за переключение контекста. Чтобы понять, как это работает, рассмотрим ситуацию, в которой процесс работает в пользовательском режиме, но его временной квант истек. Вот что происходит.

1. Процессор (фактическое оборудование) прерывает текущий процесс на основе внутреннего таймера, переключается в режим ядра и передает управление обратно ядру.
2. Ядро записывает текущее состояние процессора и памяти, что необходимо для возобновления только что прерванного процесса.
3. Ядро выполняет любые задачи, которые могли возникнуть в течение предыдущего временного кванта (например, сбор данных из ввода-вывода).
4. Теперь ядро готово к запуску другого процесса. Оно анализирует список процессов, готовых к запуску, и выбирает один из них.
5. Ядро подготавливает память для нового процесса, а затем готовит к нему процессор.
6. Ядро сообщает процессору длительность временного кванта для нового процесса.
7. Ядро переключает процессор в пользовательский режим и передает управление процессором процессу.

Переключение контекста позволяет понять, когда именно запускается ядро. Суть заключается в том, что ядро запускается между временными квантами процесса во время переключения контекста.

В случае многопроцессорной системы, как и в большинстве современных машин, все немного сложнее, потому что ядро не перестает управлять текущим процессором, чтобы позволить процессу работать на другом процессоре, и одновременно могут выполняться несколько процессов. Но чтобы максимально задействовать все доступные процессоры, ядро в любом случае выполняет необходимые шаги (и может использовать определенные лазейки, чтобы занять больше времени процессора для себя).

1.3.2. Управление памятью

Ядро должно управлять памятью во время переключения контекста, а это довольно сложная задача. Должны выполняться следующие условия.

- Ядро должно иметь в памяти выделенную область, к которой пользовательские процессы не могут получить доступ.
- Каждому пользовательскому процессу необходима собственная область памяти.
- Один пользовательский процесс не может получить доступ к области памяти, выделенной другому процессу.
- Пользовательские процессы могут совместно работать с памятью.
- Часть памяти пользовательских процессов может быть доступна только для чтения.
- Система может использовать больше памяти, чем ее существует физически, задействуя дисковое пространство в качестве вспомогательного механизма.

К счастью для ядра, оно не одно выполняет всю работу. Современные процессоры включают в себя блок управления памятью (memory management unit, MMU), который обеспечивает схему доступа к памяти, называемую *виртуальной памятью*. При использовании виртуальной памяти процесс не получает прямого доступа к памяти через ее физическое местоположение в компьютерной системе. Вместо этого ядро настраивает каждый процесс, чтобы он действовал так, будто ему доступна вся система. Когда процесс обращается к части своей памяти, MMU перехватывает обращение и с помощью таблицы соответствий преобразует адрес памяти с точки зрения процесса в фактическое физическое местоположение памяти в системе. Ядро по-прежнему должно инициализировать, постоянно поддерживать и изменять таблицу соответствий адресов памяти. Например, во время переключения контекста ядро должно заменить таблицу соответствий исходного процесса на таблицу последующего процесса.

ПРИМЕЧАНИЕ

Реализация таблицы соответствий адресов памяти называется таблицей страниц.

Подробнее о том, как отслеживать производительность памяти, описано в главе 8.

1.3.3. Управления драйверами устройств

Роль ядра в работе с устройствами относительно проста. Устройство обычно доступно только в режиме ядра, поскольку неправильный доступ (например, пользовательский процесс, запрашивающий отключение питания) может привести к сбою системы. Значительная проблема заключается в том, что различные устройства редко имеют один и тот же интерфейс программирования, даже если устройства

выполняют одну и ту же задачу (например, две разные сетевые карты). Поэтому драйверы устройств традиционно являются частью ядра, и они стремятся представить единый интерфейс для пользовательских процессов, чтобы упростить работу разработчика программного обеспечения.

1.3.4. Системные вызовы и поддержка

Существует несколько других функций ядра, доступных пользовательским процессам. К примеру, системные вызовы (system calls, syscalls) выполняют определенные задачи, которые сам по себе пользовательский процесс выполнить не может. Например, все действия по открытию, чтению и записи файлов связаны с системными вызовами.

Два системных вызова, `fork()` и `exec()`, важны для понимания того, как запускаются процессы.

- `fork()` — когда процесс вызывает `fork()`, ядро создает почти идентичную копию процесса.
- `exec()` — когда процесс вызывает `exec(program)`, ядро загружает и запускает программу `program`, заменяя текущий процесс.

Все новые пользовательские процессы в системе Linux, за исключением процесса `init` (см. главу 6), запускаются в результате вызова `fork()`, и в большинстве случаев `exec()` применяется для запуска новой программы вместо запуска копии существующего процесса. Возьмем простой пример — любую программу, которую вы запускаете в командной строке, например, команду `ls`, предназначенную для отображения содержимого каталога. Когда вы вводите `ls` в окно терминала, оболочка, работающая внутри этого окна, вызывает `fork()` для создания копии оболочки, а затем новая копия оболочки вызывает `exec(ls)` для запуска команды `ls`. На рис. 1.2 показана последовательность исполнения процессов и системных вызовов для запуска команды `ls`.

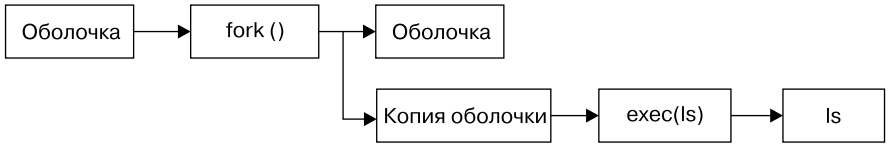


Рис. 1.2. Запуск нового процесса

ПРИМЕЧАНИЕ

Системные вызовы обычно обозначаются круглыми скобками. В примере на рис. 1.2 процесс, запрашивающий ядро для создания другого процесса, должен выполнить системный вызов fork (). Использование круглых скобок зависит от того, как именно вызов будет описан на языке программирования С. Вам не нужно знать язык С, чтобы понять

эту книгу, просто помните, что системный вызов — это взаимодействие между процессом и ядром. Кроме того, книга упрощает некоторые группы системных вызовов. Например, exec() относится ко всему семейству системных вызовов, которые выполняют одну и ту же задачу, но различаются по способу программирования. Существует также вариант процесса, называемый потоком, который мы рассмотрим в главе 8.

Ядро поддерживает пользовательские процессы с функциями, отличными от традиционных системных вызовов, наиболее распространенными из которых являются *псевдоустройства*. Они выглядят как устройства для пользовательских процессов, но реализуются исключительно в программном обеспечении. Это означает, что технически их не должно быть в ядре, но обычно они там присутствуют из практической необходимости. Например, устройство генератора случайных чисел ядра (`/dev/random`) было бы трудно безопасно реализовать с помощью пользовательского процесса.

ПРИМЕЧАНИЕ

Технически пользовательский процесс, который обращается к псевдоустройству, должен задействовать системный вызов, чтобы открыть устройство, поэтому процессы не могут полностью избежать применения системных вызовов.

1.4. Пользовательское пространство

Как уже упоминалось, оперативная память, выделяемая ядром для пользовательских процессов, называется *пользовательским пространством*. Поскольку процесс — это просто состояние (или образ) в памяти, пользовательское пространство соответствует и всем запущенным процессам в памяти. (Для обозначения пользовательского пространства применяется также более неформальный термин *userland*, иногда он означает программы, запущенные в пользовательском пространстве.)

Бóльшая часть реальных действий в системе Linux происходит в пользовательском пространстве. Хотя все процессы, по существу, одинаковы для ядра, они выполняют разные задачи для пользователей. Существует элементарная структура уровней (или слоев) обслуживания для различных типов системных компонентов, которые представляют пользовательские процессы. На рис. 1.3 показано, как примерный набор компонентов сочетается и взаимодействует в системе Linux. Основные службы находятся на нижнем уровне (ближе всего к ядру), служебные — посередине, а приложения, с которыми соприкасаются пользователи, — сверху. Рисунок 1.3 представляет собой значительно упрощенную схему, поскольку отображает только шесть компонентов, но на ней видно, что компоненты вверху находятся ближе всего к пользователю (пользовательский интерфейс и браузер), компоненты на среднем уровне включают сервер кэширования доменных имен, используемый веб-браузером, и есть несколько меньших компонентов снизу.

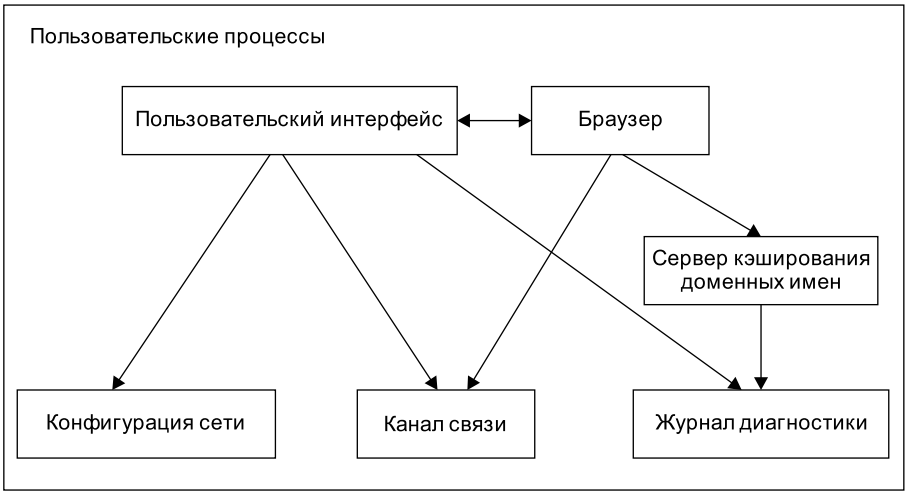


Рис. 1.3. Типы процессов и их взаимодействие

Нижний уровень, как правило, состоит из небольших компонентов, которые выполняют отдельные несложные задачи. На среднем уровне находятся более крупные компоненты, такие как службы почты, печати и баз данных. Наконец, компоненты на верхнем уровне выполняют сложные задачи, которыми пользователь часто управляет напрямую. Кроме того, одни компоненты используют другие. Как правило, если один компонент хочет использовать другой, то последний находится либо на том же уровне обслуживания, либо на уровень ниже.

Рисунок 1.3 лишь приблизительно показывает, как организовано пользовательское пространство. На самом деле в нем нет никаких правил. Например, основная масса приложений и служб записывают диагностические сообщения в логи *(logs)*. Большинство программ задействуют для этого стандартную службу логирования syslog, но некоторые предпочитают писать логи самостоятельно.

Кроме того, некоторые компоненты пользовательского пространства трудно классифицировать. Серверные компоненты, такие как веб-серверы и серверы баз данных, можно считать приложениями очень высокого уровня, так как они выполняют сложные задачи, поэтому их можно поместить на верхний уровень в схеме, изображенной на рис. 1.3. Однако пользовательские приложения могут зависеть от этих серверов, так что такие компоненты можно разместить и на среднем уровне.

1.5. Пользователи

Ядро Linux поддерживает традиционную концепцию пользователя Unix. *Пользователь (user)* — это сущность, которая может запускать процессы и владеть файлами. Чаще всего он ассоциируется с именем пользователя (username), например,

в системе может быть пользователь с именем *billyjoe*. Однако ядро не управляет именами пользователей, оно лишь идентифицирует пользователей с помощью простых *числовых идентификаторов пользователей* (user ID, UID). (Подробнее о соответствии имен пользователей их идентификаторам вы узнаете в главе 7.)

Пользователи существуют в основном для поддержки прав и границ в системе. Каждый процесс пользовательского пространства имеет владельца (owner) и, как говорят, выполняется от его имени. Пользователь может прекратить свои процессы или изменить их поведение (в определенных пределах), но не может вмешиваться в процессы других пользователей. Кроме того, пользователи могут владеть файлами и выбирать, делиться ли ими с другими пользователями.

Система Linux обычно имеет несколько пользователей в дополнение к тем, которые соответствуют реальным пользователям. Подробнее об этом написано в главе 3, однако уже сейчас нужно знать о самом важном пользователе — *суперпользователе (superuser, root)*. Суперпользователь — это исключение из рассмотренных ранее правил, поскольку он может завершить и изменить процессы других пользователей и получить доступ к любому файлу в локальной системе. Человек, который может действовать как суперпользователь, то есть имеет корневой доступ (root access), считается администратором в традиционной системе Unix.

ПРИМЕЧАНИЕ

Работа в системе в качестве суперпользователя может быть опасной. Трудно бывает выявить и исправить ошибки, потому что система позволит вам делать все, что угодно, даже если это навредит ей. По этой причине разработчики систем стараются сделать подобный доступ как можно более ненужным: например, система не требует корневого доступа для переключения между беспроводными сетями на ноутбуке. Кроме того, как бы ни был силен суперпользователь, он по-прежнему задействует пользовательский режим операционной системы, а не режим ядра.

Группы (groups) — это наборы пользователей. Основная цель групп — позволить пользователю получать доступ к файлам совместно с другими членами группы.

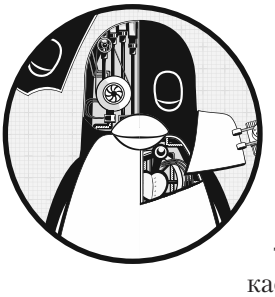
1.6. Что дальше?

Мы рассмотрели, что представляет собой работающая система Linux. Пользовательские процессы образуют среду, с которой вы непосредственно взаимодействуете, ядро управляет процессами и оборудованием. И ядро, и процессы находятся в памяти.

Это важная информация, но ее недостаточно: нельзя узнать подробности о системе Linux, не углубившись в нее и «не запачкав руки». В следующей главе мы начнем путешествие в глубины Linux с рассмотрения основ пользовательского пространства. Попутно вы узнаете о важной части системы Linux, которая не обсуждалась в этой главе, — долговременное *хранилище* (диски, файлы и т. п.). В конце концов, вам же нужно где-то хранить свои программы и данные, верно?

2

Основные команды и иерархия каталогов



В этой главе мы разберем основные команды и утилиты Unix, с которыми вы будете сталкиваться на протяжении всей книги. Эта информация начального уровня, и вам, возможно, уже известна бóльшая ее часть. Даже если вам кажется, что вы все знаете, потратьте несколько минут на то, чтобы пролистать главу и убедиться в этом, особенно это касается темы иерархии каталогов в разделе 2.19.

Вы можете задать вопрос: «Почему же команды Unix? Разве это не книга о том, как работает Linux?» Это так, но Linux — это разновидность Unix. В этой главе слово «Unix» будет появляться чаще, чем «Linux», а все, что вы узнаете, относится также к системам BSD и другим, основанным на Unix-системе. Я постарался уменьшить количество расширений пользовательского интерфейса, специфичных для Linux, не только для того, чтобы дать основы для применения других операционных систем, но и потому, что эти расширения, как правило, нестабильны. Вы сможете гораздо быстрее адаптироваться к новым версиям Linux, если будете знать основные команды. Кроме того, знание этих команд поможет вам лучше понять ядро, так как многие из них непосредственно соответствуют системным вызовам.

ПРИМЕЧАНИЕ

Чтобы больше узнать о системе Unix, прочитайте книги для начинающих пользователей: второе издание книги The Linux Command Line (No Starch Press, 2019), второе издание книги Пола Абрахама UNIX for the Impatient (Addison-Wesley Professional, 1995), а также пятое издание книги Learning the UNIX Operating System (O'Reilly, 2001).

2.1. Оболочка Bourne Shell (bash): /bin/sh

Оболочка (shell) — одна из самых важных частей системы Unix. Это программа, выполняющая команды, которые пользователи вводят в окно терминала. Эти команды могут быть другими программами или встроенными функциями оболочки. Оболочка — это еще и среда программирования. Программисты Unix часто разбивают типичные задачи на более мелкие компоненты и применяют оболочку для управления ими.

Многие важные части системы на самом деле являются *сценариями* (скриптами, shell scripts) оболочки — текстовыми файлами, которые содержат последовательность команд. Если раньше вы работали с системой MS-DOS, вы можете думать о скриптах оболочки как о мощных .BAT-файлах. Поскольку скрипты важны для работы системы, вся глава 11 полностью посвящена им.

По ходу книги вы будете пополнять свои знания о том, как обращаться с командами с помощью оболочки. Одно из важных ее преимуществ состоит в том, что если вы допустили ошибку, то можете увидеть, какой текст набрали, исправить ошибку, а затем повторить снова.

Существует множество различных оболочек Unix, но все они — производные от оболочки Bourne shell (`/bin/sh`), стандартной оболочки, разработанной в компании Bell Labs для ранних версий Unix. Любая система Unix требует ту или иную версию Bourne shell для корректной работы, что будет показано на протяжении всей этой книги.

Система Linux использует расширенную версию оболочки Bourne под названием `bash`, или Bourne-again. Оболочка `bash` — это оболочка по умолчанию в большинстве дистрибутивов Linux, и каталог `/bin/sh` обычно указывает на `bash` в системе Linux. Для выполнения заданий из книги необходимо использовать оболочку `bash`.

ПРИМЕЧАНИЕ

Оболочка bash может быть не установлена в качестве оболочки по умолчанию, если вы пользуетесь этой главой как руководством для работы с системой Unix в организации, где не являетесь системным администратором. Вы можете изменить свою оболочку с помощью команды chsh или обратиться за помощью к системному администратору.

2.2. Использование оболочки

После установки системы Linux необходимо создать как минимум одного обычного пользователя для своей личной учетной записи. В этой главе зайдите в систему как обычный пользователь.

2.2.1. Окно оболочки

После входа в систему откройте окно оболочки (часто называется терминалом). Самый простой способ — войти с помощью графического интерфейса, такого как

Gnome или KDE, — открыть приложение Terminal, которое запускает оболочку внутри нового окна. После того как вы откроете оболочку, в верхней части окна появится подсказка, которая обычно заканчивается знаком доллара (`$`). В системе Ubuntu это приглашение должно выглядеть следующим образом: `name@host:path$`, а в Fedora — `[name@host path]$`, где `name` — ваше имя пользователя, `host` — имя вашей машины, а `path` — текущий рабочий каталог (см. подраздел 2.4.1). Если вы знакомы с системой Windows, окно оболочки будет выглядеть примерно как командная строка DOS, в macOS приложение Terminal, по сути, совпадает с окном оболочки Linux.

Эта книга содержит множество команд, которые вы будете вводить в командной строке. Все они начинаются с одного знака `$`, который обозначает приглашение оболочки. Например, наберите следующую команду (только часть, которая выделена жирным шрифтом, а не знак `$`) и нажмите клавишу Enter:

```
$ echo Hello there.
```

ПРИМЕЧАНИЕ

Многие команды оболочки в этой книге начинаются со знака #. Вы должны запускать их от имени суперпользователя (root), поэтому они требуют особой осторожности. Лучше всего при запуске таких команд применять команду sudo, чтобы защитить систему и записать все действия в лог, который вы позже сможете просмотреть на предмет возможных ошибок. О том, как это сделать, говорится в разделе 2.20.

Теперь введите команду

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

Она отображает содержимое системного файла `/etc/passwd`, а затем возвращает приглашение оболочки. Пока не обращайте внимания на вывод команды, вы узнаете все об этом в главе 7.

Текст команды обычно начинается с названия программы для запуска и может сопровождаться аргументами (arguments), которые сообщают программе, с чем работать и как именно. В приведенном примере используется программа `cat`, в нее добавлен один аргумент, `/etc/passwd`. Многие аргументы являются параметрами (options), которые изменяют поведение программы по умолчанию и обычно начинаются с дефиса (`-`). Далее это будет показано при описании команды `ls`. Однако есть исключения, которые не соответствуют этой обычной структуре команд, — встроенные модули оболочки и переменные окружения.

2.2.2. Программа cat

Программа `cat` одна из самых простых для понимания в системах Unix, она просто выводит содержимое одного или нескольких файлов или другого источника ввода. Синтаксис команды `cat` выглядит следующим образом:

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

При выполнении `cat` выводит содержимое файлов *`file1`*, *`file2`* и любых других файлов, указанных в качестве аргументов (в примере это обозначено многоточием `...`), а затем завершает работу. Программа называется `cat`, потому что она выполняет *конкатенацию* (concatenation, присоединение), когда выводит содержимое нескольких файлов. Существует много способов запустить `cat`. Изучим процесс ввода-вывода Unix с помощью этой программы.

2.2.3. Стандартный поток ввода (stdin) и стандартный поток вывода (stdout)

Процессы Unix используют потоки ввода-вывода (Input/Output, I/O streams) для чтения и записи данных. Они считывают данные из входных потоков и записывают данные в выходные потоки. Сами по себе потоки очень гибкие. Например, источником потока ввода могут быть файл, устройство, окно терминала или даже выходной поток из другого процесса.

Чтобы увидеть поток ввода в работе, введите команду `cat` (без аргументов) и нажмите клавишу Enter. На этот раз вывод появится не сразу, и вы не получите приглашение оболочки, потому что `cat` все еще работает. Теперь введите что-нибудь и нажмите клавишу Enter в конце каждой строки. Теперь команда `cat` повторяет любую строку, которую вы вводите. Как только вам это наскучит, нажмите сочетание клавиш Ctrl+D в пустой строке, чтобы завершить команду `cat` и вернуться к командной строке.

Причина, по которой команда `cat` так сработала, связана с потоками. Если вы не указываете входное имя файла, `cat` считывает данные из стандартного потока ввода (standard input), предоставляемого ядром Linux, а не из потока, подключенного к файлу. В этом случае стандартный поток ввода подключен к терминалу, где вы запускаете команду `cat`.

ПРИМЕЧАНИЕ

Нажатие сочетания клавиш Ctrl+D на пустой строке останавливает текущую стандартную запись ввода с терминала с сообщением EOF (end-of-file, конец файла) и в большинстве случаев завершит и саму программу. Не путайте с сочетанием клавиш Ctrl+C, которое обычно завершает программу независимо от ее ввода или вывода.

Стандартный поток вывода (standard output) работает точно так же. Ядро предоставляет каждому процессу стандартный поток вывода, в который процессы могут записывать свои выходные данные. Команда `cat` всегда записывает свои выходные данные в стандартный поток вывода. При запуске `cat` в терминале в приведенных ранее примерах стандартный вывод был подключен к этому терминалу, так что именно там отображался вывод.

Стандартный ввод и вывод часто сокращают как *stdin* и *stdout* соответственно. Многие команды работают так же, как команда `cat`: если вы не указываете входной файл,

команда читает из `stdin`. Вывод же немного отличается. Некоторые программы (например, `cat`) отправляют выходные данные только в `stdout`, но другие имеют возможность отправлять выходные данные непосредственно в файлы.

Существует и третий стандартный поток ввода-вывода, называемый *стандартной ошибкой (standard error)*. О нем рассказывается в подразделе 2.14.1.

Одна из самых полезных особенностей стандартных потоков заключается в том, что вы можете легко управлять ими для чтения и записи не только в терминале (см. раздел 2.14). В частности, вы узнаете, как подключать потоки к файлам и другим процессам.

2.3. Основные команды

Теперь рассмотрим еще несколько команд Unix. Большинство из приводимых здесь программ работают только с несколькими аргументами, а некоторые имеют так много параметров и форматов, что их сокращенный список был бы бессмысленным. Далее представлен упрощенный список основных команд — вам пока не нужны все детали.

2.3.1. Команда ls

Команда `ls` перечисляет (lists) содержимое каталога. По умолчанию используется текущий каталог, но вы можете добавить любой каталог или файл в качестве аргумента, и для этой операции существует много полезных параметров. Например, применяйте `ls -l` для подробного (long, длинного) списка и `ls -F` для отображения информации о типе файла. Вот пример длинного списка, он включает владельца файла (столбец 3), группу (столбец 4), размер файла (столбец 5) и дату/время изменения (между столбцом 5 и именем файла):

```
$ ls -l
total 3616
-rw-r--r-- 1 juser users 3804   May 28 10:40 abusive.c
-rw-r--r-- 1 juser users 4165   Aug 13 10:01 battery.zip
-rw-r--r-- 1 juser users 131219 Aug 13 10:33 beav_1.40-13.tar.gz
-rw-r--r-- 1 juser users 6255   May 20 14:34 country.c
drwxr-xr-x 2 juser users 4096   Jul 17 20:00 cs335
-rwxr-xr-x 1 juser users 7108   Jun 16 13:05 dhry
-rw-r--r-- 1 juser users 11309  Aug 13 10:26 dhry.c
-rw-r--r-- 1 juser users 56     Jul 9 15:30 doit
drwxr-xr-x 6 juser users 4096   Feb 20 13:51 dw
drwxr-xr-x 3 juser users 4096   Jul 1 16:05 hough-stuff
```

Подробнее о столбце 1 этого вывода рассказывается в разделе 2.17. Вы можете пока игнорировать столбец 2 — это количество жестких ссылок на файл, о нем говорится в разделе 4.6.

2.3.2. Команда cp

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

```
$ cp file1 file2
```

Вы также можете скопировать файл в другой каталог, сохранив в нем то же имя файла:

```
$ cp file dir
```

Чтобы скопировать более одного файла в каталог (папку) с именем `dir`, попробуйте применить команду из следующего примера, которая копирует три файла:

```
$ cp file1 file2 file3 dir
```

2.3.3. Команда mv

Команда `mv` (move) работает так же, как и команда `cp`. В своей простейшей форме она переименовывает файл. Например, чтобы переименовать файл `file1` в `file2`, введите следующее:

```
$ mv file1 file2
```

Можно использовать команду `mv` для перемещения файлов в другие каталоги таким же образом, что и команду `cp`.

2.3.4. Команда touch

Команда `touch` может создать файл. Если целевой файл уже существует, `touch` не изменяет его, но обновляет временную метку (timestamp) его изменения. Например, чтобы создать пустой файл, введите следующее:

```
$ touch file
```

Затем запустите в нем команду `ls -l`. Появится вывод, как в следующем примере, где дата и время указывают, когда была запущена команда `touch`:

```
$ ls -l file
-rw-r--r-- 1 juser users 0 May 21 18:32 file
```

Чтобы увидеть обновление метки времени, подождите не менее минуты, а затем снова выполните ту же команду `touch`. Метка времени, возвращенная из команды `ls -l`, будет обновлена.

2.3.5. Команда rm

Команда `rm` удаляет (removes) файл. После этого он обычно исчезает из вашей системы и, как правило, не может быть восстановлен, если вы не создали его резервную копию:

```
$ rm file
```

2.3.6. Команда echo

Команда `echo` выводит свои аргументы в стандартный вывод:

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

Команда `echo` очень полезна для шаблонов поиска и подстановки имен файлов оболочки (подстановочных знаков (wildcards), таких как `*`) и переменных (таких, как `$HOME`), о которых вы узнаете позже в этой главе.

2.4. Перемещение по каталогам

Иерархия каталогов Unix начинается с каталога `/`, называемого также *корневым (root directory)*. Разделителем каталогов является косая черта (`/`), но не обратная косая черта (`\`). В корневом каталоге есть несколько стандартных подкаталогов, таких как `/usr`, о которых вы узнаете из раздела 2.19.

Ссылаясь на файл или каталог, вы указываете путь (path) или имя пути (pathname). Путь, начинающийся со знака `/` (например, `/usr/lib`), — это полный, или *абсолютный путь*.

Компонент пути, обозначенный двумя точками (`..`), указывает на то, что это *родительский каталог*. Например, если вы работаете в каталоге `/usr/lib`, путь `..` ведет к `/usr`. Аналогично `../bin` ведет к `/usr/bin`.

Одна точка (`.`) относится к *текущему каталогу*. Например, если вы находитесь в каталоге `/usr/lib`, путь `.` по-прежнему ведет к `/usr/lib`, а путь `./X11` — к каталогу `/usr/lib/X11`. Нет нужды применять одну точку `.` очень часто, потому что большинство команд по умолчанию задействуют текущий каталог, если путь не начинается со знака `/` (поэтому вместо `./X11` можно использовать `X11`).

Путь, не начинающийся с косой черты `/`, называется *относительным*. Большую часть времени вы будете работать с относительными путями, потому что уже находитесь в нужном каталоге или рядом с ним. Теперь, когда у вас есть представление об основных механизмах работы с каталогами, перечислим некоторые основные команды каталогов.

2.4.1. Команда cd

Текущий рабочий каталог (current working directory) — это каталог, в котором в данный момент находится процесс (например, командная оболочка). В дополнение к командной строке по умолчанию в большинстве дистрибутивов Linux вы можете просмотреть текущий каталог с помощью команды `pwd`, описанной в подразделе 2.5.3.

Каждый процесс может самостоятельно установить собственный текущий рабочий каталог. Команда `cd` изменяет текущий рабочий каталог оболочки:

```
$ cd dir
```

Если вы опустите `dir`, оболочка вернется в ваш *домашний каталог (home directory)* — первоначальный каталог входа в систему. Некоторые программы обозначают домашний каталог символом ~ (волнистая черта, *тильда*).

> **ПРИМЕЧАНИЕ**
>
> Команда cd встроена в командную оболочку. Она не работает как отдельная программа, потому что, если бы она запускалась как подпроцесс, то не смогла бы (чаще всего) изменить текущий родительский рабочий каталог. В данный момент это может показаться не особо важным отличием, однако бывают моменты, когда этот факт может прояснить путаницу.

2.4.2. Команда mkdir

Команда `mkdir` создает новый каталог `dir`:

```
$ mkdir dir
```

2.4.3. Команда rmdir

Команда `rmdir` удаляет каталог `dir`:

```
$ rmdir dir
```

Если в каталоге `dir` есть данные, эта команда не завершается и выводит ошибку. Однако вы можете не захотеть сначала тщательно удалить все файлы и подкаталоги внутри `dir`. Удаляйте каталог и его содержимое с помощью команды `rm -r dir`, но будьте осторожны! Это одна из немногих команд, которая способна нанести серьезный ущерб, особенно если вы запускаете ее от имени суперпользователя. Параметр `-r` указывает на рекурсивное удаление (recursive delete), убирающее все внутри `dir`. Не применяйте флаг `-r` с шаблонами поиска, такими как знак звездочки (*). И, конечно же, всегда перепроверяйте команду, прежде чем запускать ее.

2.4.4. Шаблоны поиска (переменные Wildcards)

Оболочка может сопоставлять простые шаблоны с именами файлов и каталогов, этот процесс известен как *подстановка имен файлов* (globbing). Это похоже на концепцию подстановочных знаков в других системах. Самым простым из них является символ *, обозначающий любое количество произвольных символов. Например, следующая команда выводит список файлов, находящихся в текущем каталоге:

```
$ echo *
```

Оболочка сопоставляет шаблоны с именами файлов, подставляет имена файлов в соответствии с аргументами, а затем запускает модифицированную команду. Подстановка называется *расширением шаблона*, потому что оболочка подставляет все совпадающие имена файлов под упрощенное выражение. Вот несколько способов использования символа * для расширения имен файлов.

- Команда `at*` расширяет вывод до всех имен файлов, которые начинаются с `at`.
- Команда `*at` расширяет вывод до всех имен файлов, которые заканчиваются на `at`.
- Команда `*at*` расширяет вывод до всех имен файлов, которые содержат `at`.

Если ни один файл не соответствует шаблону, оболочка `bash` не производит расширение и команда выполняется как написано буквально, без подстановки специальных символов, таких как *. Например, попробуйте выполнить команду `echo *dfkdsafh`.

ПРИМЕЧАНИЕ

Если вы привыкли к командной строке Windows, то можете машинально ввести *.*, чтобы обратиться ко всем файлам. Откажитесь от этой привычки. В Linux и других версиях Unix используется * как символ соответствия всем файлам. В оболочке Unix *.* соответствует только файлам и каталогам, в именах которых есть символ точки (.). Имена файлов Unix не нуждаются в расширениях и часто не имеют их.

Другой символ шаблона поиска — знак вопроса (`?`) — указывает на соответствие ровно одному произвольному символу. Например, `b?at` соответствует `boat` и `brat`.

Если вы не хотите, чтобы оболочка расширяла шаблон в команде, заключите его в одинарные кавычки (`'`). Например, команда `echo '*'` выводит звездочку. Это удобно для выполнения некоторых команд, описанных в следующем разделе, таких как `grep` и `find`. (Вы узнаете больше о заключении в кавычки в разделе 11.2.)

ПРИМЕЧАНИЕ

Важно помнить, что оболочка выполняет расширения перед выполнением команд. Поэтому, если символ * превращается в команду без расширения, оболочка больше ничего с ней не сделает: дальнейшие интерпретации зависят от самой команды.

Конечно, это еще не все возможности шаблонов оболочки, однако пока достаточно изучить символы * и `?`. В разделе 2.7 описывается, как ведут себя шаблоны с теми забавными файлами, которые начинаются с точки.

2.5. Команды среднего уровня

В этом разделе описаны наиболее важные команды Unix среднего уровня.

2.5.1. Команда grep

Команда `grep` выводит строки из файла или потока ввода, соответствующие выражению. Например, чтобы напечатать строки из файла `/etc/passwd`, содержащие текст `root`, введите следующее:

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

Команда `grep` чрезвычайно удобна, когда нужно работать с несколькими файлами одновременно, поскольку она печатает имя файла в дополнение к найденной строке. Например, если вы хотите проверить каждый файл в `/etc`, содержащий слово `root`, используйте команду

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

Два наиболее важных параметра `grep` — это `-i` (для поиска без учета регистра) и `-v` (инвертированный поиск, вывод всех строк, которые *не* подпадают под выражение). Существует также более мощный вариант команды под названием `egrep` (является просто синонимом `grep -E`).

Команда `grep` понимает *регулярные выражения (regular expressions, regexp)*, шаблоны, которые основаны на теории информатики и очень распространены в утилитах Unix. Регулярные выражения более мощны, чем шаблоны в стиле подстановочных знаков, и у них другой синтаксис. Есть три важные вещи, которые следует помнить о регулярных выражениях:

- `.*` соответствует любому количеству символов, в том числе ни одному (например, * в шаблонах и подстановочных знаках);
- `.+` соответствует любому одному или нескольким символам;
- `.` соответствует ровно одному произвольному символу.

ПРИМЕЧАНИЕ

Страница руководства grep(1) содержит подробное описание регулярных выражений, но его довольно трудно читать. Чтобы узнать больше, вы можете прочитать третье издание книги Джеффри Э. Ф. Фридла «Регулярные выражения» (Символ-плюс, 2008) или просмотреть главу о регулярных выражениях в книге Тома Кристенсена и др. «Программирование на Perl» (Символ-плюс, 2014). Если вы любите математику и интересуетесь, откуда берутся регулярные выражения, изучите книгу Джеффри Уллмана и Джона Хопкрофта «Введение в теорию автоматов, языков и вычислений» (Диалектика, 2019).

2.5.2. Команда less

Команда `less` используется, когда файл довольно большой или вывод команды длинный и не помещается на экране.

Чтобы просмотреть большой файл, например `/usr/share/dict/words`, можете применить команду `less /usr/share/dict/words`. Содержимое файла будет

отображаться фрагментами по размеру окна терминала. Нажмите Пробел, чтобы перейти вперед по тексту файла, нажатие клавиши b (в нижнем регистре) поможет вернуться назад. Чтобы выйти, нажмите клавишу q.

ПРИМЕЧАНИЕ

Команда less — это расширенная версия старой программы more. Настольные компьютеры и серверы Linux используют команду less, но она не применяется по умолчанию для многих встроенных систем и других систем Unix. Если вы когда-нибудь столкнетесь с ситуацией, когда не сможете задействовать команду less, берите команду more.

Вы также можете искать текст внутри вывода команды `less`. Например, для поиска слова вперед по тексту можно ввести `/word`, а для поиска назад — `-?ord`. Когда будет найдено совпадение, нажмите клавишу n, чтобы продолжить поиск.

Как описано в разделе 2.14, вы можете отправить стандартный вывод практически любой программы непосредственно на стандартный ввод другой программы. Это полезно, когда у вас есть команда с большим количеством выходных данных и вы хотели бы использовать команду `less` для их просмотра. Вот пример отправки выходных данных команды `grep` в `less`:

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

Попробуйте сделать это. Вероятно, вы найдете много подобных применений для команды `less`.

2.5.3. Команда pwd

Команда `pwd` (print working directory, вывод рабочего каталога) просто выводит имя текущего рабочего каталога. Вы можете задаться вопросом, зачем она нужна, если большинство дистрибутивов Linux настраивают учетные записи пользователей с текущим рабочим каталогом в приглашении командной строки. На это есть две причины.

Во-первых, не все приглашения включают текущий рабочий каталог, особенно если в дальнейшем от него нужно избавиться, потому что он занимает много места. Для этого вам понадобится команда `pwd`.

Во-вторых, символические ссылки, о которых вы узнаете в разделе 2.17.2, иногда могут скрывать истинный полный путь к текущему рабочему каталогу. Используйте команду `pwd -P`, чтобы устранить эту путаницу.

2.5.4. Команда diff

Чтобы увидеть различия (difference) между двумя текстовыми файлами, задействуйте команду `diff`:

```
$ diff file1 file2
```

Несколько параметров могут управлять форматом вывода, и он по умолчанию наиболее понятен для пользователя. Однако большинство программистов предпочитают вывод из команды `diff -u`, когда им нужно перенаправить этот вывод куда-то еще, потому что автоматизированным инструментам легче работать с таким форматом.

2.5.5. Команда file

Если вы видите файл и не уверены в его формате, попробуйте команду `file`, чтобы узнать, сможет ли система распознать его:

```
$ file file
```

Удивительно, как много может сделать эта на первый взгляд незначительная команда.

2.5.6. Команды find и locate

Обидно, когда вы знаете, что определенный файл точно существует в каком-то из каталогов, но не знаете, где именно. Чтобы найти файл, запустите команду `find` с указанием каталога *dir*, как в следующем примере:

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

Как и большинство программ, описанных в этом разделе, команда `find` способна еще на некоторые необычные вещи. Однако не используйте такие параметры, как `-exec`, пока не выучите наизусть форму, показанную здесь, и не поймете, зачем нужны параметры `-name` и `-print`. Команда `find` принимает специальные символы, шаблоны, такие как *, но их нужно заключить в одинарные кавычки (`'*'`), чтобы не дать специальным символам выполниться непосредственно в оболочке. (Как говорилось в подразделе 2.4.4, оболочка расширяет шаблоны *перед* выполнением команд.)

В большинстве систем для поиска файлов используется также команда `locate`. Вместо того чтобы искать файл в режиме реального времени, `locate` выполняет поиск по индексу, который периодически создается системой. Поиск с помощью команды `locate` намного быстрее, чем с командой `find`, но если нужный файл новее созданного индекса, `locate` не найдет его.

2.5.7. Команды head и tail

Команды `head` и `tail` позволяют быстро просматривать часть файла или потока данных. Например, команда `head /etc/passwd` показывает первые 10 строк файла с паролями, а `tail /etc/passwd` — последние 10 строк.

Чтобы изменить количество отображаемых строк, используйте параметр `-n`, где `n` — это количество строк (например, `head -5 /etc/passwd`). Для вывода строк, начинающихся со строки `n`, используйте `tail +n`.

2.5.8. Команда sort

Команда `sort` сортирует строки текстового файла в алфавитно-цифровом порядке. Если строки файла начинаются с цифр и вы хотите отсортировать их в числовом порядке, используйте параметр `-n`. Параметр `-r` изменяет порядок сортировки на обратный.

2.6. Смена пароля и оболочки

Используйте команду `passwd` для изменения пароля. Система попросит ввести старый пароль, а затем дважды ввести новый.

Лучше всего создавать длинные пароли, которые представляют собой бессмысленные предложения, но которые легко запомнить. Чем длиннее пароль (с точки зрения числа символов), тем лучше; попробуйте использовать 16 символов или более. (Раньше количество символов было ограничено, поэтому система советовала добавлять необычные символы и т. п.)

Вы можете изменить свою оболочку с помощью команды `chsh` (на альтернативу, такую как `zsh`, `ksh` или `tcsh`), но имейте в виду, что в этой книге применяется оболочка `bash`, поэтому, если вы смените оболочку, некоторые примеры могут не сработать.

2.7. Файлы с точками

Перейдите в свой домашний каталог, введите команду `ls`, чтобы осмотреться, а затем запустите команду `ls -a`. Видите ли вы разницу в результатах? При запуске команды `ls` без параметра `-a` файлы конфигурации, называемые *файлами с точками* (дот-файлами, скрытыми файлами), не отображаются. Это файлы и каталоги, имена которых начинаются с точки (`.`). Обычные файлы с точками — это `.bashrc` и `.login`, а также каталоги с точками, такие как `.ssh`.

В файлах и каталогах с точками нет ничего особенного. Некоторые программы не показывают их по умолчанию, так что полного беспорядка при перечислении содержимого вашего домашнего каталога не будет. Например, команда `ls` не перечисляет файлы с точками, если вы не используете параметр `-a`. Кроме того, шаблоны оболочки не подпадают под файлы с точками, если вы явно не применяете такой шаблон, как `.*`.

ПРИМЕЧАНИЕ

Вы можете столкнуться с проблемой с шаблонами, потому что символ `.*` совпадает с точками . и .. (текущий и родительский каталоги). Используйте шаблон `.[^.]*` или `.??*`, чтобы вывести все файлы с точками, кроме текущего и родительского каталогов.

2.8. Переменные окружения и оболочки

Оболочка может хранить временные переменные, называемые *переменными оболочки*, содержащие значения текстовых строк. Переменные оболочки очень полезны для отслеживания значений в скриптах, а некоторые из них управляют поведением оболочки. (Например, оболочка `bash` считывает переменную `PS1` перед отображением приглашения.)

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

```
$ STUFF=blah
```

Здесь значение переменной с именем `STUFF` устанавливается равным значению `blah`. Чтобы обратиться к этой переменной, используйте `$STUFF` (например, попробуйте запустить команду `echo $STUFF`). Вы узнаете о многих примерах использования переменных оболочки в главе 11.

> **ПРИМЕЧАНИЕ**
>
> Не ставьте никаких пробелов вокруг знака равенства (=) при присвоении значения переменной.

Переменная окружения похожа на переменную оболочки, но она не специфична для оболочки. Все процессы в системах Unix имеют хранилище переменных окружения. Основное различие между переменными окружения и оболочки заключается в том, что операционная система передает все переменные окружения оболочки программам, выполняемым оболочкой, в то время как переменные оболочки недоступны в запускаемых вами командах.

Переменная окружения назначается с помощью команды `export`. Например, если вы хотите превратить переменную оболочки `$STUFF` в переменную окружения, введите следующее:

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

Поскольку дочерние процессы наследуют переменные окружения от родительских, многие программы считывают их для собственной настройки и получения параметров. Например, вы можете поместить свои любимые параметры командной строки `less` в переменную окружения `LESS`, и команда `less` будет использовать их при запуске. (Во многих руководствах есть раздел «Окружение» (ENVIRONMENT), описывающий эти переменные.)

2.9. Переменная пути PATH

`PATH` — это специальная переменная окружения, содержащая путь к команде (или сокращенно путь) — список системных каталогов, где оболочка пытается найти

команду. Например, при запуске команды `ls` оболочка выполняет поиск в каталогах, перечисленных в `PATH` для команды `ls`. Если программы с одинаковым именем появляются в нескольких каталогах в пути, оболочка запускает первую из найденных программ.

Если запустить команду `echo $PATH`, можно увидеть, что компоненты пути разделены двоеточиями (`:`), например:

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

Чтобы дать оболочке задание искать программы в большем количестве мест, измените переменную окружения `PATH`. Например, с помощью приведенной далее команды можно добавить в начало пути каталог `dir`, чтобы оболочка просматривала его перед поиском в любом другом каталоге переменной `PATH`:

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

Или можете добавить имя каталога в конец переменной `PATH`, в результате чего оболочка будет просматривать каталог `dir` последним:

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

ПРИМЕЧАНИЕ

Вы можете случайно стереть весь свой путь, если неправильно наберете $PATH при его изменении. Если это произошло, не паникуйте! Это не навсегда — можно просто запустить новую оболочку. (Для длительного эффекта вам нужно ввести путь неправильно при редактировании определенного файла конфигурации, но даже это нетрудно исправить.) Самый простой способ вернуться к нормальному состоянию оболочки — закрыть текущее окно терминала и запустить другое.

2.10. Специальные символы

При обсуждении Linux необходимо знать названия нескольких специальных символов, с которыми вы столкнетесь во время работы. Если вас забавляют подобные вещи, изучите словарь для хакеров *Jargon File* (www.catb.org/jargon/html/) или его печатную версию — книгу Эрика С. Рэймонда *New Hacker's Dictionary* (MIT Press, 1996).

В табл. 2.1 приведены избранные специальные символы, многие из которых вы уже встречали в этой главе. Некоторые утилиты, такие как язык программирования Perl, используют почти все эти специальные символы! (В скобках приведены американские названия специальных символов.)

ПРИМЕЧАНИЕ

Вы часто будете встречать управляющие символы, отмеченные знаком карет, например, ^C для Ctrl+C.

Таблица 2.1. Специальные символы

Символ	Название	Значение
*	Звездочка, или астериск (star, asterisk)	Регулярное выражение, символ шаблона
.	Точка (dot)	Текущий каталог, разделитель имени файла/хоста
!	Восклицательный знак (bang)	Отрицание, история команд
|	Вертикальная черта (pipe)	Конвейер (канал) для команд
/	Косая черта (slash)	Разделитель каталогов, команда поиска
\	Обратная косая черта, (backslash)	Литералы, макросы (но НЕ каталоги)
$	Доллар (dollar)	Переменные, конец строки
'	Одинарные кавычки (tick, quote)	Не интерпретируемые как шаблоны литеральные строки
`	Обратный апостроф (backtick, backquote)	Подстановка команд
"	Двойные кавычки (double quote)	Частично интерпретируемые как шаблоны (полулитеральные) строки
^	Карет (caret)	Отрицание, начало строки
~	Тильда, волнистая линия (tilda, squiggle)	Отрицание, ярлык каталога
#	Октоторп, решетка, хеш, фунт, шарп (hash, sharp, pound)	Комментарии, препроцессор, подстановка
[ ]	Квадратные скобки (square brackets)	Диапазоны
{ }	Фигурные скобки (curly brackets, braces)	Блоки выражений, диапазоны
_	Нижнее подчеркивание (underscore, under)	Заменитель пробела, когда пробелы не нужны или не разрешены либо когда запутываются алгоритмы автозаполнения

2.11. Редактирование в командной строке

Практикуясь в работе с командной оболочкой, обратите внимание на то, что вы можете редактировать команды прямо в командной строке с помощью клавиш со стрелками влево и вправо, а также просматривать предыдущие команды с помощью стрелок вверх и вниз. Это стандарт большинства систем Linux.

Однако лучше забыть о клавишах со стрелками и вместо этого использовать сочетания клавиш управления (control key). Если вы изучите перечисленные в табл. 2.2 сочетания, то обнаружите, что во многих программах Unix с их помощью вводить текст гораздо удобнее, чем с использованием клавиш со стрелками.

Таблица 2.2. Сочетания клавиш командной строки

Сочетание клавиш	Действие
Ctrl+B	Переместить курсор влево
Ctrl+F	Переместить курсор вправо
Ctrl+P	Переместить курсор выше (просмотреть предыдущую команду)
Ctrl+N	Переместить курсор ниже (просмотреть следующую команду)
Ctrl+A	Переместить курсор в начало строки
Ctrl+E	Переместить курсор в конец строки
Ctrl+W	Стереть предыдущее слово
Ctrl+U	Стереть все от курсора до начала строки
Ctrl+K	Стереть все от курсора до конца строки
Ctrl+Y	Вставить стертый текст (например, после использования Ctrl+U)

2.12. Текстовые редакторы

Поскольку мы перешли к теме редактирования, пришло время изучить текстовые редакторы. Чтобы серьезно работать в системах Unix, необходимо уметь редактировать текстовые файлы, не повреждая их. Большинство частей системы используют обычные текстовые файлы конфигурации (например, файл `/etc`). Редактировать файлы несложно, но вы будете делать это так часто, что вам точно понадобится мощный инструмент для этой работы.

Необходимо изучить один из двух стандартных текстовых редакторов Unix: vi или Emacs. Большинство пользователей Unix непреклонны в своем выборе редактора, но не слушайте их. Просто выбирайте сами. Если выберете тот, который соответствует вашему стилю работы, вам будет легче научиться. В принципе, выбор сводится к следующему.

- Если вам нужен редактор, который может делать практически все, имеет обширное онлайн-руководство и вы не возражаете против дополнительного ввода текста, чтобы использовать эти функции, попробуйте редактор Emacs.
- Если скорость — это самое важное, попробуйте редактор vi, его действия и возможности похожи на видеоигру.

Книга Арнольда Роббинса, Элберта Ханна и Линды Ламб *Learning the vi and Vim Editors: Unix Text Processing* (O'Reilly, 2008) подробно расскажет о редакторе vi. Что касается редактора Emacs, используйте онлайн-учебник: запустите Emacs, нажмите сочетание клавиш Ctrl+H, а затем введите T. Или прочитайте книгу Ричарда М. Столлмана «Руководство по GNU Emacs» (1999).

У вас может появиться соблазн поэкспериментировать с более дружелюбным редактором, например nano, Pico или одним из прочих бесчисленных графических редакторов, но если вы склонны привыкать к тому, что попробовали первым, лучше не начинать с них.

ПРИМЕЧАНИЕ

Редактирование текста — это этап, где появляется видимая разница между терминалом и графическим интерфейсом. Редакторы, такие как vi, запускаются внутри окна терминала, используя стандартный интерфейс ввода-вывода терминала. Графические редакторы запускают свое окно и представляют собственный интерфейс, независимый от терминалов. Редактор Emacs по умолчанию работает в графическом интерфейсе, но будет работать и в окне терминала.

2.13. Онлайн-поддержка

Системы Linux поставляются с большим количеством документации. Страницы руководства (или *справочные страницы, man pages*) расскажут вам все, что нужно знать об основных командах. Например, чтобы просмотреть страницу руководства для команды `ls`, выполните команду `man` следующим образом:

```
$ man ls
```

Большинство страниц руководства дают в основном справочную информацию, возможно, с некоторыми примерами и перекрестными ссылками, но не более. Не ждите, что вам предоставят целый учебник с красивым литературным стилем описания.

Когда у программ много параметров, на странице руководства они некоторым образом систематизированы (например, перечисляются в алфавитном порядке), но там не сказано, какие из них важны. Обладая определенным терпением, можно найти на справочной странице нужные параметры. Если вы нетерпеливы, попросите друга — или заплатите кому-нибудь, чтобы он стал вашим другом, — который все найдет за вас.

Для поиска страницы руководства по ключевому слову (keyword) используйте параметр `-k`:

```
$ man -k keyword
```

Эта возможность полезна, если вы не знаете точно название нужной команды. Например, если ищете команду для сортировки чего-либо, выполните:

```
$ man -k sort
--пропуск--
comm (1) - построчное сравнение двух отсортированных файлов
qsort (3) - сортировка массива
sort (1) - сортировка строк текстового файла
sortm (1) - сортировка сообщений
tsort (1) - выполняет топологическую сортировку
--пропуск--
```

Выходные данные включают название страницы руководства, раздел руководства (см. далее) и краткое описание того, что там содержится.

ПРИМЕЧАНИЕ

Используйте команду man для поиска информации о командах из предыдущих разделов.

Разделы руководства пронумерованы. Когда кто-то ссылается на страницу руководства, он указывает номер раздела в скобках рядом с именем, например `ping(8)`. В табл. 2.3 перечислены разделы и их номера.

Таблица 2.3. Разделы онлайн-руководства

Раздел	Описание
1	Пользовательские команды
2	Системные вызовы
3	Высокоуровневая документация библиотек программирования Unix
4	Интерфейсы устройств и информация о драйверах
5	Описание файлов конфигурации системы
6	Игры
7	Форматы файлов, соглашения и кодировки (ASCII, суффиксы и др.)
8	Системные команды и серверы

Разделы 1, 5, 7 и 8 — хорошее дополнение к этой книге. Раздел 4 используется нечасто, а раздел 6 был бы великолепен, если бы был немного побольше. Если вы не программист, скорее всего, раздел 3 вам не понадобится. Вы сможете понять материалы раздела 2, когда прочитаете больше о системных вызовах в этой книге.

Общие термины упоминаются на множестве страниц руководства в нескольких разделах. По умолчанию команда `man` отображает первую найденную страницу. Вы можете выбрать страницу руководства, относящуюся к определенному разделу. Например, чтобы прочитать описание файла `/etc/passwd` (а не команды

passwd), можете вставить номер раздела перед именем страницы следующим образом:

```
$ man 5 passwd
```

Страницы руководства охватывают основные вопросы, но существует еще много способов получить онлайн-помощь (помимо поиска в интернете). Если вы просто ищете определенный параметр для команды, попробуйте ввести ее имя, за которым следует `--help` или `-h` (параметр варьируется в зависимости от команды). Вы или утонете в ненужной информации (как в случае с `ls --help`), или получите именно то, что ищете.

Не так давно сообщество проекта GNU решило, что ему не очень нравятся страницы руководства, и переключилось на другой формат, называемый `info` (или `texinfo`). Эта документация шире, чем обычная страница руководства, но и сложнее. Чтобы получить доступ к справочному руководству, используйте команду `info` с указанием команды *`command`*:

```
$ info command
```

Если вам не нравится формат вывода `info`, можете отправить вывод в команду `less` (просто добавьте команду `| less`).

Некоторые пакеты сбрасывают доступную документацию в каталог `/usr/share/doc`, не обращая внимания на `man` или `info`. Просматривайте этот каталог в своей системе и, конечно же, ищите информацию в интернете.

2.14. Ввод и вывод командной оболочки

Теперь, познакомившись с основными командами, файлами и каталогами Unix, вы готовы научиться перенаправлять стандартный ввод и вывод. Начнем со стандартного вывода.

Чтобы отправить вывод команды *`command`* в файл *`file`* вместо терминала, используйте символ перенаправления `>`:

```
$ command > file
```

Оболочка создает файл *`file`*, если он еще не существует. Если же *`file`* существует, оболочка сначала *стирает (затирает)* исходный файл. Некоторые оболочки имеют параметры, которые предотвращают затирание файла. (Например, можете ввести команду `set -C`, чтобы избежать затирания в `bash`.)

Вы можете добавить выходные данные в файл, вместо того чтобы перезаписывать его с помощью синтаксиса перенаправления `>>`:

```
$ command >> file
```

Это удобный способ сбора выходных данных в одном месте при выполнении последовательностей связанных команд.

Отправить стандартный вывод команды на стандартный ввод другой команды можно с помощью символа конвейера (|). Чтобы увидеть, как это работает, попробуйте выполнить следующие две команды:

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

Вы можете отправлять выходные данные через столько команд, сколько захотите, просто добавляйте еще один конвейер перед каждой дополнительной командой.

2.14.1. Стандартная ошибка

Бывает так, что при перенаправлении стандартного вывода программа все еще продолжает что-то выводить на терминал. Это называется *стандартной ошибкой* (standard error, `stderr`) и представляет собой дополнительный поток вывода для диагностики и отладки. Например, эта команда выдает ошибку:

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

После завершения файл `f` должен быть пустым, но все равно как стандартная ошибка на терминале появится следующее сообщение об ошибке:

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

По желанию вы можете перенаправить стандартную ошибку. Например, чтобы отправить стандартный вывод в файл `f` и стандартную ошибку в файл `e`, используйте синтаксис `2>`, как здесь:

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

Число `2` указывает на идентификатор потока (stream ID), который изменяется с помощью оболочки. Идентификатор потока `1` — стандартный вывод (по умолчанию), а `2` — стандартная ошибка. Вы также можете отправить стандартную ошибку в то же место, куда отправляется `stdout`, с помощью синтаксиса `>&`. Например, чтобы отправить как стандартный вывод, так и стандартную ошибку в файл с именем `f`, попробуйте выполнить команду

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

2.14.2. Стандартное перенаправление ввода

Чтобы направить файл на стандартный ввод программы, используйте оператор `<`:

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

Некоторые программы требуют подобный тип перенаправления, но поскольку большинство команд Unix принимают имена файлов в качестве аргументов, такое перенаправление не очень распространено. Например, команда из примера могла быть записана как `head /proc/cpuinfo`.

2.15. Сообщения об ошибках

Столкнувшись с проблемой в Unix-подобной системе, такой как Linux, вы увидите сообщение об ошибке. В отличие от сообщений в других операционных системах, ошибки Unix обычно точно сообщают, что пошло не так.

2.15.1. Структура сообщений об ошибках в Unix

Большинство программ Unix генерируют сообщения об одних и тех же основных ошибках, но между выводами любых двух программ могут быть тонкие различия. Вот пример, с которым вы наверняка столкнетесь в той или иной форме:

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

В этом сообщении есть три компонента.

- Название программы — `ls`. Некоторые программы опускают эту информацию, что может раздражать, когда пишется сценарий оболочки, но на самом деле это не так уж и важно.
- Имя файла — `/dsafsda` — дает более конкретную информацию. В пути к файлу есть проблема.
- Ошибка `No such file or directory` (Нет такого файла или каталога) указывает на проблему с именем файла.

Если все это объединить, получается что-то вроде «команда `ls` пыталась открыть файл `/dsafsda`, но не смогла, потому что его не существует». Это может показаться очевидным, но эти сообщения могут сбить с толку, когда вы запускаете сценарий оболочки, содержащий команду с ошибкой в имени.

При устранении ошибок всегда сначала исправляйте первую из них. Некоторые программы сообщают, что они ничего не могут сделать, прежде чем проинформировать о множестве других проблем. Допустим, вы запускаете фиктивную программу под названием `scumd` и видите сообщение об ошибке:

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

Далее идет огромный список других сообщений об ошибках. Не отвлекайтесь на них. Скорее всего, вам просто нужно создать каталог `/etc/scumd/config`.

ПРИМЕЧАНИЕ

Не путайте сообщения об ошибках (error messages) с предупреждениями (warning messages). Последние часто выглядят как ошибки, но содержат слово «Предупреждение» (warning). Предупреждение обычно означает: что-то пошло не так, но программа все равно попытается продолжить работу. Чтобы устранить проблему, отмеченную в преду-

преждающем сообщении, вам, возможно, придется найти нужный процесс и завершить его, прежде чем исправлять другие ошибки. (О перечислении и завершении процессов вы узнаете в разделе 2.16.)

2.15.2. Распространенные ошибки

Многие ошибки, с которыми вы столкнетесь в программах Unix, появляются в результате того, что с файлами и процессами что-то пошло не так. Многие из этих ошибок вызываются непосредственно обстоятельствами, с которыми сталкиваются системные вызовы ядра, поэтому необходимо узнать, как ядро отправляет проблемы обратно в процессы.

Ошибка No such file or directory (Нет такого файла или каталога)

Это самая распространенная ошибка, возникающая при попытке получить доступ к несуществующему файлу. Поскольку файловая система ввода-вывода Unix не сильно различает файлы и каталоги, это сообщение об ошибке относится к обоим случаям. Сообщение появляется, когда вы пытаетесь прочитать несуществующий файл, перейти в несуществующий каталог, записать файл в несуществующий каталог и т. д. Эта ошибка также известна как `ENOENT` (сокращение от Error NO ENTity, ошибка «нет сущности»).

ПРИМЕЧАНИЕ

Системный вызов в таком случае обычно является результатом того, что open() возвращает ENOENT. Дополнительную информацию об ошибках, с которыми он может столкнуться, см. на странице руководства open(2).

Ошибка File exists (Файл существует)

В этом случае вы, вероятно, пытались создать файл, который уже существует. Чаще всего это происходит при попытке создать каталог с тем же именем, что и файл.

Ошибка Not a directory, Is a directory (Нет такого каталога/Это каталог)

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`. При появлении такой проблемы вам потребуется время, чтобы отыскать компонент пути, который был принят за каталог.

Ошибка No space left on device (На устройстве нет места)

Ошибка говорит о том, что на вашем устройстве закончилось свободное пространство на жестком диске.

Ошибка Permission denied (Доступ запрещен)

Ошибка возникает при попытке выполнить чтение или запись, указав файл или каталог, доступ к которым вам не разрешен (вы не обладаете достаточными правами). Эта ошибка говорит также о том, что вы пытаетесь запустить файл, для которого не установлен бит права доступа на выполнение (даже если вы можете читать этот файл). В разделе 2.17 вы больше узнаете о правах доступа.

Ошибка Operation not permitted (Операция не разрешена)

Ошибка появляется при попытке принудительно завершить процесс, который вам не принадлежит.

Ошибка Segmentation fault, Bus error (Ошибка сегментации/Ошибка шины)

Ошибка сегментации, по сути, означает, что человек, написавший программу, которую вы только что запустили, где-то допустил ошибку. Программа попыталась получить доступ к части памяти, к которой ей не разрешалось обращаться, и операционная система завершила ее. Аналогично, ошибка шины означает, что программа пыталась получить доступ к части памяти так, как не должна была. Если вы получаете одну из этих ошибок, то, вероятно, передали на ввод программы какие-либо неожиданные для нее данные. В редких случаях причиной может быть неисправное оборудование памяти.

2.16. Перечисление процессов и управление ими

Как сказано в главе 1, процесс — это запущенная программа. Каждый процесс в системе имеет *числовой идентификатор процесса (process ID, PID)*. Для быстрого перечисления запущенных процессов просто запустите команду `ps` в командной строке. Появится примерно такой список:

```
$ 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
```

Поля, которые входят в вывод:

- `PID` — идентификатор процесса;
- `TTY` — терминальное устройство, на котором выполняется процесс. Подробнее об этом позже;
- `STAT` — состояние (status) процесса, то есть информация о том, что делает процесс и где находится его память. Например, `S` означает, что процесс приостановлен (sleeping), а `R` — что он запущен (running) (описание всех символов см. на странице руководства `ps(1)`);
- `TIME` — количество процессорного времени в минутах и секундах, которое процесс успел использовать. Другими словами, общее количество времени, которое процессор потратил на выполнение инструкций процесса. Помните, что поскольку процессы не выполняются постоянно, значение отличается от времени, прошедшего с момента запуска процесса (или времени настенных часов — wall-clock time);
- `COMMAND` — очевидно, это команда, используемая для запуска программы, однако имейте в виду, что процесс может изменить это поле по сравнению с его исходным значением. Кроме того, оболочка может выполнять расширение шаблона, и это поле будет отражать расширенную команду, а не то, что вы вводите в командной строке.

ПРИМЕЧАНИЕ

Идентификаторы PID уникальны для каждого процесса, запущенного в системе. Однако после завершения процесса ядро может повторно применить тот же PID для нового процесса.

2.16.1. Параметры команды ps

Команда `ps` имеет множество параметров. Они могут различаться в стилях Unix, BSD и GNU. Многие пользователи считают стиль BSD наиболее удобным (возможно, потому что он требует меньшего набора текста), поэтому именно его мы будем применять в этой книге. Вот некоторые из наиболее полезных комбинаций параметров:

- `ps x` — отображает все запущенные вами процессы;
- `ps ax` — выводит все процессы в системе, а не только ваши;
- `ps u` — отображает более подробную информацию о процессах;
- `ps w` — выводит полные имена команд, а не только то, что помещается в одной строке.

Как и в других командах, вы можете комбинировать параметры, образуя `ps aux` и `ps auxw`.

Чтобы проверить конкретный процесс, добавьте его PID в список аргументов команды `ps`. Например, для проверки текущего процесса оболочки можно использовать `ps` и `$$` (`$$` — это переменная оболочки, которая содержит PID текущей оболочки). Информация о командах администрирования `top` и `lsof` дается в главе 8. Они могут быть полезны для определения местоположения процессов, даже если вы занимаетесь чем-то, кроме обслуживания системы.

2.16.2. Завершение процесса

Чтобы завершить процесс, вы посылаете ему сигнал (signal) — сообщение процессу из ядра — с помощью команды `kill`. В большинстве случаев вам нужно всего лишь ввести команду

```
$ kill pid
```

Существует множество типов сигналов. Значение по умолчанию (использованное ранее) — `TERM` или завершение (terminate). Вы можете посылать различные сигналы, добавив дополнительный параметр к команде `kill`. Например, чтобы заморозить процесс, а не завершить его, примените сигнал `STOP`:

```
$ kill -STOP pid
```

Остановленный процесс все еще находится в памяти и готов продолжаться с того места, где остановился. Задействуйте сигнал `CONT`, чтобы продолжить (continue) выполнение процесса:

```
$ kill -CONT pid
```

> **ПРИМЕЧАНИЕ**
>
> Использование сочетания клавиш Ctrl+C для завершения процесса, запущенного в текущем терминале, аналогично применению команды kill для завершения процесса с помощью сигнала INT (interrupt, прерывание).

Ядро дает большинству процессов возможность навести порядок после получения сигналов (с помощью механизма обработчика сигналов). Однако некоторые процессы могут выбрать действие, отличное от завершения, в ответ на сигнал, застрять в процессе его обработки или просто проигнорировать его, поэтому процесс все еще будет работать после попытки его завершения. Если это произойдет, а вам действительно нужно завершить процесс, самый жестокий способ сделать это — сигнал `KILL`. В отличие от других сигналов, `KILL` нельзя игнорировать, операционная система просто не дает процессу шанса продолжить работу. Ядро завершает процесс и принудительно удаляет его из памяти. Используйте этот метод только в крайнем случае.

Не завершайте процессы без разбора, особенно если вы не знаете, что они делают. Как говорится, не руби сук, на котором сидишь.

Ядро обозначает различные сигналы числами. Вы можете добавить к `kill` числовой сигнал, если знаете номер сигнала, который хотите отправить, например, `kill -9` вместо `kill -KILL`. Запустите `kill -l`, чтобы получить таблицу соотвествий номеров сигналов с их именами.

2.16.3. Управление заданиями

Оболочки поддерживают управление заданиями (job control) — способ отправки сигналов `TSTP` (аналогично `STOP`) и `CONT` в программы с помощью различных сочетаний клавиш и команд. Это позволяет приостанавливать работу и переключаться между используемыми программами. Например, вы можете отправить сигнал `TSTP` с помощью сочетания клавиш Ctrl+Z, а затем снова запустить процесс, введя команду `fg` (вывести на передний план, foreground) или `bg` (переместить на задний план, background, см. следующий раздел). Но несмотря на свою пользу, управление заданиями не является необходимой функцией и может сбить с толку новичков. Обычно пользователи нажимают Ctrl+Z вместо Ctrl+C, забывают о том, что они выполняли, и в итоге получают множество приостановленных процессов.

ПРИМЕЧАНИЕ

Чтобы проверить, не приостановили ли вы случайно какие-либо процессы на текущем терминале, выполните команду jobs.

Если хотите запустить несколько программ, запустите каждую из них в отдельном окне терминала, переведите неинтерактивные процессы в фоновый режим (как описано в следующем разделе) и научитесь использовать утилиты `screen` и `tmux`.

2.16.4. Фоновые процессы

Обычно, запуская команду Unix из оболочки, вы не можете вводить другие команды, пока программа не завершит выполнение. Однако можно отделить процесс от оболочки и перевести его в фоновый режим с помощью символа «амперсанда» (`&`) — что снова выведет приглашение командной строки. Например, если у вас есть большой файл, который нужно распаковать с помощью команды `gunzip` (об этом в разделе 2.18), и вы хотите сделать что-то еще во время его работы, выполните команду

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

Оболочка должна ответить, напечатав PID нового фонового процесса, и приглашение должно немедленно вернуться, чтобы вы могли продолжить работу. Если процесс занимает очень много времени, он может продолжать работать и после выхода из системы, что особенно удобно, если вам нужно запустить программу, которая выполняет много вычислительных операций. Если процесс завершается до выхода из системы или закрытия окна терминала, оболочка обычно уведомляет вас об этом (это зависит от настроек).

ПРИМЕЧАНИЕ

Если вы удаленно подключаетесь к компьютеру и хотите, чтобы программа работала после выхода из системы, вам может потребоваться команда `nohup` (подробности см. на странице руководства для нее).

Недостаток запущенных фоновых процессов заключается в том, что они могут ожидать работы со стандартным вводом или, что еще хуже, чтения непосредственно с терминала. Если программа пытается прочитать что-то из стандартного ввода, когда находится в фоновом режиме, она может зависнуть (используйте команду `fg`, чтобы вернуть фоновую задачу в оболочку) или завершиться. Кроме того, если программа записывает в стандартный вывод или стандартную ошибку, вывод может появиться в окне терминала, не обращая внимания на то, что там уже запущено. Это означает, что вы можете получить неожиданный вывод, когда работаете над чем-то другим.

Лучший способ убедиться, что фоновый процесс вас не побеспокоит, — перенаправить его вывод (и, возможно, ввод), как описано в разделе 2.14.

Если вам мешают ложные выходные данные фоновых процессов, узнайте, как обновить содержимое окна терминала. Оболочка `bash` и большинство полноэкранных интерактивных программ поддерживают сочетание клавиш Ctrl+L для обновления всего экрана. Если программа считывает данные со стандартного ввода, сочетание Ctrl+R обычно обновляет текущую строку, но нажатие неправильной последовательности в неправильное время может лишь ухудшить ситуацию. Например, ввод сочетания Ctrl+R в командной строке `bash` переводит вас в режим обратного инкрементного поиска (нажмите Esc для выхода).

2.17. Режимы файлов и права доступа

Каждый файл Unix имеет набор доступов, которые определяют, можете ли вы читать, записывать или запускать файл. Команда `ls -l` отображает права доступа. Вот пример такого отображения:

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

Режим ❶ файла представляет права доступа к файлу и некоторую дополнительную информацию. Он состоит из четырех частей, как показано на рис. 2.1.

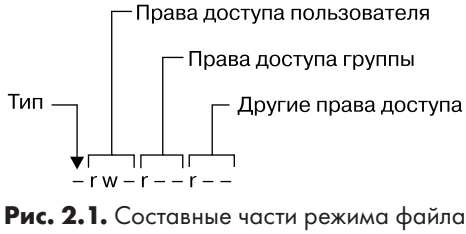


Рис. 2.1. Составные части режима файла

Первый символ режима файла — это *тип файла*. Дефис (`-`) в этой позиции, как и в примере, обозначает *обычный* файл, то есть в файле нет ничего особенного — это просто двоичные или текстовые данные. Это, безусловно, самый распространенный вид файлов. Каталоги также являются общими и обозначаются буквой `d` (directory) в слоте типа файла. (В разделе 3.1 перечислены прочие типы файлов.)

Оставшаяся часть режима файла содержит права доступа, которые разбиваются на три набора: *пользователь (user)*, *группа (groups)* и *остальные (other)*. Например, символы `rw-` в примере являются правами доступа пользователя, следующие символы `r--` — правами доступа группы, а последние символы `r--` — права доступа остальных.

Каждый набор прав доступа может содержать четыре основных варианта:

- `r` — файл доступен для чтения;
- `w` — файл доступен для записи;
- `x` — файл является исполняемым (можете запустить его как программу);
- `-` означает «ничего» (более конкретно — никаких прав на этот слот не было предоставлено).

Права доступа пользователя (первый набор) принадлежат пользователю — владельцу файла. В примере это пользователь `juser`. Второй набор — права доступа группы — предназначен для группы файла (в примере — для группы `somegroup`). Любой пользователь, относящийся к данной группе, может воспользоваться этими правами. (Примените команду `groups`, чтобы узнать, в какой группе вы находитесь, и см. подраздел 7.3.5, чтобы получить дополнительную информацию.)

Все остальные в системе имеют доступ в соответствии с третьим набором — правами доступа всех остальных, которые иногда называются правами доступа world (то, что доступно всему миру)

ПРИМЕЧАНИЕ

Каждое право доступа на чтение, запись и выполнение иногда называют битом доступа, поскольку лежащее в основе операционной системы представление представляет собой серию битов. Поэтому вы можете услышать, как люди называют части наборов доступа «битами чтения».

У некоторых исполняемых файлов в наборе прав доступа пользователя вместо буквы `x` указана буква `s`. Это говорит о том, что исполняемый файл имеет значение `setuid`, что означает: при выполнении программы она запускается так, как будто файлом владеет пользователь, а не вы. Многие программы используют бит `setuid` для запуска от имени суперпользователя, чтобы получить привилегии, необходимые для изменения системных файлов. Один из примеров — программа `passwd`, которая должна изменять файл `/etc/passwd`.

2.17.1. Изменение прав доступа

Чтобы изменить права доступа для файла или каталога, используйте команду `chmod` (изменение режима, change mode). Сначала выберите набор прав доступа, которые вы хотите изменить, а затем — бит для изменения. Например, чтобы добавить в файл *`file`* права доступа для группы (`g`, от *group*) и остальных (`o`, от *other*) на чтение (`r`, от *read*), можете выполнить следующие две команды:

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

Или только одну команду:

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

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

> **ПРИМЕЧАНИЕ**
>
> Очевидно, что не нужно делать файлы доступными для записи всем, потому что это позволяет любому пользователю вашей системы изменять их. Но позволит ли это любому, кто подключен к интернету, изменить их? Вероятно, нет, если только у вашей системы нет дыры в сетевой безопасности. В этом случае права доступа к файлам вам все равно не помогут.

Пользователи могут менять права доступа и с помощью цифр, например:

```
$ chmod 644 file
```

Это называется *абсолютным* изменением, потому что оно устанавливает *все* биты прав доступа одновременно. Чтобы понять, как это работает, вам нужно уметь представлять биты прав доступа в восьмеричной системе счисления (каждая цифра представляет число с основанием 8, от 0 до 7, и соответствует набору прав доступа). Дополнительные сведения см. на странице руководства `chmod(1)` или в справочном руководстве.

На самом деле вам не нужно знать, как конструировать абсолютные режимы, если вы хотите с ними работать, просто запомните режимы, которые будете применять чаще всего. В табл. 2.4 перечислены наиболее распространенные из них.

Каталоги также имеют права доступа. Вы можете перечислить содержимое каталога, если оно доступно для чтения, но получить доступ к файлу в каталоге можете, только если каталог исполняемый. В большинстве случаев вам нужно и то и другое. Распространенная ошибка, которую пользователи совершают при настройке разрешений каталогов, — это случайное удаление разрешения на выполнение при использовании абсолютных режимов.

Наконец, вы можете указать набор разрешений по умолчанию с помощью команды оболочки `umask`, которая применяет предопределенный набор прав к любому вновь

создаваемому файлу. В частности, используйте `umask 022`, если хотите, чтобы все могли видеть все файлы и каталоги, которые вы создаете, и `umask 077`, чтобы не открывать доступ для всех. Если вы хотите, чтобы набор желаемых разрешений применялся к новым окнам и дальнейшим сеансам, поместите команду `umask` с нужным режимом в один из файлов запуска, как описано в главе 13.

Таблица 2.4. Абсолютные режимы назначения прав доступа

Режим	Значение	Использование
644	Пользователь — чтение и запись; группа — только чтение; остальные — только чтение	В файлах
600	Пользователь — чтение и запись; группа — нет прав; остальные — нет прав	В файлах
755	Пользователь — чтение, запись, выполнение; группа — чтение и выполнение; остальные — чтение и выполнение	В каталогах, программах
700	Пользователь — чтение, запись, выполнение; группа — нет прав; остальные — нет прав	В каталогах, программах
711	Пользователь — чтение, запись, выполнение; группа — только выполнение; остальные — только выполнение	В каталогах

2.17.2. Использование символических ссылок

Символическая ссылка («мягкая» ссылка, symbolic link, symlink) — это файл, который указывает на другой файл или каталог, создавая его псевдоним (alias) (например, как ярлык в Windows). Символические ссылки обеспечивают быстрый доступ, скрывая пути каталогов.

В длинном списке каталогов символические ссылки выглядят следующим образом (обратите внимание на цифру `l` как тип файла в режиме 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 вместо этого выдаст вам `/home/origdir`. Символические ссылки — это просто имена файлов, которые указывают на другие имена. Их имена и пути, на которые они указывают, не должны ничего значить. В предыдущем примере каталог `/home/origdir` может даже не существовать.

На самом деле, если `/home/origdir` не существует, любая программа, которая обращается к `somedir`, возвращает сообщение об ошибке, информирующее, что `somedir` не существует (за исключением `ls somedir` — команды, которая только и сообщает вам, что `somedir` — это `somedir`). Это может сбить с толку, потому что само название `somedir` у вас перед глазами.

Символические ссылки могут вас запутать не только этим. Другая проблема заключается в том, что вы не можете определить характеристики объекта, на который указывает ссылка, просто взглянув на ее имя, — вы должны перейти по ссылке, чтобы увидеть, указывает ли она на файл или каталог. В вашей системе также могут быть ссылки, указывающие на другие ссылки, которые называются *связанными символическими ссылками* и могут быть помехой при попытке отслеживания.

Чтобы создать символическую ссылку от *целевого объекта* `target` к *имени ссылки* `linkname`, используйте команду `ln -s` следующим образом:

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

Аргумент `linkname` — это имя символической ссылки, аргумент `target` — путь к файлу или каталогу, на который указывает ссылка, а флаг `-s` указывает на создание символической ссылки (см. врезку далее).

При создании символической ссылки дважды проверьте команду перед ее запуском, потому что могут возникнуть ошибки. Например, если вы случайно изменили порядок аргументов (цель `ln -s linkname target`), то возникнет ошибка, если `linkname` — уже существующий каталог. Если это так (и это довольно часто так), команда `ln` создает ссылку с именем `target` внутри каталога `linkname`, и ссылка будет указывать на себя, если `linkname` не содержит абсолютный путь. Если что-то пойдет не так при создании символической ссылки на каталог, проверьте этот каталог на наличие ошибочных символических ссылок и удалите их.

Символические ссылки могут быть той еще головной болью, если вы не знаете, что они существуют. Например, вы можете легко отредактировать то, что посчитали копией файла, но это оказалась символическая ссылка на оригинал.

ВНИМАНИЕ

Не забывайте добавлять параметр `-s` при создании символической ссылки. Без этого команда `ln` создает жесткую ссылку, давая дополнительное реальное имя файла данному файлу. Новое имя файла имеет такой же статус, как и старое, оно указывает (ссылается) непосредственно на данные файла, а не на другое имя файла, как делает символическая ссылка. Жесткие ссылки могут быть еще более запутанными, чем символические ссылки. Пока вы не поймете материал раздела 4.6, избегайте их использования.

Учитывая все предупреждения относительно символических ссылок, вы можете задаться вопросом: зачем вообще кому-то их использовать? Как оказалось, их подводные камни значительно компенсируются преимуществами, которые они

обеспечивают для организации файлов, и способностью легко исправлять небольшие проблемы. Один из распространенных случаев применения — поиск программой определенного файла или каталога, который уже существует где-то еще в вашей системе. Вы не хотите делать копию, и если не можете изменить программу, то просто создайте символическую ссылку из нее на фактическое расположение файла или каталога.

2.18. Архивирование и сжатие файлов

Теперь, когда вы узнали о файлах, правах доступа и возможных ошибках, вам необходимо освоить `gzip` и `tar` — две общие утилиты для сжатия и архивирования файлов и каталогов.

2.18.1. Утилита gzip

Программа `gzip` (GNU Zip) — это одна из современных стандартных программ сжатия Unix. Файл, заканчивающийся на `.gz`, является Zip-архивом GNU. Используйте команду `gunzip` *`file`*`.gz`, чтобы распаковать файл <*`file`*>`.gz` и удалить суффикс, а чтобы снова сжать файл, примените команду `gzip` *`file`*.

2.18.2. Утилита tar

В отличие от ZIP-программ для других операционных систем, `gzip` не создает архивы файлов, то есть не упаковывает несколько файлов и каталогов в один файл. Чтобы создать архив из нескольких файлов и каталогов, используйте команду `tar`:

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

Архивы, созданные командой `tar`, обычно имеют суффикс `.tar` (общепринято, необязательное условие). Например, в предыдущей команде *`file1`*, *`file2`* и т. д. — это имена файлов и каталогов, которые нужно заархивировать в <*`archive`*>`.tar`. Флаг `c` активирует *режим создания архива (от create)*. Флаги `v` и `f` имеют более конкретные роли.

Флаг `v` (от verbose — «подробный») активирует подробный диагностический вывод, заставляя `tar` печатать имена файлов и каталогов, находящихся в архиве, при их обнаружении. Добавление второго флага `v` приводит к тому, что `tar` печатает сведения о размере файла и о правах доступа к нему. Если вы не хотите, чтобы команда `tar` выводила эти данные, не добавляйте флаг `v`.

Флаг `f` (от file — «файл») обозначает файл-параметр. Следующим аргументом в командной строке после флага `f` должно быть имя создаваемого архива `tar` (в предыдущем примере это <*`archive`*>`.tar`). Вы *должны* использовать этот параметр, за которым следует имя файла, везде, за исключением ленточных накопителей. Для использования стандартного ввода или вывода установите для имени файла дефис (`-`).

Распаковка файлов .tar

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

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

В этой команде флаг `x` переводит `tar` в *режим извлечения (распаковки)*. Вы можете извлечь отдельные части архива, введя имена частей в конце командной строки, но должны знать их точные имена. (Чтобы убедиться в этом, см. раздел «Режим содержания» далее.)

> **ПРИМЕЧАНИЕ**
>
> При использовании режима извлечения помните, что команда tar не удаляет архивированный файл .tar после извлечения его содержимого.

Режим содержания

Перед распаковкой обычно рекомендуется проверить содержимое файла `.tar` *в режиме содержания*, задействуя флаг `t` вместо флага `x`. Этот режим проверяет базовую целостность архива и выводит имена всех файлов внутри него. Если не протестируете архив перед его распаковкой, то можете в итоге добавить множество беспорядочных файлов в текущий каталог, который может оказаться очень трудно очистить.

При проверке архива в режиме `t` убедитесь, что создана рациональная структура каталогов, то есть все пути к файлам в архиве начинаются с одного и того же каталога. Если вы не уверены в этом, создайте временный каталог, перейдите в него, а затем извлеките содержимое архива. (Всегда можно использовать команду `mv * ..`, если архив не создал беспорядка.)

При распаковке вы можете указать параметр `-p` (от permissions) для сохранения прав доступа. Используйте его в режиме распаковки, чтобы переопределить команду `umask` и добавить точные права доступа, указанные в архиве. Параметр `-p` применяется по умолчанию во время работы от имени суперпользователя. Если у вас возникли проблемы с правами доступа и владельцами при распаковке архива от имени суперпользователя, дождитесь, когда команда завершится и приглашение оболочки появится снова. Хотя вы можете извлечь лишь небольшую часть архива, команда `tar` должна пройти через весь набор файлов и вы не должны прерывать процесс, потому что он устанавливает разрешения только после проверки всего архива.

Запомните *все* параметры и режимы `tar` из этого раздела, а лучше запишите в отдельном месте. Это позволит избежать ошибок с этой командой по неосторожности.

2.18.3. Сжатые архивы (.tar.gz)

Многих новичков смущает то, что архивы обычно находятся в сжатом виде, а имена файлов заканчиваются на `.tar.gz`. Чтобы распаковать сжатый архив, начинайте

с конца: сначала избавьтесь от `.gz`, а затем от `.tar`. Например, две следующие команды выполняют декомпрессию и распаковку файла `<file>.tar.gz`:

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

Поначалу нормально делать по одному шагу за раз: сначала запустить команду `gunzip` для декомпрессии, а затем команду `tar` для проверки и распаковки. Чтобы создать сжатый архив, сделайте обратное: сначала запустите `tar`, а затем `gzip`. При частом выполнении процесса архивации и сжатия вы быстро запомните, как он протекает. Но даже если это происходит не так уж часто, вы поймете, насколько утомительным может стать набор текста, и вам понадобится кратчайший путь. Посмотрим, какие есть варианты.

2.18.4. Утилита zcat

Метод, описанный ранее, — не самый быстрый или эффективный способ вызова команды `tar` для работы со сжатым архивом, и он тратит дисковое пространство и время ввода-вывода ядра. Лучший способ — объединить функции архивирования и сжатия в конвейер. Например, следующий конвейер распаковывает файл `<file>.tar.gz`:

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

Команда `zcat` действует так же, как и `gunzip -dc`. Параметр `-d` распаковывает (decompress), а параметр `-c` отправляет результат в стандартный вывод (в данном случае в команду `tar`).

Поскольку чаще используется команда `zcat`, версия команды `tar`, поставляемая с Linux, имеет свое общепринятое сокращение. Вы можете применять `z` в качестве параметра для автоматического вызова `gzip`, это работает как для извлечения архива (с режимами `x` или `t` в `tar`), так и для его создания (с помощью параметра `c`). Например, проверяйте сжатый архив с помощью команды

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

И все же постарайтесь запомнить: используя это сокращение, на самом деле вы выполняете два шага.

ПРИМЕЧАНИЕ

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

2.18.5. Другие утилиты сжатия

Еще две программы сжатия — это `xz` и `bzip2`, сжатые файлы которых заканчиваются на `.xz` и `.z2` соответственно. Хотя они немного медленнее, чем `gzip`, однако сильнее сжимают текстовые файлы. Для распаковки используются программы `unxz` и `bunzip2`, и параметры обеих довольно близки к их аналогам `gzip`.

Большинство дистрибутивов Linux поставляются с программами `zip` и `unzip`, совместимыми с ZIP-архивами в системах Windows. Они работают с обычными zip-файлами, а также с самораспаковывающимися архивами, заканчивающимися на `.exe`. Но если вы столкнулись с файлом, который заканчивается на `.z`, знайте: вы нашли древнюю реликвию, созданную программой сжатия `compress`, которая когда-то была стандартом для систем Unix. Программа `gunzip` может распаковать эти файлы, но создать не способна.

2.19. Основная иерархия каталогов Linux

Теперь, когда вы знаете, как просматривать файлы, сменять каталоги и читать страницы руководства, можете приступить к изучению системных файлов и каталогов. Подробные сведения о структуре каталогов Linux изложены в Стандарте иерархии файловой системы или в FHS (refspecs.linuxfoundation.org/fhs.shtml), но пока достаточно краткого пошагового руководства.

На рис. 2.2 представлен упрощенный вид иерархии, отображающий каталоги в `/`, `/usr` и `/var`. Обратите внимание, что в разделе `/usr` содержатся некоторые из тех же имен каталогов, что и в `/`.

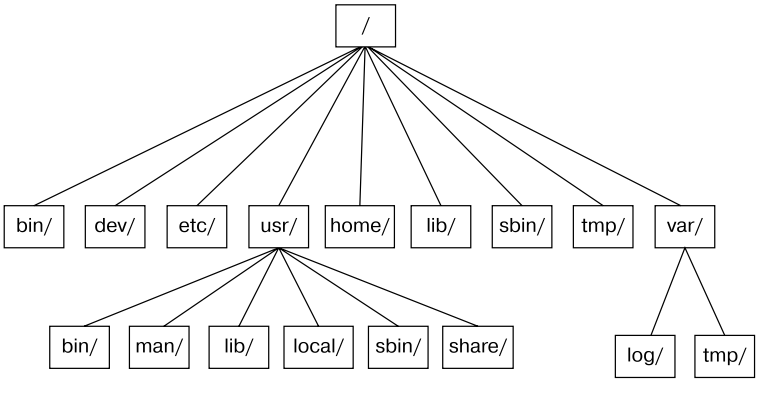


Рис. 2.2. Иерархия каталогов в системе Linux

Вот наиболее важные подкаталоги из имеющихся в корневом каталоге.

- `/bin`. Содержит готовые к запуску программы (также известные как *исполняемые файлы*), включая большинство основных команд Unix, таких как `ls` и `cp`. Большинство программ в `/bin` созданы в двоичном формате на языке C, но некоторые из них являются сценариями оболочки в современных системах.
- `/dev`. Содержит файлы устройств. Подробнее об этом вы узнаете в главе 3.
- `/etc`. Этот центральный каталог конфигурации системы (произносится этси́) содержит пароль пользователя, загрузочные файлы, файлы устройств, сетевые настройки и др.

- `/home`. Содержит домашние (личные) каталоги для обычных пользователей. Большинство установок Unix соответствуют этому стандарту.
- `/lib`. Сокращение от *library* (библиотека). В этом каталоге находятся файлы библиотек с кодом, который могут использовать исполняемые файлы. Существует два типа библиотек: статические и разделяемые. Каталог `/lib` должен содержать только разделяемые библиотеки, но другие каталоги `lib`, такие как `/usr/lib`, включают обе разновидности, а также другие вспомогательные файлы. (Мы обсудим разделяемые библиотеки более подробно в главе 15.)
- `/proc`. Предоставляет системную статистику через доступный для просмотра интерфейс каталогов и файлов. Бóльшая часть структуры подкаталогов `/proc` в Linux уникальна, но многие другие варианты Unix имеют аналогичные функции. Каталог `/proc` содержит информацию о запущенных в данный момент процессах, а также некоторые параметры ядра.
- `/run`. Содержит данные времени выполнения, относящиеся к системе, включая определенные идентификаторы процессов, файлы сокетов, записи состояния и во многих случаях системный журнал. Это относительно недавнее дополнение к корневому каталогу, в старых системах вы можете найти его в `/var/run`. В более новых системах `/var/run` — это символическая ссылка на `/run`.
- `/sys`. Этот каталог похож на каталог `/proc` тем, что он предоставляет интерфейс устройствам и системе. Подробнее о `/sys` вы узнаете в главе 3.
- `/sbin`. Место для системных исполняемых файлов. Программы в каталогах `/sbin` связаны с управлением системой, поэтому простые пользователи обычно не имеют компонентов `/sbin` в своих путях команд. Многие из утилит в этом каталоге работают, только если запущены от имени суперпользователя.
- `/tmp`. Место для хранения небольших, временных, не особо важных файлов. Любой пользователь может читать из каталога `/tmp` и записывать в него, но у пользователя может не быть доступа к файлам другого пользователя. Многие программы задействуют этот каталог в качестве рабочей области. Если какой-то файл важен, не помещайте его в каталог `/tmp`, потому что большинство дистрибутивов очищают его при загрузке, а некоторые даже периодически удаляют старые файлы. Кроме того, не позволяйте `/tmp` заполняться мусором, потому что обычно он делит пространство с важными каталогами (например, с остальной частью каталога `/`).
- `/usr`. Сокращение от *user* (пользователь), однако в этом подкаталоге нет пользовательских файлов. Вместо этого он содержит большую иерархию каталогов, включая основную часть системы Linux. Многие имена каталогов в `/usr` совпадают с именами в корневом каталоге (например, `/usr/bin` и `/usr/lib`), и они содержат файлы одного типа. (Причина, по которой в корневом каталоге не содержится вся система, в первую очередь историческая — в прошлом это было сделано для того, чтобы снизить требования к пространству для корневого каталога.)

- `/var`. Подкаталог переменных, куда программы записывают информацию, которая может изменяться с течением времени. Здесь находятся системные журналы, отслеживание активности пользователей, кэши и другие файлы, создаваемые системными программами и управляемые ими. (Здесь также есть каталог `/var/tmp`, но система не стирает его при загрузке.)

2.19.1. Другие корневые подкаталоги

В корневом каталоге есть еще несколько интересных подкаталогов.

- `/boot`. Содержит файлы загрузчика ядра. Эти файлы относятся к самому первому этапу запуска Linux, поэтому в этом каталоге вы не найдете информации о том, как Linux запускает свои службы. Подробнее об этом — в главе 5.
- `/media`. Базовый каталог для съемных носителей, таких как флеш-накопители. Этот каталог встречается во многих дистрибутивах.
- `/opt`. Может содержать дополнительное программное обеспечение сторонних производителей. Многие системы не используют каталог `/opt`.

2.19.2. Каталог /usr

Каталог `/usr` на первый взгляд может показаться относительно чистым, но если взглянуть на `/usr/bin` и `/usr/lib`, можно найти множество данных: `/usr` — это место, где находится бóльшая часть пользовательских программ и данных. В дополнение к `/usr/bin`, `/usr/sbin` и `/usr/lib` каталог `/usr` содержит следующее.

- `/include`. Файлы заголовков, используемые компилятором языка C.
- `/local`. Место, где администраторы могут устанавливать собственное программное обеспечение. Его структура должна выглядеть так же, как у каталогов `/` и `/usr`.
- `/man`. Страницы руководства.
- `/share`. Файлы, которые должны работать на других типах систем Unix без потери функциональности. Обычно это вспомогательные файлы данных, которые программы и библиотеки читают по мере необходимости. В прошлом сети машин совместно пользовались бы этим каталогом с файлового сервера, но сегодня общий каталог, применяемый таким образом, встречается редко, поскольку в современных системах нет конкретных ограничений на пространство для таких файлов. Вместо этого в дистрибутивах Linux вы найдете каталоги `/man`, `/info` и многие другие.

2.19.3. Местонахождение ядра

В системах Linux ядро обычно представляет собой двоичный файл `/vmlinuz` или `/boot/vmlinuz`. Загрузчик (boot loader) загружает этот файл в память и приводит его в действие при загрузке системы. (Подробнее о загрузчике — в главе 5.)

Как только загрузчик запускает ядро, основной файл ядра больше не используется запущенной системой. Однако вы найдете множество модулей, которые ядро загружает и выгружает по требованию в ходе нормальной работы системы. Они называются *загружаемыми модулями ядра* и расположены в папке `/lib/modules`.

2.20. Запуск команд от имени суперпользователя

Прежде чем идти дальше, вы должны научиться выполнять команды от имени суперпользователя root. У вас может возникнуть соблазн запустить корневую оболочку, но у этого действия множество недостатков.

- Нет записей о командах, изменяющих систему.
- Нет записей о пользователях, которые выполняли команды, изменяющие систему.
- Нет доступа к вашей обычной среде оболочки.
- Вы должны ввести пароль суперпользователя (если он у вас есть).

2.20.1. Команда sudo

Большинство дистрибутивов задействуют пакет под названием `sudo`, чтобы позволить администраторам запускать команды от имени суперпользователя, если они вошли в систему от своего имени. Например, из главы 7 вы узнаете о применении команды `vipw` для редактирования файла `/etc/passwd`. Вот как это можно сделать:

```
$ sudo vipw
```

При выполнении данной команды `sudo` регистрирует это действие с помощью службы syslog в системном журнале для устройства `local2`. Больше о системных журналах вы узнаете в главе 7.

2.20.2. Файл /etc/sudoers

Конечно, система не позволяет любому пользователю выполнять команды от имени суперпользователя — вы должны настроить привилегии пользователей в файле `/etc/sudoers`. Пакет `sudo` имеет множество параметров (которые вы, вероятно, никогда не будете использовать), что несколько усложняет синтаксис в файле `/etc/sudoers`. Например, этот файл дает пользователям `user1` и `user2` возможность запускать любую команду от имени суперпользователя, не вводя пароль:

```
User_Alias ADMINS = user1, user2

ADMINS ALL = NOPASSWD: ALL

root ALL=(ALL) ALL
```

Первая строка определяет псевдоним пользователя `ADMINS` для двух пользователей, а вторая строка предоставляет привилегии. Часть `ALL = NOPASSWD` означает, что пользователи с псевдонимом `ADMINS` могут применять `sudo` для выполнения команд от имени суперпользователя. Второе `ALL` означает «любая команда». Первое `ALL` означает «любой хост». (Если у вас более одной машины, можете установить различные виды доступа для каждой машины или группы машин, но мы не будем рассматривать эту функцию.)

Часть `root ALL=(ALL) ALL` просто означает, что суперпользователь может использовать `sudo` для выполнения любой команды на любом хосте. Дополнительный параметр `(ALL)` говорит о том, что суперпользователь может выполнять команды, как и любой другой пользователь. Вы можете расширить эту привилегию для пользователей `ADMINS`, добавив параметр `(ALL)` во вторую строку `/etc/sudoers`:

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

ПРИМЕЧАНИЕ

Применяйте команду visudo для редактирования файла /etc/sudoers. Эта команда проверяет файл на синтаксические ошибки после сохранения.

2.20.3. Журналы sudo

Более подробно мы рассмотрим журналы позже, однако вы уже сейчас можете найти журналы `sudo` в большинстве систем с помощью команды

```
$ journalctl SYSLOG_IDENTIFIER=sudo
```

В старых системах вам нужно будет искать файл журнала в `/var/log`, например, `/var/log/auth.log`.

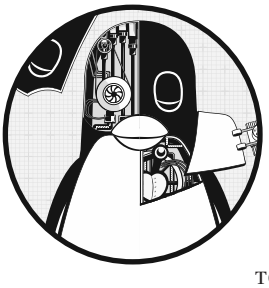
На этом закончим с командой `sudo`. Если вам нужно задействовать ее расширенные функции, см. страницы руководства `sudoers(5)` и `sudo(8)`. (Фактическая механика переключения пользователей рассматривается в главе 7.)

2.21. Что дальше?

Теперь вы знаете, как в командной строке запускать программы, перенаправлять вывод, взаимодействовать с файлами и каталогами, просматривать списки процессов, страницы руководства и перемещаться по пользовательской среде системы Linux. А еще можете выполнять команды от имени суперпользователя. Возможно, вы еще не очень много знаете о внутренних компонентах пользовательского интерфейса или о том, что происходит в ядре, но основы файлов и процессов вы уже изучили. В нескольких следующих главах вы будете работать как с ядром, так и с компонентами системы пользовательского пространства, применяя только что изученные инструменты командной строки.

3

Устройства



В этой главе описана базовая инфраструктура устройств, обеспечиваемая ядром, в функционирующей системе Linux.

За время существования Linux значительно изменилось то, как именно ядро представляет устройства пользователю. Мы начнем с рассмотрения традиционной системы файлов устройств, чтобы увидеть, как ядро предоставляет информацию о конфигурации устройства по пути `sysfs`. Наша цель состоит в том, чтобы иметь возможность извлекать информацию об устройствах, имеющихся в системе, и понять, как протекают элементарные операции. В последующих главах будет подробнее рассмотрено взаимодействие с конкретными типами устройств.

Важно понимать, как ядро взаимодействует с пользовательским пространством при появлении новых устройств. Система `udev` позволяет программам пользовательского пространства автоматически настраивать и задействовать новые устройства. Вы увидите основные принципы того, как ядро отправляет сообщение процессу пользовательского пространства через `udev`, а также что именно процесс делает с ним.

3.1. Файлы устройств

Управлять большинством устройств в системе Unix легко, поскольку ядро представляет множество интерфейсов ввода-вывода устройств пользовательским процессам в виде файлов. Эти файлы устройств иногда называют *узлами устройств*. Некоторые устройства доступны не только программистам, применяющим обычные файловые операции для работы с ними, но и стандартным программам, таким как

`cat`, поэтому вам не нужно быть программистом, чтобы использовать устройства. Однако существует ограничение на то, что можно сделать с файловым интерфейсом, поэтому не все устройства или возможности устройств доступны в стандартном файловом вводе-выводе.

Система Linux использует тот же дизайн для файлов устройств, что и другие системы Unix. Файлы устройств находятся в каталоге `/dev`, и запуск команды `ls /dev` отображает довольно много файлов в `/dev`. Итак, как же работать с устройствами?

Для начала рассмотрим команду:

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

Как и любая другая команда с перенаправленным выводом, она отправляет данные из стандартного вывода в файл. Однако файл является устройством `/dev/null`, поэтому ядро отказывается от своих обычных файловых операций и использует драйвер устройства для данных, записанных на это устройство. В случае с `/dev/null` ядро просто принимает входные данные и отбрасывает их.

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

```
$ 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 каждой строки (первый символ режима файла). Если это символы `b`, `c`, `p` или `s`, то файл является устройством. Эти буквы обозначают *block* (блочное устройство), *character* (символьное устройство), *pipe* (конвейер) и *socket* (сокет) соответственно.

- **Блочное устройство.** Программы получают доступ к данным с блочного устройства фиксированными частями. Устройство `sda1` в приведенном ранее примере — это *дисковое устройство*, тип блочного устройства. Диски можно легко разбить на блоки данных. Поскольку общий размер блочного устройства фиксирован и его легко индексировать, процессы имеют быстрый произвольный доступ к любому блоку в устройстве с помощью ядра.
- **Символьное устройство.** Символьные устройства работают с потоками данных. Вы можете считывать символы с символьных устройств или записывать их на символьные устройства, как было показано ранее на устройстве `/dev/null`. Символьные устройства не имеют размера: когда вы читаете из одного из них или записываете в него, ядро обычно выполняет на нем операцию чтения или записи. Принтеры, непосредственно подключенные к компьютеру, представляются символьными устройствами. Важно отметить, что во время взаимодействия с символьным устройством ядро не может создавать

резервные копии и повторно выполнять проверку после передачи данных устройству или процессу.

- **Конвейер.** *Именованный конвейер* по структуре такой же, как символьные устройства, но с другим процессом на другом конце потока ввода-вывода вместо драйвера ядра.
- **Сокет.** Сокеты — это специальные интерфейсы, которые используются для межпроцессной связи. Они часто находятся за пределами каталога `/dev`. Файлы сокетов представляют собой доменные сокеты Unix, подробнее о них вы узнаете в главе 10.

В списках файлов вывода команды `ls -l` с блочных и символьных устройств номера перед датами являются *основными (major)* и *второстепенными (minor)* номерами устройств, которые ядро использует для их идентификации. Подобные устройства обычно имеют одинаковое основное число, например `sda3` и `sdb1` (оба являются разделами жесткого диска).

ПРИМЕЧАНИЕ

Не у всех устройств есть файлы, поскольку интерфейсы ввода-вывода блочных и символьных устройств подходят не во всех случаях. Например, сетевые интерфейсы не содержат файлов устройств. Теоретически возможно взаимодействовать с сетевым интерфейсом с помощью одного символьного устройства, но поскольку это проблематично, ядро предлагает другие интерфейсы ввода-вывода.

3.2. Путь к устройству sysfs

Традиционный каталог Unix `/dev` удобен для того, чтобы пользовательские процессы ссылались на устройства, поддерживаемые ядром, и взаимодействовали с ними, но это очень упрощенная схема. Имя устройства в `/dev` немногое говорит об устройстве. Другая проблема заключается в том, что ядро назначает устройства в том порядке, в котором они найдены, поэтому между перезагрузками устройство может получить другое имя.

Чтобы обеспечить единообразное представление подключенных устройств на основе их фактических аппаратных атрибутов, ядро Linux предлагает интерфейс `sysfs` для обозначения файлов и каталогов.

Базовый путь для устройств — `/sys/devices`. Например, жесткий диск SATA в `/dev/sda` может иметь следующий путь в интерфейсе `sysfs`:

```
/sys/devices/pci0000:00/0000:00:17.0/ata3/host0/target0:0:0/0:0:0:0/block/sda
```

Как видите, путь довольно длинный по сравнению с именем файла `/dev/sda`, которое является также именем каталога. Но на самом деле нельзя сравнивать эти два пути, потому что у них разные цели. Файл `/dev` позволяет пользовательским процессам применять устройство, в то время как путь `/sys/devices` задействуется

для просмотра информации и управления устройством. Если вы перечислите содержимое пути устройства, как в примере, то получите подобный вывод:

```
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
```

Файлы и подкаталоги здесь предназначены для чтения в основном программами, а не пользователями, но вы можете получить представление о том, что они содержат и представляют, посмотрев на пример файла `/dev`. Запуск команды `cat dev` в этом каталоге отображает числа `8:0`, которые являются основными и второстепенными номерами устройств `/dev/sda`.

В каталоге `/sys` есть несколько ярлыков. Например, `/sys/block` должен содержать все блочные устройства, доступные в системе. Однако это всего лишь символические ссылки: лучше запустить команду `ls -l /sys/block`, чтобы выявить истинные `sysfs` пути.

Может быть трудно найти местоположение `sysfs` устройства в файле `/dev`. Используйте команду `udevadm`, как в следующем примере, чтобы отобразить путь и другие интересные атрибуты:

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

Более подробную информацию о команде `udevadm` и всей системе `udev` вы найдете в разделе 3.5.

3.3. Утилита dd и устройства

Программа `dd` чрезвычайно полезна в работе с блочными и символьными устройствами. Ее единственная функция заключается в чтении из входного файла или потока и записи в выходной файл или поток с выполнением некоторых преобразований кодирования по мере необходимости. Одна особенно полезная функция команды `dd` в отношении блочных устройств заключается в том, что вы можете обрабатывать фрагмент данных в середине файла, игнорируя то, что происходит до или после.

> **ВНИМАНИЕ**
>
> Команда dd — очень мощный инструмент, поэтому при ее запуске убедитесь, что уверены в своих действиях. Очень легко повредить файлы и данные на устройствах, по неосторожности допустив ошибку. Часто бывает полезно записать выходные данные в новый файл.

Команда `dd` копирует данные в блоках фиксированного размера. Вот пример того, как можно использовать команду `dd` с символьным устройством, задействуя несколько распространенных параметров:

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

Как видите, формат параметров dd отличается от форматов параметров большинства других команд Unix, он основан на старом стиле языка управления заданиями IBM (JCL, Job Control Language). Вместо того чтобы использовать символ дефиса (-) для обозначения параметра, вводится имя параметра и устанавливается его значение со знаком равенства (=). В предыдущем примере один блок размером 1024 байта копируется из /dev/zero (непрерывный поток нулевых байтов) в файл *new_file*.

Важные параметры команды dd.

- if=*file*. Файл ввода. По умолчанию применяется стандартный ввод.
- of=*file*. Файл вывода. По умолчанию используется стандартный вывод.
- bs=size. Размер блока. Команда dd считывает и записывает такое количество байтов данных за один раз. Чтобы сокращенно указать большие куски данных, можете с помощью b и k обозначать 512 и 1024 байта соответственно. Поэтому в предыдущем примере можно указать bs=1k вместо bs=1024.
- ibs=*size*, obs=*size*. Размеры блока ввода и вывода. Если вы можете использовать один и тот же размер блока как для ввода, так и для вывода, применяйте параметр bs, если нет — параметры ibs и obs для ввода и вывода соответственно.
- count=*num*. Общее количество блоков для копирования. Во время работы с огромным файлом (или с устройством, которое предоставляет бесконечный поток данных, например /dev/zero) необходимо, чтобы команда dd останавливалась в фиксированной точке, в противном случае вы можете потратить много дискового пространства, процессорного времени или и того и другого. Используйте команду count с параметром skip для копирования небольшого фрагмента из большого файла или устройства.
- skip=*num*. Параметр позволяет пропустить первые несколько (num) блоков в файле ввода или потоке и не копировать их в выходные данные.

3.4. Имена устройств

Иногда бывает трудно найти имя устройства (например, при разбиении диска на разделы). Вот несколько способов узнать его.

- Выполните udevd с помощью команды udevadm (см. раздел 3.5).
- Найдите устройство в каталоге /sys.
- Угадайте имя из выходных данных команды journalctl -k (которая выводит сообщения ядра) или системного журнала ядра (см. раздел 7.1). Этот вывод может содержать описание устройств в вашей системе.
- Для дискового устройства, которое уже видно системе, можно проверить вывод команды mount.
- Запустите команду cat /proc/devices, чтобы просмотреть блочные и символьные устройства, для которых в вашей системе в настоящее время имеются

драйверы. Каждая строка состоит из номера и имени. Этот номер является основным номером устройства, как сказано в разделе 3.1. Если вы можете угадать устройство по имени, найдите в каталоге `/dev` символьные или блочные устройства с соответствующим основным номером — и найдете файлы устройств.

Среди этих методов надежен только первый, но он требует утилиты `udev`. Если вы попадете в ситуацию, когда `udev` недоступна, попробуйте другие методы, но имейте в виду, что в ядре может не быть файла устройства для вашего оборудования.

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

3.4.1. Жесткие диски: /dev/sd*

Большинство жестких дисков, подключенных к текущим системам Linux, соответствуют именам устройств с префиксом `sd`, например, `/dev/sda`, `/dev/sdb` и т. д. Эти устройства представляют собой целые диски, ядро создает отдельные файлы устройств `/dev/sda1` и `/dev/sda2` для разделов на диске.

Давайте посмотрим, откуда взялось такое название. `sd` — это часть имени, которое расшифровывается как *SCSI-диск*. Интерфейс *Small Computer System Interface* (*SCSI, интерфейс малых вычислительных систем*) первоначально был разработан как стандарт аппаратного обеспечения и протокола для связи между устройствами, например дисками и другими периферийными устройствами. Хотя традиционное аппаратное обеспечение SCSI в большинстве современных систем не используется, протокол SCSI широко распространен благодаря своей адаптивности. К примеру, его применяют USB-накопители. История с дисками SATA (Serial ATA — общее пространство хранения на Windows) немного сложнее, однако ядро Linux все еще использует команды SCSI в определенный момент при обращении к подобным дискам.

Чтобы перечислить устройства SCSI, имеющиеся в вашей системе, задействуйте утилиту, которая просматривает sysfs-пути к устройствам. Одним из самых лаконичных инструментов является утилита `lsscsi`. Пример вывода команды `lsscsi`:

```
$ lsscsi
[0:0:0:0]❶   disk❷ ATA    WDC WD3200AAJS-2  01.0 /dev/sda❸
[2:0:0:0]    disk   FLASH  Drive UT_USB20    0.00 /dev/sdb
```

В первом столбце ❶ указывается адрес устройства в системе, во втором ❷ описывается, что это за устройство, а в последнем (❸) говорится, где найти файл устройства. Все остальное — это информация о поставщике устройства.

Система Linux назначает устройства файлам устройств в том порядке, в котором их драйверы находят устройства. Таким образом, в примере ядро сначала нашло диск, а затем флеш-накопитель.

К сожалению, эта схема назначения устройств обычно вызывала проблемы при перенастройке оборудования. Предположим, у вас есть система с тремя дисками: `/dev/sda`, `/dev/sdb` и `/dev/sdc`. Если `/dev/sdb` выйдет из строя, его необходимо удалить, чтобы система снова заработала, в таком случае прежний диск `/dev/sdc` превратится в `/dev/sdb`, а диска `/dev/sdc` в системе больше не будет. Если при этом вы ссылались на имена устройств непосредственно в файле `fstab` (см. подраздел 4.2.8), вам пришлось бы внести изменения в него, чтобы вернуть все (практически все) в нормальное состояние. Для решения этой проблемы многие системы Linux используют *универсальный уникальный идентификатор* (Universally Unique Identifier, UUID; см. подраздел 4.2.4) и/или таблицу постоянных соответствий дисковых устройств с помощью *менеджера логических томов* (Logical Volume Manager, LVM).

В этом разделе мы не говорили о том, как применять диски и другие устройства хранения данных в системах Linux. Дополнительные сведения об использовании дисков см. в главе 4. Позже в этой главе мы рассмотрим, как работает поддержка SCSI в ядре Linux.

3.4.2. Виртуальные диски: /dev/xvd*, /dev/vd*

Некоторые дисковые устройства оптимизированы для виртуальных машин, таких как AWS и VirtualBox. Система виртуализации Xen использует префиксы `/dev/xvd` и `/dev/vd`.

3.4.3. Устройства долговременной памяти: /dev/nvme*

Некоторые системы теперь применяют интерфейс NVMe для работы с определенными типами твердотельных накопителей. В системе Linux эти устройства отображаются в файле `/dev/nvme*`. Используйте команду `nvme list`, чтобы вывести список этих устройств, имеющихся в вашей системе.

3.4.4. Подсистема создания виртуальных блочных устройств: /dev/dm-*, /dev/mapper/*

В некоторых системах используется менеджер LVM, что на уровень выше по сравнению с дисками и другими прямыми блочными хранилищами. Этот менеджер задействует систему ядра, называемую *сопоставителем устройств (device mapper)*. Если вы видите блочные устройства, начинающиеся с `/dev/dm`, и символические ссылки в файл `/dev/mapper`, значит, ваша система работает с данным менеджером. Подробнее об этом в главе 4.

3.4.5. CD- и DVD-приводы: /dev/sr*

Система Linux распознает большинство оптических накопителей как устройства SCSI `/dev/sr0`, `/dev/sr1` и т. д. Однако если диск использует устаревший интерфейс,

он может отображаться как устройство PATA (описано далее). Устройства `/dev/sr*` предназначены только для чтения с дисков. Для записи и перезаписи оптических устройств вы будете применять общие устройства SCSI, например `/dev/sg0`.

3.4.6. Жесткие диски PATA: /dev/hd*

PATA (Parallel ATA) — это более старый тип интерфейса хранения. Блочные устройства `/dev/hda`, `/dev/hdb`, `/dev/hdc` и `/dev/hdd` распространены в более старых версиях ядра Linux и на устаревшем оборудовании. Это фиксированные назначения, основанные на парах устройств с интерфейсами 0 и 1. Вы можете обнаружить, что диск SATA распознается как один из этих дисков. Это означает, что диск SATA работает в режиме совместимости, что снижает производительность. Проверьте настройки BIOS, чтобы узнать, можно ли переключить контроллер SATA в его собственный режим.

3.4.7. Терминалы: /dev/tty*, /dev/pts/* и /dev/tty

Терминалы — это устройства для перемещения символов между пользовательским процессом и устройством ввода-вывода (ввод текста на экран терминала). Интерфейс терминала уходит корнями в далекие времена, когда терминалы были устройствами на базе пишущих машинок.

Большинство терминалов сейчас — это *псевдотерминальные* устройства, эмулированные терминалы, которые понимают особенности ввода-вывода реальных терминалов. Вместо того чтобы обращаться к реальному аппаратному обеспечению, ядро представляет интерфейс ввода-вывода — окно терминала оболочки, в которое и вводятся команды.

Двумя общими терминальными устройствами являются `/dev/tty1` (первая виртуальная консоль) и `/dev/pts/0` (первое псевдотерминальное устройство). Сам каталог `/dev/pts` — это выделенная файловая система.

Устройство `/dev/tty` — управляющий терминал текущего процесса. Если программа в настоящее время считывает и записывает данные с терминала, это устройство является синонимом этого терминала. Процесс не обязательно подключать к терминалу.

Режимы отображения и виртуальные консоли

Система Linux имеет два основных режима отображения: текстовый и графический (в главе 14 представлены оконные системы, использующие этот режим). Хотя системы Linux традиционно загружались в текстовом режиме, большинство дистрибутивов теперь применяют параметры ядра и промежуточные механизмы графического отображения (графические экраны загрузки, например *plymouth*), чтобы полностью скрыть текстовый режим во время загрузки системы. В таких случаях система переключается в полный графический режим ближе к концу загрузки.

Linux поддерживает виртуальные консоли для мультиплексирования дисплея. Каждая виртуальная консоль может работать в графическом или текстовом режиме. В текстовом режиме вы можете переключаться между консолями с помощью комбинации функциональных клавиш и Alt. Например, сочетание Alt+F1 переводит на терминал `/dev/tty1`, Alt+F2 — на терминал `/dev/tty2` и т. д. Многие из этих виртуальных консолей могут быть заняты процессом `getty`, запускающим приглашение консоли, как описано в разделе 7.4.

Виртуальная консоль, используемая в графическом режиме, отличается от консоли в текстовом режиме. Вместо того чтобы получать назначение виртуальной консоли из настроек инициализации, графическая среда задействует свободную виртуальную консоль, если не указана какая-то определенная. Например, если у вас есть процессы `getty`, запущенные на терминалы `tty1` и `tty2`, новая графическая среда займет `tty3`. Кроме того, оказавшись в графическом режиме, необходимо нажать сочетание функциональных клавиш Ctrl+Alt, чтобы переключиться на другую виртуальную консоль, вместо более простой комбинации функциональных клавиш и Alt.

Если вы хотите переключиться на текстовую консоль после загрузки системы, нажмите сочетание клавиш Ctrl+Alt+F1. Чтобы вернуться в графическую среду, нажмите Alt+F2, Alt+F3 и т. д., пока не попадете в нужную графическую среду.

ПРИМЕЧАНИЕ

Некоторые дистрибутивы используют терминал tty1 в графическом режиме. В этом случае вам нужно будет задействовать другие консоли.

Если у вас возникли проблемы с переключением консолей из-за неисправного механизма ввода или каких-то других обстоятельств, можете попытаться заставить систему сменить консоли с помощью команды `chvt`. Например, чтобы переключиться на `tty1`, выполните следующую команду от суперпользователя root:

```
# chvt 1
```

3.4.8. Последовательные порты: /dev/ttyS*, /dev/ttyUSB*, /dev/ttyACM*

Устаревший тип порта RS-232 и аналогичные последовательные порты представлены как самые настоящие терминальные устройства. Количество действий в командной строке с последовательным портом ограничено из-за слишком большого количества настроек, о которых нужно побеспокоиться, например скорости передачи данных и управлении потоком. Однако вы можете использовать команду `screen` для подключения к терминалу, добавив путь к устройству в качестве аргумента. Вам может потребоваться разрешение на чтение и запись на устройство, получить его можно, добавив себя в определенную группу, такую как `dialout`.

Порт, известный в Windows как COM1, — это `/dev/ttyS0`, COM2 — это `/dev/ttyS1` и т. д. Подключаемые последовательные адаптеры USB отображаются вместе с USB и ACM с именами `/dev/ttyUSB0`, `/dev/ttyACM0`, `/dev/ttyUSB1`, `/dev/ttyACM1` и т. д.

Наиболее интересные приложения, связанные с последовательными портами, — это платы на базе микроконтроллеров, которые вы можете подключить к своей системе Linux для разработки и тестирования. Например, вы можете получить доступ к консоли и циклу чтения — выполнения — вывода платы CircuitPython через последовательный интерфейс USB. Нужно лишь подключить один из них, найти устройство (обычно это `/dev/ttyACM0`) и подключиться к нему с помощью команды `screen`.

3.4.9. Параллельные порты: /dev/lp0 и /dev/lp1

Тип интерфейса, который в значительной степени был заменен USB и сетями, где однонаправленные параллельные портовые устройства `/dev/lp0` и `/dev/lp1` соответствуют LPT1: и LPT2: в системе Windows. Вы можете отправлять файлы (например, файл для печати) непосредственно на параллельный порт с помощью команды `cat`, но может потребоваться дополнительно подать страницу или перезагрузить принтер. Сервер печати, такой как CUPS, гораздо эффективнее справляется при взаимодействии с принтером.

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

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

Linux имеет два набора аудиоустройств. Существуют отдельные устройства для системного интерфейса Advanced Linux Sound Architecture (продвинутая звуковая архитектура, ALSA) и более старой открытой звуковой системы (Open Sound System, OSS). Устройства ALSA находятся в каталоге `/dev/snd`, но с ними трудно работать напрямую. Системы Linux, использующие ALSA, поддерживают устройства с обратной совместимостью OSS, если при этом загружена поддержка OSS ядром.

Некоторые элементарные операции возможны с драйвером dsp OSS и аудиоустройствами. Например, компьютер воспроизводит любой WAV-файл, который вы отправляете в каталог `/dev/dsp`. Однако с воспроизведением могут возникнуть проблемы из-за несоответствия частоты. Кроме того, войдя в большинство систем, вы обнаруживаете, что устройство уже занято.

ПРИМЕЧАНИЕ

Звуки в системе Linux — это целая запутанная история из-за множества задействованных слоев. Мы только что говорили об устройствах уровня ядра, но обычно в пользовательском пространстве существуют серверы, такие как pulse-audio, которые управляют звуком из разных источников и выступают посредниками между звуковыми устройствами и другими процессами пользовательского пространства.

3.4.11. Создание файла устройства

В любой относительно новой системе Linux вы не создаете собственные файлы устройств — они создаются с помощью утилит `devtmpfs` и `udev` (см. раздел 3.5).

Однако полезно узнать, как это сделать: хоть и редко, но вам может потребоваться создать именованный конвейер или файл сокета.

Команда `mknod` создает одно устройство. Необходимо знать его имя, а также основные и второстепенные номера. Например, создать файл `/dev/sda1` можно с помощью следующей команды:

```
# mknod /dev/sda1 b 8 1
```

Параметр `b 8 1` указывает на блочное устройство с основным номером 8 и второстепенным номером 1. Для устройств символьных или именованных каналов используйте параметр `carp` вместо `b` (опустите главные и второстепенные числа для именованных конвейеров).

В более старых версиях систем Unix и Linux поддерживать каталог `/dev` было сложной задачей. С каждым значительным обновлением ядра или добавлением драйверов ядро может поддерживать больше типов устройств, а это означает, что для имен файлов устройств будет назначен новый набор основных и второстепенных номеров. Для решения этой проблемы в каждой системе в каталоге `/dev` была программа `MAKEDEV` для создания групп устройств. После обновления системы необходимо было обновить и программу `MAKEDEV`, а затем запустить ее, чтобы создать новые устройства.

В конце концов эта статическая система стала неэффективной, так что ей начали искать замену. Первой попыткой стала утилита `devfs` — реализация каталога `/dev` в пространстве ядра, которая содержала все устройства, поддерживаемые текущим ядром. Однако существовал ряд ограничений, которые привели в итоге к разработке `udev` и `devtmpfs`.

3.5. Менеджер устройств udev

Мы уже говорили о том, что неоправданная сложность процессов в ядре опасна, потому что она может легко сделать всю систему нестабильной. Например, управление файлами устройств: вы можете создавать файлы устройств в пользовательском пространстве, так зачем вам делать это в ядре? Ядро Linux может отправлять уведомления процессу пользовательского пространства, называемому `udevd`, при обнаружении нового устройства в системе (например, когда кто-то подключает USB-накопитель). Процесс `udevd` может изучить характеристики нового устройства, создать его файл, а затем выполнить любую инициализацию устройства.

ПРИМЕЧАНИЕ

Скорее всего, вы увидите, что утилита udevd в вашей системе работает как systemd-udevd, потому что это часть механизма запуска, который описан в главе 6.

Это все работает в теории. К сожалению, с этим подходом есть проблема — файлы устройств необходимы на ранних этапах загрузки, поэтому утилита `udevd` также

должна запускаться в это время. Но чтобы создавать файлы устройств, `udevd` не может зависеть от каких-либо устройств, которые она должна создать, поэтому ей необходимо очень быстро выполнить первоначальный запуск, чтобы остальная часть системы не простаивала в ожидании запуска `udevd`.

3.5.1. Файловая система devtmpfs

Файловая система `devtmpfs` была разработана для решения проблемы доступности устройства во время загрузки (более подробную информацию о файловых системах см. в разделе 4.2). Эта файловая система похожа на старую `devfs`, но она более простая. Ядро создает файлы устройств по мере необходимости, а также уведомляет `udevd` о наличии нового устройства. Получив этот сигнал, `udevd` не создает файлы устройств, но выполняет инициализацию устройства вместе с настройкой прав доступа и уведомлением других процессов о наличии новых устройств. Кроме того, она создает ряд символических ссылок в каталоге `/dev` для дальнейшей идентификации устройств. Примеры можно найти в каталоге `/dev/disk/by-id`, где каждому подключенному диску соответствует одна запись или несколько.

Например, рассмотрим ссылки на типичный диск (подключенный в `/dev/sda`) и его разделы в `/dev/disk/by-id`:

```
$ ls -l /dev/disk/by-id
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
```

Процесс `udevd` называет ссылки по типу интерфейса, а затем по информации о производителе и модели, серийному номеру и разделу (если есть).

> **ПРИМЕЧАНИЕ**
>
> Часть tmp в devtmpfs указывает на то, что файловая система находится в оперативной памяти с возможностью чтения/записи процессами пользовательского пространства. Данная характеристика позволяет udevd создавать эти символические ссылки. Более подробную информацию мы рассмотрим в подразделе 4.2.12.

Но как `udevd` узнает, какие символические ссылки создавать, и как именно она их создает? В следующем разделе описывается, как `udevd` выполняет свою работу. Однако нет необходимости знать наизусть материал этой главы, чтобы продолжать изучать книгу. На самом деле, если вы впервые сталкиваетесь с устройствами Linux, настоятельно рекомендую перейти к следующей главе, чтобы начать изучать, как использовать диски.

3.5.2. Работа и настройка менеджера udevd

Демон `udevd` работает следующим образом.

1. Ядро отправляет `udevd` уведомление о событии, называемое *uevent*, по внутренней сетевой ссылке.
2. Демон `udevd` загружает все атрибуты события `uevent`.
3. Демон `udevd` анализирует свои правила, на их основе фильтрует и обновляет событие `uevent`, а также выполняет соответствующие действия или устанавливает дополнительные атрибуты.

Входящее событие `uevent`, которое демон `udevd` получает от ядра, может выглядеть следующим образом (вы узнаете, как получить этот вывод с помощью команды `udevadm monitor --property`, в подразделе 3.5.4):

```
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
```

Это конкретное событие — смена устройства. После получения события `uevent` демон `udevd` узнает имя устройства, путь к устройству `sysfs` и ряд других атрибутов, связанных со свойствами, и теперь он готов начать обработку правил.

Файлы правил находятся в каталогах `/lib/udev/rules.d` и `/etc/udev/rules.d`. Правила в каталоге `/lib` являются значениями по умолчанию, а правила в `/etc` — переопределениями. Полное объяснение правил и больше информации можно найти на странице руководства `udev(7)`. Приведем некоторые основные сведения о том, как демон `udevd` читает правила.

1. `udevd` считывает правила от начала до конца файла правил.
2. После чтения правила и, возможно, его выполнения `udevd` продолжает чтение текущего файла правил для получения следующих применимых правил.
3. Существуют директивы (например, `GOTO`), позволяющие при необходимости пропускать части правил в файле. Они обычно помещаются в верхней части файла правил, чтобы пропустить весь файл, если он не имеет отношения к конкретному устройству, которое настраивает `udevd`.

Рассмотрим символические ссылки из примера `/dev/sda`, приведенного в подразделе 3.5.1. Они были определены правилами в файле `/lib/udev/rules.d/60-persistent-storage.rules`. Внутри вы увидите следующие строки:

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

# ATAPI devices (SPC-3 or later)
KERNEL=="sd*[!0-9]|sr*", ENV{ID_SERIAL}!="?*", SUBSYSTEMS=="scsi",
ATTRS{type}=="5",ATTRS{scsi_ level}=="[6-9]*", IMPORT{program}="ata_id
--export $devnode"
```

Эти правила соответствуют дискам ATA и оптическим носителям, представленным через подсистему SCSI ядра (см. раздел 3.6). Из примера видно, что есть несколько правил для различных способов представления устройств, но суть заключается в том, что `udevd` попытается подобрать устройство, начинающееся с `sd` или `sr`, но без номера (с выражением `KERNEL=="sd*[!0-9]|sr*"`), а также подсистему (`SUBSYSTEMS=="scsi"`) и, наконец, некоторые другие атрибуты в зависимости от типа устройства. Если все эти условные выражения истинны в любом из правил, `udevd` переходит к следующему и последнему выражению:

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

Это не условие, а директива для импорта переменных из команды `/lib/udev/ata_id`. Если у вас есть похожий диск, попробуйте выполнить команду самостоятельно в командной строке. Это будет выглядеть так:

```
# /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
ID_REVISION=01.03E10
ID_SERIAL=WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671
--пропуск--
```

Импорт теперь настраивает среду таким образом, чтобы все имена переменных в этом выводе были установлены в указанные значения. Например, любое следующее правило теперь распознает `ENV{ID_TYPE}` как диск `disk`.

В двух правилах, которые мы видели ранее, особое внимание уделяется параметру `ID_SERIAL`. В каждом правиле это условие следует вторым:

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

Это выражение принимает истинное значение `true`, если значение `ID_SERIAL` не задано. Поэтому, если значение `ID_SERIAL` уже задано, условие будет ложным (`false`), все текущее правило не применяется и `udevd` переходит к следующему правилу.

Почему так происходит? Цель этих двух правил — запустить процесс `ata_id`, чтобы найти серийный номер дискового устройства, а затем добавить эти атрибуты в текущую рабочую копию `uevent`. Вы можете найти эту общую закономерность во многих правилах `udev`.

Если значение переменной ENV{ID_SERIAL} установлено, udevd может оценить это правило позже в файле правил, который ищет любые подключенные диски SCSI:

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

Из примера видно, что это правило требует установки ENV{ID_SERIAL} и у него есть одна директива:

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

Эта директива указывает демону udevd добавить символическую ссылку для обнаруженного устройства. Итак, теперь вы знаете, откуда берутся символические ссылки на устройства!

Возможно, вам интересно, как отличить условное выражение от директивы. Условные выражения обозначаются двумя знаками равенства (==) или восклицательным знаком и знаком равенства и (!=), а директивы — одним знаком равенства (=), плюсом и знаком равенства (+=) или двоеточием и знаком равенства (:=).

3.5.3. Команда udevadm

Программа udevadm — это инструмент администрирования udevd. С помощью нее вы можете перезагрузить правила udevd и инициировать события, однако самыми мощными функциями udevadm являются возможность поиска и изучения системных устройств, а также возможность мониторинга событий uevents при получении их из ядра. Однако синтаксис команд может быть довольно сложным. Для большинства вариантов существуют длинные и короткие формы, в книге мы будем применять длинные.

Начнем с изучения системного устройства. Возвращаясь к примеру из раздела 3.5.2, чтобы просмотреть все атрибуты udev, используемые и генерируемые в сочетании с правилами для устройства, такого как /dev/sda, выполните следующую команду:

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

Вывод команды выглядит 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
--пропуск--
```

Префикс в каждой строке указывает на атрибут или другую характеристику устройства. В данном случае `P:` — это путь к устройству в `sysfs`, `N:` — узел устройства (то есть имя, данное файлу `/dev`), `S:` — указание на символическую ссылку на узел устройства, который демон `udevd` поместил в `/dev` в соответствии с его правилами, и `E:` — дополнительная информация об устройстве, извлеченная в правилах `udevd`. (В этом примере отображено гораздо больше выходных данных, чем описано, попробуйте выполнить команду самостоятельно, чтобы понять, как она работает.)

3.5.4. Отслеживание устройств

Чтобы отслеживать события `uevents` с помощью утилиты `udevadm`, используйте команду `monitor`:

```
$ udevadm monitor
```

Вывод (например, при использовании устройства флеш-накопителя) выглядит следующим образом:

```
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)
--пропуск--
```

В этом выводе по две копии каждого сообщения, поскольку поведение по умолчанию заключается в печати как входящего сообщения из ядра (помеченного `KERNEL`), так и обрабатываемых сообщений из `udevd`. Чтобы видеть только события ядра, добавьте параметр `--kernel`, а чтобы видеть только события, обрабатываемые `udevd`, используйте `--udev`. Чтобы просмотреть все входящие события, включая атрибуты, как показано в подразделе 3.5.2, добавьте параметр `--property`. Параметры `--udev` и `--property` вместе показывают событие после обработки.

Вы также можете фильтровать события по подсистемам. Например, чтобы просмотреть только сообщения ядра, относящиеся к изменениям в подсистеме SCSI, используйте команду

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

Дополнительные сведения об `udevadm` см. на странице руководства `udevadm(8)`.

Возможности `udev` этим не ограничиваются — их гораздо больше. Например, существует демон под названием `udisksd`, который прослушивает события, чтобы автоматически присоединять диски и уведомлять другие процессы о наличии новых дисков.

3.6. Подробнее об интерфейсе SCSI и ядре Linux

В этом разделе мы рассмотрим поддержку SCSI в ядре Linux как способ изучения части архитектуры ядра Linux. Чтобы начать работать с дисками и устройствами, знать это не обязательно, поэтому, если торопитесь, перейдите к главе 4. Кроме того, материал этого раздела более продвинутого уровня и имеет менее практический характер, чем то, о чем мы говорили прежде, поэтому, если не хотите забираться в дебри теории Linux, вам определенно стоит перейти к следующей главе.

Начнем с небольшой предыстории. Традиционная настройка оборудования SCSI представляет собой адаптер хоста, связанный с цепочкой устройств по шине SCSI, как показано на рис. 3.1. Адаптер хоста подключается к компьютеру. Адаптер хоста и устройства имеют идентификатор SCSI ID, и на шину может приходиться 8 или 16 идентификаторов в зависимости от версии SCSI. Некоторые администраторы используют термин *«цель SCSI»* для обозначения устройства и его идентификатора SCSI ID, поскольку один конец сеанса в протоколе SCSI называется целью.

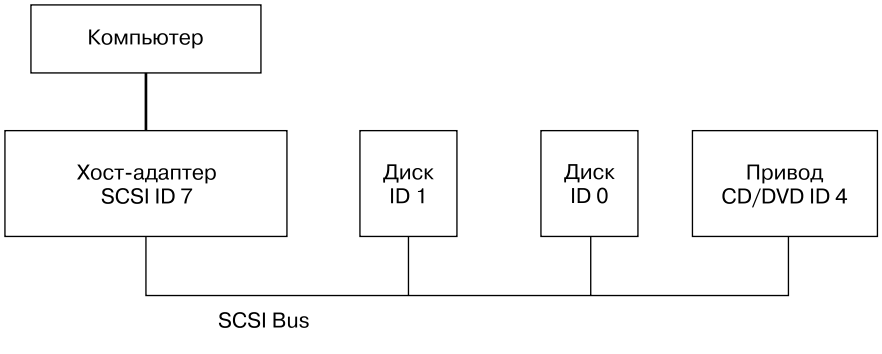


Рис. 3.1. Шина SCSI с хост-адаптером и устройствами

Любое устройство может взаимодействовать с другим устройством на равных с помощью команд SCSI. Компьютер не подключен непосредственно к цепочке устройств, поэтому он должен проходить через адаптер хоста, чтобы взаимодействовать

с дисками и другими устройствами. Как правило, компьютер отправляет команды SCSI адаптеру хоста для ретрансляции на устройства, а устройства ретранслируют ответы обратно через адаптер хоста.

Более новые версии SCSI, такие как Serial Attached SCSI (SAS, интерфейс SCSI с последовательным подключением), обеспечивают исключительную производительность, но вы, вероятно, не найдете реальных устройств SCSI на большинстве машин. Вы чаще будете сталкиваться с USB-накопителями, применяющими команды SCSI. Кроме того, устройства, поддерживающие ATAPI (например, приводы CD/DVD-ROM), используют версию набора команд SCSI.

Диски SATA также отображаются в вашей системе как устройства SCSI, но они немного отличаются, поскольку большинство их взаимодействуют через уровень трансляции в библиотеке libata (см. подраздел 3.6.2). Некоторые контроллеры SATA (особенно высокопроизводительные RAID-контроллеры) выполняют этот перевод через аппаратное обеспечение.

Как все это сочетается и работает вместе? Рассмотрим 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
```

Цифры в квадратных скобках — это (слева направо) номер адаптера хоста SCSI, номер шины SCSI, идентификатор SCSI-устройства и LUN (Logical Unit Number, номер логического блока, дополнительное подразделение устройства). В этом примере имеется четыре подключенных адаптера (`scsi0`, `scsi1`, `scsi2` и `scsi3`), каждый из которых имеет одну шину (все с номером 0) и только одно устройство на каждой шине (все с целевым номером 0). Считыватель USB-карт в 2:0:0 имеет четыре логических блока — по одному для каждого типа флеш-накопителя. Ядро назначило каждому логическому блоку отдельный файл устройства.

Несмотря на то что они не являются устройствами SCSI, устройства NVMe иногда могут отображаться на выходе `lsscsi` с *N* в качестве номера адаптера.

ПРИМЕЧАНИЕ

Если вы хотите самостоятельно попробовать работать с утилитой lsscsi, установите ее в качестве дополнительного пакета.

На рис. 3.2 показана иерархия драйверов и интерфейсов внутри ядра для конкретной конфигурации системы, начиная с отдельных драйверов устройств и заканчивая драйверами блоков. Она не включает в себя обобщенные драйверы SCSI generic *(sg)*.

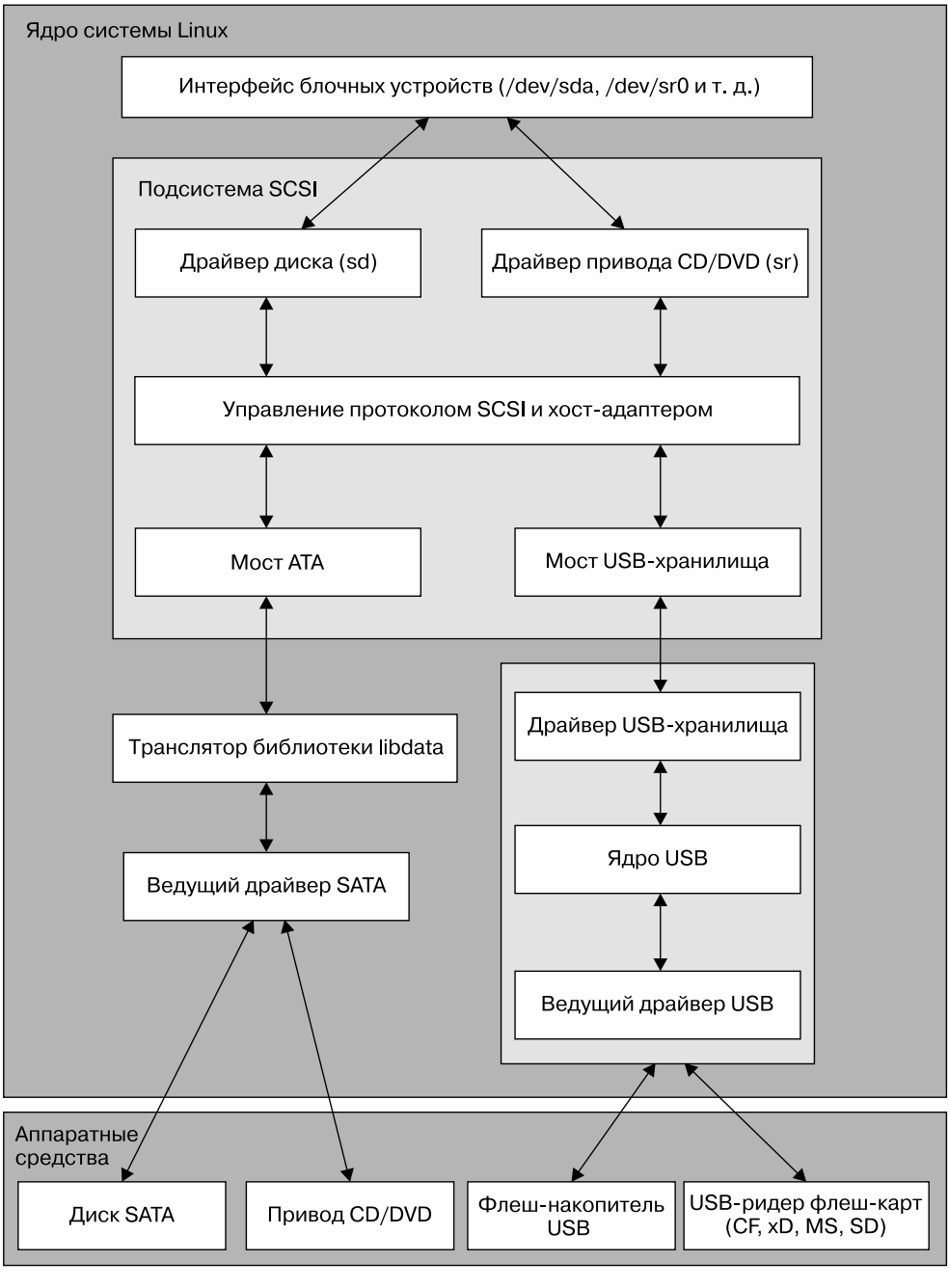


Рис. 3.2. Схема подсистемы SCSI Linux

Это большая структура, и на первый взгляд она может показаться ошеломляющей, однако поток данных на рисунке крайне линейный. Начнем с анализа подсистемы SCSI и ее трех уровней драйверов.

- Верхний уровень обрабатывает операции для класса устройств. Например, драйвер *sd* (SCSI disk) находится на этом уровне, он знает, как переводить запросы из интерфейса устройства блока ядра в команды для конкретного диска в протоколе SCSI и наоборот.
- Средний уровень модерирует и направляет сообщения SCSI между верхним и нижним уровнями, а также отслеживает все шины SCSI и устройства, подключенные к системе.
- Нижний уровень обрабатывает действия, зависящие от оборудования. Драйверы здесь отправляют исходящие сообщения протокола SCSI определенным адаптерам хоста или оборудованию и принимают входящие сообщения из оборудования. Причина такого отделения от верхнего уровня заключается в том, что хотя сообщения SCSI одинаковы для класса устройств (например, класса дисков), различные типы адаптеров хоста имеют различные процедуры отправки одних и тех же сообщений.

Верхний и нижний уровни содержат множество различных драйверов, но важно помнить, что для любого файла устройства в системе ядро почти всегда использует по одному драйверу верхнего и нижнего уровней. Для диска в каталоге `/dev/sda` в нашем примере ядро применяет драйвер верхнего уровня sd и драйвер нижнего уровня моста ATA.

Бывают случаи, когда вы можете задействовать более одного драйвера верхнего уровня для одного аппаратного устройства (см. раздел 3.6.3). Для реальных аппаратных устройств SCSI, таких как диск, подключенный к адаптеру хоста SCSI или аппаратному RAID-контроллеру, драйверы нижнего уровня напрямую связываются с нижестоящим оборудованием. Однако для большинства аппаратных средств, подключенных к подсистеме SCSI, все работает иначе.

3.6.1. USB-накопитель и SCSI

Для того чтобы подсистема SCSI могла взаимодействовать с обычными USB-накопителями, как показано на рис. 3.2, ядру требуется нечто большее, чем просто драйвер SCSI нижнего уровня. Флеш-накопитель USB, представленный файлом `/dev/sdf`, понимает команды SCSI, но для того чтобы фактически взаимодействовать с накопителем, ядро должно знать, как общаться через систему USB.

Теоретически, система USB очень похожа на систему SCSI — у нее есть классы устройств, шины и контроллеры хостов. Поэтому неудивительно, что ядро Linux включает в себя трехуровневую подсистему USB, схожую с подсистемой SCSI, с драйверами класса устройств вверху, ядром управления шиной посередине и драйверами хост-контроллера внизу. Подобно тому как подсистема SCSI передает

команды SCSI между своими компонентами, подсистема USB передает сообщения USB между своими. Есть даже команда `lsusb`, похожая на команду `lsscsi`.

Компонент, который нас интересует, — это драйвер USB-накопителя в верхней части иерархии. Он выступает в роли переводчика. На одном конце драйвер говорит на «языке» SCSI, а на другом — на «языке» USB. Поскольку оборудование для хранения данных включает команды SCSI в свои USB-сообщения, драйвер выполняет относительно простую работу: он в основном переупаковывает данные.

Благодаря наличию подсистем SCSI и USB у вас есть почти все необходимое для работы с флеш-накопителями. Последним недостающим звеном является драйвер нижнего уровня в подсистеме SCSI, поскольку драйвер USB-накопителя — это часть подсистемы USB, а не подсистемы SCSI. (По организационным причинам две подсистемы не должны совместно использовать один драйвер.) Чтобы заставить подсистемы взаимодействовать друг с другом, простой драйвер SCSI-моста нижнего уровня подключается к драйверу хранения USB-подсистемы.

3.6.2. Интерфейсы SCSI и ATA

Жесткий диск SATA и оптический привод, показанные на рис. 3.2, работают с одним и тем же интерфейсом SATA. Для подключения специфичных для SATA драйверов ядра к подсистеме SCSI ядро использует драйвер-мост, как и в случае с USB-накопителями, но с другим механизмом и дополнительными сложностями. Оптический привод задействует ATAPI — версию команд SCSI, закодированных в протоколе ATA. Однако жесткий диск не использует ATAPI и не кодирует никаких команд SCSI!

Ядро Linux использует часть библиотеки libata для согласования дисков SATA (и ATA) с подсистемой SCSI. Для оптических приводов, говорящих на языке ATAPI, упаковка команд SCSI в протокол ATA и извлечение из него — это относительно простая задача. Но для жесткого диска задача намного усложняется, потому что библиотека должна выполнить полный перевод команд.

Работа оптического привода похожа на ввод текста книги на английском языке в компьютер. Вам не нужно понимать, о чем она, чтобы выполнить эту задачу, и даже не нужно понимать английский язык. Но задача для жесткого диска больше похожа на чтение немецкой книги и перевод ее текста на английский язык. В этом случае вам нужно понимать оба языка, а также само содержание книги.

Несмотря на сложность этой задачи, libata выполняет ее и позволяет подключать интерфейсы ATA/SATA и устройства к подсистеме SCSI. (Обычно задействовано больше драйверов, чем один драйвер хоста SATA, показанный на рис. 3.2.)

3.6.3. Обобщенные устройства SCSI

Когда процесс пользовательского пространства взаимодействует с подсистемой SCSI, он обычно выполняет задачу через уровень блочных устройств и/или другую

службу ядра, которая находится поверх драйвера класса устройств SCSI (например, sd или sr). Другими словами, большинству пользовательских процессов ничего не нужно знать об устройствах SCSI или их командах.

Однако пользовательские процессы могут обходить драйверы классов устройств и передавать команды протокола SCSI непосредственно устройствам через их обобщенные устройства. Например, рассмотрим систему, описанную в разделе 3.6, но на этот раз взгляните на то, что происходит, если добавить 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 каждая запись содержит обобщенный файл устройства SCSI в последнем столбце (❶). Например, обобщенным устройством для оптического привода в `/dev/sr0` является `/dev/sg1`.

Почему может понадобиться обобщенное устройство? Это обусловлено сложностью кода ядра: когда задачи становятся особенно сложными, лучше их вывести за пределы ядра. Представьте запись и чтение на CD/DVD. Чтение оптического диска — это довольно простая операция, и для нее есть специальный драйвер ядра.

Однако запись на оптический диск значительно сложнее, чем чтение, и никакие критические системные службы не зависят от действия записи. Нет никаких причин использовать пространство ядра для этой задачи. Поэтому для записи на оптический диск в Linux вы запускаете программу пользовательского пространства, которая взаимодействует с обобщенным устройством SCSI, таким как `/dev/sg1`. Эта программа может быть менее эффективной, чем драйвер ядра, но ее гораздо проще создавать и поддерживать.

3.6.4. Методы множественного доступа к одному устройству

На рис. 3.3 показаны две точки доступа (sr и sg) для оптического привода из пространства пользователя для подсистемы SCSI Linux (любые драйверы ниже нижнего уровня SCSI не рассматриваются). Процесс А считывает данные с диска с помощью драйвера sr, а процесс Б записывает данные на диск с помощью драйвера sg. Однако подобные процессы обычно не запускаются одновременно для доступа к одному и тому же устройству.

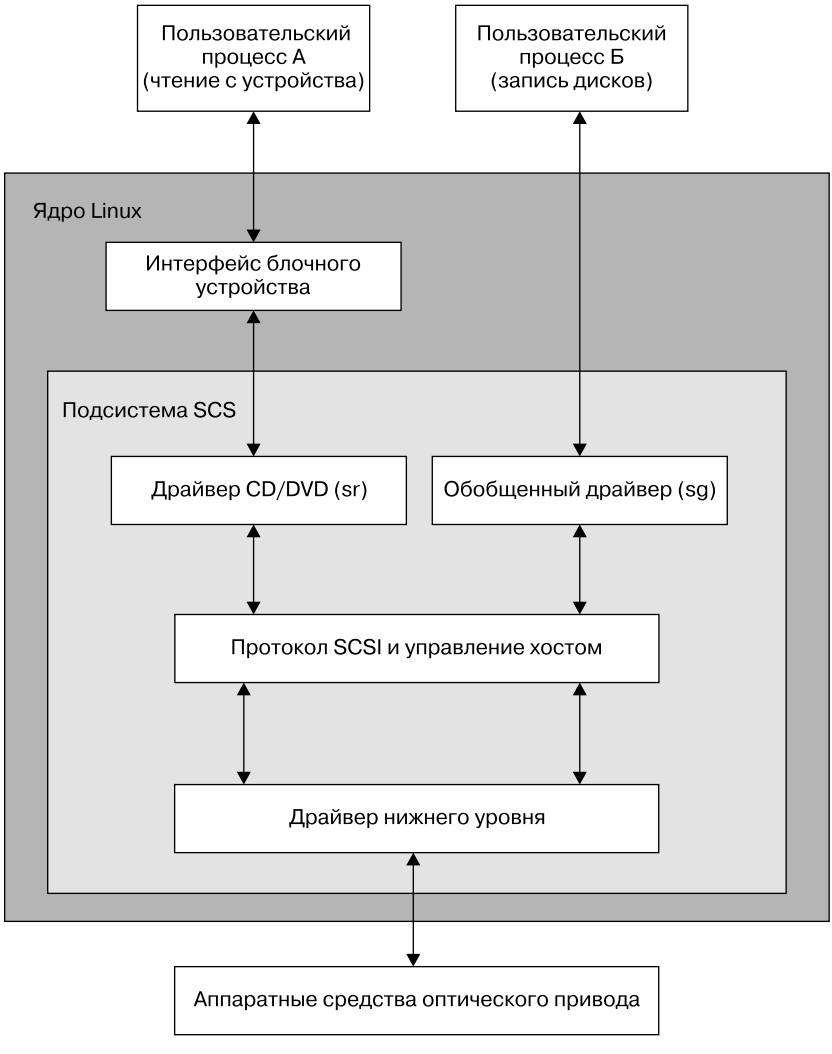
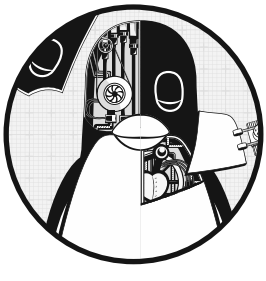


Рис. 3.3. Схема драйверов оптического устройства

На рис. 3.3 процесс А считывает данные с блочного устройства. Но действительно ли пользовательские процессы читают данные таким образом? Ответ — нет, не так, ведь поверх блочных устройств имеется больше слоев и еще больше точек доступа для жестких дисков. Об этом вы узнаете из следующей главы.

4

Диски и файловые системы



В главе 3 мы рассмотрели дисковые устройства верхнего уровня, доступные ядру. Здесь же мы подробно обсудим, как работать с дисками в системе Linux. Вы узнаете, как разбивать диски на разделы, создавать и поддерживать файловые системы, которые находятся внутри разделов диска, а также работать с пространством подкачки.

Напомню, что дисковые устройства имеют такие имена, как `/dev/sda`, первый диск подсистемы SCSI. Этот тип блочного устройства представляет собой весь диск, но внутри самого диска есть множество различных компонентов и слоев.

На рис. 4.1 отображена схема простого диска Linux (рисунок не масштабирован). В этой главе вы узнаете, где находится каждая его часть.

Разделы являются более мелкими частями всего диска. В Linux они обозначаются цифрой после блочного устройства, поэтому у них есть такие имена, как `/dev/sda1` и `/dev/sdb3`. Ядро представляет каждый раздел как блочное устройство, как представляло бы весь диск. Разделы определяются на небольшой области диска, называемой *таблицей разделов* (также *метка диска*).

Ядро позволяет одновременно получать доступ как ко всему диску, так и к одному из его разделов, но обычно в этом нет необходимости, если вы не копируете диск целиком.

Менеджер логических томов Linux (Logical Volume Manager, LVM) позволяет более гибко управлять традиционными дисковыми устройствами и разделами и в настоящее время применяется во многих системах. Мы рассмотрим менеджер LVM в разделе 4.4.

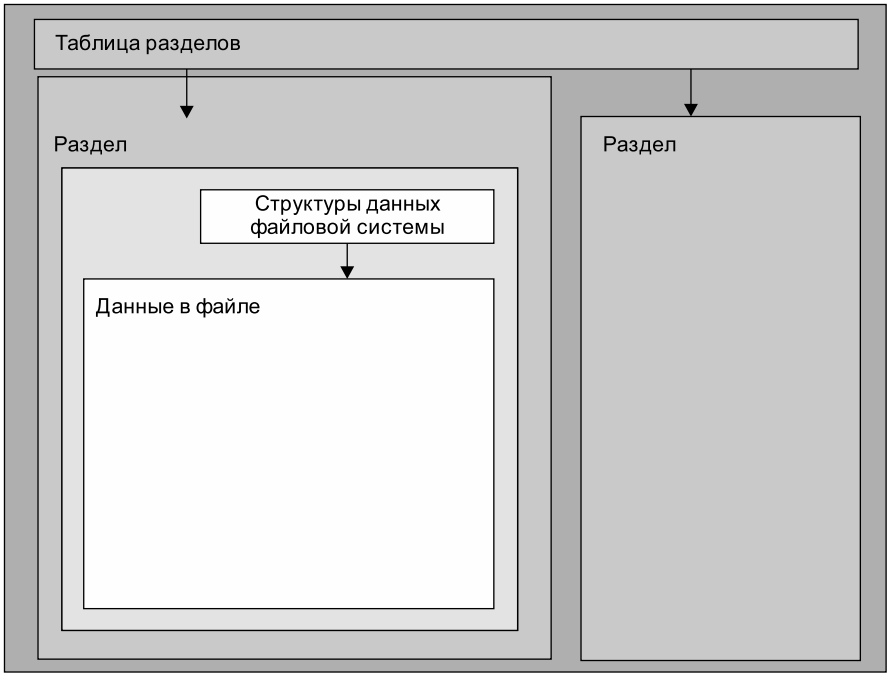


Рис. 4.1. Типичная схема диска Linux

ПРИМЕЧАНИЕ

Несколько разделов данных когда-то были распространены в системах с большими дисками, поскольку старые компьютеры могли загружаться только с определенных частей диска. Кроме того, администраторы задействовали части для резервирования определенного объема пространства для областей операционной системы. Например, они не хотели, чтобы пользователи могли заполнять всю систему и препятствовать работе критически важных служб. Эта практика не уникальна для систем Unix — вы найдете множество новых систем Windows с несколькими разделами на одном диске. Кроме того, большинство систем имеют отдельный раздел подкачки.

Следующий уровень разбиения на разделы — это *файловая система*, база данных файлов и каталогов, с которыми вы привыкли взаимодействовать в пользовательском пространстве. Файловые системы рассмотрим в разделе 4.2.

Как видно на рис. 4.1, если вы хотите получить доступ к данным в файле, вам необходимо использовать соответствующее расположение раздела из таблицы разделов, а затем выполнить поиск в базе данных файловой системы в этом разделе.

Для доступа к данным на диске ядро Linux и применяет систему слоев, показанную на рис. 4.2. Подсистема SCSI и все остальное, описанное в разделе 3.6, представлены одним блоком. Обратите внимание: вы можете работать с диском как через

файловую систему, так и непосредственно через дисковые устройства. В этой главе мы рассмотрим, как работают оба метода. Для упрощения менеджер LVM не представлен на рис. 4.2, но у него есть компоненты в интерфейсе блочного устройства и несколько компонентов управления в пользовательском пространстве.

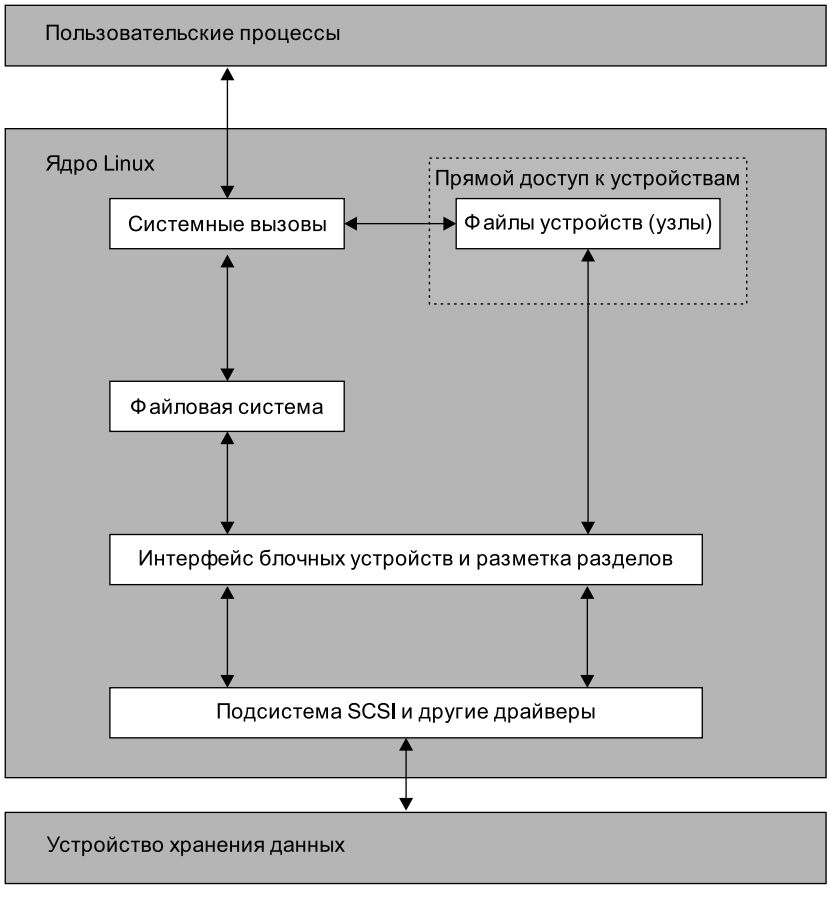


Рис. 4.2. Схема ядра с доступом к диску

Чтобы понять, как все сочетается между собой, начнем с самого начала — с разделов.

4.1. Разбиение дисков на разделы

Существует много вариаций таблиц разделов. В таблице разделов нет ничего особенного — это просто набор данных, которые сообщают о том, как разделяются блоки на диске.

Традиционная таблица, относящаяся к временам Windows, находится внутри *главной загрузочной записи (Master Boot Record, MBR)*, и у нее есть множество ограничений. Большинство новых систем используют глобальную таблицу разделов с уникальными идентификаторами GPT *(Globally Unique Identifier Partition Table)*.

Инструменты разделения в Linux:

- `parted` (сокращение от PARTition EDitor, редактор разделов) — текстовый инструмент, поддерживающий таблицы как MBR, так и GPT;
- графическая версия инструмента `parted`;
- `fdisk` — традиционный текстовый инструмент разбиения дисков Linux на разделы. Последние версии `fdisk` поддерживают MBR, GPT и многие другие типы таблиц разделов, но более старые версии ограничены поддержкой MBR.

Поскольку инструмент `parted` уже некоторое время поддерживает как MBR, так и GPT и позволяет легко запускать отдельные команды для получения меток разделов, мы будем использовать его для отображения таблиц разделов. Однако при создании и изменении таблиц разделов задействуем `fdisk`. Так мы проиллюстрируем оба интерфейса. Многие пользователи предпочитают интерфейс `fdisk` из-за его интерактивного характера и того факта, что он не вносит никаких изменений на диск, пока вы их не просмотрите (мы обсудим это в ближайшее время).

ПРИМЕЧАНИЕ

Существует огромная разница между созданием разделов и управлением файловой системой: таблица разделов определяет простые границы на диске, в то время как файловая система — это гораздо более сложная система данных. По этой причине мы будем использовать отдельные инструменты для работы с разделами и создания файловых систем (см. подраздел 4.2.2).

4.1.1. Просмотр таблицы разделов

Вы можете просмотреть таблицу разделов своей системы с помощью команды `parted -l`. В выводе далее показаны два дисковых устройства с двумя различными типами таблиц разделов:

```
# parted -l
Model: ATA KINGSTON SM2280S (scsi)
Disk /dev/sda: 240GB ❶
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number Start  End   Size   Type     File system  Flags
 1     1049kB 223GB 223GB  primary  ext4         boot
 2     223GB  240GB 17.0GB extended
 5     223GB  240GB 17.0GB logical  linux-swap(v1)
```

```
Model: Generic Flash Disk (scsi)
Disk /dev/sdf: 4284MB ❷
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:

Number Start  End    Size   File system Name     Flags
 1     1049kB 1050MB 1049MB             myfirst
 2     1050MB 4284MB 3235MB             mysecond
```

Первое устройство (`/dev/sda`) (❶) использует традиционную таблицу разделов MBR, которую команда `parted` назвала `msdos`, а второе (`/dev/sdf`) (❷) содержит GPT. Обратите внимание на то, что два типа таблиц хранят разные наборы параметров. В частности, в таблице MBR нет столбца `Name`, поскольку в этой схеме не существует имен. (В GPT произвольно выбраны имена `myfirst` и `mysecond`.)

ПРИМЕЧАНИЕ

Следите за размерами блоков при чтении таблиц разделов. Вывод команды parted показывает приблизительный размер, который легче всего читать. В то же время команда fdisk -l показывает точное число, но в большинстве случаев единицы измерения представляют собой 512-байтовые секторы, что может сбить с толку, поскольку может показаться, что фактические размеры вашего диска и разделов были удвоены. При внимательном изучении таблицы разделов fdisk вы увидите, что в ней отображается также информация о размере сектора.

Основы MBR

Таблица MBR в этом примере содержит основной, расширенные и логические разделы. *Основной раздел* — это обычный раздел диска, например раздел 1. Базовый MBR имеет ограничение в четыре основных раздела, поэтому, если вам нужно больше четырех, необходимо назначить один из них *расширенным разделом*. Расширенный раздел разбивается на *логические разделы*, которые операционная система затем может использовать, как и любой другой раздел. В этом примере раздел 2 представляет собой расширенный раздел, содержащий логический раздел 5.

ПРИМЕЧАНИЕ

Тип файловой системы, выводимый командой parted, не обязательно совпадает с полем идентификатора системы в его записях MBR. Идентификатор системы MBR — это просто число, определяющее тип раздела: например, 83 — это раздел Linux, а 82 — область подкачки Linux. Кроме того, parted самостоятельно определяет, какая файловая система находится в этом разделе. Если вам необходимо знать идентификатор системы для MBR, используйте команду fdisk -l.

Разделы LVM: краткий обзор

Если при просмотре таблицы разделов вы видите разделы, помеченные как LVM (код `8e` в качестве типа раздела), устройства с именем `/dev/dm -*` или ссылки на device mapper, значит, ваша система использует менеджер LVM. Мы начнем

с традиционного прямого разделения дисков, которое выглядит немного иначе, чем в системе, применяющей LVM.

Чтобы вы знали, чего ожидать, взглянем на некоторые примеры вывода команды `parted -l` в системе с менеджером LVM (свежеустановленная система Ubuntu с использованием LVM в VirtualBox). Во-первых, есть описание фактической таблицы разделов, которая выглядит ожидаемо, за исключением флага `lvm`:

```
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sda: 10.7GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number Start  End    Size   Type    File system Flags
 1     1049kB 10.7GB 10.7GB primary             boot, lvm
```

Затем есть некоторые устройства, которые выглядят так, как будто они должны быть разделами, но называются дисками:

```
Model: Linux device-mapper (linear) (dm)
Disk /dev/mapper/ubuntu--vg-swap_1: 1023MB
Sector size (logical/physical): 512B/512B
Partition Table: loop
Disk Flags:

Number Start End    Size   File system   Flags
 1     0.00B 1023MB 1023MB linux-swap(v1)


Model: Linux device-mapper (linear) (dm)
Disk /dev/mapper/ubuntu--vg-root: 9672MB
Sector size (logical/physical): 512B/512B
Partition Table: loop
Disk Flags:

Number Start End    Size   File system Flags
 1     0.00B 9672MB 9672MB ext4
```

Простой способ представить процесс — это понять, что разделы отделены от таблицы разделов. Что происходит на самом деле, вы узнаете из раздела 4.4.

ПРИМЕЧАНИЕ

При использовании команды fdisk -l вывод окажется менее подробным: в приведенном ранее примере вы не увидите ничего, кроме одного физического раздела с меткой LVM.

Первичное чтение ядром

При первоначальном чтении таблицы MBR ядро Linux выдает вывод отладки, как в следующем примере (можно просмотреть его с помощью команды `journalctl -k`):

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

Часть вывода `sda2 < sda5 >` указывает на то, что `/dev/sda2` — это расширенный раздел, содержащий один логический раздел `/dev/sda5`. Обычно сам расширенный раздел игнорируется, поскольку необходим доступ именно к логическим разделам, которые он содержит.

4.1.2. Редактирование таблиц разделов

Просмотр таблиц разделов — довольно простая и безвредная операция. Изменение таблиц разделов также относительно несложная задача, но внесение такого рода изменений на диск сопряжено с риском. Помните:

- изменение таблицы разделов затрудняет восстановление любых данных в разделах, которые вы удаляете или переопределяете, поскольку это может привести к удалению расположения файловых систем на этих разделах. Если диск содержит важные данные, убедитесь, что у вас есть резервная копия;
- никакие разделы на целевом диске в момент изменения не должны использоваться. Это важно, поскольку большинство дистрибутивов Linux автоматически монтируют любую обнаруженную файловую систему. (Дополнительные сведения о монтировании и демонтировании см. в подразделе 4.2.3.)

Когда будете готовы, выберите программу разбиения на разделы. Если хотите использовать `parted`, примените утилиту `parted` командной строки или графический интерфейс, например `gparted`, а с `fdisk` довольно легко работать в командной строке. У всех этих утилит есть онлайн-руководства, и сами они просты в применении. (Попробуйте использовать их на флеш-накопителе, если у вас нет запасных дисков.)

Тем не менее существует большая разница в том, как работают `fdisk` и `parted`. С помощью `fdisk` вы создаете новую таблицу разделов перед внесением фактических изменений на диск, и она реализует изменения только при выходе из программы. Но при использовании `parted` разделы создаются, изменяются и удаляются *по мере выполнения команд*. У вас нет возможности просмотреть таблицу разделов перед ее изменением. Эти различия — ключ к пониманию того, как эти две утилиты взаимодействуют с ядром. Утилита `fdisk`, как и `parted`, полностью изменяет разделы в пользовательском пространстве — нет необходимости предоставлять поддержку ядра для перезаписи таблицы разделов, поскольку пользовательское пространство может считывать и изменять все блочные устройства.

Однако в какой-то момент ядро должно прочитать таблицу разделов, чтобы представить разделы как блочные устройства, которые можно использовать. Утилита `fdisk` делает это довольно простым способом. После изменения таблицы разделов `fdisk` выдает один системный вызов, чтобы сообщить ядру, что оно должно перечитать таблицу разделов диска (далее будет приведен пример взаимодействия ядра с `fdisk`). Затем ядро генерирует вывод отладки, который можно просмотреть с помощью команды `journalctl -k`. Например, создав два раздела в `/dev/sdf`, вы увидите следующее:

```
sdf: sdf1 sdf2
```

Инструменты `parted` не используют один системный вызов для всего диска, вместо этого они сигнализируют ядру, когда изменяются отдельные разделы. После обработки одного изменения раздела ядро не выдает предыдущий вывод отладки.

Перечислим несколько способов увидеть изменения раздела.

- Используйте `udevadm` для просмотра изменений событий ядра. Например, команда `udevadm monitor --kernel` покажет, как удаляются старые устройства разделов и добавляются новые.
- Проверьте файл `/proc/partitions` для получения полной информации о разделах.
- Проверьте файл `/sys/block/device/` на наличие измененных системных интерфейсов разделов или `/dev` на наличие измененных устройств разделов.

ПРИНУДИТЕЛЬНАЯ ПЕРЕЗАГРУЗКА ТАБЛИЦЫ РАЗДЕЛОВ

Если вам необходимо удостовериться в правильности вносимых изменений в таблицу разделов, можете использовать команду `blockdev` для выполнения системного вызова в старомодном стиле, который выдает `fdisk`. Например, чтобы заставить ядро перезагрузить таблицу разделов в `/dev/sdf`, выполните следующую команду:

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

4.1.3. Создание таблицы разделов

Давайте применим только что полученные знания и создадим новую таблицу разделов на новом пустом диске. В следующем примере мы исходим из таких условий:

- диск размером 4 Гбайт (небольшое неиспользуемое USB-флеш-устройство — возьмите устройство любого размера, которое есть под рукой);
- таблица разделов MBR;
- два раздела, предназначенные для файловой системы ext4: 200 Мбайт и 3,8 Гбайт;
- дисковое устройство в `/dev/sdd` (вам нужно будет найти местоположение устройства с помощью команды `lsblk`).

Для выполнения этой задачи используйте утилиту `fdisk`. Напомним, что это интерактивная команда, поэтому, убедившись, что к диску ничего не примонтировано, начните с команды

```
# fdisk /dev/sdd
```

Появится вводное сообщение, а затем командная строка:

```
Command (m for help):
```

В первую очередь выведите текущую таблицу с помощью команды `p` (команды утилиты `fdisk` довольно короткие). Вот как это будет выглядеть:

```
Command (m for help): p
Disk /dev/sdd: 4 GiB, 4284481536 bytes, 8368128 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x88f290cc

Device     Boot Start     End Sectors Size Id Type
/dev/sdd1        2048 8368127 8366080   4G c W95 FAT32 (LBA)
```

Большинство устройств уже содержат один раздел в стиле FAT, как в примере в `/dev/sdd1`. Поскольку вы хотите создать новые разделы для системы Linux, то можете удалить существующие разделы (конечно, если уверены, что важных данных в них нет), например:

```
Command (m for help): d
Selected partition 1
Partition 1 has been deleted.
```

Помните, что утилита `fdisk` не вносит изменений, пока вы не сохраните таблицу разделов, поэтому диск остается неизменным до последнего. Если вы допустили ошибку, которую не можете исправить, с помощью команды `q` выйдите из редактора `fdisk`, не сохраняя изменений. Теперь создадим первый раздел размером 200 Мбайт с помощью команды `n`:

```
Command (m for help): n
Partition type
   P   primary (0 primary, 0 extended, 4 free)
   E   extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-8368127, default 2048): 2048
Last sector, +sectors or +size{K,M,G,T,P} (2048-8368127, default 8368127): +200M

Created a new partition 1 of type 'Linux' and of size 200 MiB.
```

В примере `fdisk` предлагает раздел MBR, номер раздела, начало раздела и его конец (или размер). Значения по умолчанию довольно часто отображают то, что нужно. Единственное, что здесь изменилось, — это конец/размер раздела с синтаксисом `+` для указания размера и единицы измерения.

Создание второго раздела происходит почти так же, за исключением того, что вы будете использовать все значения по умолчанию, поэтому мы не будем рассматривать

этот пример. Закончив разделение, введите команду `p` (print, вывод) для просмотра содержимого:

```
Command (m for help): p
[--пропуск--]
Device     Boot Start     End Sectors Size Id Type
/dev/sdd1        2048  411647  409600 200M 83 Linux
/dev/sdd2      411648 8368127 7956480 3.8G 83 Linux
```

Когда будете готовы сохранить таблицу разделов, используйте команду `w` (write, запись):

```
Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
```

Обратите внимание, что утилита `fdisk` не спрашивает вас, уверены ли вы в безопасности своих действий, — она просто выполняет свою работу.

Если вас интересует дополнительная диагностика, задействуйте команду `journalctl -k`, чтобы просмотреть сообщения о чтении ядром, упомянутые ранее, но помните, что они появятся только при использовании утилиты `fdisk`.

На данный момент мы рассмотрели все основы, и можно приступать к разбиению дисков на разделы, однако если вы хотите больше узнать о дисках, продолжайте читать следующий раздел. В противном случае переходите к разделу 4.2, где рассказывается о размещении файловой системы на диске.

4.1.4. Геометрия дисков и разделов

Любое устройство с движущимися частями вносит сложность в систему, поскольку существуют физические элементы, которые не поддаются абстрагированию. Жесткий диск — не исключение: даже если представлять его как блочное устройство с произвольным доступом к любому блоку, если система не будет внимательна к тому, как она размещает данные на диске, это может иметь серьезные последствия для производительности. Рассмотрим физические свойства простого диска с одной пластиной (рис. 4.3).

Диск состоит из вращающейся пластины на шпинделе с головкой, прикрепленной к движущемуся коромыслу, которое может перемещаться по радиусу диска. Когда диск вращается под головкой, последняя считывает данные. Когда коромысло находится в одном положении, головка может считывать данные только с фиксированного круга, называемого *цилиндром*. Большие диски имеют более одного «блина» (жесткого магнитного диска), которые вращаются вокруг одного и того же шпинделя. Каждый «блин» может иметь одну или две головки в верхней и/или нижней части, все головки прикреплены к одному шпинделю (коромыслу) и движутся согласованно. Шпиндель смещается, а на диске находится множество

цилиндров, от маленьких в его центре до больших на периферии. Наконец, вы можете разделить цилиндр на доли, называемые *секторами*. Такой подход к геометрии диска кратко называется CHS (*cylinder-head-sector — цилиндр, головка, сектор*), в старых системах вы могли бы найти любую часть диска, используя эти три параметра.

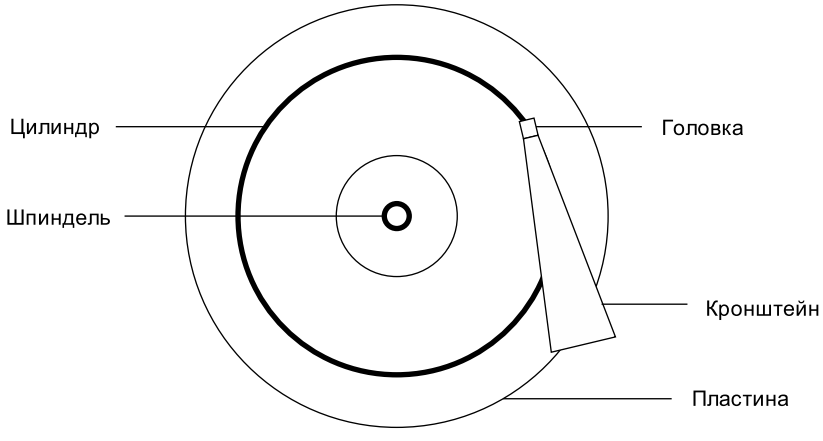


Рис. 4.3. Вид на жесткий диск сверху

ПРИМЕЧАНИЕ

Дорожка — это часть цилиндра, к которой обращается одна головка, поэтому на рис. 4.3 цилиндр также является дорожкой.

Ядро и различные программы разбиения на разделы могут показать, какое количество цилиндров имеет диск. Однако на любом из современных жестких дисков *эти значения не правдивы*! Традиционная схема передачи, применяемая системой CHS, не масштабируется на современном дисковом оборудовании и не учитывает тот факт, что появилась возможность поместить во внешние цилиндры больше данных, чем во внутренние. Аппаратное обеспечение диска поддерживает *логическую блочную адресацию (Logical Block Addressing, LBA)* для передачи местоположения на диске по номеру блока (это гораздо более простой интерфейс), но остатки CHS еще видны. Например, таблица разделов MBR содержит информацию о CHS, а также эквиваленты LBA, и некоторые загрузчики все еще доверяют значениям CHS (не волнуйтесь, большинство загрузчиков Linux используют значения LBA).

ПРИМЕЧАНИЕ

Слово «сектор» может сбивать с толку, потому что программы разделения Linux также применяют его, но для обозначения другого инструмента.

ВАЖНЫ ЛИ ГРАНИЦЫ ЦИЛИНДРА?

Сущность цилиндров когда-то имела решающее значение для работы с разделами, потому что цилиндры сами по себе — это идеальные границы для разделов. Считывание потока данных с цилиндра происходит очень быстро, потому что головка может непрерывно собирать данные по мере вращения диска. Раздел, выполненный в виде набора соседних цилиндров, также обеспечивает быстрый непрерывный доступ к данным, поскольку головке не нужно далеко перемещаться между цилиндрами.

Хотя внешне диски практически не изменились, понятие точного разбиения на разделы устарело. Некоторые старые программы разбиения уведомляют, если вы не размещаете свои разделы точно на границах цилиндра. Игнорируйте это предупреждение — с этим мало что можно сделать, потому что сообщаемые CHS параметры современных дисков просто не соответствуют действительности. Схема LBA наряду с улучшенной логикой в новых утилитах разбиения гарантирует, что ваши разделы будут расположены эффективно.

4.1.5. Чтение твердотельных дисков

Устройства хранения данных без движущихся частей, такие как *твердотельные накопители (SSD)*, радикально отличаются от вращающихся дисков с точки зрения характеристик доступа. Для них произвольный доступ к разделам — не проблема, потому что нет головки, которая двигалась бы по «блину», но некоторые характеристики все же меняют работу SSD.

Один из наиболее существенных факторов, влияющих на производительность SSD, — это *выравнивание разделов*. Считывая данные с SSD, вы читаете их частями (они называются *страницами*, не путайте со страницами виртуальной памяти), например, 4096 байт или 8192 байта за раз, и чтение должно начинаться с части, кратной этому значению. Это означает, что если ваш раздел и его данные не лежат на границе, вам придется выполнять два считывания вместо одного для небольших распространенных операций, например чтения содержимого каталога.

Новые версии утилит разбиения включают логику, позволяющую помещать вновь созданные разделы под соответствующие порядковые номера от начала дисков, поэтому вам не нужно беспокоиться о выравнивании разделов. Инструменты разбиения в настоящее время не производят никаких вычислений — они просто выравнивают разделы по границам 1 Мбайт, или, точнее, 2 048 512-байтовых блоков. Это довольно консервативно, потому что граница совпадает с размерами страниц 4096, 8192 и т. д. вплоть до 1 048 576. Однако, если вам интересно или вы хотите убедиться, что разделы начинаются на границе, вы легко найдете эту информацию в каталоге `/sys/block`. Вот пример кода для раздела `/dev/sdf2`:

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

Вывод в примере — это смещение раздела от начала устройства в единицах 512 байт (система Linux также называет их *секторами*). Если этот твердотельный накопитель использует 4096-байтовые страницы, на них находятся восемь таких секторов. Вам нужно лишь посмотреть, сможете ли вы равномерно разделить порядковые номера раздела на 8. В приведенном примере этого сделать нельзя, а значит, раздел не достигнет оптимальной производительности.

4.2. Файловые системы

Последним связующим звеном между ядром и пользовательским пространством для дисков обычно является *файловая система* — то, с чем вы взаимодействуете при выполнении таких команд, как `ls` и `cd`. Как упоминалось ранее, файловая система — это форма базы данных, она обеспечивает структуру для преобразования простого блочного устройства в сложную иерархию файлов и подкаталогов, понятную пользователям.

В свое время все файловые системы располагались на дисках и других физических носителях, предназначенных исключительно для хранения данных. Однако древовидная структура каталогов и интерфейс ввода-вывода файловых систем довольно универсальны, поэтому файловые системы теперь выполняют множество задач, к примеру, роль системных интерфейсов, которые отображаются в каталогах в `/sys` и `/proc`. Файловые системы традиционно реализуются в ядре, но инновационный протокол 9P из системы Plan9 (en.wikipedia.org/wiki/9P_(protocol)) вдохновил разработчиков на создание файловых систем пользовательского пространства. Функция *файловой системы в пользовательском пространстве (File System in User Space, FUSE)* позволяет создавать файловые системы в пользовательском пространстве в Linux.

Уровень абстракции *виртуальной файловой системы (Virtual File System, VFS)* завершает реализацию файловой системы. Подобно тому как подсистема SCSI стандартизирует связь между различными типами устройств и командами управления ядром, VFS гарантирует, что все реализации файловой системы поддерживают стандартный интерфейс, чтобы приложения пользовательского пространства могли одинаково обращаться к файлам и каталогам. Поддержка VFS позволила Linux поддерживать чрезвычайно большое количество файловых систем.

4.2.1. Типы файловых систем

Поддержка файловой системы Linux включает в себя собственные проекты, оптимизированные для Linux, а также внешние типы файловых систем, например Windows FAT, универсальные файловые системы, такие как ISO 9660, и многие другие. Далее перечислены наиболее распространенные типы файловых систем для хранения данных. Имена типов, распознанные Linux, заключены в круглые скобки рядом с именами файловых систем.

ЭВОЛЮЦИЯ ФАЙЛОВОЙ СИСТЕМЫ LINUX

Большинство пользователей давно применяют серию расширенных файловых систем (Extended filesystem), и тот факт, что она так долго оставалась стандартом, свидетельствует не только о ее эффективности, но и об адаптивности. Сообщество разработчиков Linux имеет тенденцию полностью заменять компоненты, которые не отвечают сиюминутным потребностям, но каждый раз, когда расширенной файловой системе чего-то не хватает, кто-то ее обновляет. Тем не менее в технологии файловых систем есть много того, что даже ext4 не может использовать из-за требований обратной совместимости. Это связано в первую очередь с улучшением масштабируемости, относящимся к очень большому количеству файлов, большим файлам и аналогичным сценариям.

На момент написания этого текста система Btrfs является стандартом по умолчанию для одного основного дистрибутива Linux. Если этот выбор окажется успешным, вполне вероятно, что Btrfs заменит серию расширенных файловых систем.

- *Четвертая расширенная файловая система* (Fourth Extended filesystem, ext4) — это текущая итерация линейки файловых систем, родных для Linux. *Вторая расширенная файловая система* (ext2) долго использовалась по умолчанию для систем Linux, вдохновленных традиционными файловыми системами Unix, такими как файловая система Unix (Unix File System,UFS) и быстрая файловая система (Fast File System, FFS). *Третья расширенная файловая система* (ext3) добавила функцию журналирования (небольшой кэш вне обычной структуры данных файловой системы) для повышения целостности данных и ускорения загрузки. Файловая система ext4 является дальнейшим улучшением и поддерживает файлы большего размера, чем ext2 или ext3, а также большее количество подкаталогов.
- В расширенной серии файловых систем существует определенная обратная совместимость. Например, вы можете монтировать файловые системы ext2 и ext3 взаимозаменяемо и монтировать файловые системы ext2 и ext3 как ext4, но *не можете* монтировать ext4 как ext2 или ext3.
- *Файловая система Btrfs* (B-tree filesystem) — это новейшая файловая система, родная для Linux и расширяющая возможности файловой системы ext4.
- *Файловые системы FAT* (msdos, vfat, exfat) относятся к системам Microsoft. Простой тип msdos поддерживает очень примитивное однообразное множество систем MS-DOS. Большинство съемных флеш-носителей, таких как SD-карты и USB-накопители, по умолчанию содержат разделы vfat (до 4 Гбайт) или exfat (4 Гбайт и более). Системы Windows могут использовать либо файловую систему на основе FAT, либо более продвинутую файловую систему NT (ntfs).

- *XFS* — это высокопроизводительная файловая система, применяемая по умолчанию некоторыми дистрибутивами, такими как Red Hat Enterprise Linux 7.0 и более высоких версий.
- *HFS+* (hfsplus) — это стандарт файловой системы Apple, используемый в большинстве систем Macintosh.
- *Файловая система ISO9660* (iso9660) — это стандарт CD-ROM. Большинство CD-приводов применяют разновидности стандарта ISO 9660.

4.2.2. Создание файловой системы

После завершения разбиения на разделы, описанного в разделе 4.1, можно приступить к созданию файловой системы. Как и разбиение дисков, файловая система создается в пользовательском пространстве, потому что его процесс может напрямую обращаться к блочному устройству и управлять им.

ЧТО ТАКОЕ MKFS

Это всего лишь внешний интерфейс для ряда программ создания файловых систем, `mkfs.fs`, где `fs` — тип файловой системы. Поэтому, когда вы запускаете команду `mkfs -t ext4`, `mkfs` в свою очередь запускает `mkfs.ext4`. Но здесь еще больше неочевидного. Проверьте файлы `mkfs.*`, стоящие после команд, 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
```

Как видно из примера, `mkfs.ext4` — это просто символическая ссылка на `mke2fs`. Это важно помнить, если вы столкнулись с системой без определенной команды `mkfs` или просматриваете документацию конкретной файловой системы. Утилиты создания каждой файловой системы имеют собственную страницу руководства, например `mke2fs(8)`. В большинстве систем это не проблема, потому что доступ к странице руководства `mkfs.ext4(8)` должен перенаправить вас на страницу руководства `mke2fs(8)`, просто имейте это в виду.

Утилита `mkfs` может создавать множество типов файловых систем. Например, вы можете создать раздел ext4 в `/dev/sda2` с помощью команды

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

Программа `mkfs` автоматически определяет количество блоков в устройстве и устанавливает подходящие значения по умолчанию. Если вы не представляете, что делаете, и не хотите подробно ознакомиться с документацией, не меняйте их.

При создании файловой системы `mkfs` по ходу работы осуществляет диагностический вывод, включая относящийся к суперблоку. *Суперблок* — ключевой компонент верхнего уровня базы данных файловой системы, и он настолько важен, что `mkfs` создает несколько резервных копий на случай проблем с оригиналом. Запишите несколько номеров резервных копий суперблоков при запуске `mkfs` на случай, если вам потребуется восстановить суперблок после сбоя диска (см. подраздел 4.2.11).

ВНИМАНИЕ

Создание файловой системы — это задача, которую следует выполнять только после добавления нового диска или повторного разбиения старого. Вы должны создать файловую систему только один раз для каждого нового раздела, в котором нет ранее существовавших данных (или данные есть, но их можно удалить). Создание новой файловой системы поверх существующей уничтожает старые данные.

4.2.3. Монтирование файловой системы

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

Чтобы примонтировать файловую систему, необходимо знать:

- устройство, местоположение или идентификатор файловой системы (например, раздел диска, в котором хранятся ее фактические данные). Некоторые файловые системы специального назначения, такие как proc и sysfs, не имеют местоположения;
- тип файловой системы;
- *точку монтирования* (mountpoint) — место в иерархии каталогов текущей системы, к которому будет присоединена файловая система. Точка монтирования всегда является обычным каталогом. Например, вы можете использовать каталог `/music` в качестве точки монтирования для файловой системы, содержащей музыку. Точка монтирования не обязательно должна находиться непосредственно под корневым каталогом `/` — она может располагаться в любом месте системы.

Общепринято, что для монтирования файловой системы необходимо смонтировать устройство в точке монтирования. Чтобы выяснить текущее состояние файловой

системы, запустите команду `mount`. Вывод (который может быть довольно длинным) должен выглядеть следующим образом:

```
$ 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)
fusectl on /sys/fs/fuse/connections type fusectl (rw)
debugfs on /sys/kernel/debug type debugfs (rw)
securityfs 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)
--пропуск--
```

Каждая строка соответствует одной смонтированной в данный момент файловой системе, элементы которой расположены в таком порядке.

1. Устройство, например `/dev/sda3`. Обратите внимание на то, что некоторые из устройств не реальные (например, proc), но зарезервированы для реальных имен устройств, поскольку эти файловые системы специального назначения не нуждаются в устройствах.
2. Слово `on`.
3. Точка монтирования.
4. Слово `type`.
5. Тип файловой системы, обычно в виде короткого идентификатора.
6. Параметры монтирования (в скобках). Более подробную информацию см. в подразделе 4.2.6.

Чтобы смонтировать файловую систему вручную, используйте команду `mount` следующим образом с указанием типа файловой системы, устройства и желаемой точки монтирования:

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

Например, чтобы смонтировать четвертую расширенную файловую систему, найденную на устройстве `/dev/sdf2` в `/home/extra`, применяйте команду

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

Обычно не нужно указывать параметр *типа* `-t`, потому что команда `mount` применяет его сама. Однако иногда необходимо различать два похожих типа систем, например различные файловые системы в стиле FAT.

Чтобы демонтировать (отсоединить) файловую систему, используйте команду `umount` следующим образом:

```
# umount mountpoint
```

Вы также можете демонтировать файловую систему через ее устройство вместо точки монтирования.

ПРИМЕЧАНИЕ

Почти все системы Linux включают временную точку монтирования /mnt, которая обычно используется для тестирования. Задействуйте ее при экспериментах с системой, но если собираетесь монтировать файловую систему для расширенного применения, найдите или создайте для нее другое местоположение.

4.2.4. Идентификатор UUID файловой системы

Способ монтирования файловых систем, рассмотренный в предыдущем разделе, зависит от имен устройств. Однако имена устройств могут изменяться, поскольку они зависят от порядка, в котором ядро находит устройства. Чтобы решить эту проблему, можете идентифицировать и монтировать файловые системы по их *универсально-уникальному идентификатору UUID (Universally Unique Identifier)* — стандарту уникальных «серийных номеров» идентификации объектов в компьютерной системе. Программы создания файловой системы, такие как mke2fs, генерируют UUID при инициализации структуры данных файловой системы.

Для просмотра списка устройств и соответствующих файловых систем и UUID в своей системе используйте программу `blkid` (идентификатор блока, Block Id):

```
# blkid
/dev/sdf2: UUID="b600fe63-d2e9-461c-a5cd-d3b373a5e1d2" TYPE="ext4"
/dev/sda1: UUID="17f12d53-c3d7-4ab3-943e-a0a72366c9fa" TYPE="ext4"
PARTUUID="c9a5ebb0-01"
/dev/sda5: UUID="b600fe63-d2e9-461c-a5cd-d3b373a5e1d2" TYPE="swap"
PARTUUID="c9a5ebb0-05"
/dev/sde1: UUID="4859-EFEA" TYPE="vfat"
```

В этом примере программа `blkid` обнаружила четыре раздела с данными: два с файловыми системами ext4, один с сигнатурой пространства подкачки (см. раздел 4.3) и один с файловой системой на основе FAT. Все собственные разделы Linux имеют стандартные UUID, лишь у раздела FAT его нет. Вы можете ссылаться на раздел FAT через его серийный номер тома FAT (в данном случае 4859-EFEA).

Чтобы смонтировать файловую систему по ее идентификатору UUID, используйте параметр монтирования UUID. Например, чтобы смонтировать первую файловую систему из предыдущего списка в `/home/extra`, введите:

```
# mount UUID=b600fe63-d2e9-461c-a5cd-d3b373a5e1d2 /home/extra
```

Обычно файловые системы не монтируются вручную по UUID подобным образом, потому что вы знаете устройство, а монтировать его по имени гораздо проще, чем по непонятному UUID. Тем не менее важно понимать, что это такое. В первую очередь идентификатор UUID является предпочтительным способом автоматического

монтирования файловых систем, отличных от LVM, в `/etc/fstab` во время загрузки (см. подраздел 4.2.8). Кроме того, многие дистрибутивы используют UUID в качестве точки монтирования при вставке съемных носителей. В предыдущем примере файловая система FAT находится на флеш-карте. Система Ubuntu смонтирует этот раздел в `/media/user/4859-EFEA` после вставки носителя. Демон `udevd`, описанный в главе 3, обрабатывает начальное событие для вставки устройства.

При необходимости вы можете изменить UUID файловой системы (например, если скопировали полную файловую систему откуда-то еще и теперь вам нужно отличить ее от оригинала). См. страницу руководства `tune2fs(8)`, чтобы узнать, как это сделать в файловой системе ext2/ext3/ext4.

4.2.5. Буферизация диска, кэширование и файловые системы

Система Linux, как и другие варианты Unix, буферизует запись на диск. Это означает, что ядро обычно не сразу записывает изменения в файловые системы, когда процессы запрашивают изменения. Вместо этого она сохраняет эти изменения в оперативной памяти до тех пор, пока ядро не определит подходящее время для их фактической записи на диск. Эта система буферизации невидима для пользователя и обеспечивает значительный прирост производительности.

Когда вы размонтируете файловую систему с помощью команды `umount`, ядро автоматически *синхронизируется* с диском, записывая изменения из своего буфера на диск. Вы можете заставить ядро сделать это в любое время, выполнив команду `sync`, которая по умолчанию синхронизирует все диски в системе. Если по какой-то причине вы не можете размонтировать файловую систему перед выключением системы, обязательно сначала запустите команду `sync`.

Кроме того, ядро использует оперативную память для кэширования блоков по мере их чтения с диска. Поэтому, если один или несколько процессов повторно обращаются к файлу, ядру не нужно снова и снова обращаться к диску, оно может просто считывать данные из кэша и экономить время и ресурсы.

4.2.6. Параметры монтирования файловой системы

Существует множество способов изменить поведение команды монтирования, что часто требуется в ходе работы со съемными носителями или при обслуживании системы. На самом деле общее количество вариантов монтирования ошеломляет. У команды `mount(8)` есть обширная страница руководства, но даже прочитав ее, трудно понять, с чего начать и что можно без опаски игнорировать. Рассмотрим наиболее полезные параметры в этом разделе.

Параметры делятся на две основные категории: общие и подходящие для конкретной файловой системы. Общие параметры обычно работают во всех типах файловых систем и включают параметр `-t` для указания типа файловой системы, как было

показано ранее. Напротив, параметр, специфичный для файловой системы, относится только к определенным типам файловых систем.

Чтобы активировать параметр для файловой системы, используйте переключатель `-o`, за которым следует параметр. Например, команда `-o remount,rw` перемонтирует файловую систему, уже смонтированную как доступную только для чтения, в режим чтения и записи.

Основные короткие параметры

Общие параметры имеют короткий синтаксис. Наиболее важными из них являются следующие.

- `-r` — монтирует файловую систему в режиме только для чтения. Это полезно для ряда задач, от защиты от записи до начальной загрузки. Вам не нужно указывать его при доступе к устройству, которое уже доступно только для чтения, например к CD-ROM, система сделает это за вас (а также сообщит о состоянии только для чтения).
- `-n` — гарантирует, что команда `mount` не попытается обновить исполняемую системную базу данных монтирования `/etc/mtab`. По умолчанию операция `mount` завершается неудачно, если не может выполнить запись в этот файл, поэтому данный параметр важен во время загрузки, поскольку корневой раздел (включая системную базу данных монтирования) сначала доступен только для чтения. Этот параметр задействуется также для устранения системной проблемы в однопользовательском режиме, поскольку системная база данных монтирования может быть недоступна в текущий момент.
- `-t`. Параметр `-t` *`type`* указывает на тип файловой системы.

Длинные параметры

Рамки коротких вариантов параметров, таких как `-r`, слишком узки для постоянно растущего числа вариантов монтирования — в алфавите недостаточно букв, чтобы охватить все возможные варианты. К тому же короткие варианты вызывают проблемы, потому что трудно определить значение варианта на основе одной буквы. Многие общие параметры и все параметры, относящиеся к файловой системе, используют более длинный и гибкий формат.

Чтобы задействовать длинные параметры с командой `mount` в командной строке, начните с параметра `-o`, после которого стоят соответствующие ключевые слова, разделенные запятыми. Вот полный пример с длинными параметрами, следующими за параметром `-o`:

```
# mount -t vfat /dev/sde1 /dos -o ro,uid=1000
```

Два длинных параметра в примере — это `ro` и `uid=1000`. Параметр `ro` указывает режим только для чтения и совпадает с коротким параметром `-r`. Параметр `uid=1000`

приказывает ядру обрабатывать все файлы в файловой системе так, как если бы владельцем был пользователь ID 1000.

Наиболее полезные длинные параметры:

- `exec`, `noexec` — включает или отключает выполнение программ в файловой системе;
- `suid`, `nosuid` — включает или отключает программы `setuid`;
- `ro` — монтирует файловую систему в режиме только для чтения (read-only mode) (как и короткий параметр `-r`);
- `rw` — монтирует файловую систему в режиме чтения и записи (read-write mode).

ПРИМЕЧАНИЕ

Разница между текстовыми файлами Unix и DOS состоит главным образом в том, как заканчиваются строки. В Unix только символ новой строки (\n, ASCII 0x0A) отмечает конец строки, а DOS использует возврат каретки (\r, ASCII 0x0D), за которым следует символ новой строки. Было много попыток автоматического преобразования на уровне файловой системы, но они всегда вызывают проблемы. Текстовые редакторы, такие как vim, могут автоматически определять стиль новой строки файла и поддерживать его соответствующим образом. Таким образом легче сохранить единообразие стилей.

4.2.7. Повторное монтирование файловой системы

Бывают случаи, когда нужно изменить параметры монтирования для уже смонтированной файловой системы. Наиболее распространенная ситуация — вам нужно сделать файловую систему, доступную только для чтения, доступной для записи во время аварийного восстановления. В этом случае требуется повторно подключить файловую систему в той же точке монтирования.

Следующая команда заново монтирует корневой каталог в режиме чтения и записи (вам нужен параметр `-n`, потому что команда `mount` не может записывать в системную базу данных монтирования, когда корневой каталог доступен только для чтения):

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

Эта команда предполагает, что правильный список устройств для `/` находится в каталоге `/etc/fstab` (как описано в следующем разделе). Если это не так, вы должны указать устройство в качестве дополнительного параметра.

4.2.8. Таблица файловой системы /etc/fstab

Чтобы смонтировать файловые системы во время загрузки и избавиться от необходимости использовать команду `mount`, системы Linux хранят постоянный список

файловых систем и параметров в файле `/etc/fstab`. Это текстовый файл в очень простом формате, как показано в листинге 4.1.

Листинг 4.1. Список файловых систем и параметров в файле `/etc/fstab`

```
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 соответствует одной файловой системе и разбита на шесть полей. Значения этих полей (слева направо):

- **Устройство или идентификатор UUID.** Большинство современных систем Linux больше не используют устройство в файле `/etc/fstab`, предпочитая UUID.
- **Точка монтирования (mount point).** Указывает, куда следует присоединить файловую систему.
- **Тип файловой системы.** Вам может быть незнаком параметр `swap` в этом списке, это раздел подкачки (см. раздел 4.3).
- **Параметры.** Длинные параметры, разделенные запятыми.
- **Информация о резервном копировании для команды dump.** Команда `dump` — давно устаревшая утилита резервного копирования, это поле больше не актуально. Вы всегда должны устанавливать его равным `0`.
- **Порядок проверки целостности файловой системы.** Чтобы убедиться, что программа `fsck` всегда запускается сначала в корневом каталоге, устанавливайте значение `1` для корневой файловой системы и `2` — для любых других локально подключенных файловых систем на жестком диске или SSD. Задавайте значение `0`, чтобы отключить проверку загрузки для всех других файловых систем, включая устройства только для чтения, подкачку swap и файловую систему `/proc` (см. информацию о команде `fsck` в подразделе 4.2.11).

При использовании команды `mount` вы можете добавлять ярлыки, если файловая система, с которой хотите работать, находится в файле `/etc/fstab`. Например, ориентируясь на листинг 4.1, смонтируйте CD, просто запустив команду `mount /cd rom`.

Можете попытаться одновременно смонтировать все записи в файле `/etc/fstab`, которые не содержат параметр `noauto`, с помощью команды

```
# mount -a
```

В листинге 4.1 представлены некоторые новые параметры, а именно `errors`, `noauto` и `user`, поскольку они не применяются за пределами файла `/etc/fstab`. Кроме того, здесь чаще всего стоит параметр `defaults`. Вот что означают эти параметры:

- `defaults`. Устанавливает параметры `mount` по умолчанию: режим чтения и записи, включение файлов устройств, исполняемых файлов, бит `setuid` и т. д.

Используйте его, если не хотите предоставлять файловой системе какие-то специальные параметры, но нужно заполнить все поля в файле `/etc/fstab`;

- `errors`. Параметр, относящийся к файловым системам ext2/ext3/ext4. Он задает поведение ядра, когда в системе возникают проблемы с монтированием файловой системы. Значение по умолчанию обычно равно `errors=continue`, что означает: ядро должно возвращать код ошибки и продолжать работать. Чтобы ядро снова попыталось выполнить монтирование в режиме только для чтения, используйте параметр `errors=remount-ro`. Параметр `errors=panic` указывает ядру (и системе) остановиться, когда возникает проблема с монтированием;
- `noauto`. Указывает команде `mount -a` игнорировать запись. Применяйте его для предотвращения монтирования во время загрузки устройства со съемным носителем, например устройства флеш-памяти;
- `user`. Позволяет непривилегированным пользователям запускать команду `mount` на определенной строке, что может быть удобно для предоставления определенных видов доступа к съемным носителям. Поскольку пользователи могут поместить файл `setuid-root` на съемный носитель с другой системой, этот параметр также устанавливает значения для `nosuid`, `noexec` и `nodev` (для запрета специальных файлов устройств). Имейте в виду, что для съемных носителей и других общих случаев этот параметр теперь используется ограниченно, поскольку большинство систем применяют `ubus` наряду с другими механизмами для автоматического монтирования вставленных носителей. Однако этот параметр может быть полезен в особых случаях, когда вы хотите предоставить контроль над монтированием определенных каталогов.

4.2.9. Альтернативы файлу /etc/fstab

Хотя файл `/etc/fstab` был традиционным способом представления файловых систем и их точек монтирования, существуют две альтернативы. Первая — это каталог `/etc/fstab.d`, который содержит отдельные файлы конфигурации файловой системы (по одному для каждой файловой системы). По сути он очень похож на многие другие каталоги конфигурации, которые встречаются в этой книге.

Второй альтернативой является настройка модулей `systemd` units для файловых систем. Вы узнаете больше о системе `systemd` и ее блоках в главе 6. Однако конфигурация блока `systemd` часто генерируется из файла `/etc/fstab` (или на его основе), поэтому можно обнаружить совпадения в системах.

4.2.10. Емкость файловой системы

Чтобы просмотреть размер и способы применения смонтированных в данный момент файловых систем, используйте команду `df`. Вывод может быть очень обширным (и он становится все длиннее благодаря специализированным файловым системам), но он должен включать информацию о ваших реальных устройствах хранения:

```
$ df
Filesystem          1K-blocks      Used Available Use% Mounted on
/dev/sda1           214234312 127989560  75339204  63% /
/dev/sdd2             3043836      4632   2864872   1% /media/user/uuid
```

Краткое описание полей в выводе команды `df`:

- `Filesystem` (файловая система). Устройство, на котором расположена файловая система.
- `1K-blocks` (1К-блоки). Общая емкость файловой системы в блоках по 1024 байта.
- `Used` (занято). Количество занятых блоков.
- `Available` (свободно). Количество свободных блоков.
- `Use%` (используется). Процент используемых блоков.
- `Mounted on` (смонтировано). Точка монтирования.

ПРИМЕЧАНИЕ

Если у вас возникли проблемы с поиском правильной строки в выводе df, соответствующей определенному каталогу, выполните команду df dir, где dir — это каталог, который вы хотите найти. Команда ограничит вывод файловой системы для этого каталога. Очень часто применяется параметр df, который ограничивает вывод устройством, содержащим ваш текущий каталог.

Из примера видно, что две файловые системы имеют размер примерно 215 Гбайт и 3 Гбайт. Однако значения емкости могут показаться немного странными, потому что 127 989 560 плюс 75 339 204 не равно 214 234 312, а 127 989 560 — это не 63 % от 214 234 312. В обоих случаях не учтены 5 % от общей емкости. На самом деле пространство есть, но оно скрыто в зарезервированных (reserved) блоках. Только суперпользователь может задействовать зарезервированные блоки файловой системы, когда она начинает заполняться. Эта функция предохраняет системные серверы от немедленного сбоя, когда у них заканчивается дисковое пространство.

ПРИМЕЧАНИЕ

Стандарт POSIX (Portable Operating System Interface for Unix, переносимый интерфейс операционных систем Unix) определяет размер блока в 512 байт. Однако этот размер труднее читать, поэтому по умолчанию вывод df и du в большинстве дистрибутивов Linux выполняется в блоках размером 1024 байта. Если вы хотите отобразить числа в 512-байтовых блоках, установите переменную среды POSIXLY_CORRECT. Чтобы явно указать блоки размером 1024 байта, примените параметр -k (обе команды его поддерживают). Программы df и du также имеют параметр -m для перечисления емкости в блоках размером 1 Мбайт и параметр -h, который составляет вывод так, чтобы его было легче всего читать пользователю, основываясь на общих размерах файловых систем.

ВЫВОД ВСЕХ ИСПОЛЬЗУЮЩИХСЯ КАТАЛОГОВ

Если ваш диск заполняется и вам нужно знать, где находятся все занимающие место медиафайлы, примените команду `du`. Без аргументов она выводит использование диска для каждого каталога в иерархии каталогов, начиная с текущего рабочего каталога. (Это может быть длинный список, для примера просто запустите команду `cd/; du`, чтобы понять, о чем речь. Нажмите сочетание клавиш Ctrl+C, когда вам это наскучит.) Команда `du -s` включает режим общего подсчета для вывода только итоговой суммы. Чтобы оценить все файлы и подкаталоги в определенном каталоге, перейдите в него и запустите команду `du -s *`, но имейте в виду, что могут существовать каталоги с точкой, которые эта команда не увидит.

4.2.11. Проверка и восстановление файловых систем

Оптимизация, которую предлагают файловые системы Unix, становится возможной благодаря сложному механизму базы данных. Для бесперебойной работы файловых систем ядро должно быть уверено в том, что смонтированная файловая система не содержит ошибок, а оборудование надежно хранит данные. Если ошибки существуют, это может привести к потере данных и сбоям системы.

Помимо физических проблем, ошибки файловой системы обычно возникают из-за того, что пользователь неправильно выключает систему (например, выдергивает шнур питания). В таких случаях предыдущий кэш файловой системы в памяти может не совпадать с данными на диске, а система в этот момент может находиться в процессе изменения файловой системы. Хотя многие файловые системы поддерживают журналирование, но чтобы сделать их повреждения менее частыми, вы всегда должны правильно завершать работу системы. Любую файловую систему необходимо время от времени проверять на наличие ошибок и проблем.

Инструментом для проверки файловой системы является утилита `fsck`. Как и в случае с программой `mkfs`, для каждого типа файловой системы, поддерживаемого Linux, существует своя версия `fsck`. Например, при запуске в расширенной серии файловых систем (ext2/ext3/ext4) `fsck` распознает тип файловой системы и запускает утилиту `e2fsck`. Поэтому, как правило, вам не нужно вводить команду `e2fsck`, за исключением случаев, когда `fsck` не может определить тип файловой системы или вы ищете страницу руководства `e2fsck`.

Информация, представленная в этом разделе, относится к серии расширенных файловых систем и `e2fsck`.

Чтобы запустить команду `fsck` в интерактивном ручном режиме, укажите в качестве аргумента устройство или точку монтирования (как указано в файле `/etc/fstab`), например:

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

ВНИМАНИЕ

Никогда не используйте fsck для смонтированной файловой системы, так как ядро может изменить данные на диске при выполнении проверки, что приведет к несоответствиям во время выполнения, которые в свою очередь могут вызвать сбой системы и повреждение файлов. Существует лишь одно исключение: монтируя корневой раздел только для чтения в однопользовательском режиме, можете задействовать на нем команду fsck.

В ручном режиме команда `fsck` выводит подробные отчеты о состоянии на своем пути, которые должны выглядеть примерно так, если нет проблем:

```
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 команда `fsck` обнаруживает проблему в ручном режиме, она останавливается и задает вопрос о ее устранении. Эти вопросы касаются внутренней структуры файловой системы, такой как повторное подключение незакрепленных дескрипторов *inode* и очистка блоков (дескрипторы *inodes* являются строительными блоками файловой системы, вы увидите, как они работают, в разделе 4.6). Если `fsck` спрашивает вас о повторном подключении дескриптора inode, значит, она обнаружила файл, у которого, по-видимому, нет имени. При повторном подключении такого файла `fsck` помещает файл в каталог `lost+found` (потеряно + найдено) в файловой системе с номером в качестве имени файла. Если это произойдет, вам нужно угадать имя на основе содержимого файла, а исходное имя файла, скорее всего, было удалено.

В общем, бессмысленно сразу подключать процесс восстановления `fsck`, если вы только что неправильно вышли из системы, потому что `fsck` может найти множество мелких ошибок, которые нужно исправить. К счастью, у команды `e2fsck` есть параметр `-p`, который автоматически исправляет обычные проблемы без запроса и прерывает работу, когда возникает серьезная ошибка. Фактически дистрибутивы Linux запускают вариант `fsck -p` во время загрузки. (Вы также можете встретить команду `fsck -a`, которая делает то же самое.)

Если вы подозреваете, что в системе возникла серьезная ошибка, например произошел сбой оборудования или неправильно сконфигурировано устройство, вам необходимо решить, как действовать, потому что команда `fsck` действительно может повредить файловую систему, у которой и так уже большие проблемы. (Одним из явных признаков проблем у системы является то, что `fsck` задает *много* вопросов в ручном режиме.)

Если вы думаете, что проблема действительно есть, попробуйте запустить команду `fsck -n`, чтобы проверить файловую систему, ничего в ней не изменяя. Если возникла

проблема с конфигурацией устройства, которую, по вашему мнению, можно исправить (например, незакрепленные кабели или неправильное количество блоков в таблице разделов), сделайте это перед основным запуском команды `fsck`, иначе вы, скорее всего, потеряете много данных.

Если вы подозреваете, что поврежден только суперблок (например, потому что кто-то добавил запись в начало раздела диска), можете восстановить файловую систему с помощью одной из резервных копий суперблока, созданных `mkfs`. Используйте команду `fsck -b num` для замены поврежденного суперблока на альтернативный блок `num` и надейтесь на лучшее.

Если вы не знаете, где найти резервный суперблок, запустите команду `mkfs -n` на устройстве, чтобы просмотреть список номеров резервных копий суперблоков, не повреждая свои данные. (Опять же убедитесь, что вы применяете параметр `-n`, иначе *действительно* уничтожите файловую систему.)

Проверка файловых систем ext3 и ext4

Обычно вам не нужно проверять файловые системы ext3 и ext4 вручную, поскольку журнал обеспечивает целостность данных (напомню, что *журнал (journal)* представляет собой небольшой кэш данных, который еще не был записан в определенную область файловой системы). Если вы не выключили свою систему аккуратно, журнал будет содержать часть данных. Чтобы сбросить журнал в файловой системе ext3 или ext4 в обычную базу данных файловой системы, запустите команду `e2fsck` следующим образом:

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

Однако вы можете смонтировать поврежденную файловую систему ext3 или ext4 в режиме ext2, поскольку ядро не будет монтировать эти файловые системы с непустым журналом.

Наихудший случай

Если с диском случилось что-то серьезное, у вас не так уж много способов решить эту проблему.

- Вы можете попытаться извлечь весь образ файловой системы с диска с помощью команды `dd` и перенести его в раздел на другом диске того же размера.
- Можете смонтировать файловую систему в режиме только для чтения и спасти все, что успели.
- Можете использовать команду `debugfs`.

В первых двух случаях все равно нужно восстановить файловую систему перед ее монтированием, если вы не хотите вручную просматривать необработанные

данные. А если хотите просмотреть их вручную, то можете ввести команду `fsck -y` и ответить на все ее вопросы. Но делайте это в крайнем случае, потому что в процессе восстановления могут возникнуть проблемы, которые лучше решить вручную.

Инструмент `debugfs` позволяет просматривать файлы в файловой системе и копировать их в другое место. По умолчанию он открывает файловые системы в режиме только для чтения. При восстановлении данных сохранить ваши файлы в целости и сохранности, чтобы не испортить все еще больше, — это хорошая идея.

Если проблема приобретает масштаб катастрофы, нет резервных копий и вы уже отчаялись, то необходимо обращаться к профессиональным службам и надеяться, что они смогут «соскрести» данные с диска.

4.2.12. Файловые системы специального назначения

Не все файловые системы представляют собой хранилище на физических носителях. У большинства версий систем Unix есть файловые системы, которые служат системными интерфейсами. То есть вместо того, чтобы быть просто средством хранения данных на устройстве, файловая система может представлять системную информацию, такую как идентификаторы процессов и диагностические сообщения ядра. Идея таких файловых систем восходит к механизму `/dev`, который является ранней моделью использования файлов для интерфейсов ввода-вывода. Идея `/proc` возникла в восьмой версии Unix, реализованной Томом Дж. Киллианом, а ее производство ускорилось, когда компания Bell Labs (включая многих из тех, кто изначально разрабатывал Unix) создала Plan 9 — исследовательскую операционную систему, которая вывела абстракцию файловой системы на совершенно новый уровень (en.wikipedia.org/wiki/Plan_9_from_Bell_Labs).

Перечислим некоторые из специальных файловых систем, широко используемых в Linux.

- `proc`. Монтируется в каталоге `/proc`. Название — сокращенная версия от слова *process (процесс)*. Каждый *пронумерованный* каталог внутри `/proc` ссылается на идентификатор текущего процесса в системе, файлы в каждом каталоге представляют различные аспекты этого процесса. Каталог `/proc/self` представляет текущий процесс. Файловая система `proc` в Linux содержит большое количество дополнительной информации о ядре и оборудовании в таких файлах, как `/proc/cpuinfo`. Имейте в виду, что в руководстве по созданию ядра рекомендуется перемещать информацию, не связанную с процессами, из `/proc` в `/sys`, поэтому системная информация в `/proc` может представлять не самый актуальный интерфейс.
- `sysfs`. Монтируется в каталоге `/sys` (о нем рассказывалось в главе 3).
- `tmpfs`. Монтируется в каталоге `/run` и в других местах. С помощью `tmpfs` вы можете использовать физическую память и пространство подкачки

swap в качестве временного хранилища. Также можете монтировать `tmpfs` там, где вам нужно, применяя параметры `size` и `nr_blocks` для управления максимальным размером. Тем не менее будьте осторожны, чтобы постоянно не добавлять данные в папку `tmpfs`, потому что в системе в конечном итоге закончится память и программы начнут сбоить.

- `squashfs`. Тип файловой системы, доступной только для чтения, в которой содержимое хранится в сжатом формате и извлекается по требованию с помощью устройства обратной связи. Одним из примеров использования является система управления пакетами snap, которая монтирует пакеты в каталоге `/snap`.
- `overlay`. Файловая система, которая объединяет каталоги в составную систему. Контейнеры часто задействуют файловые системы overlay (наслоения) (вы узнаете, как они работают, в главе 17).

4.3. Область подкачки swap

Не каждый раздел на диске содержит файловую систему. А еще можно увеличить объем оперативной памяти на машине за счет дискового пространства. Если у вас заканчивается реальная память, система виртуальной памяти Linux может автоматически перемещать фрагменты памяти в дисковое хранилище и обратно. Это называется *подкачкой*, потому что части бездействующих программ отправляются на диск в обмен на активные части, находящиеся на диске. Область диска, используемая для хранения страниц памяти, называется *областью подкачки* (или просто *подкачкой*, *swap*).

Выходные данные команды `free` подключают использование подкачки в килобайтах следующим образом:

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

4.3.1. Использование раздела диска в качестве области подкачки

Чтобы задействовать весь раздел диска в качестве области подкачки, выполните следующие действия.

1. Убедитесь, что раздел пуст.
2. Запустите команду `mkswap dev`, где `dev` — устройство раздела. Эта команда помещает *сигнатуру области подкачки* на раздел, помечая его (а не файловую систему или что-либо другое) как область подкачки.
3. Запустите команду `swapon dev`, чтобы зарегистрировать область в ядре.

После создания раздела подкачки вы можете поместить новую запись подкачки в файл `/etc/fstab`, чтобы система задействовала область подкачки сразу после загрузки компьютера. Вот пример записи, которая использует `/dev/sda5` в качестве раздела подкачки:

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

Сигнатуры области подкачки имеют идентификаторы UUID, поэтому учтите, что многие системы теперь применяют их вместо необработанных имен устройств.

4.3.2. Использование файла в качестве области подкачки

Вы можете использовать обычный файл в качестве области подкачки, если необходимо переделать разбиение диска, чтобы создать основной раздел подкачки, и при этом не должно быть никаких проблем.

Примените следующие команды, чтобы создать пустой файл, инициализировать его как область подкачки и добавить в пул подкачки:

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

В примере *`swap_file`* — это имя нового файла подкачки, а *`num_mb`* — его желаемый размер в мегабайтах.

Чтобы удалить раздел подкачки или файл из активного пула ядра, используйте команду `swapoff`. В вашей системе должно быть достаточно свободной памяти (реальной и подкачки вместе взятых) для размещения любых активных страниц в удаляемой части пула подкачки.

4.3.3. Определение необходимого размера области подкачки

Уже давно в системах Unix было общепринятым резервировать по крайней мере вдвое больший объем области подкачки, чем существует реальной оперативной памяти. Сегодня проблема заключается не только в огромных объемах доступных дисков и памяти, но и в способах использования системы. С одной стороны, дискового пространства так много, что возникает соблазн выделить более чем вдвое больший объем памяти. С другой — вы можете никогда и не задействовать область подкачки, потому что реальной памяти и так много.

Правило «удвоенного объема реальной памяти» было введено в то время, когда несколько пользователей работали на одном компьютере. Однако не все они были активны, поэтому было удобно иметь возможность задействовать память неактивных пользователей, если активному требовалось больше памяти.

То же самое работает и для компьютера с одним пользователем. Если вы запускаете множество процессов, можно переместить в область подкачки части неактивных процессов или даже неактивные части активных процессов. Однако если многие активные процессы хотят использовать память одновременно и вы часто обращаетесь к области подкачки, возникнут серьезные проблемы с производительностью, поскольку ввод-вывод диска (даже SSD) слишком медлителен и отстает от остальной системы. В таком случае можно либо докупить дополнительный объем памяти, либо завершить некоторые процессы, либо пожаловаться.

Ядро Linux может самостоятельно перевести процесс в область подкачки, чтобы обеспечить немного больший объем дискового кэша. Чтобы предотвратить такое поведение, администраторы настраивают определенные системы вообще без области подкачки. Например, высокопроизводительные серверы никогда не должны использовать пространство подкачки и должны избегать доступа к диску, если это вообще возможно.

ПРИМЕЧАНИЕ

Опасно не добавлять пространство подкачки на компьютере общего назначения. Если на компьютере полностью заканчивается как реальная память, так и область подкачки, ядро Linux вызывает процесс OOM Killer (Out-Of-Memory Killer), чтобы принудительно завершить процессы и освободить немного памяти. Очевидно, это не должно происходить с настольными приложениями. В то же время высокопроизводительные серверы включают в себя сложные системы мониторинга, резервирования и балансировки нагрузки, гарантирующие, что они никогда не достигнут опасной зоны.

Вы узнаете гораздо больше о том, как работает система памяти, прочитав главу 8.

4.4. Менеджер логических томов LVM

До сих пор мы рассматривали только прямое управление и использование дисков через разделы с точным указанием на устройствах хранения места, где должны находиться определенные данные. Теперь вы знаете, что доступ к блочному устройству, такому как `/dev/sda1`, приведет вас к месту на определенном устройстве в соответствии с таблицей разделов в `/dev/sda`, даже если точное местоположение может быть неизвестно.

Обычно все работает нормально, но и у этого способа управления есть недостатки, особенно когда дело доходит до внесения изменений в диски уже после установки. Например, если вы хотите обновить диск, необходимо установить новый диск, раздел, добавить файловые системы, возможно, внести некоторые изменения в загрузчик, выполнить другие задачи и, наконец, переключиться на новый диск. Этот процесс чреват ошибками и требует нескольких перезагрузок. Есть случай еще хуже: когда вы хотите установить дополнительный диск, чтобы увеличить емкость, вам нужно выбрать новую точку монтирования для файловой системы

на этом диске и надеяться, что сможете вручную распределить свои данные между старым и новым дисками.

Менеджер LVM решает эти проблемы, добавляя еще один уровень управления между устройствами физического блока и файловой системой. Суть этого приема заключается в том, что вы выбираете набор *физических томов* (Phisycal Volumes, PV, обычно просто блочных устройств, таких как разделы диска), чтобы добавить их в *группу томов (Volume Groups, VG)*, которая действует как своего рода общий пул данных. Затем выделяете из группы томов *логические тома (Logical Volumes, LV)*.

На рис. 4.4 проиллюстрирована описанная выше схема для одной группы томов. Здесь показаны несколько физических и логических томов, но многие системы на базе LVM имеют только один физический и только два логических тома (для корневого и подкачки).

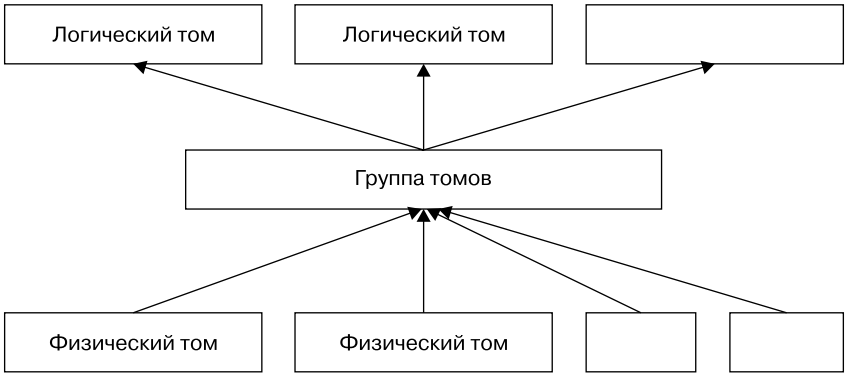


Рис. 4.4. Сочетание физических и логических томов в группе томов

Логические тома — это просто блочные устройства, они содержат файловые системы или сигнатуры подкачки, поэтому вы можете рассматривать взаимосвязь между группой томов и ее логическими томами как аналогичную взаимосвязь диска и его разделов. Критическое различие заключается в том, что не вы определяете самостоятельно, как логические тома расположены в группе томов, — это делает менеджер LVM.

LVM позволяет выполнять мощные и чрезвычайно полезные задачи:

- добавлять больше физических томов (например, другой диск) в группу томов, увеличивая ее размер;
- удалять физические тома, если для размещения существующих логических томов внутри группы томов достаточно места;
- изменять размер логических томов (и, как следствие, изменять размер файловых систем с помощью утилиты `fsadm`).

Вы можете сделать все это без перезагрузки компьютера и в большинстве случаев без размонтирования каких-либо файловых систем. Хотя для добавления нового оборудования на физических дисках может потребоваться завершить работу системы, облачные вычислительные среды часто позволяют добавлять новые блочные устройства хранения данных и без этого, что делает менеджер LVM отличным решением для систем, которым требуется такая гибкость и эффективность.

Мы рассмотрим менеджер LVM довольно подробно. Сначала изучим, как взаимодействовать с логическими томами и их компонентами и управлять ими, а затем более детально поговорим о том, как работают LVM и драйвер ядра, на котором он построен. Однако обсуждение менеджера LVM не слишком важно для понимания остальной части книги, поэтому можете сразу переходить к главе 5.

4.4.1. Работа с менеджером LVM

LVM имеет ряд инструментов пользовательского пространства для управления томами и группами томов. Большинство из них основаны на команде `lvm` — интерактивном инструменте общего назначения. Существуют отдельные команды (которые являются просто символическими ссылками на LVM) для выполнения определенных задач. Например, команда `vgs` имеет тот же эффект, что и ввод `vgs` в командной строке `lvm>` инструмента `lvm`, и `vgs` (обычно в `/sbin`) является символической ссылкой на `lvm`. В этой книге мы будем применять отдельные команды.

В следующих нескольких разделах рассмотрим компоненты системы, использующей логические тома. Первые примеры взяты из стандартного дистрибутива Ubuntu с применением параметров разделения LVM, поэтому многие имена будут содержать слово *Ubuntu*. Однако все технические детали относятся не только к этому конкретному дистрибутиву.

Понятие групп томов и их перечисление

Упомянутая ранее команда `vgs` отображает группы томов, настроенные в настоящее время в системе. Вывод команды довольно лаконичен. Вот что вы можете увидеть в примере работы LVM:

```
# vgs
  VG        #PV #LV #SN Attr   VSize   VFree
  ubuntu-vg  1   2   0 wz--n- <10.00g 36.00m
```

Первая строка — это заголовок, а каждая последующая строка представляет группу томов. Столбцы выглядят следующим образом:

- `VG`. Имя группы томов. `ubuntu-vg` — это общее имя, которое инсталлятор Ubuntu присваивает при настройке системы с помощью LVM.
- `#PV`. Количество физических томов, которые содержит хранилище группы томов.

- `#LV`. Количество логических томов внутри группы томов.
- `#SN`. Количество слепков (snapshots) логических томов (мы не будем вдаваться в подробности об этом столбце).
- `Attr`. Ряд атрибутов состояния группы томов. В примере есть атрибуты `w` (writable, запись), `z` (resizable, изменение размера) и `n` (normal allocation policy, обычная политика распределения).
- `VSize`. Размер группы томов.
- `VFree`. Объем нераспределенного пространства в группе томов.

Этого краткого набора параметров группы томов достаточно для большинства задач. Если вы хотите углубиться в группу томов, используйте команду `vgdisplay`, которая помогает разобраться в ее свойствах. Вот та же группа томов после применения команды `vgdisplay`:

```
# vgdisplay
  --- Volume group ---
  VG Name               ubuntu-vg
  System ID
  Format                lvm2
  Metadata Areas        1
  Metadata Sequence No  3
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                2
  Open LV               2
  Max PV                0
  Cur PV                1
  Act PV                1
  VG Size               <10.00 GiB
  PE Size               4.00 MiB
  Total PE              2559
  Alloc PE / Size       2550 / 9.96 GiB
  Free PE / Size        9 / 36.00 MiB
  VG UUID               0zs0TV-wnT5-laOy-vJ0h-rUae-YPdv-pPwaAs
```

Вывод похож на предыдущий, однако появились и новые значения:

- `Open LV`. Количество логических томов, используемых в настоящее время.
- `Cur PV`. Количество физических томов, входящих в группу томов.
- `Act LV`. Количество активных физических томов в группе томов.
- `VG UUID`. Универсальный идентификатор группы томов. В системе может быть более одной группы томов с одинаковыми именами, в этом случае UUID помогает опознать конкретную группу. Большинство инструментов LVM (к примеру, `vgrename`, который как раз и помогает разрешить подобные ситуации) понимает идентификатор UUID как альтернативу имени группы

томов. Имейте в виду, что вы увидите множество разных UUID — у каждого компонента LVM есть свой.

Физический экстент (physical extent (PE) в выводе команды `vgdisplay`) — это часть физического объема, похожая на блок, но гораздо большего масштаба. В приведенном ранее примере размер PE составляет 4 Мбайт. Из него видно, что большинство PE в этой группе томов используются, но опасаться нечего. Это просто объем пространства в группе томов, выделенный для логических разделов (в данном случае для файловой системы и области подкачки), он не отражает фактическое применение в файловой системе.

Перечисление логических томов

Как и в ситуации с группами томов, команды для перечисления логических томов — это `lvs` (краткий вывод) и `lvdisplay` (подробный вывод). Пример вывода команды `lvs`:

```
# lvs
  LV     VG        Attr       LSize    Pool Origin Data% Meta% Move Log Cpy%Sync Convert
  root   ubuntu-vg -wi-ao---- <9.01g
  swap_1 ubuntu-vg -wi-ao---- 976.00m
```

В базовых настройках LVM важно понимать только первые четыре столбца, а остальные могут оставаться пустыми, как в примере (мы не будем их рассматривать). Первые четыре столбца означают следующее:

- `LV`. Имя логического тома.
- `VG`. Группа томов, в которой находится логический том.
- `Attr`. Атрибуты логического тома. В данном случае это `w` (запись), `i` (inherited allocation policy, унаследованная политика распределения), `a` (активны) и `o` (открыты). В более продвинутых настройках групп томов активны многие из этих слотов, в частности первый, седьмой и девятый.
- `LSize`. Размер логического тома.

Запуск более подробной команды `lvdisplay` помогает понять, где именно логический том встроен в систему. Пример выходных данных одного из логических томов:

```
# lvdisplay /dev/ubuntu-vg/root
  --- Logical volume ---
  LV Path                /dev/ubuntu-vg/root
  LV Name                root
  VG Name                ubuntu-vg
  LV UUID                CELZaz-PWr3-tr3z-dA3P-syC7-KWsT-4YiUW2
  LV Write Access        read/write
  LV Creation host, time ubuntu, 2018-11-13 15:48:20 -0500
  LV Status              available
  # open                 1
```

```
  LV Size                <9.01 GiB
  Current LE             2306
  Segments               1
  Allocation             inherit
  Read ahead sectors     auto
  - currently set to     256
  Block device           253:0
```

В этом выводе много интересного, и большая часть довольно понятна (обратите внимание на то, что UUID логического тома отличается от идентификатора его группы томов). Возможно, самое важное в этом выводе — первая строка, `LV Path` — путь к устройству логического тома. Некоторые системы, но не все, используют его в качестве точки монтирования файловой системы или области подкачки (в модуле монтирования `systemd` или в каталоге `/etc/fstab`).

Несмотря на то что вы видите основные и второстепенные номера устройств блочного устройства логического тома (здесь 253 и 0), а также что-то, похожее на путь к устройству, на самом деле это не тот путь, который задействует ядро. Взгляните на каталог `/dev/ubuntu-vg/root`:

```
$ ls -l /dev/ubuntu-vg/root
lrwxrwxrwx 1 root root 7 Nov 14 06:58 /dev/ubuntu-vg/root -> ../dm-0
```

Как видите, это всего лишь символическая ссылка на `/dev/dm-0`. Давайте вкратце обсудим это.

Устройства логических томов

Как только LVM выполнит работу по настройке системы, устройства с блоками логических томов будут доступны по адресам `/dev/dm-0`, `/dev/dm-1` и т. д. и могут располагаться в любом порядке. Из-за непредсказуемости этих имен устройств LVM создает также символические ссылки на устройства, которые имеют стабильные имена на основе имен групп томов и логических томов. Это было показано ранее в примере с `/dev/ubuntu-vg/root`.

В большинстве реализаций системы есть дополнительное место расположения для символических ссылок — `/dev/mapper`. Формат имени здесь также основан на группе томов и логическом томе, но иерархии каталогов нет, вместо этого ссылки носят такие имена, как `ubuntu--vg-root`. В этом случае `udev` преобразует одинарную черту в группе томов в двойную, а затем разделяет имена групп томов и логических томов одинарной чертой.

Многие системы используют ссылки в `/dev/mapper` в своих конфигурациях каталога `/etc/fstab`, системы `systemd` и загрузчика, чтобы указать системе логические тома, применяемые для файловых систем и области подкачки.

В любом случае эти символические ссылки указывают на блочные устройства для логических томов, и вы можете взаимодействовать с ними так же, как с любым

другим блочным устройством: создавать файловые системы, разделы подкачки и т. д.

ПРИМЕЧАНИЕ

Изучив каталог /dev/mapper, вы найдете файл с именем control. Возможно, вам будет интересно узнать о нем, а также о том, почему файлы реальных блочных устройств начинаются с dm- и совпадает ли это как-то с /dev/mapper. Мы рассмотрим эти вопросы в конце главы.

Работа с физическими томами

Последняя важная часть LVM — это *физический том* (Physical Volume, PV). Группа томов создается из одного или нескольких физических томов. Хотя физический том может показаться простой частью системы LVM, она содержит немного больше информации, чем кажется на первый взгляд. Как для группы томов и логических томов, команды LVM для просмотра физических томов — это `pvs` (краткий список) и `pvdisplay` (подробный список). Вот примерный вывод команды `pvs` для системы:

```
# pvs
  PV         VG        Fmt Attr PSize   PFree
  /dev/sda1 ubuntu-vg lvm2 a-- <10.00g 36.00m
```

А вот более подробный вывод команды `pvdisplay`:

```
# pvdisplay
  --- Physical volume ---
  PV Name               /dev/sda1
  VG Name               ubuntu-vg
  PV Size               <10.00 GiB / not usable 2.00 MiB
  Allocatable           yes
  PE Size               4.00 MiB
  Total PE              2559
  Free PE               9
  Allocated PE          2550
  PV UUID               v2Qb1A-XC2e-2G4l-NdgJ-lnan-rjm5-47eMe5
```

Ранее вы читали о группе томов и логических томах, так что можете понять большую часть этого вывода. Добавим несколько примечаний.

- Для физического тома нет специального названия, кроме имени блочного устройства. В этом и нет необходимости — все имена, требующиеся для ссылки на логический том, находятся на уровне группы томов и выше. Однако у физического тома есть UUID, который нужен для создания группы томов.
- В этом случае количество PE (phiysical extent, напомним, это увеличенный блок физического тома) соответствует числу используемых PE в группе томов (как показано ранее), потому что это единственный физический том в группе.

- Существует небольшое пространство, которое LVM помечает как непригодное для применения, потому что его недостаточно для размещения полного PE.
- Значение атрибута `a` в выводе физических томов `pvs` соответствует значению параметра Allocatable вывода `pvdisplay`, и это просто означает, что если вы хотите выделить место для логического тома в группе томов, LVM может выбрать использование этого физического тома. Однако в этом случае имеется только девять нераспределенных PEs (в общей сложности 36 Мбайт), поэтому для новых логических томов доступно не так уж много места.

Как упоминалось ранее, физические тома содержат больше чем просто сведения о себе в группе томов. Каждый физический том содержит метаданные физического тома (physical volume metadata), обширную информацию о его группе томов и его логических томах. В ближайшее время мы изучим *метаданные физических томов*, но сначала перейдем к практике, чтобы увидеть, как то, что мы узнали, сочетается друг с другом.

Построение системы логических томов

Давайте рассмотрим, как создать новую группу томов и некоторые логические тома из двух дисковых устройств. Мы объединим два дисковых устройства объемом 5 Гбайт и 15 Гбайт в группу томов, а затем разделим это пространство на два логических тома по 10 Гбайт каждый — практически невыполнимая задача без менеджера LVM. В приведенном здесь примере используются диски VirtualBox. Хотя для любой современной системы приведенные размеры дисков довольно малы, для примера этого достаточно.

На рис. 4.5 показана схема тома. Новые диски находятся в `/dev/sdb` и `/dev/sdc`, новая группа томов будет называться `myvg`, а два новых логических тома — `mylv1` и `mylv2`.

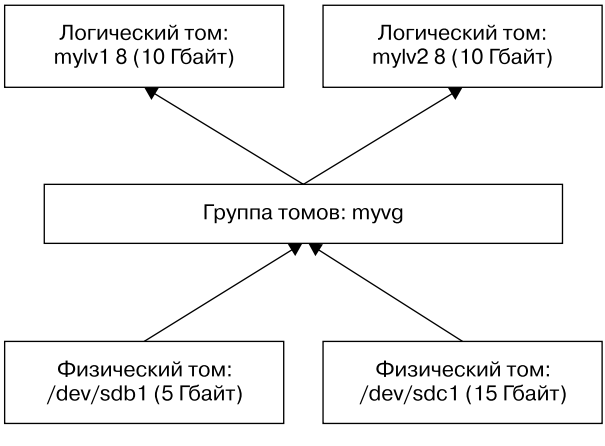


Рис. 4.5. Построение системы логических томов

Первая задача — создать один раздел на каждом из этих дисков и отметить его для менеджера LVM. Сделайте это с помощью программы разбиения на разделы (см. подраздел 4.1.2), используя идентификатор типа раздела `8e`, чтобы таблицы разделов выглядели следующим образом:

```
# parted /dev/sdb print
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sdb: 5616MB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number Start  End    Size   Type    File system Flags
 1     1049kB 5616MB 5615MB primary             lvm
# parted /dev/sdc print
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sdc: 16.0GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number Start  End    Size   Type    File system Flags
 1     1049kB 16.0GB 16.0GB primary             lvm
```

Не обязательно разбивать диск на разделы, чтобы сделать его физическим томом. Физический том может быть любым блочным устройством, даже устройством с целым диском, таким как `/dev/sdb`. Однако разбиение на разделы позволяет выполнять загрузку с диска, а также предоставляет средства идентификации блочных устройств как физических томов LVM.

Создание физических томов и группы томов

Необходимо с помощью LVM назначить новой группе томов новые разделы `/dev/sdb1` и `/dev/sdc1` и сделать один из них физическим томом. Эти задачи выполняет команда `vgcreate`. Вот пример того, как создать группу томов под названием `myvg` с `/dev/sdb1` в качестве начального физического тома:

```
# vgcreate myvg /dev/sdb1
  Physical volume "/dev/sdb1" successfully created.
  Volume group "myvg" successfully created
```

ПРИМЕЧАНИЕ

Вы также можете сначала создать физический том отдельно с помощью команды pvcreate. Однако vgcreate выполняет этот шаг в разделе, если в данный момент ничего другого нет.

На этом этапе большинство систем автоматически обнаруживают новую группу томов. Для проверки выполните команду `vgs` (имейте в виду, что в вашей системе

могут быть существующие группы томов, которые отображаются вдобавок к только что созданной):

```
# vgs
  VG  #PV #LV #SN Attr   VSize  VFree
  myvg  1   0   0 wz--n- <5.23g <5.23g
```

ПРИМЕЧАНИЕ

Если вы не видите новую группу томов, попробуйте сначала запустить команду pvscan. Если ваша система автоматически не обнаруживает изменения в LVM, нужно будет запускать pvscan каждый раз, когда вы вносите изменения.

Теперь можете добавить второй физический том в `/dev/sdc1` в группу томов с помощью команды `vgextend`:

```
# vgextend myvg /dev/sdc1
  Physical volume "/dev/sdc1" successfully created.
  Volume group "myvg" successfully extended
```

При запуске команды `vgs` отображаются два физических тома, а их размер равен размеру двух объединенных разделов:

```
# vgs
  VG   #PV #LV #SN Attr   VSize   VFree
  my-vg  2   0   0 wz--n- <20.16g <20.16g
```

Создание логических томов

Последним шагом на уровне блочного устройства является создание логических томов. Как упоминалось ранее, мы собираемся создать два логических тома по 10 Гбайт каждый, но вы можете экспериментировать, к примеру, добавить один большой логический том или несколько маленьких.

Команда `lvcreate` выделяет новый логический том в группе томов. Единственными реальными сложностями при создании простых логических томов являются определение размеров, когда группа томов состоит из более чем одного тома, и указание типа логического тома.

Помните, что физические тома делятся на экстенты PE, а количество доступных PE может *не совсем* соответствовать желаемому размеру тома. Однако количество экстентов должно быть указано приближенно, чтобы не вызывать проблем, поэтому, если вы впервые работаете с LVM, вам не нужно изучать физические экстенты.

При использовании команды `lvcreate` можете указать размер логического тома как емкость в байтах с параметром `--size` или как количество физических экстентов с параметром `--extents`.

Итак, чтобы увидеть, как это все работает, и завершить обсуждение схемы LVM, изображенной на рис. 4.5, мы создадим логические тома с именами `mylv1` и `mylv2` с помощью параметра `--size`:

```
# lvcreate --size 10g --type linear -n mylv1 myvg
  Logical volume "mylv1" created.
# lvcreate --size 10g --type linear -n mylv2 myvg
  Logical volume "mylv2" created.
```

В примере тип (type) — это линейное (linear) отображение, самый простой тип, когда вам не нужна избыточность или какие-то другие специальные функции (мы не будем работать с другими типами в этой книге). В этом случае `--type linear` оказывается необязательным, поскольку является параметром по умолчанию.

После выполнения этих команд убедитесь, что логические тома LV существуют, с помощью команды `lvs`, а затем внимательно изучите состояние группы томов VG с помощью `vgdisplay`:

```
# vgdisplay myvg
  --- Volume group ---
  VG Name               myvg
  System ID
  Format                lvm2
  Metadata Areas        2
  Metadata Sequence No  4
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                2
  Open LV               0
  Max PV                0
  Cur PV                2
  Act PV                2
  VG Size               20.16 GiB
  PE Size               4.00 MiB
  Total PE              5162
  Alloc PE / Size       5120 / 20.00 GiB
  Free PE / Size        42 / 168.00 MiB
  VG UUID               1pHrOe-e5zy-TUtK-5gnN-SpDY-shM8-Cbokf3
```

Обратите внимание на то, что в выводе есть 42 свободных физических экстента PE, потому что размеры, которые мы выбрали для логических томов, не полностью занимали все доступные экстенты в группе томов VG.

Управление логическими томами: создание разделов

Созданные логические тома можно использовать, установив файловые системы на устройства, таким же образом, как и любой обычный раздел диска. Как упоминалось ранее, в каталоге `/dev/mapper` и в данном случае в каталоге `/dev/myvg`

для группы томов будут находиться символические ссылки на устройства. Так, например, вы можете выполнить следующие три команды, чтобы создать файловую систему, временно смонтировать ее и посмотреть, сколько фактически места на логическом томе:

```
# mkfs -t ext4 /dev/mapper/myvg-mylv1
mke2fs 1.44.1 (24-Mar-2018)
Creating filesystem with 2621440 4k blocks and 655360 inodes
Filesystem UUID: 83cc4119-625c-49d1-88c4-e2359a15a887
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632
Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done
# mount /dev/mapper/myvg-mylv1 /mnt
# df /mnt
Filesystem            1K-blocks Used Available Use% Mounted on
/dev/mapper/myvg-mylv1 10255636 36888  9678076   1% /mnt
```

Удаление логических томов

Мы еще не рассматривали какие-либо операции с другим логическим томом, `mylv2`, поэтому давайте используем его, чтобы сделать пример более интересным. Допустим, вы обнаружили, что не задействуете второй логический том. Вы решаете удалить его и изменить размер первого логического тома, чтобы занять оставшееся место в группе томов. На рис. 4.6 отображена наша цель.

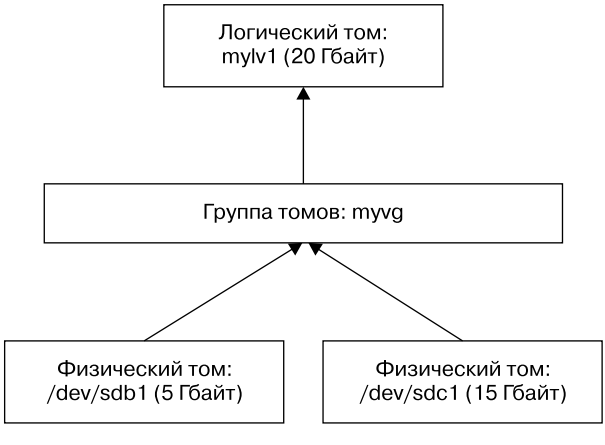


Рис. 4.6. Перераспределение логических томов

Предположив, что вы уже переместили важные данные, содержавшиеся на логическом томе, который собираетесь удалить, или создали их резервную копию

и том уже не используется в текущей системе (то есть вы размонтировали его), сначала удалите его с помощью команды `lvremove`. Работая с логическими томами, при помощи этой команды вы будете ссылаться на них, применяя другой синтаксис, то есть разделяя имена групп томов и логических томов косой чертой (`myvg/mylv2`):

```
# lvremove myvg/mylv2
Do you really want to remove and DISCARD active logical volume myvg/mylv2? [y/n]: y
  Logical volume "mylv2" successfully removed
```

> **ВНИМАНИЕ**
>
> Будьте внимательны при запуске команды lvremove. Поскольку вы не применяли этот синтаксис с другими командами LVM ранее, то можете случайно поставить пробел вместо косой черты. Если вы совершите эту ошибку в данном конкретном случае, lvremove предположит, что вы хотите удалить все логические тома в группах томов myvg и mylv2. (Скорее всего, у вас и не будет группы томов с именем mylv2, но на данный момент это не самая большая проблема.) Поэтому, если вы не обратите внимания на синтаксис, команда может удалить вообще все логические тома в группе томов.

Как видно из примера, команда `lvremove` пытается защитить вас от ошибок, дважды проверяя, действительно ли вы хотите удалить каждый логический том, отмеченный для удаления. К тому же она не попытается удалить используемый том. Однако не нужно отвечать на каждый вопрос согласием (команда `y`).

Изменение размера логических томов и файловых систем

Теперь вы можете изменить размер первого логического тома `mylv1`. Это можно сделать, даже если он используется и его файловая система уже смонтирована. При этом важно понимать, что необходимо выполнить два действия. Чтобы задействовать больший логический том, необходимо изменить размер как его, так и файловой системы внутри него (это тоже возможно, если том уже смонтирован). Но поскольку это распространенная задача, команда `lvresize`, которая изменяет размер логического тома, имеет и параметр (`-r`) для изменения размера файловой системы.

Только для примера применим две отдельные команды, чтобы увидеть, как они работают. Существует несколько способов изменить размер логического тома, но в данном случае наиболее простой вариант — добавление всех свободных физических экстентов в группе томов на логический том. Напомним, что вы можете узнать это число с помощью команды `vgdisplay`, в нашем примере это 2602. Вот пример использования команды `lvresize` на томе `mylv1`:

```
# lvresize -l +2602 myvg/mylv1
  Size of logical volume myvg/mylv1 changed from 10.00 GiB (2560 extents)
  to 20.16 GiB (5162 extents).
  Logical volume myvg/mylv1 successfully resized.
```

Теперь вам нужно изменить размер файловой системы внутри с помощью команды `fsadm`. Забавно наблюдать, как она работает в подробном (verbose) режиме (используйте параметр `-v`):

```
# fsadm -v resize /dev/mapper/myvg-mylv1
fsadm: "ext4" filesystem found on "/dev/mapper/myvg-mylv1".
fsadm: Device "/dev/mapper/myvg-mylv1" size is 21650997248 bytes
fsadm: Parsing tune2fs -l "/dev/mapper/myvg-mylv1"
fsadm: Resizing filesystem on device "/dev/mapper/myvg-mylv1" to 21650997248 bytes
(2621440 -> 5285888 blocks of 4096 bytes)
fsadm: Executing resize2fs /dev/mapper/myvg-mylv1 5285888
resize2fs 1.44.1 (24-Mar-2018)
Filesystem at /dev/mapper/myvg-mylv1 is mounted on /mnt; on-line resizing required
old_desc_blocks = 2, new_desc_blocks = 3
The filesystem on /dev/mapper/myvg-mylv1 is now 5285888 (4k) blocks long.
```

Как видно из вывода команды, `fsadm` — это просто скрипт, который знает, как преобразовать свои аргументы в аргументы, используемые специфичными для файловой системы инструментами, такими как `resize2fs`. Если вы не укажете определенный размер, он изменится самостоятельно, чтобы соответствовать размеру всего устройства.

После детального изучения подробностей изменения размеров томов вам, вероятно, понадобятся быстрые команды. Проще всего применить другой синтаксис для изменения размера и заставить команду `lvresize` изменить размер раздела с помощью единственной команды:

```
# lvresize -r -l +100%FREE myvg/mylv1
```

Удобно, что можно расширить файловую систему ext2/ext3/ext4, пока она смонтирована. К сожалению, в обратном направлении это не работает: нельзя урезать размер файловой системы, когда она уже смонтирована. В таком случае необходимо не только размонтировать файловую систему, но и выполнить все действия в обратном порядке, как того требует процесс урезания размера логического тома. Поэтому при изменении размера вручную вам нужно изменить размер раздела перед изменением размера логического тома, убедившись, что у нового логического тома все еще достаточно места для файловой системы. Опять же *гораздо* проще использовать команду `lvresize` с параметром `-r`, чтобы она могла координировать размеры файловой системы и логических томов за вас.

4.4.2. Реализация менеджера LVM

Теперь, когда мы рассмотрели практические основы работы менеджера LVM, можем кратко взглянуть на его реализацию. Как и практически все то, что рассматривается в любой другой теме этой книги, LVM содержит ряд слоев и компонентов с довольно тщательным разделением частей в ядре и в пользовательском пространстве. Как вы вскоре увидите, поиск физических томов для определения

структуры групп томов и логических томов довольно сложен, и ядро Linux предпочло бы не заниматься этим. Нет необходимости, чтобы все это происходило в пространстве ядра: физические тома — это всего лишь блочные устройства, а пользовательское пространство имеет произвольный доступ к блочным устройствам. На самом деле менеджер LVM (более конкретно — LVM2 в современных системах) — просто название для набора утилит пользовательского пространства, которые знают структуру LVM.

В то же время ядро выполняет работу по маршрутизации запроса на местоположение на блочном устройстве логического тома в истинное местоположение на реальном устройстве. Для этого применяется драйвер *device mapper* (иногда сокращенный до *devmapper*) — новый слой, расположенный между обычными блочными устройствами и файловой системой. Как следует из названия, задача, которую выполняет device mapper, похожа на поиск маршрута на карте, к примеру, перевод адреса улицы в абсолютное местоположение, то есть координаты глобальной широты/долготы. (Это форма виртуализации; виртуальная память, которую мы увидим далее в книге, работает аналогично.)

Существует связь между инструментами пользовательского пространства LVM и драйвером device mapper — несколько утилит, которые запускаются в пользовательском пространстве для управления картой устройств в ядре. Давайте рассмотрим подробнее как сторону LVM, так и сторону ядра.

Утилиты LVM и поиск физических томов

Прежде чем что-либо предпринять, утилита LVM должна просканировать доступные блочные устройства в поисках физических томов. Перечислим шаги, которые LVM следует выполнить в пространстве пользователя.

1. Найти все физические тома в системе.
2. Найти все группы томов, к которым принадлежат физические тома, по UUID (эта информация содержится в самих физических томах).
3. Убедиться, что все готово, то есть присутствуют все необходимые физические тома, принадлежащие группе томов.
4. Найти все логические тома в группах томов.
5. Определить схему сопоставления данных из физических в логические тома.

В начале вывода каждого физического тома есть заголовок, который идентифицирует том, а также его группы томов и логические тома внутри. Утилиты LVM могут объединить эту информацию и определить, присутствуют ли в системе все физические тома, необходимые для группы томов и ее логических томов. Если все верно, LVM начнет передавать данные в ядро.

ПРИМЕЧАНИЕ

Если вас интересует внешний вид заголовка LVM на физическом томе, выполните следующую команду:

```
# dd if=/dev/sdb1 count=1000 | strings | less
```

В этом примере используется каталог /dev/sdb1 в качестве физического тома. Не ожидайте, что вывод будет удобно читать, но он показывает информацию, необходимую для LVM.

Любая утилита LVM, такая как команда `pvscan`, `lvs` или `vgcreate`, способна сканировать и обрабатывать физические тома.

Драйвер device mapper

После того как менеджер LVM определил структуру логических томов из всех заголовков на физических томах, он связывается с драйвером ядра device mapper, чтобы инициализировать блочные устройства для логических томов и загрузить их таблицы сопоставления. Это достигается с помощью системного вызова `ioctl(2)` (обычно используемого интерфейса ядра) в файле устройства `/dev/mapper/control`. На самом деле нецелесообразно пытаться отслеживать это взаимодействие, но можно просмотреть подробные результаты с помощью команды `dmsetup`.

Чтобы вывести список сопоставленных устройств, которые в настоящее время обслуживаются драйвером device mapper, примените команду `dmsetup info`. Пример вывода от одного из логических томов, созданных ранее в этой главе:

```
# dmsetup info
Name:              myvg-mylv1
State:             ACTIVE
Read Ahead:        256
Tables present:    LIVE
Open count:        0
Event number:      0
Major, minor:      253, 1
Number of targets: 2
UUID: LVM-1pHrOee5zyTUtK5gnNSpDYshM8Cbokf3OfwX4T0w2XncjGrwct7nwGhpp7l7J5aQ
```

Основной (major) и второстепенный (minor) номера устройства соответствуют файлу `/dev/dm-*` устройства для сопоставленного устройства, основной номер для данного device mapper — 253. Поскольку второстепенное число равно 1, файл устройства называется `/dev/dm-1`. Обратите внимание на то, что у ядра есть имя и еще один UUID для подключенного устройства. LVM предоставил их ядру (идентификатор UUID ядра — это просто объединение идентификаторов группы томов и логических томов).

ПРИМЕЧАНИЕ

Помните, что символические ссылки, такие как /dev/mapper/myvg-mylv1?, утилита udev создает в ответ на новые устройства из драйвера device mapper, используя файл правил, как описано в подразделе 3.5.2.

Вы также можете просмотреть таблицу, которую LVM передал драйверу device mapper, выполнив команду `dmsetup table`. Вот как это выглядит в предыдущем примере, когда два логических тома объемом 10 Гбайт (`mylv1` и `mylv2`) были распределены по двум физическим томам объемом 5 Гбайт (`/dev/sdb1`) и 15 Гбайт (`/dev/sdc1`):

```
# dmsetup table
myvg-mylv2: 0 10960896 linear 8:17 2048
myvg-mylv2: 10960896 10010624 linear 8:33 20973568
myvg-mylv1: 0 20971520 linear 8:33 2048
```

Каждая строка содержит сегмент местоположения для данного отображаемого устройства. Для устройства `myvg-mylv2` есть две части, а для `myvg-mylv1` — одна. Порядок полей после имени таков:

1. Начальное смещение отображаемого устройства. Блоки расположены в 512-байтовых секторах или в блоке обычного размера, который видно на многих других устройствах.
2. Длина этого сегмента.
3. Схема отображения. В примере это простая линейная схема «один к одному».
4. Пара номеров основного и второстепенного устройств исходного устройства, то есть то, что LVM называет физическими томами. Здесь `8:17` — это `/dev/sdb1`, а `8:33` — `/dev/sdc1`.
5. Начальное смещение на исходном устройстве.

Интересно, что в нашем примере LVM решил использовать пространство в каталоге `/dev/sdc1` для первого созданного нами логического тома (`mylv1`). Менеджер LVM решил также, что он хочет разместить первый логический том объемом 10 Гбайт, не прерывая его, и единственный способ сделать это — добавить его в `/dev/sdc1`. Однако при создании второго логического тома (`mylv2`) у LVM не было иного выхода, кроме как разделить его на два сегмента между двумя физическими томами. На рис. 4.7 показано полученное расположение.

Впоследствии, когда мы удалили том `mylv2` и расширили том `mylv1`, чтобы заполнить оставшееся пространство в группе томов, исходное начальное смещение в физическом томе осталось там, где оно было в `/dev/sdc1`, а все остальное изменилось, позволяя добавить оставшуюся часть физического тома:

```
# dmsetup table
myvg-mylv1: 0 31326208 linear 8:33 2048
myvg-mylv1: 31326208 10960896 linear 8:17 2048
```

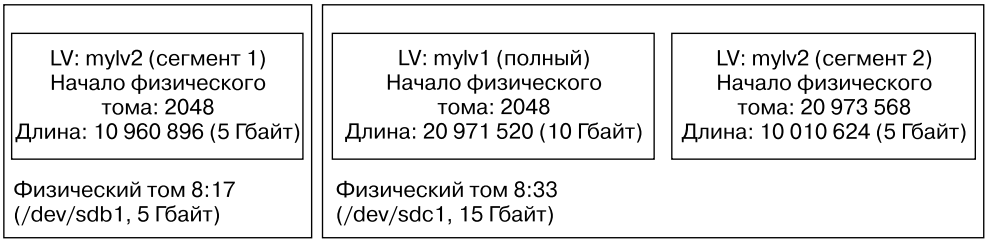


Рис. 4.7. Распределение томов mylv1 и mylv2 с помощью LVM

На рис. 4.8 отображено полученное размещение.

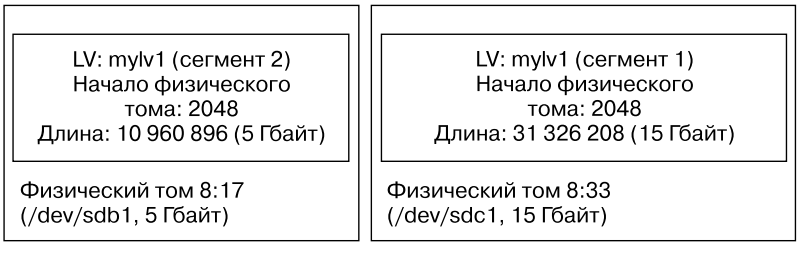


Рис. 4.8. Расположение томов после удаления тома mylv2 и расширения тома mylv1

Вы можете поэкспериментировать с логическими томами и сопоставлением устройств с помощью драйвера device mapper и виртуальных машин и узнать, каким будет результат. Многие функции, такие как программа RAID и зашифрованные диски, встроены в драйвер device mapper.

4.5. Что дальше? Диски и пользовательское пространство

В системе Unix у компонентов, связанных с дисками, границы между пользовательским пространством и ядром могут быть трудноразличимыми. Как вы видели, ядро обрабатывает необработанный блочный ввод-вывод с устройств, и инструменты пользовательского пространства могут применять блочный ввод-вывод через файлы устройств. Однако пользовательское пространство обычно задействует блочный ввод-вывод только для инициализации операций, таких как разбиение на разделы, создание файловой системы и области подкачки. В обычной ситуации пользовательское пространство применяет только поддержку файловой системы, которую ядро предоставляет в верхнем слое блочного ввода-вывода. Аналогично, ядро также обрабатывает большинство мелких деталей в ходе работы с областью подкачки в системе виртуальной памяти.

В оставшейся части этой главы мы кратко рассмотрим внутренние компоненты файловой системы Linux. Эта информация более продвинутого уровня, и для продолжения работы с книгой она необязательна. Если вы только знакомитесь с системой, перейдите к следующей главе и начните изучать, как загружается Linux.

4.6. Что находится внутри традиционной файловой системы

Традиционная файловая система Unix состоит из двух основных компонентов: пула блоков данных, в которых можно хранить информацию, и системы баз данных, управляющей пулом данных. База данных сосредоточена вокруг структуры данных inode. Дескриптор *inode* — это набор данных, описывающих конкретный файл, включая его тип, права доступа и, что самое важное, где именно в пуле данных находятся данные файла. Дескрипторы inodev идентифицируются по номерам, перечисленным в таблице inode.

Имена файлов и каталогов также реализованы в виде дескрипторов inode . Дескриптор каталога содержит список имен файлов и ссылок, соответствующих другим дескрипторам.

Для примера я создал новую файловую систему, смонтировал ее и изменил каталог на точку монтирования. Затем добавил некоторые файлы и каталоги с помощью следующих команд:

```
$ 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 `dir_2/file_5` как прямую ссылку на `dir_1/file_3`, что означает: эти два имени файлов на самом деле представляют один и тот же файл (подробнее об этом позже). Попробуйте и вы выполнить эту задачу (не обязательно в новой файловой системе).

Если исследовать каталоги в этой файловой системе, можно заметить, что ее содержимое выглядело бы так, как на рис. 4.9.

ПРИМЕЧАНИЕ

В вашей системе номера дескрипторов, вероятно, будут другими, особенно если вы запустите команды для создания файлов и каталогов в существующей файловой системе. Конкретные цифры не важны, все дело в данных, на которые они указывают.

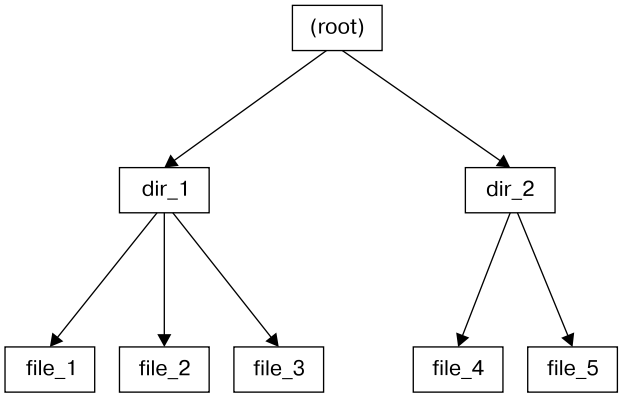


Рис. 4.9. Представление файловой системы на уровне пользователя

Фактическое содержимое файловой системы в виде набора дескрипторов, показанное на рис. 4.10, выглядит далеко не так очевидно, как представление на уровне пользователя.

Как понять эту таблицу? Для любой файловой системы ext2/ext3/ext4 вы начинаете с дескриптора под номером 2, который является *корневым дескриптором inode* (постарайтесь не путать его с корневой файловой системой всей системы). Из таблицы дескрипторов, приведенной на рис. 4.10, видно, что это дескриптор каталога (`dir`), поэтому вы можете перейти по стрелке к пулу данных, где видно содержимое корневого каталога: две записи с именами `dir_1` и `dir_2` для дескрипторов в 12 и 7633 соответственно. Чтобы изучить эти записи, вернитесь в таблицу дескрипторов и просмотрите любой из них.

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

1. Определяет компоненты пути — каталог с именем `dir_1`, за которым следует компонент с именем `file_2`.
2. Следует за корневым дескприптором к его данным каталога.
3. Находит имя `dir_1` в данных каталога `inode 2`, которое указывает на дескриптор с номером 12.
4. Ищет дескриптор `inode 12` в таблице дескрипторов и подтверждает, что это дескриптор каталога.
5. Следует по ссылке дескриптора `inode 12` к его информации каталога (второе поле внизу в пуле данных).
6. Находит второй компонент пути (`file_2`) в данных каталога `inode 12`. Запись указывает на дескриптор номер 14.
7. Находит `inode 14` в таблице каталогов. Это дескриптор файла.

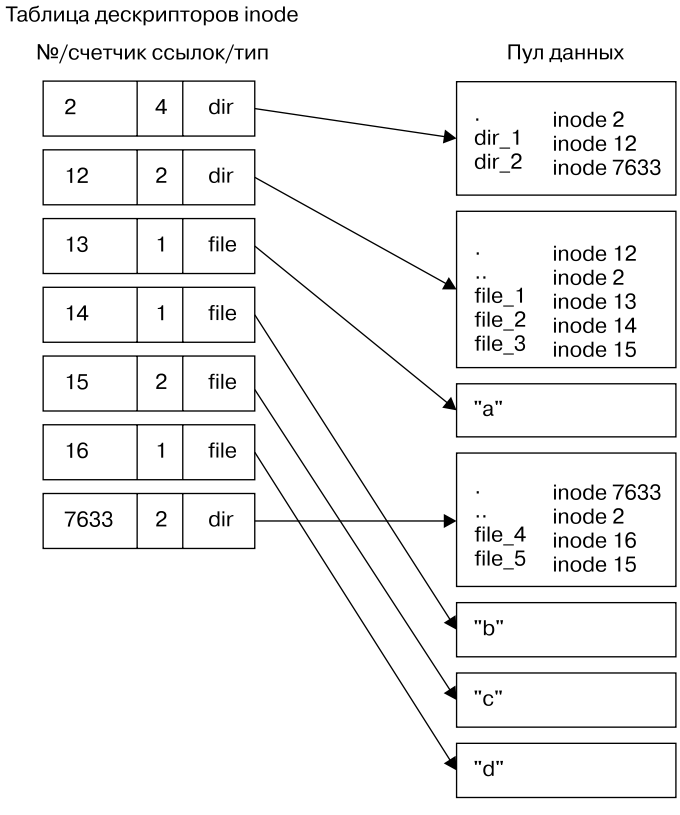


Рис. 4.10. Структура дескрипторов файловой системы, показанной на рис. 4.9

На этом этапе ядро знает свойства файла и может открыть его, перейдя по ссылке на данные дескриптора `inode 14`.

Система дескрипторов, указывающих на структуры данных каталогов, и структур данных, указывающих на дескрипторы, позволяет создавать привычную иерархию файловой системы. Кроме того, обратите внимание на то, что дескрипторы каталога содержат записи для `.` (текущий каталог) и `..` (родительский каталог, за исключением корневого каталога). Это позволяет легко найти точку отсчета при переходе на уровень выше в структуре каталогов.

4.6.1. Сведения о дескрипторе и количество ссылок

Чтобы просмотреть значение дескриптора для любого каталога, используйте команду `ls -i`. Вот что вы увидите, если применить ее к корневому каталогу (получить более подробную информацию о дескрипторе можно с помощью команды `stat`):

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

Вы, вероятно, задаетесь вопросом о *количестве ссылок* в таблице дескрипторов. Вы уже видели количество ссылок в выводе обычной команды `ls -l`. Как количество ссылок соотносится с файлами на рис. 4.9, в частности, с жесткой ссылкой на файл `file_5`? Поле `link count` — это общее количество записей каталога (во всех каталогах), указывающих на дескриптор. Большинство файлов в этом поле имеют значение 1, потому что они встречаются только один раз в записях каталога, что вполне ожидаемо. В большинстве случаев, создавая файл, вы создаете новую запись в каталоге и новый дескриптор для нее. Однако в выводе `inode 15` встречается дважды. Сначала он создается как `dir_1/file_3`, а затем связывается как `dir_2/file_5`. Жесткая ссылка — это просто созданная вручную запись в каталоге с уже существующим дескриптором. Команда `ln` (без параметра `-s`) позволяет создавать новые жесткие ссылки вручную.

Именно поэтому удаление файла называют удалением *ссылки*. Если вы запустите команду `rm dir_1/file_2`, ядро будет искать запись с именем `file_2` в записях каталога `inode 12`. Обнаружив, что `file_2` соответствует `inode 14`, ядро удаляет запись каталога, а затем вычитает 1 из числа ссылок для дескриптора 14. В результате количество ссылок в `inode 14` будет равно 0, и ядро будет знать, что больше нет имен, ссылающихся на дескриптор. Таким образом, теперь он может удалить дескриптор и любые связанные с ним данные.

Однако если вы запустите команду `rm dir_1/file_3`, то количество ссылок в `inode 15` изменится с двух до одной (потому что `dir_2/file_5` все еще указывает на нее), и ядро знает, что дескриптор удалять не нужно.

Количество ссылок примерно так же работает и для каталогов. Обратите внимание на то, что количество ссылок `inode 12` равно 2, потому что там есть две ссылки на дескриптор: одна для `dir_1` в записях каталога для `inode 2`, а вторая — свое же название (`.`) в его собственных записях каталога. Если вы создадите новый каталог `dir_1/dir_3`, количество ссылок для `inode 12` увеличится до трех, поскольку новый каталог будет содержать родительскую (`..`) запись, которая ссылается на `inode 12` так же, как родительская ссылка `inode 12` указывает на `inode 2`.

В подсчете ссылок есть одно небольшое исключение. У корневого `inode 2` четыре ссылки. Однако на рис. 4.10 показаны только три ссылки для входа в каталог. Четвертая находится в суперблоке файловой системы, так как он сообщает, где найти корневой дескриптор *inode*.

Не бойтесь экспериментировать с системой. Создание структуры каталогов, а затем применение команды `ls -i` или начало просмотра фрагментов безвредно. Для этого вам не нужно быть суперпользователем (если только вы не смонтируете и не создадите новую файловую систему).

4.6.2. Распределение блоков

В нашем обсуждении все еще не хватает одного фрагмента. Как файловая система узнает, какие блоки задействуются и какие доступны при выделении блоков пула данных для нового файла? Один из наиболее простых способов — применение дополнительной структуры данных управления, называемой *битовой картой блоков (block bitmap)*. В этой схеме файловая система резервирует ряд байтов, каждый бит которых соответствует одному блоку в пуле данных. Значение `0` говорит о том, что блок свободен, а `1` — что он используется. Таким образом, выделение и освобождение блоков — это вопрос переключения состояния битов.

Проблемы в файловой системе возникают, когда данные таблицы дескрипторов не совпадают с данными о распределении блоков или когда количество ссылок неверно (это может произойти, например, если вы не полностью выключили систему). Поэтому, когда вы проверяете файловую систему, как описано в подразделе 4.2.11, программа `fsck` просматривает таблицу дескрипторов и структуру каталогов, чтобы сгенерировать новое количество ссылок и новую карту распределения блоков (например, битовую карту блоков), а затем сравнивает вновь сгенерированные данные с файловой системой на диске. Если есть несоответствия, `fsck` должна исправить количество ссылок и определить, что делать с любыми дескрипторами и/или данными, которые не появились при пересечении структуры каталогов. Большинство программ `fsck` создают эти «осиротевшие» новые файлы в каталоге `lost+found`.

4.6.3. Работа с файловыми системами в пользовательском пространстве

В ходе работы с файлами и каталогами в пользовательском пространстве вам не нужно сильно беспокоиться о низкоуровневой реализации. Процессы получают доступ к содержимому файлов и каталогов смонтированной файловой системы с помощью системных вызовов ядра. Любопытно, однако, что у вас есть доступ к определенной информации о файловой системе, которая не вписывается в пространство пользователя, в частности, системный вызов `stat()` возвращает номера дескрипторов inode и количество ссылок.

Нужно ли пользователю думать о количестве дескрипторов, количестве ссылок и других деталях реализации, если у него файловая система не поддерживается? В целом, нет. Эти данные доступны программам пользовательского пространства в первую очередь для обеспечения обратной совместимости. Кроме того, не все файловые системы, доступные в Linux, имеют эти внутренние компоненты файловой системы. Уровень интерфейса VFS гарантирует, что системные вызовы всегда возвращают значения дескрипторов inode и количество ссылок, но они не обязательно что-то значат.

Возможно, вы не сможете выполнять обычные Unix-операции с файловой системой в нетрадиционных файловых системах. Например, не сможете применять

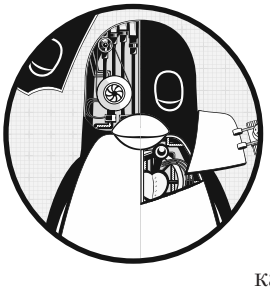
команду `ln` для создания жесткой ссылки в смонтированной файловой системе VFAT, поскольку ее структура записей каталогов, разработанная для Windows, а не для Unix/Linux, не поддерживает эту функцию.

К счастью, системные вызовы, доступные для пользовательского пространства в системах Linux, обеспечивают достаточный уровень абстракции для свободного доступа к файлам, то есть вам не нужно ничего знать о базовой реализации, чтобы получить доступ к файлам. Кроме того, имена файлов имеют гибкий формат и смешанный регистр, что облегчает поддержку других файловых систем с иерархией компонентов.

Помните, что поддержка определенной файловой системы не обязательно должна происходить в ядре. Например, в файловых системах пользовательского пространства ядру необходимо действовать только в качестве канала для системных вызовов.

5

Загрузка ядра Linux



Ранее мы изучили физическую и логическую структуру системы Linux, говорили о том, что такое ядро и как работать с процессами. В этой главе рассмотрим, как запускается и *загружается* ядро. Другими словами, вы узнаете, как ядро перемещается в память и что оно делает до момента запуска первого пользовательского процесса.

Упрощенно процесс загрузки выглядит следующим образом.

1. BIOS или загрузочная программа компьютера загружается и запускает загрузчик.
2. Загрузчик находит образ ядра на диске, загружает его в память и запускает.
3. Ядро инициализирует устройства и их драйверы.
4. Ядро монтирует корневую файловую систему.
5. Ядро запускает программу под названием `init` с идентификатором процесса `1`. Эта точка является *началом пользовательского пространства*.
6. Программа `init` приводит в действие остальные системные процессы.
7. В какой-то момент запускается процесс, позволяющий пользователю войти в систему, обычно в конце или ближе к концу процесса загрузки.

В этой главе рассматриваются первые четыре этапа, основное внимание уделяется загрузчику и ядру. В главе 6 продолжается изучение загрузки пользовательского пространства: подробно описывается программа `systemd` — наиболее распространенная версия программы `init` в системах Linux.

Умение узнавать каждый этап процесса загрузки окажет вам неоценимую помощь при устранении проблем с загрузкой и понимании системы в целом. Однако поведение системы, заданное по умолчанию во многих дистрибутивах Linux, часто затрудняет (если не делает невозможной) идентификацию первых нескольких этапов загрузки по мере их выполнения, поэтому вы, вероятно, сможете хорошо изучить их только после того, как они завершатся и вы войдете в систему.

5.1. Сообщения при загрузке

Традиционные системы Unix при загрузке выдают множество диагностических сообщений, которые рассказывают вам о ее ходе. Сообщения поступают сначала от ядра, а затем от процессов и систем инициализации, которые постепенно запускаются. Однако они не последовательны, а в некоторых случаях даже не особо информативны. Кроме того, аппаратные улучшения привели к тому, что теперь ядро запускается намного быстрее, чем раньше, и сообщения мелькают так быстро, что бывает трудно понять, что происходит. В результате большинство современных дистрибутивов Linux делают все возможное, чтобы скрыть диагностику загрузки с помощью заставок и других форм заполнения экрана, чтобы отвлечь вас во время запуска системы.

Лучший вариант просмотреть диагностические сообщения о загрузке ядра и времени выполнения — открыть системный журнал ядра с помощью команды `journalctl`. При запуске `journalctl -k` отображаются сообщения текущей загрузки, а с помощью параметра `-b` можно просмотреть предыдущие загрузки. Мы рассмотрим этот журнал более подробно в главе 7.

Если у вас нет системы `systemd`, можете поискать файл журнала `/var/log/kern.log` или выполнить команду `dmesg` для просмотра сообщений в *кольцевом буфере ядра*.

Пример вывода команды `journalctl -k`:

```
microcode: microcode updated early to revision 0xd6, date = 2019-10-03
Linux version 4.15.0-112-generic (buildd@lcy01-amd64-027) (gcc version 7.5.0
(Ubuntu 7.5.0-3ubuntu1~18.04)) #113-Ubuntu SMP Thu Jul 9 23:41:39 UTC 2020
(Ubuntu 4.15.0-112.113-generic 4.15.18)
Command line: BOOT_IMAGE=/boot/vmlinuz-4.15.0-112-generic root=UUID=17f12d53-c3d7-
4ab3-943e-a0a72366c9fa ro quiet splash vt.handoff=1
KERNEL supported cpus:
--пропуск--
scsi 2:0:0:0: Direct-Access ATA KINGSTON SM2280S 01.R PQ: 0 ANSI: 5
sd 2:0:0:0: Attached scsi generic sg0 type 0
sd 2:0:0:0: [sda] 468862128 512-byte logical blocks: (240 GB/224 GiB)
sd 2:0:0:0: [sda] Write Protect is off
sd 2:0:0:0: [sda] Mode Sense: 00 3a 00 00
sd 2:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO
or FUA
sda: sda1 sda2 < sda5 >
sd 2:0:0:0: [sda] Attached SCSI disk
--пропуск--
```

Начинающийся после запуска ядра процесс запуска пользовательского пространства также генерирует сообщения. Просматривать их, вероятно, будет сложнее, потому что в большинстве систем вы не найдете их ни в одном файле журнала. Скрипты запуска спроектированы для отправки на консоль сообщений, которые удаляются после завершения загрузки. Однако в системах Linux это не проблема, поскольку программа `systemd` фиксирует диагностические сообщения, которые обычно поступают на консоль при запуске и во время выполнения.

5.2. Параметры инициализации и загрузки ядра

При запуске ядро Linux инициализируется в таком порядке:

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

Первые два шага не слишком примечательны, но когда ядро добирается до устройств, возникает вопрос о зависимостях. Например, драйверы дисковых устройств могут зависеть от поддержки шины и поддержки подсистемы SCSI, как говорилось в главе 3. Позже, в процессе инициализации, ядро должно смонтировать корневую файловую систему перед инициализацией.

В целом, вам не нужно беспокоиться о зависимостях, за исключением того, что некоторые необходимые компоненты могут быть загружаемыми модулями ядра, а не частью основного ядра. Некоторым системам может потребоваться загрузить эти модули ядра до того, как будет смонтирована основная корневая файловая система. Мы рассмотрим эту проблему и ее решения для начальной файловой системы оперативной памяти (`initrd`) в разделе 6.7.

Ядро выдает определенные виды сообщений, указывающих на то, что оно готовится запустить свой первый пользовательский процесс:

```
Freeing unused kernel memory: 2408K
Write protecting the kernel read-only data: 20480k
Freeing unused kernel memory: 2008K
Freeing unused kernel memory: 1892K
```

Здесь оно не только очищает часть незадействованной памяти, но и защищает собственные данные. Затем, если вы запускаете достаточно новое ядро, то увидите, что оно запускает первый процесс в пользовательском пространстве как `init`:

```
Run /init as init process
   with arguments:
    --пропуск-
```

После этого монтируется корневая файловая система и запускается systemd, отправляя несколько собственных сообщений в журнал ядра:

```
EXT4-fs (sda1): mounted filesystem with ordered data mode. Opts: (null)
systemd[1]: systemd 237 running in system mode. (+PAM +AUDIT +SELINUX +IMA
+APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT +GNUTLS +ACL +XZ +LZ4
+SECCOMP +BLKID +ELFUTILS +KMOD -IDN2 +IDN -PCRE2 default-hierarchy=hybrid)
systemd[1]: Detected architecture x86-64.
systemd[1]: Set hostname to <duplex>.
```

На этом этапе пользовательское пространство запущено.

5.3. Параметры ядра

Когда ядро Linux запускается, оно получает набор текстовых *параметров ядра*, содержащих дополнительные сведения о системе. Параметры определяют множество различных характеристик поведения, таких как объем диагностических выходных данных, которые должно выдавать ядро, и параметры, зависящие от драйвера устройства.

Вы можете изучить параметры, переданные в активное ядро вашей системы, просмотрев файл `/proc/cmdline`:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-4.15.0-43-generic root=UUID=17f12d53-c3d7-4ab3-943e
-a0a72366c9fa ro quiet splash vt.handoff=1
```

Параметры представляют собой либо простые однословные флаги, такие как `ro` и `quiet`, либо пары *`key=value`*, например `vt.handoff=1`. Многие параметры не особо значимы для работы системы (например, флаг `splash` используется для отображения экрана загрузки), однако `root` — один из важнейших. Он отвечает за расположение корневой файловой системы, без него ядро не сможет правильно запустить пользовательское пространство. Корневую файловую систему можно указать как файл устройства, например:

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

В большинстве современных систем существуют две более распространенные альтернативы вывода. Первый способ позволяет отобразить логический том подобно этому:

```
root=/dev/mapper/my-system-root
```

Второй — отображает UUID (см. подраздел 4.2.4):

```
root=UUID=17f12d53-c3d7-4ab3-943e-a0a72366c9fa
```

Оба варианта эффективны, поскольку не зависят от конкретного сопоставления устройств ядра.

Параметр `ro` указывает ядру смонтировать корневую файловую систему в режиме «только для чтения» при запуске пользовательского пространства. Так и должно быть: режим «только для чтения» гарантирует, что утилита `fsck` сможет безопасно проверить корневую файловую систему, прежде чем пытаться выполнить более серьезные задачи. После проверки процесс загрузки повторно подключает корневую файловую систему в режиме чтения/записи.

Обнаружив незнакомый параметр, ядро Linux сохраняет его. Позже оно передает параметр программе `init` при запуске пользовательского пространства. Например, если вы добавляете параметр `-s` к параметрам ядра, оно передает `init` указание, что та должна запускаться в однопользовательском режиме.

Если вас интересуют основные параметры загрузки, на странице руководства `bootparam(7)` приведен их обзор. Если вам нужен конкретный параметр, проверьте файл `kernel-params.txt`, который поставляется с ядром Linux.

Изучив основы параметров, вы можете смело перейти к главе 6, чтобы узнать особенности запуска пользовательского пространства, начального диска оперативной памяти и программы init, которую ядро запускает в качестве своего первого процесса. В оставшейся части данной главы подробно описано, как ядро загружается в память, запускается и получает свои параметры.

5.4. Загрузчики

В начале загрузки, до загрузки ядра и команды init, программа *загрузчика* запускает ядро. Задача загрузчика проста: ему нужно загрузить ядро в память с диска, а затем запустить его с набором определенных параметров. Однако выполнить ее сложнее, чем кажется. Чтобы понять, почему так, рассмотрим вопросы, на которые должен ответить загрузчик:

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

Ответы, как правило, заключаются в том, что ядро и его параметры обычно находятся в корневой файловой системе. Кажется, что параметры ядра легко найти, но помните, что само ядро еще не запущено, а чаще всего именно оно просматривает файловую систему, чтобы найти необходимые файлы. Хуже того, драйверы устройств ядра, обычно применяемые для доступа к диску, также недоступны. Это похоже на извечный вопрос: что было раньше, курица или яйцо? В данном случае ситуация еще сложнее, но сейчас давайте рассмотрим, как загрузчик преодолевает препятствия, связанные с драйверами и файловой системой.

Загрузчику действительно нужен драйвер для доступа к диску, но не тот, который использует ядро. В системе Windows для доступа к дискам загрузчики применяют традиционную базовую систему ввода-вывода (BIOS, Basic Input-Output System) или более новый интерфейс *Unified Extensible Firmware Interface (UEFI)*. (Интерфейсы *Extensible Firmware Interface (EFI)* и *UEFI* подробнее рассмотрим в подразделе 5.8.2.) Современное дисковое оборудование включает встроенное ПО, позволяющее BIOS или UEFI получать доступ к подключенному оборудованию хранения данных через *стандартизованный механизм адресации (Logical Block Addressing, LBA)*. LBA — это универсальный простой способ доступа к данным с любого диска, но его производительность довольно низка. Это не проблема, поскольку загрузчики часто являются единственными программами, которые должны использовать этот режим для доступа к диску: после запуска ядро получает доступ к собственным высокопроизводительным драйверам.

ПРИМЕЧАНИЕ

Чтобы определить, задействует ли ваша система BIOS или UEFI, запустите утилиту efibootmgr. Если вы получите список целей загрузки, значит, в системе применяется UEFI. Если же придет сообщение, что переменные EFI не поддерживаются, — применяется BIOS. Кроме того, можете проверить, есть ли в вашей системе файл /sys/firmware/efi. Если это так, она использует интерфейс UEFI.

Как только доступ к необработанным данным диска разрешен, загрузчик должен выполнить работу по поиску нужных данных в файловой системе. Большинство распространенных загрузчиков могут считывать таблицы разделов и имеют встроенную поддержку доступа к файловым системам только для чтения. Таким образом, они могут находить и считывать файлы, необходимые для загрузки ядра в память. Эта возможность значительно упрощает динамическую настройку и усовершенствование загрузчика. Загрузчики Linux не всегда имели такую возможность, а без нее настроить загрузчик сложнее.

Для ядра наблюдается тенденция добавления новых функций (особенно в том, что касается технологии хранения данных), за которыми следуют загрузчики, добавляющие отдельные упрощенные версии этих функций для компенсации.

5.4.1. Задачи загрузчика

Основная функция загрузчика Linux включает в себя возможность выполнять следующие задачи:

- выбирать одно из нескольких ядер;
- переключаться между наборами параметров ядра;
- разрешить пользователю вручную переопределять и редактировать имена и параметры образов ядра (например, для входа в однопользовательский режим);
- обеспечить поддержку загрузки других операционных систем.

За время, прошедшее с момента создания ядра Linux, загрузчики стали более эффективными, появились новые функции, например история командной строки и системы меню, но основной задачей всегда была гибкость в выборе образа ядра и параметров. (Удивительно, но необходимость в некоторых задачах действительно снизилась. Например, поскольку вы можете выполнить аварийную или восстановительную загрузку с USB-накопителя, не нужно беспокоиться о ручном вводе параметров ядра или переходе в однопользовательский режим.) Современные загрузчики обеспечивают бóльшую мощность, чем когда-либо, что может быть особенно удобно, если вы создаете собственные ядра или хотите настроить параметры ядра.

5.4.2. Обзор загрузчиков

Вот основные загрузчики, которые вы можете встретить в системах Unix:

- **GRUB.** Почти универсальный загрузчик для систем Linux с версиями BIOS/MBR и UEFI.
- **LILO.** Один из первых загрузчиков Linux. ELILO — версия UEFI.
- **SYSLINUX.** Может быть настроен для запуска из множества файловых систем.
- **LOADLIN.** Загружает ядро из MS-DOS.
- **systemd-boot.** Простой менеджер загрузки UEFI.
- **coreboot (ранее LinuxBIOS).** Высокопроизводительная замена BIOS для персональных компьютеров, которая может содержать ядро.
- **Linux Kernel EFISTUB.** Плагин ядра для загрузки ядра непосредственно из системного раздела EFI/UEFI (ESP).
- **efilinux.** Загрузчик UEFI, предназначенный для применения в качестве модели и эталона для других загрузчиков UEFI.

Эта книга посвящена в основном загрузчику GRUB. Причины использования других загрузчиков заключаются в том, что их проще настроить, чем GRUB, они быстрее или предоставляют другие функции специального назначения.

Чтобы изучить информацию о загрузчике, перейдите в приглашение загрузки и введите имя ядра и параметры. Однако для этого нужно знать, как попасть в приглашение загрузки или меню. К сожалению, иногда это сложно найти, потому что дистрибутивы Linux сильно видоизменяют поведение и внешний вид загрузчика. Бывает, что, просто наблюдая за процессом загрузки, невозможно определить, какой именно загрузчик используется дистрибутивом.

В следующих разделах рассказывается, как получить приглашение загрузки, чтобы ввести имя ядра и параметры. После этого вы сможете узнать, как настроить и установить загрузчик.

5.5. Введение в загрузчик GRUB

GRUB расшифровывается как *Grand Unified Boot Loader — главный унифицированный загрузчик*. Мы рассмотрим версию GRUB 2, но существует также устаревшая версия под названием GRUB Legacy, которая применяется все реже.

Одной из наиболее важных возможностей GRUB является навигация по файловой системе, которая позволяет легко выбирать образ ядра и его параметры. Лучший способ увидеть это в действии и узнать о загрузчике GRUB в целом — заглянуть в его меню. Интерфейс прост в навигации, но есть большая вероятность, что ранее вы никогда его не видели.

Чтобы получить доступ к меню GRUB, нажмите и удерживайте клавишу Shift при первом появлении экрана запуска BIOS или ESC, если в вашей системе есть UEFI. В противном случае конфигурация загрузчика может не приостанавливаться перед загрузкой ядра. На рис. 5.1 показано меню GRUB.

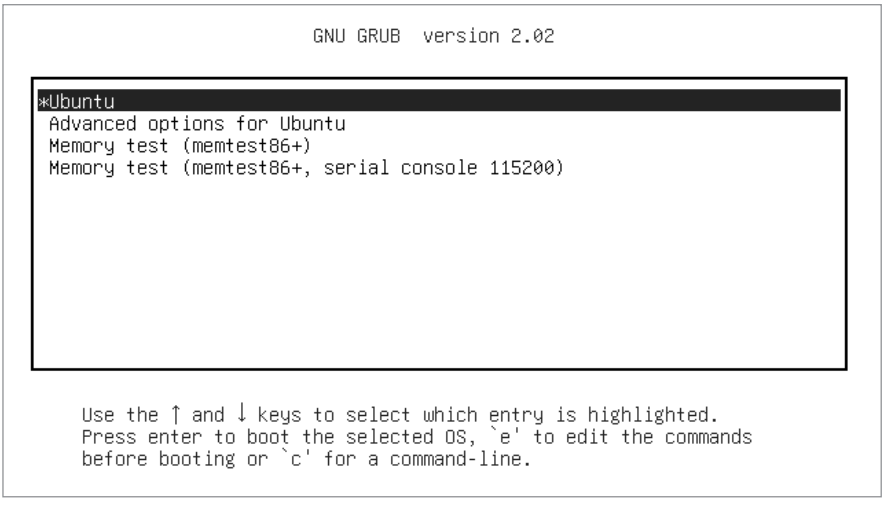


Рис. 5.1. Меню загрузчика GRUB

Выполните следующие действия, чтобы изучить загрузчик:

1. Перезагрузите или включите свою систему Linux.
2. Удерживайте нажатой клавишу Shift во время загрузки BIOS или Esc на экране загрузки, чтобы открыть меню GRUB. (Иногда заставка загрузки экрана не отображается, поэтому нужно угадать, когда нажимать кнопку.)
3. Нажмите клавишу e, чтобы просмотреть команды конфигурации загрузчика для параметра загрузки по умолчанию. Появится экран, как на рис. 5.2 (прокрутите страницу вниз, чтобы увидеть все детали).

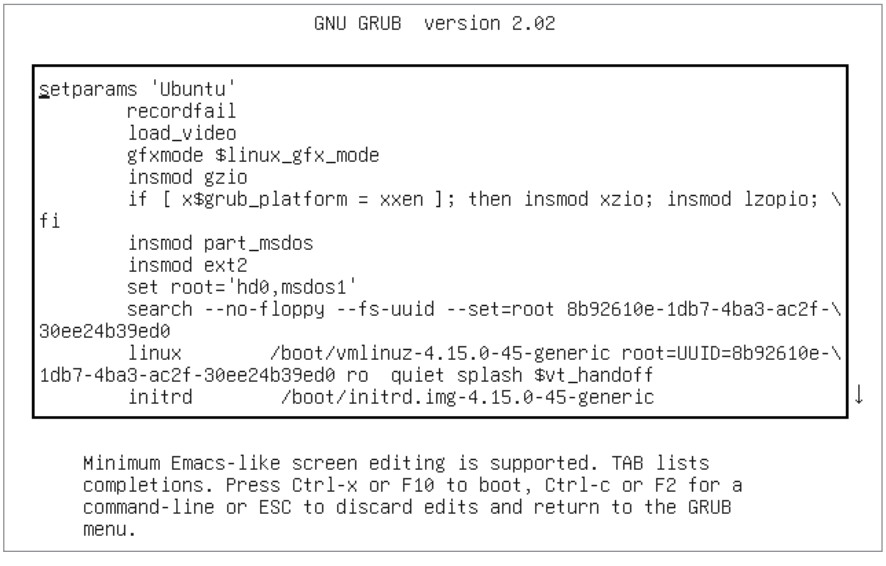


Рис. 5.2. Редактор настроек GRUB

Данный экран сообщает нам, что для этой конфигурации корневой каталог задан с идентификатором UUID, образ ядра — это `/boot/vmlinuz-4.15.0-45-generic` и параметры ядра включают `ro`, `quiet` и `splash`. Начальная файловая система оперативной памяти — это `/boot/initrd.img-4.15.0-45-generic`. Но если вы никогда раньше не видели подобной конфигурации, она может вас несколько сбить с толку. Почему существует несколько ссылок на `root` и почему они разные? Почему здесь указан параметр `insmod`? Если же вы встречали ее раньше, то, возможно, помните, что это функция ядра Linux, обычно запускаемая `udevd`.

Дубли в выводе обусловлены тем, что GRUB не использует ядро Linux (помните, что его задача — это именно *запустить* ядро). Конфигурация, которую вы видите, полностью состоит из функций и команд GRUB, она существует в собственном отдельном мире. Путаница частично связана с тем фактом, что GRUB заимствует терминологию из многих источников. GRUB имеет собственное «ядро» и собственную команду `insmod` для динамической загрузки модулей GRUB, полностью независимую от ядра Linux. Многие команды GRUB похожи на команды оболочки Unix, есть даже команда `ls` для перечисления файлов.

ПРИМЕЧАНИЕ

Существует модуль GRUB для LVM, необходимый для загрузки систем, в которых ядро находится на логическом томе. Вы можете увидеть это и в своей системе.

Безусловно, наибольшая путаница возникает из-за использования GRUB слова `root`. Обычно `root` применяется для определения корневой файловой системы

вашей системы. В конфигурации GRUB это параметр ядра, расположенный после имени образа команды `linux`.

Все остальные ссылки на слово `root` в конфигурации относятся к корню GRUB, который существует только внутри GRUB. Корень GRUB — это файловая система, в которой GRUB выполняет поиск файлов образов файловой системы ядра и оперативной памяти.

На рис. 5.2 корень GRUB сначала устанавливается на устройство, специфичное для GRUB (`hd0,msdos1`), значение по умолчанию для этой конфигурации ❶. В следующей команде GRUB выполняет поиск определенного UUID для раздела ❷. Если он найдет UUID, то установит корень GRUB в данный раздел.

Чтобы завершить процесс, первым аргументом команды `linux` (`/boot/vmlinuz-...`) должно стать расположение файла образа ядра Linux ❸. GRUB загружает этот файл из корневого каталога GRUB. Аналогично работает команда `initrd`, указывающая файл для описанной в главе 6 начальной файловой системы оперативной памяти ❹.

Вы можете отредактировать эту конфигурацию внутри GRUB — обычно это самый простой способ временно исправить ошибочную загрузку. Чтобы навсегда устранить проблему с загрузкой, вам потребуется изменить конфигурацию (см. подраздел 5.5.2), но сейчас давайте немного углубимся и рассмотрим некоторые внутренние компоненты GRUB с помощью интерфейса командной строки.

5.5.1. Изучение устройств и разделов с помощью командной строки GRUB

Как показано на рис. 5.2, GRUB имеет собственную схему адресации устройств. Например, первый найденный жесткий диск называется `hd0`, за ним следует `hd1` и т. д. Присвоение имен устройствам может быть изменено, но, к счастью, GRUB может выполнить поиск UUID во всех разделах, чтобы найти тот, в котором находится ядро, с помощью команды `search`, как вы только что видели на рис. 5.2.

Перечисление устройств

Чтобы понять, как GRUB ссылается на устройства в вашей системе, откройте командную строку GRUB, нажав клавишу `c` в меню загрузки или редакторе конфигурации. GRUB выведет приглашение:

```
grub>
```

Здесь можно ввести любую команду, отображенную в конфигурации, но для начала попробуйте выполнить диагностическую команду `ls`. Без аргументов вывод представляет собой список устройств, известных GRUB:

```
grub> ls
(hd0) (hd0,msdos1)
```

В этом случае имеются одно основное дисковое устройство, обозначаемое (hd0), и один раздел (hd0,msdos1). Если бы на диске был раздел подкачки, он также отображался бы, например как (hd0,msdos5). Префикс msdos в названиях разделов сообщает, что диск содержит таблицу разделов MBR; префикс будет начинаться с gpt для GPT таблицы в системах UEFI. (Существуют еще более глубокие комбинации с третьим идентификатором, когда карта разметки диска BSD находится внутри раздела, но обычно он не требуется, если вы не задействуете несколько операционных систем на одной машине.)

Для вывода более подробной информации используйте команду ls -l. Она может быть особенно полезна, поскольку отображает любые идентификаторы UUID разделов диска, например:

```
grub> ls -l
Device hd0: No known filesystem detected — Sector size 512B - Total size
32009856KiB
        Partition hd0,msdos1: Filesystem type ext* — Last modification time
          2019-02-14 19:11:28 Thursday, UUID 8b92610e-1db7-4ba3-ac2f-
30ee24b39ed0 - Partition start at 1024Kib - Total size 32008192KiB
```

Этот конкретный диск имеет файловую систему Linux ext2/ext3/ext4 на первом разделе MBR. Системы, применяющие раздел подкачки, покажут другой раздел, но вы не сможете определить его тип по выводу.

Перемещение по файлам

Теперь рассмотрим возможности навигации по файловой системе GRUB. Определите корень GRUB с помощью команды echo (напомним, что именно здесь GRUB ищет ядро):

```
grub> echo $root
hd0,msdos1
```

Чтобы использовать команду ls в GRUB для перечисления файлов и каталогов в этом корневом каталоге, добавьте косую черту в конец названия раздела:

```
grub> ls (hd0,msdos1)/
```

Поскольку неудобно вводить имя фактического корневого раздела, можете использовать переменную root, чтобы сэкономить время:

```
grub> ls ($root)/
```

Вывод представляет собой краткий список имен файлов и каталогов в файловой системе этого раздела, таких как etc/, bin/ и dev/. Теперь это совершенно другая функция команды ls для GRUB. Раньше вы перечисляли устройства, таблицы разделов и некоторую информацию о заголовке файловой системы. А сейчас можете увидеть содержимое файловых систем.

Аналогичным образом можно глубже изучить файлы и каталоги в разделе. Например, чтобы проверить каталог `/boot`, начните со следующей команды:

```
grub> ls ($root)/boot
```

ПРИМЕЧАНИЕ

Используйте клавиши со стрелками ↑ и ↓, чтобы просмотреть историю команд GRUB, и стрелки ← и → для редактирования текущей командной строки. Стандартные клавиши чтения строки (Ctrl+N, Ctrl+P и т. д.) тоже сработают.

Вы можете просмотреть все текущие установленные переменные GRUB с помощью команды `set`:

```
grub> set
?=0
color_highlight=black/white
color_normal=white/black
--пропуск--
prefix=(hd0,msdos1)/boot/grub
root=hd0,msdos1
```

Одной из наиболее важных переменных является `$prefix` — файловая система и каталог, в которых GRUB ищет необходимую конфигурацию и вспомогательную поддержку. Далее мы обсудим конфигурацию GRUB.

Закончив изучать интерфейс командной строки GRUB, нажмите клавишу Esc, чтобы вернуться в меню GRUB. В качестве альтернативы, если установили всю необходимую конфигурацию для загрузки (включая переменные `linux` и, возможно, `initrd`), можете ввести команду `boot` для загрузки этой конфигурации. В любом случае загрузите свою систему. Мы собираемся изучить конфигурацию GRUB, и делать это лучше всего, когда у вас есть полноценная и доступная система.

5.5.2. Конфигурация GRUB

Каталог конфигурации GRUB — это обычно `/boot/grub` или `/boot/grub2`. Он содержит центральный файл конфигурации `grub.cfg`, каталог для конкретной архитектуры, например `i386-pc`, содержащий загружаемые модули с суффиксом `.mod`, и несколько других элементов, таких как шрифты и информация о локализации. Мы не будем изменять `grub.cfg` напрямую — вместо этого используем команду `grub-mkconfig` (или `grub2-mkconfig` в Fedora).

Файл конфигурации grub.cfg

Если взглянуть на файл `grub.cfg`, можно увидеть, как GRUB инициализирует свое меню и параметры ядра. Вы также увидите, что файл состоит из команд GRUB, которые обычно начинаются с ряда шагов инициализации, за которыми следует ряд пунктов меню для различных конфигураций ядра и загрузки. Процесс

инициализации в целом несложен, но изначально есть множество условий, которые могут вас в этом разуверить. Первая часть состоит из набора определений функций, значений по умолчанию и команд настройки видео, таких как эта:

```
if loadfont $font ; then
set gfxmode=auto
load_video
insmod gfxterm
--пропуск—
```

ПРИМЕЧАНИЕ

Многие переменные, такие как $font, происходят из вызова load_env в начале файла grub.cfg.

Далее в файле конфигурации вы найдете доступные конфигурации загрузки, каждая из которых начинается с команды `menuentry`. Вы уже можете прочитать и понять этот пример, основываясь на том, что узнали в предыдущем разделе:

```
menuentry 'Ubuntu' --class ubuntu --class gnu-linux --class gnu --class os
$menuentry_id_option 'gnulinux-simple-8b92610e-1db7-4ba3-ac2f-30ee24b39ed0' {
        recordfail
        load_video
        gfxmode $linux_gfx_mode
        insmod gzio
        if [ x$grub_platform = xxen ]; then insmod xzio; insmod lzopio; fi
        insmod part_msdos
        insmod ext2
        set root='hd0,msdos1'
        search --no-floppy --fs-uuid --set=root 8b92610e-1db7-4ba3-ac2f-
          30ee24b39ed0
        linux   /boot/vmlinuz-4.15.0-45-generic root=UUID=8b92610e-1db7-4ba3-ac2f-
          30ee24b39ed0 ro quiet splash $vt_handoff
        initrd /boot/initrd.img-4.15.0-45-generic
}
```

Проверьте файл `grub.cfg` на наличие команд `submenu`, содержащих несколько команд `menuentry`. Многие дистрибутивы используют команду `submenu` для более старых версий ядра, чтобы они не заполняли меню в GRUB.

Создание нового файла конфигурации

Если вы хотите внести изменения в конфигурацию GRUB, не редактируйте файл `grub.cfg` напрямую, потому что он создается автоматически и система иногда перезаписывает его. Создайте новый файл конфигурации в другом месте, а затем запустите `grub-mkconfig` для создания новой конфигурации.

Чтобы увидеть, как работает генерация конфигурации, посмотрите на самое начало файла `grub.cfg`. Там должны быть строки комментариев, как в примере:

```
### BEGIN /etc/grub.d/00_header ###
```

При дальнейшем изучении вы обнаружите, что почти каждый файл в `/etc/grub.d` является сценарием оболочки, который создает фрагмент файла `grub.cfg`. Сама команда `grub-mkconfig` представляет собой сценарий оболочки, который запускает все в `/etc/grub.d`. Имейте в виду, что GRUB не запускает эти сценарии во время загрузки — мы запускаем их в пользовательском пространстве для создания файла `grub.cfg`, который запускает сам GRUB.

Попробуйте выполнить настройку от имени суперпользователя. Не беспокойтесь о перезаписи своей текущей конфигурации. Сама по себе эта команда просто отображает конфигурацию на стандартный вывод:

```
# grub-mkconfig
```

Что делать, если нужно добавить пункты меню и другие команды в конфигурацию GRUB? Если кратко, то вы должны поместить свои настройки в новый файл `custom.cfg` в каталоге конфигурации GRUB (обычно `/boot/grub/custom.cfg`).

А если развернуто, то все немного сложнее. Каталог конфигурации `/etc/grub.d` предоставляет вам два варианта сценария: `40_custom` и `41_custom`. Первый, `40_custom`, — это сценарий, который вы можете редактировать самостоятельно, но он наименее стабилен: обновление пакета, скорее всего, уничтожит все внесенные вами изменения. Сценарий `41_custom` проще — это просто серия команд, которые загружают файл `custom.cfg` при запуске GRUB. Если вы выберете этот вариант, ваши изменения не будут отображаться при создании файла конфигурации, потому что GRUB выполняет всю работу во время загрузки.

ПРИМЕЧАНИЕ

Числа перед именами файлов влияют на порядок обработки: файлы с меньшими числами занимают первые места в файле конфигурации.

Оба варианта пользовательских файлов конфигурации не особенно обширны, вы можете добавлять собственные сценарии для создания данных конфигурации. В каталоге `/etc/grub.d` вы можете найти дополнительные настройки, относящиеся к конкретному дистрибутиву. Например, Ubuntu добавляет в конфигурацию загрузки параметры проверки памяти (`memtest86+`).

Чтобы записать и установить недавно созданный файл конфигурации GRUB, внесите в свой каталог GRUB новую конфигурацию с параметром `-o` в команду `grub-mkconfig`, как в примере:

```
# grub-mkconfig -o /boot/grub/grub.cfg
```

Обязательно создайте резервную копию старой конфигурации и убедитесь, что вы производите установку в правильный каталог.

Теперь перейдем к техническим деталям работы GRUB и загрузчиков. Если вы хотите пропустить информацию о загрузчиках и ядре, открывайте сразу главу 6.

5.5.3. Установка GRUB

Установка загрузчика GRUB более сложна, чем его настройка. К счастью, беспокоиться об установке не нужно, потому что дистрибутив должен справиться с этим сам. Однако если вы пытаетесь скопировать или восстановить загрузочный диск либо подготавливаете собственную последовательность загрузки, вам может потребоваться установить загрузчик самостоятельно.

Прежде чем устанавливать загрузчик, прочтите раздел 5.4, чтобы получить представление о том, как загружаются компьютеры, и определите, какой тип загрузки вы используете — MBR или UEFI. Затем соберите программное обеспечение GRUB и определите, где будет находиться ваш каталог GRUB (по умолчанию применяется `/boot/grub`). Возможно, вам не понадобится выполнять сборку загрузчика GRUB, если это сделает сам дистрибутив, но если вы занимаетесь сборкой самостоятельно, см. главу 16 о том, как создавать программное обеспечение из исходного кода. Убедитесь, что вы создали правильную *цель*: она различается у загрузок MBR или UEFI (существуют различия даже между 32- и 64-разрядным EFI).

Установка GRUB в систему

Установка загрузчика требует, чтобы вы или программа установки определили следующее:

- целевой каталог GRUB, как его видит запущенная вами система. Как упоминалось ранее, обычно используется каталог `/boot/grub`, но он может быть и другим, если вы устанавливаете GRUB на другой диск для применения в другой системе;
- текущее устройство целевого диска GRUB;
- для загрузки UEFI — текущую точку монтирования системного раздела EFI (обычно это `/boot/efi`).

Помните, что GRUB — это модульная система, но для загрузки модулей она должна прочитать файловую систему, содержащую каталог GRUB. Ваша задача состоит в том, чтобы создать версию GRUB, способную считывать эту файловую систему, чтобы она могла загружать остальную часть своей конфигурации (`grub.cfg`) и любые необходимые модули. В Linux это означает создание версии GRUB с предварительно загруженным модулем `ext2.mod` (и, возможно, `lvm.mod`). При использовании этой версии вам нужно лишь поместить ее на загрузочную часть диска, а прочие необходимые файлы — в `/boot/grub`.

К счастью, GRUB поставляется с утилитой `grub-install` (не путать с `install-grub`, которую можно встретить в некоторых старых системах), которая выполняет бóльшую часть работы по установке файлов GRUB и настройке. Например, если ваш диск находится в `/dev/sda` и вы хотите установить GRUB на его MBR с текущим каталогом `/boot/grub`, используйте команду:

```
# grub-install /dev/sda
```

ВНИМАНИЕ

Неправильная установка GRUB может нарушить последовательность загрузки в системе, поэтому относитесь к этой команде серьезно. Изучите, как получить резервную копию MBR с помощью команды dd, создайте резервную копию любого другого установленного в данный момент каталога GRUB и убедитесь, что у вас есть план аварийной загрузки.

Установка GRUB с помощью MBR на внешнее запоминающее устройство

Чтобы установить GRUB на устройство хранения вне используемой системы, необходимо вручную указать каталог GRUB на этом устройстве в том же виде, в каком его теперь видит система. Предположим, что у вас есть целевое устройство `/dev/sdc` и его корневая файловая система, содержащая `/boot` (например, `/dev/sdc1`), смонтирован на `/mnt` текущей системы. Это означает, что при установке GRUB эта система увидит файлы GRUB в файле `/mnt/boot/grub`. При запуске `grub-install` установки укажите, куда должны быть помещены эти файлы, следующим образом:

```
# grub-install --boot-directory=/mnt/boot /dev/sdc
```

В большинстве систем MBR `/boot` является частью корневой файловой системы, но некоторые установки помещают `/boot` в отдельную файловую систему. Убедитесь, что знаете, где находится ваша цель `/boot`.

Установка GRUB с помощью UEFI

Предполагается, что установка с помощью UEFI проще, потому что вам нужно лишь скопировать загрузчик в нужное место. Но необходимо также сообщить о загрузчике прошивке, то есть сохранить конфигурацию загрузчика в NVRAM, с помощью команды `efibootmgr`. Команда `grub-install` запускает это, если она доступна, поэтому обычно можно установить GRUB в системе UEFI, как в следующем примере:

```
# grub-install --efi-directory=efi_dir --bootloader-id=name
```

Здесь *`efi_dir`* — это место, где каталог UEFI отображается в используемой системе (обычно задействуется `/boot/efi/EFI`, потому что раздел UEFI чаще всего монтируется в `/boot/efi`), а *`name`* — идентификатор загрузчика. К сожалению, при установке загрузчика UEFI может возникнуть множество проблем. Например, если вы устанавливаете его на диск, который в конечном итоге окажется в другой системе, необходимо выяснить, как сообщить об этом загрузчике прошивке новой системы. Проблемой может стать и то, что существуют различия в процедуре установки съемных носителей.

Но одной из самых больших проблем является функция безопасной загрузки UEFI.

5.6. Проблемы безопасной загрузки UEFI

Одна из новых проблем, влияющих на установку Linux, связана с функцией *безопасной загрузки*. Когда этот механизм UEFI активен, для запуска любого загрузчика

требуется цифровая подпись поставщика. Корпорация Microsoft потребовала от поставщиков оборудования, поставляющих Windows 8 и более поздние версии, использовать функцию безопасной загрузки. В результате, если вы попытаетесь установить в этих системах неподписанный загрузчик, встроенное ПО отклонит его и операционная система не загрузится.

У основных дистрибутивов Linux нет проблем с безопасной загрузкой, поскольку они включают уже подписанные загрузчики, обычно основанные на версии GRUB для UEFI. Часто между UEFI и GRUB проходит небольшая подписанная часть данных, UEFI запускает ее, и она, в свою очередь, выполняет GRUB. Защита от загрузки несанкционированного программного обеспечения — важная функция, если ваша система находится в ненадежной среде или должна соответствовать определенным требованиям безопасности, поэтому некоторые дистрибутивы требуют, чтобы вся последовательность загрузки, включая ядро, была подписана.

У систем с безопасной загрузкой есть некоторые недостатки, особенно значимые для тех, кто экспериментирует с созданием собственных загрузчиков. Вы можете обойти требование безопасной загрузки, отключив его в настройках UEFI. Однако это не сработает для систем с двойной загрузкой, например Windows.

5.7. Метод цепной загрузки других операционных систем

UEFI позволяет довольно легко поддерживать загрузку других операционных систем, поскольку в разделе EFI можно установить несколько загрузчиков. Однако устаревший интерфейс MBR не поддерживает эту функцию, и даже если у вас есть UEFI, то все равно может иметься отдельный раздел с загрузчиком в стиле MBR, который вы хотите использовать. Вместо настройки и запуска ядра Linux GRUB может загружать и запускать другой загрузчик в определенном разделе на диске — это называется *цепной загрузкой (chainloading)*.

Для цепной загрузки создайте новый пункт в меню конфигурации GRUB с помощью одного из методов, описанных в разделе «Создание нового файла конфигурации». Вот пример установки Windows на третьем разделе диска:

```
menuentry "Windows" {
        insmod chain
        insmod ntfs
        set root=(hd0,3)
        chainloader +1
}
```

Параметр `+1` указывает команде `chainloader` загружать все, что находится в первом секторе раздела. Вы также можете заставить ее напрямую загрузить файл, используя для загрузки `io.sys` в загрузчик MS-DOS такую строку:

```
menuentry "DOS" {
        insmod chain
        insmod fat
        set root=(hd0,3)
        chainloader /io.sys
}
```

5.8. Детали работы загрузчика

Теперь мы быстро изучим часть внутренних компонентов загрузчика. Чтобы понять, как работают загрузчики, такие как GRUB, сначала рассмотрим, как загружается компьютер при включении. Поскольку схемы загрузки должны устранять многочисленные недостатки традиционных механизмов загрузки ПК, они имеют несколько вариантов, но два основных — это, конечно, MBR и UEFI.

5.8.1. Загрузчик MBR

Дополним информацию из раздела 4.1. В MBR (Master Boot Record, главная загрузочная запись) есть небольшая область размером 441 байт, которую BIOS загружает и выполняет после самотестирования при включении питания (POST, Power-On SelfTest). К сожалению, этого пространства недостаточно для размещения практически любого загрузчика, поэтому необходимо дополнительное пространство, что вызывает появление того, что иногда называют *многоэтапным загрузчиком*. В этом случае начальный фрагмент кода в MBR не делает ничего, кроме загрузки остальной части кода загрузчика. Оставшиеся части загрузчика обычно помещаются в пространство между MBR и первым разделом на диске. Это не совсем безопасно, так как там что угодно может перезаписать код, но тем не менее большинство загрузчиков работают так, включая и установки GRUB.

Эта схема загрузки кода загрузчика после MBR не работает с диском с разделами GPT, использующим BIOS для загрузки, поскольку информация о GPT находится в области после MBR. (GPT не меняет традиционный MBR для сохранения обратной совместимости.) Обходной путь для GPT заключается в создании небольшого раздела, называемого *загрузочным разделом BIOS*, со специальным идентификатором UUID (`21686148-6449-E6F-744E-656564454649`), чтобы предоставить место для размещения полного кода загрузчика. Однако это не обычная конфигурация, поскольку GPT используется с интерфейсом UEFI, а не с традиционным BIOS. Обычно он встречается только в старых системах с очень большими дисками (более 2 Тбайт), которые для MBR слишком велики.

5.8.2. Загрузчик UEFI

Производители персональных компьютеров и компании, выпускающие программное обеспечение, поняли, что традиционный PC BIOS сильно ограничен, поэтому они разработали ему замену и создали Extensible Firmware Interface (EFI,

расширенный интерфейс прошивки), о котором мы уже говорили в этой главе. EFI потребовалось некоторое время, чтобы освоиться на большинстве систем, и сегодня это наиболее распространенный интерфейс, особенно сейчас, когда Microsoft требует безопасной загрузки для Windows. Действующим стандартом является Unified EFI (UEFI, унифицированный EFI), который включает в себя такие функции, как встроенная оболочка и возможность чтения таблиц разделов и навигации по файловым системам. Схема разделения GPT — это часть стандарта UEFI.

Загрузка в системах UEFI радикально отличается от загрузки MBR. По большей части ее гораздо легче понять. Вместо исполняемого загрузочного кода, находящегося за пределами файловой системы, существует специальная файловая система VFAT, называемая системным разделом EFI (ESP, EFI System Partition), которая содержит каталог с именем `EFI`. В системе Linux ESP обычно монтируется в каталоге `/boot/efi`, поэтому вы сможете найти бóльшую часть структуры каталогов EFI, начиная с `/boot/efi/EFI`. Каждый загрузчик имеет собственный идентификатор и соответствующий подкаталог, такой как `efi/microsoft`, `efi/apple`, `efi/ubuntu` или `efi/grub`. Файл загрузчика имеет расширение `.efi` и находится в одном из этих подкаталогов вместе с другими вспомогательными файлами, например `grubx64.efi` (версия GRUB для EFI) и `shimx64.efi`.

ПРИМЕЧАНИЕ

ESP отличается от загрузочного раздела BIOS, описанного в подразделе 5.8.1, и имеет другой UUID. В вашей системе будет только один вариант.

Однако есть одна загвоздка: вы не можете просто вставить старый код загрузчика в ESP, потому что он был написан для интерфейса BIOS. Вместо этого нужно задействовать загрузчик, написанный для UEFI. Например, при использовании GRUB необходимо установить версию GRUB для UEFI, а не версию BIOS. И как объяснялось ранее в разделе «Установка GRUB с помощью UEFI», вы должны сообщать о новых загрузчиках прошивке.

Наконец, как отмечалось в разделе 5.6, необходимо решить проблему безопасной загрузки.

5.8.3. Как работает GRUB

Завершим обсуждение GRUB, изучив весь процесс его работы.

1. PC BIOS или прошивка инициализирует аппаратное обеспечение и выполняет поиск загрузочного кода на своих устройствах хранения данных.
2. После нахождения загрузочного кода BIOS (или прошивка) загружается и выполняет его. Вот тут-то и вступает в дело GRUB.
3. Загружается ядро GRUB.

4. Ядро инициализируется. На этом этапе GRUB может получить доступ к дискам и файловым системам.
5. GRUB идентифицирует свой загрузочный раздел и загружает туда конфигурацию.
6. GRUB дает пользователю возможность изменить конфигурацию.
7. После тайм-аута или действий пользователя GRUB выполняет конфигурацию (последовательность команд в файле `grub.cfg`, как описано в разделе 5.5.2).
8. В ходе выполнения конфигурации GRUB может загружать дополнительный код (модули) в загрузочный раздел. Некоторые из них могут быть предварительно загружены.
9. GRUB выполняет команду `boot` для загрузки и выполнения ядра, как указано в конфигурации команды `linux`.

Шаги 3 и 4, на которых загружается ядро GRUB, могут осложниться из-за недостатков традиционных механизмов загрузки ПК. Самый большой вопрос: где *именно* находится ядро GRUB? Существует три основных варианта ответа:

- частично между MBR и началом первого раздела;
- в обычном разделе;
- в специальном загрузочном разделе — загрузочном разделе GPT, ESP или другом.

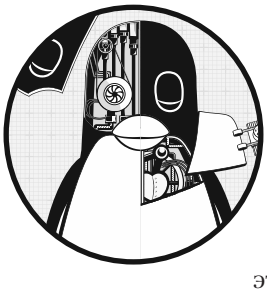
Во всех случаях, кроме тех, когда вы используете UEFI/ESP, PC BIOS загружает 512 байт из MBR, и именно отсюда начинается работа GRUB. Этот маленький фрагмент, полученный из `boot.img` в каталоге GRUB, — еще не ядро, но он содержит начальное местоположение ядра и загружает его с этой точки.

Однако если у вас есть ESP, ядро GRUB содержится там в виде файла. Прошивка может перемещаться по ESP и напрямую запускать весь загрузчик GRUB или любой другой операционной системы, расположенный там. (У вас может быть прослойка в ESP, которая идет непосредственно перед GRUB для обработки безопасной загрузки, но идея остается прежней.)

Тем не менее в большинстве систем описанный процесс — это еще не вся картина. Загрузчику также может потребоваться загрузить начальный образ файловой системы оперативной памяти перед загрузкой и исполнением ядра. Это определяет параметр конфигурации `initrd`, и мы рассмотрим его в разделе 6.7. Но прежде чем вы узнаете о файловой системе оперативной памяти, мы должны рассмотреть запуск пользовательского пространства, и с этого начинается следующая глава.

6

Запуск пользовательского пространства



Точка, в которой ядро запускает инициализацию init — свой первый процесс в пользовательском пространстве, — имеет большое значение не только потому, что память и процессор в этот момент уже готовы к полноценной работе системы, но и потому, что именно здесь вы можете увидеть, как строится остальная часть системы в целом. До этого момента ядро идет по максимально контролируемому пути, определяемому относительно небольшим числом разработчиков программного обеспечения. Пользовательское пространство является более модульным и настраиваемым и позволяет увидеть, что входит в процесс его запуска и эксплуатации. Эти знания можно применять, потому что для понимания и изменения запуска пользовательского пространства не требуется навык программирования на низком уровне.

Пользовательское пространство запускается в следующем порядке:

1. Процесс `init`.
2. Основные службы низшего уровня, такие как `udevd` и `syslogd`.
3. Конфигурация сети.
4. Службы среднего и высокого уровня (`cron`, `print` и т. д.).
5. Приглашения для входа в систему, графические интерфейсы и приложения высокого уровня, такие как веб-серверы.

6.1. Основные сведения об init

`init` (инициализация) — это программа пользовательского пространства, такая же, как и любая другая программа в системе Linux, она находится в каталоге `/sbin` вместе со многими другими системными двоичными файлами. Ее основная цель — запускать и останавливать основные служебные процессы системе.

Во всех актуальных версиях основных дистрибутивов Linux стандартной реализацией `init` является `systemd`. Эта глава посвящена тому, как работает `systemd` и как с ней взаимодействовать.

У `systemd` есть две разновидности, которые вы можете встретить в старых системах. Инициализация System V init — это традиционная последовательная инициализация (Sys V происходит из Unix System V) в Red Hat Enterprise Linux (RHEL) во всех версиях до 7.0 и Debian 8. Upstart — это инициализация дистрибутивов Ubuntu во всех версиях до 15.04.

Существуют и другие версии `init`, особенно на встроенных платформах. Например, Android имеет собственную систему `init`, и версия под названием `runit` популярна в облегченных системах. У BSD также есть своя версия `init`, но вы вряд ли увидите ее в современной системе Linux. (Некоторые дистрибутивы изменили конфигурацию инициализации System V, чтобы она была похожа на стиль BSD.)

Различные виды системы `init` были разработаны для устранения ряда недостатков в системе System V. Чтобы понять причину проблем, рассмотрим внутреннюю работу традиционного процесса инициализации. По сути, это серия сценариев, которые запускаются последовательно, по одному за раз.

Каждый сценарий обычно запускает одну службу или настраивает отдельную часть системы. В большинстве случаев довольно легко получается разобраться с зависимостями, к тому же система достаточно гибка и позволяет выполнять необычные задачи при запуске путем изменения сценариев.

Однако эта схема имеет существенные ограничения. Их можно объединить в проблемы с производительностью и проблемы с управлением системой. Наиболее важны следующие:

- Производительность проседает из-за того, что две части последовательной загрузки обычно не могут выполняться одновременно.
- Управление работающей системой может быть затруднено. Ожидается, что сценарии запуска запустят демонов службы. Чтобы найти PID демона службы, необходимо использовать команду `ps` или какой-либо другой механизм, специфичный для службы, либо полустандартную систему записи PID, такую как `/var/run/myservice.pid`.
- Сценарии запуска, как правило, содержат много стандартного, шаблонного кода, что иногда затрудняет чтение и понимание их действий.

- Информации о службах и конфигурации по требованию немного. Большинство служб запускается во время загрузки, конфигурация системы в значительной степени также устанавливается в это время. В свое время традиционный демон `inetd` мог обрабатывать сетевые службы по требованию, но в современных системах он не используется.

Современные системы инициализации справились с этими проблемами, изменив способ запуска служб, порядок их контроля и настройки зависимостей. Мы рассмотрим, как все это работает в `systemd`, но сначала необходимо удостовериться в ее наличии.

6.2. Определение системы инициализации

Определение версии инициализации вашей системы обычно не составляет труда. Страница руководства `init(1)` сообщает об этом сразу, но если вы не уверены, проверьте свою систему следующим образом:

- Если в системе есть каталоги `/usr/lib/systemd` и `/etc/systemd`, значит, она применяет `systemd`.
- Если у вас есть каталог `/etc/init`, содержащий несколько файлов `.conf`, то система использует Upstart (если только вы не работаете с Debian 7 и старше, в этом случае задействуется инициализация System V). Здесь мы не будем рассматривать Upstart, потому что эта система была вытеснена `systemd` в большинстве дистрибутивов Linux.
- Если ни одного из перечисленных каталогов в системе не существует, но есть файл `/etc/inittab`, вероятно, используется инициализация System V. Перейдите к разделу 6.5.

6.3. systemd

Команда `systemd` — это одна из новейших реализаций команды `init` в Linux. В дополнение к обработке обычного процесса загрузки `systemd` призвана включить функциональность ряда стандартных служб Unix, таких как `cron` и `inetd`. Создатели `systemd` вдохновлялись работой команды `launchd` системы Apple.

Чем `systemd` действительно отличается от своих предшественников, так это ее расширенными возможностями управления службами. В отличие от традиционной `init`, `systemd` может отслеживать отдельные демоны службы после их запуска и группировать несколько процессов, связанных со службой, что дает больше возможностей и позволяет лучше понять, что именно выполняется в системе.

Система `systemd` ориентирована на выполнение задач и целей. На верхнем уровне цель определяется с помощью *юнита* (unit) для какой-либо системной задачи. Он может содержать инструкции для общих задач запуска, таких как запуск демона, а также зависимости, которые являются другими юнитами. При запуске (или

активации) юнита `systemd` пытается активировать его зависимости, а затем переходит к деталям юнита.

При запуске служб система не придерживается жесткой последовательности, вместо этого она активирует юниты всякий раз, когда они готовы. После загрузки systemd может реагировать на системные события (например, события `uevents`, описанные в главе 3), активируя дополнительные юниты.

Начнем с изучения верхнего уровня юнита, активации и начального процесса загрузки. После этого ознакомимся со спецификой конфигурации юнита и множеством вариантов его зависимостей. По пути мы поймем, как просматривать процессы работающей системы и как ею управлять.

6.3.1. Юниты и типы юнитов

Одно из преимуществ эффективности системы `systemd` заключается в том, что она не просто управляет процессами и службами, но и может управлять монтированием файловой системы, отслеживать запросы на подключение к сети, запускать таймеры и многое другое. Каждая такая возможность называется *типом юнита*, а каждая конкретная функция (например, служба) — *юнитом*. Включая юнит, вы *активируете* его. У каждого юнита есть собственный файл конфигурации, мы рассмотрим их в подразделе 6.3.3.

Перечислим наиболее важные типы юнитов, которые выполняют задачи во время загрузки в типичной системе Linux.

- **Юниты служб (service units).** Управляют служебными демонами, найденными в системе Unix.
- **Целевые юниты (target units).** Управляют другими юнитами, обычно группируя их.
- **Сокет-юниты (socket units).** Представляют местоположения запросов на входящее сетевое подключение.
- **Юниты монтирования (mount units).** Контролируют присоединение файловых систем с системой.

ПРИМЕЧАНИЕ

Полный список типов юнитов можно найти на странице руководства systemd(1).

Служебные и целевые юниты — наиболее распространенные и простые для понимания из перечисленных. Давайте посмотрим, как они сочетаются друг с другом при загрузке системы.

6.3.2. Графики загрузки и зависимостей юнитов

При загрузке системы вы активируете целевой юнит по умолчанию, называемый `default.target`, который объединяет несколько юнитов служб и монтирования

в качестве зависимостей. Это дает частичное представление о том, что произойдет при загрузке. Можно подумать, что зависимости юнитов образуют дерево с одним общим юнитом вверху, разветвляющееся на несколько юнитов ниже для более поздних этапов процесса загрузки, — но на самом деле они образуют граф. Юнит, который появляется с опозданием в процессе загрузки, может зависеть от нескольких предыдущих юнитов, что приводит к объединению более ранних ветвей дерева зависимостей. Вы даже можете создать граф зависимостей с помощью команды `systemd-analyze dot`. Весь граф для типичной системы довольно велик (для визуализации требуется значительная вычислительная мощность), и его трудно прочитать, но есть способы фильтровать юниты и обнулять отдельные части.

На рис. 6.1 показана малая часть графа зависимостей для юнитов `default.target` из типичной системы. При активации этого юнита все юниты под ним также активируются.

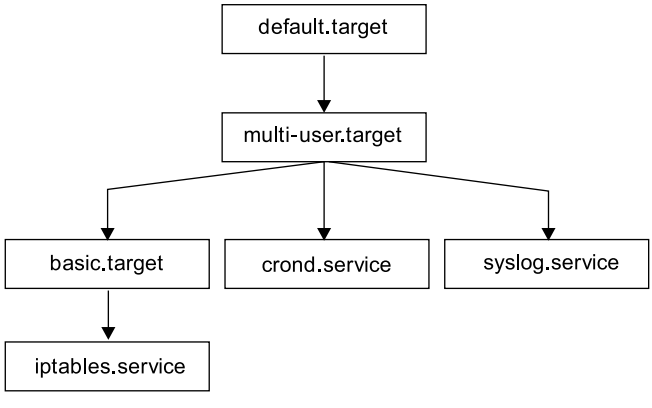


Рис. 6.1. Граф зависимостей юнитов

ПРИМЕЧАНИЕ

В большинстве систем default.target — это ссылка на какой-либо другой целевой юнит высокого уровня, например, относящийся к запуску пользовательского интерфейса. В системе, показанной на рис. 6.1, по умолчанию default.target группирует устройства, необходимые для запуска графического интерфейса.

На рисунке отображено очень упрощенное представление. В собственной системе вы не сможете набросать граф зависимостей, просто посмотрев файл конфигурации юнита сверху вниз. Мы подробнее рассмотрим, как работают зависимости, в подразделе 6.3.6.

6.3.3. Конфигурация systemd

Файлы конфигурации `systemd` распределены по многим каталогам в системе, поэтому, возможно, конкретный файл вам придется поискать. Существуют два основных

каталога для конфигурации `systemd`: каталог *системного юнита (system unit)* (глобальная конфигурация, обычно `/lib/systemd/system` или `/usr/lib/systemd/system`) и каталог *конфигурации системы (system configuration)* (локальные определения, обычно `/etc/systemd/system`).

Чтобы избежать путаницы, придерживайтесь следующего правила: не вносите изменения в каталог системного юнита, потому что дистрибутив сохраняет все изменения. Вносите локальные изменения в каталог конфигурации системы. Это общее правило применяется в масштабах всей системы. Когда вам предоставляется выбор между изменением чего-либо в каталогах `/usr` и `/etc`, всегда меняйте данные в `/etc`.

Вы можете проверить текущий путь поиска конфигурации `systemd`, включая приоритетность, с помощью команды

```
$ systemctl -p UnitPath show
UnitPath=/etc/systemd/system.control /run/systemd/system.control /run/systemd/
transient /etc/systemd/system /run/systemd/system /run/systemd/generator /lib/
systemd/system /run/systemd/generator.late
```

Чтобы просмотреть каталоги системного юнита и конфигурации в своей системе, используйте следующие команды:

```
$ pkg-config systemd --variable=systemdsystemunitdir
/lib/systemd/system
$ pkg-config systemd --variable=systemdsystemconfdir
/etc/systemd/system
```

Юнит-файлы

Формат юнит-файлов получен из спецификации записей рабочего стола XDG (используется для файлов `.desktop`, очень похожих на файлы `.ini` в системах Microsoft) с именами секций в квадратных скобках (`[]`) и присвоением значений переменных (параметров) в каждой секции. В качестве примера рассмотрим файл `dbus-daemon.service` для демона `bus`:

```
[Unit]
Description=D-Bus System Message Bus
Documentation=man:dbus-daemon(1)
Requires=dbus.socket
RefuseManualStart=yes

[Service]
ExecStart=/usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile
--systemd-activation --syslog-only
ExecReload=/usr/bin/dbus-send --print-reply --system --type=method_call --dest=
org.freedesktop.DBus / org.freedesktop.DBus.ReloadConfig
```

Существует две секции, `[Unit]` и `[Service]`. Секция `[Unit]` содержит сведения о юните, описание и информацию о зависимостях. В частности, данному юниту требуется юнит `dbus.socket` в качестве зависимости.

В служебном юните, подобном `dbus.socket`, вы найдете подробную информацию о службе в секции `[Service]`, в том числе о том, как подготовить, запустить и перезагрузить эту службу. Полный перечень можно найти на страницах руководства `systemd.service(5)` и `systemd.exec(5)`, а также в подразделе 6.3.5, где мы обсудим отслеживание процессов.

Многие другие файлы конфигурации юнитов также просты. Например, файл служебного юнита `sshd.service` позволяет удаленно безопасно входить в оболочку, запустив `sshd`.

ПРИМЕЧАНИЕ

Юнит-файлы в вашей системе могут немного отличаться от рассмотренных. В этом примере видно, что Fedora использует имя dbus-daemon.service, а Ubuntu — dbus.service. В самих файлах также могут быть отличия, но они чаще всего незначительны.

Переменные

В юнит-файлах часто можно найти различные переменные. Например, вот так выглядит секция из другого юнит-файла, на этот раз для защищенной оболочки, о которой вы узнаете в главе 10:

```
[Service]
EnvironmentFile=/etc/sysconfig/sshd
ExecStartPre=/usr/sbin/sshd-keygen
ExecStart=/usr/sbin/sshd -D $OPTIONS $CRYPTO_POLICY
ExecReload=/bin/kill -HUP $MAINPID
```

Все, что начинается со знака доллара (`$`), — это переменные. Хотя они имеют одинаковый синтаксис, их происхождение различно. Параметры `$OPTIONS` и `$CRYPTO_POLICY`, которые вы можете передать в `sshd` при активации юнита, определены в файле, указанном в параметре `EnvironmentFile`. В этом конкретном случае вы можете просмотреть файл `/etc/sysconfig/sshd`, чтобы понять, определены ли переменные, и, если да, узнать, каковы их значения.

Для сравнения: переменная `$MAINPID` содержит идентификатор *отслеживаемого процесса (tracked process)* службы (см. раздел 6.3.5). После активации устройства systemd записывает и сохраняет этот PID (Process ID), чтобы впоследствии использовать его для управления процессом службы. Файл юнита `sshd.service` задействует переменную `$MAINPID` для отправки сигнала зависания (`HUP`) в `sshd`, когда вы хотите перезагрузить конфигурацию (это очень распространенный метод работы с перезагрузками и перезапуском демонов Unix).

Спецификаторы

Спецификатор — это функция, подобная переменной, часто встречающаяся в юнит-файлах. Спецификаторы начинаются со знака процента (`%`). Например, спецификатор `%n` — это текущее имя юнита, а `%H` — имя текущего хоста.

Вы можете использовать спецификаторы для создания нескольких копий юнитов из одного юнит-файла. Одним из примеров является набор процессов `getty`, которые управляют запросами на вход в систему на виртуальных консолях, таких как tty1 и tty2. Чтобы задействовать эту функцию, добавьте символ `@` в конец имени юнита перед точкой в имени юнит-файла. Например, в большинстве дистрибутивов именем юнит-файла `getty` будет `getty@.service`, оно делает возможным динамическое создание юнитов, таких как `getty@tty1` и `getty@tty2`. Все, что стоит после символа `@`, называется *инстансом* (instance, экземпляром). Просматривая один из этих юнит-файлов, вы можете также увидеть спецификатор `%I` или `%i`. При активации службы из юнит-файла с инстансами, `systemd` расширяет указанный спецификатор `%I` или `%i` в имя экземпляра.

6.3.4. Процесс работы systemd

Вы будете взаимодействовать с `systemd` в основном с помощью команды `systemctl`, которая позволяет активировать и деактивировать службы, отображать статус, перезагружать конфигурацию и многое другое.

Существуют наиболее важные команды, которые помогут вам получить информацию о юнитах. Например, чтобы просмотреть список активных юнитов в своей системе, выполните команду `list-units` (по умолчанию она выполняется с командой `systemctl`, поэтому технически аргумент `list-units` не нужен):

```
$ systemctl list-units
```

Формат вывода типичен для информационных команд в Unix. Например, заголовок и строка для `-.mount` (корневая файловая система) выглядят следующим образом:

```
UNIT                LOAD   ACTIVE SUB        DESCRIPTION
-.mount             loaded active mounted    Root Mount
```

По умолчанию `systemctl list-units` выводит много сведений, потому что типичная система имеет множество активных юнитов, но это все равно сокращенная форма вывода, потому что `systemctl` урезает длинные имена юнитов. Чтобы просмотреть полные названия юнитов, используйте параметр `--full`, а чтобы просмотреть все юниты (не только активные), примените параметр `--all`.

Особенно полезной функцией `systemctl` является возможность отобразить статус конкретного юнита. Вот типичная команда `status` и часть ее выходных данных:

```
$ systemctl status sshd.service
· sshd.service - OpenBSD Secure Shell server
  Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled; vendor preset:
    enabled)
  Active: active (running) since Fri 2021-04-16 08:15:41 EDT; 1 months 1 days ago
 Main PID: 1110 (sshd)
    Tasks: 1 (limit: 4915)
   CGroup: /system.slice/sshd.service
           └1110 /usr/sbin/sshd -D
```

За этим выводом может следовать ряд сообщений журнала. Если вы привыкли к традиционной системе init, вас может удивить количество полезной информации, которую можно получить из вывода одной этой команды. Вы узнаете не только состояние юнита, но и процессы, связанные со службой, когда юнит был запущен, и ряд сообщений журнала, если таковые имеются.

Информационный вывод для других типов юнитов содержит аналогичную полезную информацию. Например, информационный вывод юнитов монтирования включает время, когда произошло монтирование, точную выполненную команду и конечный статус.

Интересная часть вывода — название группы управления (control group — `cgroup`). В предыдущем примере группой управления является `/system.slice/sshd.service`, а процессы в группе показаны ниже. Вы можете увидеть также группы управления с именами, начинающимися с `systemd:/system`, если процессы юнитов (например, юнитов монтирования) уже завершены. А можете просматривать системные группы управления без соответствующего статуса юнита с помощью команды `systemd-cgls`. Больше о том, как `systemd` использует группы управления cgroups, сказано в подразделе 6.3.5, а о том, как работают группы управления, — в разделе 8.6.

Команда `status` также отображает только самые последние сообщения журнала диагностики для юнита. Можете просмотреть все сообщения юнита следующим образом:

```
$ journalctl --unit=unit_name
```

О команде `journalctl` подробнее говорится в главе 7.

ПРИМЕЧАНИЕ

В зависимости от конфигурации вашей системы и пользователя вам могут потребоваться права суперпользователя для запуска команды journalctl.

Связь заданий с запуском, остановкой и перезагрузкой юнитов

Для активации, деактивации и перезапуска юнитов предназначены команды `systemctl start`, `systemctl stop` и `systemctl restart`. Однако если вы изменили файл конфигурации юнита, можете указать `systemd` перезагрузить файл одним из двух способов:

- `systemctl reload` *`unit`* — перезагружает только конфигурацию для юнита unit;
- `systemctl daemon-reload` — перезагружает все конфигурации юнитов.

Запросы на активацию, повторную активацию и перезапуск юнитов в `systemd` называются *заданиями (jobs)* и по сути представляют собой изменения состояния юнитов. Вы можете проверить текущие задания в системе с помощью команды:

```
$ systemctl list-jobs
```

Если система работала в течение некоторого времени, можно ожидать, что активных заданий не будет, поскольку все активации, необходимые для запуска системы, должны быть уже завершены. Однако во время загрузки вы можете войти в систему достаточно быстро для того, чтобы увидеть задания юнитов, которые запускаются довольно медленно, 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 этом случае задание `76` — запуск юнита `sendmail.service` — занимает очень много времени. Другие перечисленные задания находятся в режиме ожидания, скорее всего, потому что ждут, когда будет выполнено задание `76`. Когда `sendmail.service` завершит запуск и будет полностью активен, задание `76` будет завершено, остальные задания также завершатся, а список заданий окажется пустым.

ПРИМЕЧАНИЕ

Термин «задание» может сбивать с толку, особенно потому, что некоторые другие системы init используют его для обозначения функций, больше похожих на юниты systemd. Задания юнитов не имеют никакого отношения и к управлению заданиями оболочки.

См. раздел 6.6, чтобы узнать, как выключить и перезагрузить систему.

Добавление юнитов в систему systemd

Добавление юнитов в `systemd` в первую очередь связано с созданием, затем активацией и, возможно, включением юнит-файлов. Обычно необходимо размещать свои юнит-файлы в системном каталоге конфигурации (`/etc/systemd/system`), чтобы не путать их с другими файлами дистрибутива и чтобы дистрибутив не перезаписывал их при обновлении.

Поскольку создавать целевые юниты, которые на самом деле ничего не делают и не вмешиваются в вашу систему, довольно легко, попробуйте это сделать. Чтобы создать две цели, одна из которых зависит от другой, выполните следующие действия.

1. Создайте юнит-файл с именем `test1.target` в файле `/etc/systemd/system`:

   ```
   [Unit]
   Description=test 1
   ```

2. Создайте файл `test2.target` с зависимостью от `test1.target`:

   ```
   [Unit]
   Description=test 2
   Wants=test1.target
   ```

Ключевое слово `Wants` здесь определяет зависимость, которая вызывает активацию `test1.target` при активации `test2.target`. Активируйте юнит `test2.target`, чтобы увидеть его в действии:

```
# systemctl start test2.target
```

3. Убедитесь, что оба юнита активны:

```
# systemctl status test1.target test2.target
· test1.target - test 1
  Loaded: loaded (/etc/systemd/system/test1.target; static; vendor preset: enabled)
  Active: active since Tue 2019-05-28 14:45:00 EDT; 16s ago

May 28 14:45:00 duplex systemd[1]: Reached target test 1.

· test2.target - test 2
  Loaded: loaded (/etc/systemd/system/test2.target; static; vendor preset: enabled)
  Active: active since Tue 2019-05-28 14:45:00 EDT; 17s ago
```

4. Если в вашем юнит-файле есть секция `[Install]`, необходимо «включить» юнит перед активацией:

```
# systemctl enable unit
```

Раздел `[Install]` — это еще один способ создания зависимости. Рассмотрим его (и зависимости в целом) более подробно в подразделе 6.3.6.

Удаление юнитов из системы systemd

Чтобы удалить юнит, выполните следующие действия.

1. Деактивируйте юнит, если это необходимо:

```
# systemctl stop unit
```

2. Если в вашем юнит-файле есть раздел `[Install]`, отключите юнит, чтобы удалить любые символические ссылки, созданные системой зависимостей:

```
# systemctl disable unit
```

Теперь можно удалить юнит-файл.

ПРИМЕЧАНИЕ

Отключение устройства, которое включено неявно (то есть не имеет раздела [Install]), не даст никакого эффекта.

6.3.5. Отслеживание и синхронизация процессов systemd

Команде `systemd` требуются разумный объем информации и контроль над каждым процессом, который она запускает. Изначально реализовать это было непросто. Служба может запускаться по-разному: создавать новые экземпляры самой себя

или даже демонизироваться и отделяться от исходного процесса. К тому же неизвестно, сколько подпроцессов может породить сервер.

Чтобы легко управлять активированными юнитами, `systemd` использует ранее упомянутые группы управления `cgroups` — функцию ядра Linux, которая позволяет более точно отслеживать иерархию процессов. Применение `cgroups` также помогает свести к минимуму работу, которую разработчик или администратор должен выполнить, чтобы создать рабочий юнит-файл. В `systemd` не нужно учитывать любое возможное поведение службы при запуске — нужно лишь знать, разветвляется ли процесс запуска службы. Используйте параметр `Type` в файле служебного юнита, чтобы указать поведение при запуске. Существуют два основных типа запуска:

- `Type=simple` — процесс службы не разветвляется и не завершается — он остается основным процессом;
- `Type=forking` — процесс службы разветвляется, и системы ожидают, что исходный процесс завершится. После этого `systemd` предполагает, что служба готова к работе.

Параметр `Type=simple` не учитывает тот факт, что запуск службы может занять некоторое время, и в результате `systemd` не знает, когда запускать зависимые юниты, которым данная служба необходима для работы. Один из способов решить эту проблему — использовать отложенный запуск (см. подраздел 6.3.7). Однако некоторые стили запуска `Type` могут указывать на то, что сама служба уведомит `systemd`, когда она будет готова:

- `Type=notify` — когда служба готова к работе, она отправляет особое уведомление для `systemd` вызовом специальной функции;
- `Type=dbus` — когда вы будете готовы, служба зарегистрируется на D-Bus (Desktop Bus, шина рабочего стола).

Другой стиль запуска службы использует `Type=oneshot`, здесь процесс службы полностью завершается без дочерних процессов после запуска. Это похоже на действие `Type=simple`, за исключением того, что `systemd` не считает службу запущенной до тех пор, пока процесс обслуживания не завершится. Любые строгие зависимости (с которыми мы вскоре познакомимся) не будут запускаться до этого завершения. Служба, использующая `Type=oneshot`, также получает директиву `RemainAfterExit=yes` по умолчанию, чтобы `systemd` считала службу активной даже после завершения ее процессов.

Последний вариант — `Type=idle`. Он работает так же, как и `Type=simple`, но предписывает `systemd` не запускать службу до тех пор, пока все активные задания не будут завершены. Суть этого способа заключается в том, чтобы просто отложить запуск службы до тех пор, пока другие службы не будут запущены, чтобы избежать перекрывания выводов различных служб. Помните, что как только служба запущена, задание `systemd`, которое ее запустило, завершается, поэтому ожидание завершения всех прочих заданий гарантирует, что никакие иные процессы не запускаются.

Если вас интересует, как работают группы управления `cgroups`, мы рассмотрим их более подробно в разделе 8.6.

6.3.6. Зависимости systemd

Гибкая система зависимостей во время загрузки и работы требует определенной степени сложности, поскольку чрезмерно строгие правила могут привести к низкой производительности и нестабильности системы. Предположим, что вы хотите отобразить приглашение для входа в систему после запуска сервера базы данных, поэтому определяете строгую зависимость от приглашения для входа на этот сервер. Это означает, что в случае сбоя сервера БД запрос на вход также не будет выполнен и вы даже не сможете зайти на сервер, чтобы устранить проблему!

Задачи во время загрузки Unix довольно отказоустойчивы и часто могут завершаться сбоем, не вызывая серьезных проблем для стандартных служб. Например, если вы удалили системный диск с данными, но оставили его запись `/etc/fstab` (или юнит монтирования в `systemd`), монтирование файловой системы во время загрузки завершится сбоем. Хотя эта ошибка может повлиять на серверы приложений (например, веб-серверы), обычно она не влияет на стандартную работу системы.

Чтобы остаться гибкой и отказоустойчивой, система `systemd` предлагает несколько типов и стилей зависимостей. Сначала рассмотрим основные типы, обозначенные по их ключевым словам синтаксиса.

- `Requires` — строгие зависимости. При активации юнита с зависимостью `Requires` система `systemd` пытается активировать юнит зависимости. Если юнит зависимости выходит из строя, система отключает и зависимый юнит.
- `Wants` — зависимости только для активации. При активации юнита `systemd` активирует зависимости юнита `Wants`, но она не реагирует, если эти зависимости не работают.
- `Requisite` — юниты, которые уже должны быть активны. Перед активацией юнита с зависимостью `Requisite` система `systemd` сначала проверяет состояние зависимости. Если она не была активирована, `systemd` завершается сбоем при активации юнита с зависимостью.
- `Conflicts` — отрицательные зависимости. При активации юнита с зависимостью `Conflict` система `systemd` автоматически деактивирует противоположную зависимость, если та активна. Одновременная активация конфликтующих юнитов не удастся.

Тип зависимости `Wants` особенно важен, поскольку он не распространяет сбои на другие юниты. На странице руководства `systemd.service(5)` написано, что по возможности указывать зависимости следует именно с помощью `Wants`, и легко понять почему. Это создает более надежную систему, предоставляя вам преимущества традиционной команды `init`, когда сбой более раннего компонента запуска не обязательно запрещает запуск более поздних компонентов.

Вы можете просматривать зависимости юнитов с помощью команды `systemctl`, если укажете тип зависимости, например `Wants` или `Requires`:

```
# systemctl show -p type unit
```

Порядок выполнения юнитов

Синтаксис зависимостей, который мы рассматривали ранее, явно не имел особого порядка юнитов. Например, активация большинства служебных юнитов с зависимостями `Requires` или `Wants` приводит к тому, что эти юниты запускаются одновременно. Это логично, потому что вам необходимо как можно быстрее запустить как можно больше служб, чтобы сократить время загрузки. Однако бывают ситуации, когда один юнит должен загружаться после другого. Например, в системе, показанной на рис. 6.1, юнит `default.target` запускается после `multi-user.target` (порядок там не показан).

Чтобы активировать юниты в определенном порядке, используйте следующие модификаторы зависимостей:

- `Before`. Текущий юнит активируется раньше перечисленных юнитов. Например, если `Before=bar.target` отображается в `foo.target`, `systemd` активирует `foo.target` перед `bar.target`.
- `After`. Текущий юнит активируется после перечисленных юнитов.

При использовании упорядочивания юнитов `systemd` ожидает, пока юнит получит активный статус, прежде чем активировать зависимые юниты.

Зависимости по умолчанию и неявные зависимости

По мере изучения зависимостей (особенно с помощью команды `systemd-analyze`) вы начнете замечать, что некоторые юниты приобретают зависимости, которые явно не указаны в их файлах или других видимых механизмах. Вы, скорее всего, столкнетесь с этим в целевых юнитах с зависимостями `Wants`: система `systemd` добавляет модификатор `After` рядом с любым юнитом, указанным в качестве зависимости `Wants`. Эти дополнительные зависимости являются внутренними для `systemd`, рассчитываются во время загрузки и не сохраняются в файлах конфигурации.

Модификатор `After` называется *зависимостью по умолчанию*, он автоматически добавляется к конфигурации юнита и предназначен для предотвращения распространенных ошибок и сохранения небольших файлов юнитов. Эти зависимости различные для разных типов юнита. Например, `systemd` добавляет разные зависимости по умолчанию для целевых и служебных юнитов. Эти различия перечислены в разделах `DEFAULT DEPENDENCIES` на страницах руководства `systemd.service(5)` и `systemd.target(5)`.

Вы можете отключить зависимость по умолчанию в юните, добавив параметр `DefaultDependencies=no` в его файл конфигурации.

Условные зависимости

Вы можете использовать несколько параметров условной зависимости для тестирования различных состояний операционной системы, а не юнитов `systemd`, например:

- `ConditionPathExists=p` — истинно, если в системе существует путь (файл) `p`;
- `ConditionPathIsDirectory=p` — истинно, если `p` — это каталог;
- `ConditionFileNotEmpty=p` — истинно, если `p` — это файл и он ненулевой длины.

Если условная зависимость в юните ложна при попытке `systemd` активировать юнит, то он не активируется, хотя это относится только к юниту, в котором отображается условная зависимость. То есть если вы активируете юнит с условной и несколькими другими зависимостями, `systemd` попытается активировать эти зависимости от юнита независимо от того, истинно или ложно это условие.

Прочие зависимости в основном являются вариациями предыдущих. Например, зависимость `RequiresOverridable` аналогична зависимости `Requires` при нормальной работе по умолчанию, однако она действует как зависимость `Wants`, если устройство активировано вручную. Полный список см. на странице руководства `systemd.unit(5)`.

Секция [Install] и включение юнитов

До сих пор мы говорили о том, как определить зависимости в файле конфигурации зависимого юнита. То же самое можно выполнить в обратном порядке, то есть указав зависимый юнит в зависимом юнит-файле. Вы можете сделать это, добавив параметр `WantedBy` или `RequiredBy` в секции `[Install]`. Этот механизм позволяет изменять момент запуска юнита без изменения дополнительных файлов конфигурации (например, если вы не хотите редактировать файл системного юнита).

Чтобы увидеть, как это работает, рассмотрим примеры юнитов, приведенные в подразделе 6.3.4. Возьмем два юнита, `test1.target` и `test2.target`, причем `test2.target` имеет Wants-зависимость от `test1.target`. Мы можем изменить их так, чтобы `test1.target` выглядел следующим образом:

```
[Unit]
Description=test 1

[Install]
WantedBy=test2.target
```

А `test2.target` будет выглядеть так:

```
[Unit]
Description=test 2
```

Поскольку теперь есть юнит с разделом [Install], вам необходимо *включить* юнит с помощью команды systemctl, прежде чем вы сможете его запустить. Рассмотрим это на примере с test1.target:

```
# systemctl enable test1.target
Created symlink /etc/systemd/system/test2.target.wants/test1.target → /etc/
systemd/system/test1.target.
```

Обратите внимание на эффект включения юнита в выводе, который создал символическую ссылку в подкаталоге .wants, соответствующем зависимому юниту (в данном случае test2.target). Теперь можно запустить оба юнита одновременно с помощью команды systemctl start test2.target, поскольку зависимость установлена.

ПРИМЕЧАНИЕ

Включение юнита не активирует его.

Чтобы отключить юнит (и удалить символическую ссылку), используйте команду systemctl следующим образом:

```
# systemctl disable test1.target
Removed /etc/systemd/system/test2.target.wants/test1.target.
```

Два юнита из приведенного ранее примера позволяют поэкспериментировать с различными сценариями запуска. Например, посмотрите, что происходит при попытке запустить только test1.target или запустить test2.target без включения test1.target. Или попробуйте изменить WantedBy на RequiredBy. (Помните, что можно проверить состояние юнита с помощью команды systemctl status.)

Во время нормальной работы система игнорирует раздел [Install] в юните, но отмечает его наличие и по умолчанию считает юнит отключенным. «Включенный» юнит включается и после перезагрузки.

Секция [Install] обычно отвечает за каталоги .wants и .require в системном каталоге конфигурации (/etc/systemd/system). Однако каталог конфигурации юнита ([/usr]/lib/systemd/system) также содержит каталоги .wants, и вы можете добавить ссылки, которые не соответствуют разделам [Install] в юнит-файлах. Подобные добавления вручную — это простой способ добавить зависимость без изменения файла юнита, который может быть перезаписан в будущем (например, путем обновления программного обеспечения), но не стоит этим увлекаться, поскольку добавление вручную трудно отследить.

6.3.7. Запуск по запросу и параллелизация ресурсов в systemd

Одна из особенностей системы systemd — возможность откладывать запуск юнита до тех пор, пока он не станет жизненно необходимым для ее работы. Настройка подобного запуска происходит так:

1. Вы создаете юнит `systemd` (назовем его `Unit A`) для системной службы.
2. Определяете системный ресурс, такой как сетевой порт/сокет, файл или устройство, которое `Unit A` будет использовать.
3. Вы создаете другой юнит `systemd`, `Unit R`, для представления этого ресурса. Эти юниты классифицируются по типам: юниты сокетов, юниты путей и юниты устройств.
4. Определяете взаимосвязь между `Unit A` и `Unit R`. Обычно она возникает на основе названий юнитов, но может быть и явной, как мы вскоре увидим.

Далее происходит следующее:

1. При активации `Unit R` система `systemd` осуществляет мониторинг данного ресурса.
2. Когда что-либо пытается получить доступ к ресурсу, `systemd` блокирует ресурс и его входные данные буферизуются.
3. `systemd` активирует `Unit A`.
4. Когда `Unit A` будет готов, служба `Unit A` возьмет под контроль ресурс, считает буферизованные входные данные и начнет работу.

Важные замечания

- Убедитесь, что юнит ресурсов охватывает все ресурсы, предоставляемые службой. Это просто, так как большинство служб имеют только одну точку доступа.
- Убедитесь, что юнит ресурсов привязан к нужному служебному юниту. Эта связь может быть как неявной, так и явной, и в некоторых случаях многие параметры представляют различные способы, с помощью которых `systemd` может передавать данные в служебный юнит.
- Не все серверы знают, как взаимодействовать с юнитами ресурсов, которые может предоставить `systemd`.

Если вам известно, как работают традиционные утилиты, такие как `inetd`, `xinetd` и `automount`, то вы заметите много общего. Действительно, концепция ничем не отличается, `systemd` даже включает поддержку юнитов автоматического монтирования (automount).

Пример сокет-юнита и служебного юнита

Рассмотрим пример простой сетевой `echo`-службы. Информация в данном разделе может быть не слишком понятна, если вы еще не изучили работу TCP, портов и прослушивания из главы 9 и сокетов из главы 10, но основная идея довольно проста.

Смысл службы `echo` состоит в том, чтобы повторять все, что отправляет сетевой клиент после подключения, в нашем случае он будет прослушивать TCP-порт 22222.

Мы начнем создавать его с сокет-юнита, чтобы предоставить нужный порт, как показано в следующем файле `echo.socket`:

```
[Unit]
Description=echo socket

[Socket]
ListenStream=22222
Accept=true
```

Обратите внимание: в служебном юните нет упоминания о том, что этот сокет поддерживает порты внутри файла юнита. Итак, что это за соответствующий файл служебного юнита? Он называется `echo@.service`. Ссылка устанавливается в соответствии с соглашением об именовании: если файл служебного юнита имеет тот же префикс, что и файл `.socket` (в данном случае `echo`), `systemd` знает, что нужно активировать этот юнит, когда он в работе. В этом случае `systemd` создает экземпляр `echo@.service`, когда на `echo.socket` возникает активность. Вот как выглядит служебный юнит-файл `echo@.service`:

```
[Unit]
Description=echo service

[Service]
ExecStart=/bin/cat
StandardInput=socket
```

ПРИМЕЧАНИЕ

Если вам не нравится неявная активация юнитов на основе префиксов или нужно связать юниты с разными префиксами, можете использовать явный параметр в юните, определяющем ваш ресурс. Например, задействуйте Socket=bar.socket внутри foo.service, чтобы bar.socket передавал свой сокет в foo.service.

Чтобы запустить этот юнит, необходимо запустить юнит `echo.socket`:

```
# systemctl start echo.socket
```

Теперь вы можете протестировать службу, подключившись к локальному TCP-порту 22222 с помощью утилиты `telnet`. Служба повторяет то, что вы вводите, например:

```
$ telnet localhost 22222
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi there.
Hi there.
```

Когда вам наскучит это занятие, нажмите сочетание клавиш Ctrl+] на отдельной строке, а затем нажмите Ctrl+D, чтобы вернуться в свою оболочку. Чтобы прекратить деятельность службы, остановите сокет следующим образом:

```
# systemctl stop echo.socket
```

ПРИМЕЧАНИЕ

Команда telnet может не быть установлена по умолчанию в вашем дистрибутиве.

Экземпляры и передача управления

Так как юнит `echo@.service` поддерживает несколько одновременных экземпляров, в имени есть символ `@` (напомним, что спецификатор `@` означает параметризацию). Зачем же могут понадобиться несколько экземпляров? Допустим, у вас есть несколько сетевых клиентов, подключающихся к службе одновременно, и вы хотите, чтобы каждое соединение имело собственный экземпляр. В этом случае служебный юнит должен поддерживать несколько экземпляров, потому что в `echo.socket` включен параметр `Accept=true`. Он предписывает `systemd` не только прослушивать порт, но и принимать входящие соединения, а затем передавать входящее соединение юниту службы, создавая отдельный экземпляр для каждого соединения. Каждый экземпляр считывает данные из подключения в качестве стандартного ввода, но ему не обязательно знать, что они поступают из сетевого подключения.

ПРИМЕЧАНИЕ

Большинство сетевых подключений требуют большей гибкости, чем простой шлюз для стандартного ввода и вывода, поэтому не ожидайте, что сможете создавать сложные сетевые службы с помощью служебного юнит-файла, такого как echo@.service из приведенного ранее примера.

Если служебный юнит может выполнить работу по приему соединения, не включайте символ `@` в имя файла этого юнита и не ставьте параметр `Accept=true` в сокет-юнит. В этом случае служебный юнит получает полный контроль над сокетом от `systemd`, который в свою очередь не пытается снова прослушивать сетевой порт, пока служебный юнит не завершит работу.

Множество различных ресурсов и вариантов передачи управления служебных юнитов затрудняют их разбиение на категории. Кроме того, документация о параметрах размещена на нескольких страницах руководства. Для юнитов, ориентированных на ресурсы, изучите страницы `systemd.socket(5)`, `systemd.path(5)` и `systemd.device(5)`. Стоит также прочитать страницу `systemd.exec(5)`, где говорится о том, что служебный юнит может ожидать получения ресурса после активации.

Оптимизация загрузки с помощью дополнительных юнитов

Конечная цель `systemd` — упростить порядок зависимостей и ускорить время загрузки. Юниты ресурсов, такие как юниты сокетов, позволяют это осуществить, и данный способ аналогичен запуску по запросу. То есть у вас по-прежнему есть служебный и вспомогательный юниты, представляющие предлагаемый ресурс служебного юнита. Исключением в данном случае является то, что `systemd` запускает служебный юнит уже после того, как он активирует вспомогательный юнит, не дожидаясь запроса.

Суть этой схемы заключается в том, что для запуска во время загрузки основных служебных юнитов, таких как `systemd-journald.service`, требуется некоторое время, и от них зависят многие другие юниты. Однако `systemd` может очень быстро предоставить необходимый юнит ресурса (например, сокета), а затем немедленно активировать не только нужный юнит, но и любые другие юниты, зависящие от него. Как только необходимый юнит готов, он берет на себя управление ресурсом.

На рис. 6.2 показано, как это работает в традиционной последовательной системе. На временно́й шкале загрузки служба E предоставляет необходимый ресурс R. Службы A, B и C зависят от этого ресурса (но не друг от друга) и должны дождаться запуска службы E. Поскольку система не начнет новую службу, пока не завершит предыдущую, требуется довольно много времени, чтобы приступить к запуску службы C.

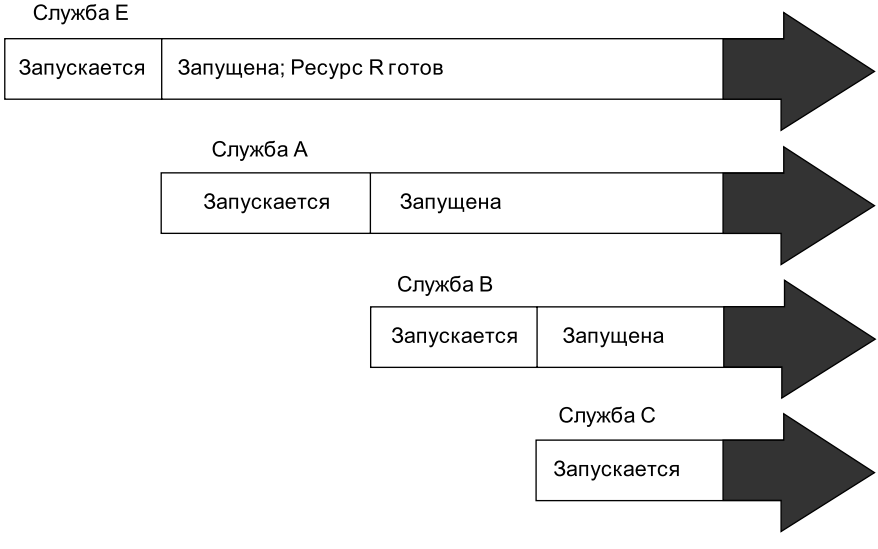


Рис. 6.2. Временна́я шкала последовательной загрузки с зависимостью от ресурсов

На рис. 6.3 показана похожая конфигурация загрузки системы. Службы представлены юнитами A, B, C и E, при этом новый юнит `R` представляет ресурс, который предоставляется юнитом E. Поскольку `systemd` может обеспечивать интерфейс для юнита R во время запуска юнитов, юниты A, B, C и E могут быть запущены одновременно. Когда все будет готово, на смену юниту E придет юнит R. Интересным моментом здесь является то, что юнитам A, B или C может не потребоваться доступ к ресурсу, предоставляемому юниту R, до завершения запуска. В этот момент пользователь дает им *возможность* получить доступ к ресурсу как можно скорее.

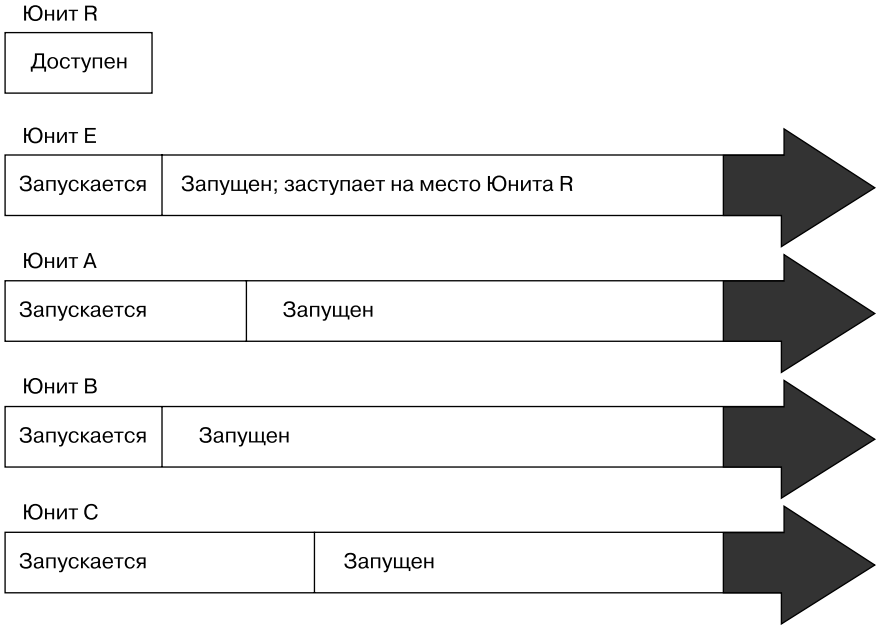


Рис. 6.3. Временна́я шкала загрузки systemd с юнитом ресурсов

ПРИМЕЧАНИЕ

Когда вы реализуете параллельный запуск подобным образом, есть вероятность, что система временно замедлится из-за одновременного запуска большого количества юнитов.

Здесь вывод таков: хотя в данном случае вы не осуществляете запуск юнита по запросу, вы используете функции, которые делают его возможным. Для примера изучите юниты конфигурации `journald` и D-Bus на машине, работающей под управлением `systemd`, — они, скорее всего, будут загружаться параллельно.

6.3.8. Вспомогательные компоненты systemd

По мере роста популярности `systemd` стала включать в себя поддержку нескольких задач, не связанных с запуском служб и управлением ими, как напрямую, так и через вспомогательные уровни совместимости. Вы можете увидеть множество программ в каталоге `/lib/systemd` — это исполняемые файлы, связанные с данными функциями.

Перечислим несколько системных служб:

- `Udevd` — описана в главе 3, это часть `systemd`.
- `journald` — служба ведения журнала, которая обрабатывает несколько различных механизмов, включая традиционную службу системного журнала Unix — `syslog`. Подробнее вы узнаете об этом из главы 7.

- `resolved` — демон кэширования службы имен для DNS, вы узнаете о нем из главы 9.

Все исполняемые файлы для этих служб имеют префикс `system-`. Например, интегрированный в `systemd` демон `udevd` называется `systemd-udevd`.

Если заглянуть глубже, то можно обнаружить, что некоторые из этих программ — довольно простые обертки. Их функция заключается в запуске стандартных системных утилит и уведомлении `systemd` о результатах. Один из примеров — утилита `systemd-fsck`.

Если вы видите в `/lib/systemd` программу, которую не можете идентифицировать, проверьте страницу руководства. Велика вероятность, что в ней будет описана не только утилита, но и тип юнита, который она дополняет.

6.4. Уровни выполнения в System V

Теперь, изучив систему `systemd` и то, как она работает, рассмотрим некоторые аспекты традиционной инициализации init в System V. В любой момент времени в системе Linux выполняется определенный базовый набор процессов, таких как `crond` и `udevd`. Для команды System V init это состояние машины называется ее *уровнем выполнения (runlevel)*, который обозначается числом от 0 до 6. Система проводит бóльшую часть своего времени на одном уровне выполнения, но когда вы выключаете машину, init переключается на другой уровень, чтобы упорядоченно завершить системные службы и остановить ядро.

Можете проверить уровень выполнения своей системы с помощью команды `who -r` следующим образом:

```
$ who -r
run-level 5 2019-01-27 16:43
```

Вывод сообщает, что текущий уровень выполнения — 5, а также показывает дату и время, когда он был установлен.

Уровни выполнения служат различным целям, наиболее распространенной из которых является различение состояний запуска, выключения системы, однопользовательского режима и режима консоли. Например, большинство систем традиционно использовали уровни выполнения 2–4 для текстовой консоли, уровень выполнения 5 означает, что система запускает вход в графический интерфейс.

Тем не менее концепция уровней выполнения уходит в прошлое. `systemd` поддерживает их, но считает устаревшими для использования, в качестве определяющих состояния системы, предпочитая им целевые юниты. Для `systemd` уровни выполнения существуют в основном для запуска служб, поддерживающих только сценарии System V init.

6.5. System V init

System V init — одна из старейших среди используемых в Linux, ее основная идея заключается в поддержке упорядоченной загрузки на различных уровнях выполнения с тщательно продуманной последовательностью запуска. В большинстве серверных и настольных систем System V init сейчас встречается редко, но вы можете столкнуться с ней в версиях RHEL до 7.0, а также во встроенных средах Linux, например в маршрутизаторах и телефонах. Кроме того, некоторые старые пакеты могут предоставлять только сценарии запуска, предназначенные для System V init, `systemd` может обрабатывать их в режиме совместимости, который обсудим в подразделе 6.5.5. Мы рассмотрим здесь основы, но имейте в виду, что в реальности вы можете и не столкнуться ни с чем из описанного в этом разделе.

Установка System V `init` состоит из двух компонентов: центрального файла конфигурации и большого набора сценариев загрузки, дополненных фермой символических ссылок. Все начинается с файла конфигурации `/etc/inittab`. Если у вас есть System V init, найдите в файле `inittab` строку, подобную следующей:

```
id:5:initdefault:
```

Она указывает на то, что уровень выполнения по умолчанию равен 5.

Все строки в `inittab` имеют четыре поля, разделенных двоеточиями и стоящих в таком порядке:

1. Уникальный идентификатор (короткая строка, `id` в предыдущем примере).
2. Применимые номера уровней выполнения.
3. Действие, которое должно выполнить init (уровень выполнения по умолчанию 5 в предыдущем примере).
4. Команда для выполнения (необязательно).

Чтобы увидеть, как работают команды в файле `inittab`, рассмотрите пример:

```
l5:5:wait:/etc/rc.d/rc 5
```

Эта конкретная строка важна, потому что она запускает бóльшую часть конфигурации системы и служб. В примере действие ожидания `wait` определяет, когда и как System V init выполнит команду: запускается `/etc/rc.d/rc 5` один раз при входе на уровень выполнения 5, а затем необходимо дождаться завершения этой команды, прежде чем делать что-то еще. Команда `rc5` выполняет в файле `/etc/rc5.d` все, что начинается с числа (в числовом порядке). В ближайшее время мы рассмотрим это более подробно.

Далее рассмотрены некоторые из наиболее распространенных действий `inittab` в дополнение к `initdefault` и `wait`.

respawn

Действие `respawn` указывает init выполнить следующую команду и, если команда завершит выполнение, запустить ее снова. Скорее всего, в файле `inittab` вы увидите что-то подобное:

```
1:2345:respawn:/sbin/mingetty tty1
```

Программы `getty` обеспечивают приглашения входа в систему. Приведенная строка используется для первой виртуальной консоли (`/dev/tty1`), которую вы видите при нажатии клавиш Alt+F1 или Ctrl+Alt+F1 (см. подраздел 3.4.7). Действие `respawn` возвращает приглашение на вход после выхода из системы.

ctrlaltdel

Действие `ctrlaltdel` управляет действиями системы при нажатии сочетания клавиш Ctrl+Alt+Del на виртуальной консоли. В большинстве систем это своего рода команда перезагрузки с использованием команды `shutdown` (обсудим в разделе 6.6).

sysinit

Действие команды `sysinit` — это первое, что должен выполнить init при запуске, до входа на какие-либо уровни выполнения.

ПРИМЕЧАНИЕ

Дополнительные доступные действия см. на странице руководства inittab(5).

6.5.1. System V init: последовательность команд при запуске

Теперь посмотрим, как System V init запускает системные службы непосредственно перед тем, как позволить пользователю войти в систему. Рассмотрим вывод `inittab` из приведенного ранее примера:

```
l5:5:wait:/etc/rc.d/rc 5
```

Эта короткая строка запускает множество других программ. Расшифровывается `rc` как *run commands — команды запуска*, которые многие называют сценариями, программами или службами. Но где же находятся эти команды?

Цифра `5` в этой строке говорит об уровне выполнения 5. Команды, вероятно, находятся либо в `/etc/rc.d/rc5.d`, либо в `/etc/rc5.d`. (Уровень выполнения 1 использует `rc1.d`, уровень выполнения 2 использует `rc2.d` и т. д.)

Например, в каталоге `rc5.d` вы можете найти следующие элементы:

```
S10sysklogd     S20ppp          S99gpm
S12kerneld      S25netstd_nfs   S99httpd
S15netstd_init S30netstd_misc S99rmnologin
S18netbase      S45pcmcia       S99sshd
S20acct         S89atd
S20logoutd S89cron
```

Команда `rc 5` запускает программы в каталоге `rc5.d`, выполняя команды в следующей последовательности:

```
S10sysklogd start
S12kerneld start
S15netstd_init start
S18netbase start
--пропуск--
S99sshd start
```

Обратите внимание на аргумент `start` в каждой команде. Прописная буква `S` в имени команды означает, что она должна выполняться в режиме запуска (Start mode), а число (от 00 до 99) определяет, в какой последовательности `rc` запускает команды. Команды `rc*.d` обычно представляют собой сценарии оболочки, которые запускают программы в `/sbin` или `/usr/sbin`.

Вы можете выяснить, что делает конкретная команда, просмотрев сценарий с помощью команды `less` или другой программы с функцией пагинации.

ПРИМЕЧАНИЕ

Некоторые каталоги rc*.d содержат команды, которые начинаются с K (для режима kill или принудительной остановки (stop mode)). В этом случае rc запускает команду с аргументом stop вместо start. Скорее всего, вы столкнетесь с командами с прописной K на уровнях выполнения, завершающих работу системы.

Эти команды можно ввести вручную, однако обычно это делается через каталог `init.d` вместо каталогов `rc*.d`, которые мы рассмотрим далее.

6.5.2. Ферма ссылок System V init

Содержимое каталогов `rc*.d` на самом деле является символическими ссылками на файлы в каталоге `init.d`. Если ваша цель в том, чтобы взаимодействовать со службами, добавлять, удалять или изменять их в каталогах `rc*.d`, вам необходимо понимать, как работают эти символические ссылки. Длинный список каталогов, таких как `rc5.d`, отображает структуру, подобную следующей:

```
lrwxrwxrwx . . . S10sysklogd -> ../init.d/sysklogd
lrwxrwxrwx . . . S12kerneld -> ../init.d/kerneld
lrwxrwxrwx . . . S15netstd_init -> ../init.d/netstd_init
```

```
lrwxrwxrwx . . . S18netbase -> ../init.d/netbase
--пропуск--
lrwxrwxrwx . . . S99httpd -> ../init.d/httpd
--пропуск--
```

Большое количество символических ссылок в нескольких подкаталогах, подобных этому, называется *фермой ссылок (link farm)*. Дистрибутивы Linux содержат эти ссылки, чтобы они могли использовать одни и те же сценарии запуска для всех уровней выполнения. Это не обязательное требование для систем, однако оно упрощает организацию системы.

Запуск и остановка служб

Чтобы запускать и останавливать службы вручную, применяйте скрипт в каталоге `init.d`. Например, один из способов запустить программу веб-сервера httpd вручную — это запустить скрипт `init.d/httpd start`. Аналогично, чтобы отключить запущенную службу, можно использовать аргумент `stop` (например, `httpd stop`).

Изменение последовательности загрузки

Изменение последовательности загрузки в System V `init` обычно выполняется изменением фермы ссылок. Наиболее распространенным изменением является предотвращение выполнения одной из команд в каталоге `init.d` на определенном уровне выполнения. Однако в таком случае вы должны быть осторожны. Например, вы можете удалить символическую ссылку в соответствующем каталоге `rc*.d`. Но если когда-нибудь ее понадобится вернуть, у вас могут возникнуть проблемы с восстановлением ее точного названия. Один из лучших подходов — добавить символ подчеркивания (_) в начало имени ссылки, например:

```
# mv S99httpd _S99httpd
```

Это приведет к тому, что команда `rc` проигнорирует `_S99httpd`, поскольку имя файла больше не начинается с `S` или `K`, но исходное имя по-прежнему указывает на его назначение.

Чтобы добавить службу, создайте сценарий, подобный находящимся в каталоге `init.d`, а затем создайте символическую ссылку в правильном каталоге `rc*.d`. Самый простой способ сделать это — скопировать и изменить один из сценариев, уже включенных в `init.d` (дополнительную информацию о сценариях оболочки см. в главе 11).

При добавлении службы выберите в последовательности загрузки подходящее место, чтобы запустить ее. Если служба запускается слишком рано, она может не сработать из-за зависимости от какой-либо другой службы. Большинство системных администраторов предпочитают давать несущественным службам номера от 90, таким образом, эти службы выполняются после основных системных служб.

6.5.3. Команда run-parts

Механизм, который System V init задействует для запуска сценариев `init.d`, нашел применение во многих системах Linux независимо от того, используют ли они System V init. Это утилита под названием `run-parts`, и единственное, что она делает, — запускает множество исполняемых программ в заданном каталоге в определенном порядке. Эта команда работает как пользователь, который вводит команду `ls` в какой-то каталог, а затем просто запускает все программы, перечисленные в выходных данных.

По умолчанию команда запускает все программы в каталоге, но у вас есть возможность выбрать определенные и игнорировать другие. В некоторых дистрибутивах не нужно контролировать запускаемые программы. Например, Fedora поставляется с очень простой утилитой `run-parts`.

Другие дистрибутивы, такие как Debian и Ubuntu, имеют более сложные утилиты `run-parts`. Их функции включают возможность запускать программы на основе регулярного выражения (например, выражения `S[0-9]{2}` для запуска всех сценариев запуска в каталоге уровня выполнения `/etc/init.d`) и передавать аргументы программам. Эти возможности позволяют запускать и останавливать уровни выполнения System V с помощью одной команды.

На самом деле вам не нужно разбираться в деталях того, как применять команду `run-parts`, большинство пользователей даже не знают, что она существует. Главное — помнить, что она время от времени появляется в сценариях и существует исключительно для запуска программ из указанного каталога.

6.5.4. Управление System V init

Иногда необходимо вручную подтолкнуть init к тому, чтобы она переключила уровень выполнения, перечитала свою конфигурацию или выключила систему. Для управления System V init используется команда `telinit`. Например, чтобы переключиться на уровень выполнения 3, введите:

```
# telinit 3
```

При переключении уровней выполнения init попытается отключить любые процессы, отсутствующие в файле `inittab` для нового уровня выполнения, поэтому будьте осторожны при смене уровней выполнения.

Когда вам нужно добавить или удалить задания либо внести другие изменения в файл `inittab`, вы должны сообщить init об изменении и попросить перезагрузить файл. Используйте команду `telinit` следующим образом:

```
# telinit q
```

Вы также можете с помощью команды `telinit s` переключиться в однопользовательский режим.

6.5.5. Совместимость systemd и System V

Особенность, которая отличает `systemd` от других init-систем нового поколения, заключается в том, что она пытается максимально отследить все службы, запускаемые совместимыми с System V сценариями инициализации. Вот как это работает:

1. Сначала `systemd` активирует уровень выполнения `<N>.target`, где `N` — уровень выполнения.
2. Для каждой символьной ссылки в `/etc/rc<N>.d` `systemd` идентифицирует скрипт в `/etc/init.d`.
3. `systemd` связывает имя сценария со служебным юнитом (например, `/etc/init.d/foo` будет `foo.service`).
4. `systemd` активирует служебный юнит и запускает сценарий с аргументом `start` или `stop`, основанным на его имени в `rc<N>.d`.
5. `systemd` пытается связать любые процессы из сценария со служебным юнитом.

Поскольку система устанавливает связь с именем служебного юнита, вы можете использовать команду `systemctl` для перезапуска службы или просмотра ее статуса. Но не ждите никаких чудес от режима совместимости System V — он все равно должен запускать сценарии init последовательно.

6.6. Завершение работы системы

Команда `init` управляет тем, как система завершает работу и перезагружается. Команды для отключения системы одинаковы независимо от того, какую версию init вы запускаете. Правильный способ выключить компьютер с Linux — использовать команду `shutdown`.

Существуют два основных способа применения команды `shutdown`. Если вы *остановите (halt)* систему, она выключит систему и оставит ее в этом состоянии. Чтобы немедленно остановить машину, выполните следующее:

```
# shutdown -h now
```

В большинстве дистрибутивов и версий Linux остановка отключает питание машины. Или можете *перезагрузить (reboot)* систему. Для этого используйте команду `-r` вместо `-h`.

Выключение занимает несколько секунд. В это время избегайте сброса или выключения компьютера.

В предыдущем примере `now` — это время отключения, то есть данный момент времени. Указывать аргумент времени обязательно, и существует множество способов сделать это. Например, если вы хотите, чтобы машина выключилась когда-нибудь

в будущем, можете применить параметр `+n`, где `n` — количество минут, которые команда `shutdown` должна подождать, прежде чем выключить компьютер. Другие параметры см. на странице руководства `shutdown (8)`.

Чтобы перезагрузить систему через 10 минут, введите:

```
# shutdown -r +10
```

В Linux `shutdown` уведомляет всех, кто вошел в систему, о том, что она отключается, но непосредственной работы она выполняет мало. Если вы укажете иное время вместо `now`, команда `shutdown` создаст файл с именем `/etc/nologin`. При наличии этого файла система запрещает вход в систему всем пользователям, кроме суперпользователя.

Когда наконец наступает время завершения работы системы, `shutdown` сообщает `init` о начале процесса завершения работы. В `systemd` это означает активацию юнитов выключения, а при запуске System V — изменение уровня выполнения на 0 (остановка) или 6 (перезагрузка). Независимо от реализации или конфигурации инициализации процедура обычно выполняется следующим образом:

1. `init` просит каждый процесс полностью завершиться.
2. Если процесс не отвечает через некоторое время, `init` принудительно завершает его, сначала применив сигнал `TERM`.
3. Если сигнал `TERM` не работает, `init` использует сигнал `KILL` для любых оставшихся активными процессов.
4. Система блокирует системные файлы на своих местах и выполняет остальные приготовления к завершению работы.
5. Система демонтирует все файловые системы, кроме корневой.
6. Система заново монтирует корневую файловую систему в режиме только для чтения.
7. Система записывает все буферизованные данные в файловую систему с помощью программы `sync`.
8. На последнем шаге система указывает ядру перезагрузиться или остановиться с помощью системного вызова `reboot(2)`. Это может быть сделано посредством `init` или вспомогательной программы, например `reboot`, `halt` или `poweroff`.

Программы `reboot` и `halt` ведут себя по-разному в зависимости от того, как они вызываются, что может вызвать путаницу. По умолчанию эти программы вызывают `shutdown` с параметрами `-r` или `-h`. Однако если система уже находится на уровне выполнения остановки или перезагрузки, программы сообщают ядру, чтобы оно немедленно отключилось. Если вы действительно хотите быстро выключить систему, не беспокоясь о потенциальном ущербе, задействуйте параметр принудительного действия (force) `-f`.

6.7. Начальная файловая система оперативной памяти

Процесс загрузки Linux в основном довольно прост. Однако один компонент всегда сбивает с толку — `initramfs`, или *начальная файловая система оперативной памяти*. Она — как небольшой участок пользовательского пространства, который существует перед запуском обычного пользовательского режима. Для начала давайте обсудим, почему она вообще существует.

Проблема связана с наличием множества различных видов устаревшего оборудования. Помните, что ядро Linux не взаимодействует с интерфейсом BIOS или EFI, чтобы получить данные с дисков, поэтому для подключения корневой файловой системы ему требуется поддержка драйверов для базового механизма хранения. Например, если корень находится в массиве RAID, подключенном к стороннему контроллеру, ядру сначала нужен драйвер для него. К сожалению, существует так много драйверов контроллеров хранения, что дистрибутивы не могут вместить их все в свои ядра, поэтому многие драйверы поставляются в виде загружаемых модулей. Но загружаемые модули — это файлы, и если в вашем ядре изначально не смонтирована файловая система, оно не сможет загрузить необходимые ему модули драйверов.

Обходной путь состоит в том, чтобы собрать небольшую коллекцию модулей драйверов ядра вместе с несколькими другими утилитами в архив. Загрузчик загружает этот архив в память перед запуском ядра. При запуске ядро считывает содержимое архива во временную файловую систему RAM (init-ramfs), монтирует ее в каталог `/` и выполняет передачу управления в пользовательском режиме команде init в файловой системе `initramfs`. Затем утилиты, включенные в `initramfs`, позволяют ядру загружать необходимые модули драйверов для реальной корневой файловой системы. Наконец, утилиты монтируют реальную корневую файловую систему и запускают полноценный init.

Реализации этого процесса различаются и постоянно развиваются. В некоторых дистрибутивах init в `initramfs` представляет собой довольно простой сценарий оболочки, который запускает `udev` для загрузки драйверов, а затем монтирует реальный корневой каталог и выполняет команду init в нем. В дистрибутивах, использующих `systemd`, вы, как правило, увидите полноценную сборку `systemd` без файлов конфигурации модулей и всего нескольких файлов конфигурации `udevd`.

Одной из основных характеристик начальной файловой системы оперативной памяти, которая (пока) оставалась неизменной с момента создания, является возможность обойти ее, если в ней нет необходимости. То есть если в вашем ядре есть все драйверы, необходимые для подключения корневой файловой системы, можете опустить начальную файловую систему в конфигурации загрузчика. В таком случае время загрузки системы сокращается. Попробуйте самостоятельно отключить начальную файловую систему во время загрузки, используя редактор

меню GRUB, чтобы удалить строку `initrd`. (Лучше не экспериментировать, изменяя файл конфигурации GRUB, так как вы можете совершить ошибку, которую будет трудно исправить.) Постепенно стало немного сложнее обойти загрузку начальной файловой системы, поскольку такие функции, как монтирование по UUID, могут быть недоступны в общих ядрах дистрибутива.

Вы можете проверить содержимое своей начальной файловой системы, но вам нужно будет найти и распаковать определенные архивы. Большинство систем теперь используют архивы, созданные `mkinitramfs`, которые вы можете распаковать с помощью `unmkinitramfs`, но встречаются и устаревшие сжатые архивы `cpio` (см. страницу руководства `cpio(1)`).

Особый интерес представляет «разворот» ближе к самому концу процесса инициализации в начальной файловой системе. Эта часть отвечает за удаление содержимого временной файловой системы (для экономии памяти) и окончательное переключение на реальную корневую файловую систему.

Обычно начальная файловая система не создается вручную, так как это довольно трудоемкий процесс. Существует ряд утилит для создания начальных образов файловой системы, и ваш дистрибутив, скорее всего, поставляется с одной из них. Две наиболее распространенные утилиты — это `mkinitramfs` и `dracut`.

ПРИМЕЧАНИЕ

Термин «начальная файловая система оперативной памяти» (initramfs) относится к реализации, которая использует архив cpio в качестве источника временной файловой системы. Существует более старая версия, называемая начальным диском оперативной памяти, или initrd, которая задействует образ диска в качестве основы временной файловой системы. Она вышла из употребления, потому что поддерживать архив cpio гораздо проще. Тем не менее встречается термин initrd, которым обозначается начальная файловая система оперативной памяти на основе cpio. Часто имена файлов и файлы конфигурации все еще содержат имя initrd.

6.8. Аварийная загрузка системы и однопользовательский режим

Когда что-то не так с системой, пользователи обычно первым делом загружают систему с помощью «живого» образа дистрибутива или специального образа восстановления, например SystemRescueCD, который можно поместить на съемный носитель. «Живой» образ — это просто система Linux, которая может загружаться и запускаться без процесса установки, установочные образы большинства дистрибутивов дублируются как «живые» образы. Настройка работы системы после сбоя включает в себя следующие действия:

- проверку файловых систем;
- сброс забытого пароля;

- исправление проблем в критически важных файлах, таких как `/etc/fstab` и `/etc/passwd`;
- восстановление резервных копий.

Еще одним вариантом быстрой загрузки является *однопользовательский режим (single-user mode)*. Суть его заключается в том, что система быстро загружается в корневую оболочку, вместо того чтобы проходить через все множество служб. В System V init однопользовательский режим обычно выполняется на уровне 1. В `systemd` он представлен как `rescue.target`. Для загрузчика применяется параметр `-s`. Возможно, вам потребуется ввести пароль суперпользователя, чтобы войти в однопользовательский режим.

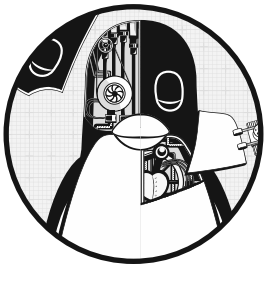
Самая большая проблема с однопользовательским режимом заключается в том, что он не особо удобен. Сеть почти наверняка будет недоступна (а если и будет доступна, то ее окажется сложно использовать), нет графического интерфейса, и даже терминал может работать неправильно. По этой причине загрузка с «живых» образов более предпочтительна и эффективна.

6.9. Что дальше?

Теперь вы ознакомились с фазами запуска ядра и пользовательского пространства системы Linux, а также с тем, как `systemd` отслеживает службы после их запуска. Далее мы немного углубимся в пространство пользователя. И начнем с ряда файлов конфигурации системы, которые все программы Linux применяют при взаимодействии с определенными элементами пользовательского пространства. Затем рассмотрим основные службы, которые запускает `systemd`.

7

Настройка системы: журналирование, системное время, пакетные задачи и пользователи



Впервые заглянув в каталог `/etc`, чтобы изучить конфигурацию своей системы, вы можете сильно удивиться количеству имеющихся в нем файлов. Но есть и хорошая новость: хотя большинство файлов в какой-то степени влияют на работу системы, важны лишь некоторые из них.

В этой главе рассматриваются части системы, которые делают всю описанную в главе 4 инфраструктуру доступной для программного обеспечения пользовательского пространства, с которым мы обычно взаимодействуем. К примеру, инструменты, о которых шла речь в главе 2. В частности, рассмотрим следующие темы:

- ведение системного журнала;
- файлы конфигурации, к которым обращаются системные библиотеки для получения информации о сервере и пользователе;
- несколько избранных серверных программ (иногда называемых *демонами* (*daemons*)), которые запускаются при загрузке системы;
- утилиты конфигураций, которые можно применять для настройки серверных программ и файлов конфигурации;
- конфигурация времени;
- конфигурация периодического планирования задач.

Широкое использование системы `systemd` привело к сокращению числа основных независимых демонов типичной системы Linux. Одним из примеров является демон системного ведения журнала (`syslogd`), функциональность которого сейчас в значительной степени обеспечивается демоном, встроенным в `systemd` (`journald`). Тем не менее часть традиционных демонов, таких как `crond` и `atd`, все еще применяются.

Как и предыдущие главы, эта глава практически не содержит информации о сети, поскольку та является отдельным строительным блоком системы. В главе 9 вы увидите, как именно сеть вписывается в систему.

7.1. Ведение системного журнала

Большинство системных программ записывают результаты диагностики в виде сообщений в службу `syslog`. Эту работу выполняет традиционный демон `syslogd`, он ожидает сообщений и, получив одно, отправляет его по соответствующему каналу, например, записывает в файл или базу данных. В большинстве современных систем `journald` (он поставляется с `systemd`) делает бóльшую часть работы. В этой книге мы сосредоточимся именно на `journald`, однако рассмотрим и многие аспекты традиционного системного журнала `syslog`.

Системное журналирование — одна из наиболее важных частей системы. Когда что-то идет не так и непонятно, как это исправить, разумно начинать с проверки журнала. Если у вас есть утилита `journald`, используйте команду `journalctl`, которую мы рассмотрим в разделе 7.1.2. В более старых системах нужно проверять сами файлы. В любом случае сообщения журнала выглядят следующим образом:

```
Aug 19 17:59:48 duplex sshd[484]: Server listening on 0.0.0.0 port 22.
```

Сообщение журнала обычно содержит важную информацию, такую как имя процесса, идентификатор процесса и метка времени. Могут иметься и два других поля: *объект* (facility, общая категория) и *важность* (severity, насколько срочным является сообщение). Мы обсудим их более подробно позже.

Понять, как работает журналирование в системе Linux, может оказаться довольно сложно из-за различных комбинаций старых и новых компонентов программного обеспечения. Некоторые дистрибутивы, такие как Fedora, перешли по умолчанию только на использование утилиты `journald`, в то время как другие применяют одну из версий устаревшего `syslogd` (например, `rsyslogd`) вместе с `journald`. Старые дистрибутивы и некоторые специализированные системы могут не задействовать `systemd` и иметь только одну из версий `syslogd`. Кроме того, некоторые программные системы вообще обходят стандартизированное ведение журнала и пишут свои собственные.

7.1.1. Проверка настроек журнала

Всегда следует проверить свою систему и узнать, какой тип ведения журнала используется именно в ней. Сделать это можно так:

1. Проверьте, есть ли у вас утилита `journald`. Скорее всего, она имеется в вашей системе, если вы применяете `systemd`. Вы можете искать `journald` в списке процессов, однако самый простой способ найти ее — запустить команду `journalctl`. Если в системе утилита `journald` активна, вы увидите список сообщений журнала постранично.
2. Проверьте, есть ли у вас утилита `rsyslogd`. Найти ее можно в списке процессов, а также поищите файл `/etc/rsyslog.conf`.
3. Если у вас нет `rsyslogd`, проверьте, есть ли утилита `syslog-ng` (другая версия `syslogd`), отыскав каталог с именем `/etc/syslog-ng`.

Далее продолжите просматривать файлы журналов в каталоге `/var/log`. Если у вас есть одна из версий утилиты `syslogd`, этот каталог будет содержать множество файлов, большинство которых создано вашим демоном системного журнала `syslog`. Однако здесь будет и несколько файлов, поддерживаемых другими службами, например, `wtmp` и `lastlog` — файлы журналов, к которым такие утилиты, как `last` и `lastlog`, обращаются для получения записей входа.

Кроме того, в `/var/log` могут находиться дополнительные подкаталоги, содержащие журналы. Они почти всегда поступают из других служб. Один из них, `/var/log/journal`, — это место, где `journald` хранит свои (двоичные) файлы журналов.

7.1.2. Поиск и мониторинг журналов

Если в вашей системе нет `journald` или вы ищете файл журнала, поддерживаемый какой-либо другой утилитой, необходимо просматривать журнал. Без аргументов команда `journalctl` лавиной выводит все сообщения, имеющиеся в журнале, начиная с самого старого (точно так же, как они будут отображаться в файле журнала). К счастью, `journalctl` по умолчанию использует утилиту `less` для отображения конкретных сообщений. Вы можете с помощью нее искать нужные сообщения и обратить вспять (reverse) вывод по времени, применяя команду `journalctl -r`. Однако существуют более эффективные способы поиска сообщений в журнале.

ПРИМЕЧАНИЕ

Чтобы получить полный доступ к сообщениям журнала, необходимо запустить команду journalctl либо от имени суперпользователя, либо от имени пользователя, принадлежащего к группам adm или systemd-journal. В большинстве дистрибутивов пользователь по умолчанию имеет доступ к журналам.

Как правило, вы можете искать отдельные поля журналов, просто добавив их в командную строку, например, запустите команду `journalctl _PID=8792` для поиска сообщений с идентификатором процесса `8792`. Однако наиболее мощные функции сортировки носят общий характер. Вы можете указывать один или несколько дополнительных критериев поиска.

Фильтрация по времени

Параметр `-S` (since — «начиная с») относится к наиболее полезным параметрам для поиска по определенному времени. Вот пример одного из самых простых и эффективных способов его использования:

```
$ journalctl -S -4h
```

Часть `-4h` этой команды может выглядеть как параметр, но на самом деле это спецификация времени, указывающая команде `journalctl` искать сообщения за последние четыре часа для вашего часового пояса. Можно применять и комбинацию определенного дня и/или времени:

```
$ journalctl -S 06:00:00
$ journalctl -S 2020-01-14
$ journalctl -S '2020-01-14 14:30:00'
```

Параметр `-U` (until — «до») работает таким же образом, указывая время, до которого `journalctl` должен вывести сообщения. Однако чаще всего это не особо эффективно, потому что пользователи обычно просто просматривают страницы, пока не найдут то, что нужно, и выходят.

Фильтрация по юнитам

Еще одним быстрым и эффективным способом получения соответствующих журналов является фильтрация по юниту `systemd`. Для этого используется параметр `-u`:

```
$ journalctl -u cron.service
```

Обычно при фильтрации по юнитам тип юнита (в данном случае `.service`) опускается. Если вы не знаете название конкретного юнита, попробуйте выполнить эту команду, чтобы перечислить все юниты, имеющиеся в журнале:

```
$ journalctl -F _SYSTEMD_UNIT
```

Параметр `-F` показывает все значения в журнале для определенного поля.

Поиск полей

Иногда необходимо узнать, в каком поле искать нужную информацию. Вы можете перечислить все доступные поля следующим образом:

```
$ journalctl -N
```

Любое поле, начинающееся с символа подчеркивания (например, `_SYSTEMD_UNIT` из приведенного ранее примера), — это защищенное поле, клиент, отправляющий сообщение в журнал, его изменять не может.

Фильтрация по тексту

Классический метод поиска файлов журналов состоит в том, чтобы запустить по всем ним `grep`, надеясь найти соответствующую строку или место в файле, где может быть больше нужной информации. Аналогично, вы можете искать сообщения журнала по регулярному выражению с помощью параметра `-g` (как в следующем примере) — он будет возвращать сообщения, содержащие `kernel`, за которым в какой-то момент следует `memory`:

```
$ journalctl -g 'kernel.*memory'
```

К сожалению, при таком способе поиска в журнале вы получите *только* те сообщения, которые точно соответствуют регулярному выражению. Часто важная информация может быть рядом в смысле времени. Попробуйте выбрать метку времени совпадений с выражением, а затем запустить команду `journalctl -S` с указанием времени, чтобы посмотреть, какие сообщения появились примерно в то же время.

ПРИМЕЧАНИЕ

Для работы параметра -g требуется версия утилиты journalctl с определенной библиотекой. Некоторые дистрибутивы не включают версию, поддерживающую параметр -g.

Фильтрация по сообщениям о загрузке системы

Часто пользователь может просматривать журналы в поисках сообщений на то время, когда система загружалась, или появившихся непосредственно перед тем, как она вышла из строя (и перезагрузилась). Система позволяет легко получать сообщения за полный цикл одной загрузки, с момента запуска машины и до ее остановки. Например, если вы ищете начало текущей загрузки (boot), используйте параметр `-b`:

```
$ journalctl -b
```

Можно также сместить сортировку — чтобы начать с предыдущей загрузки, задействуйте дополнительное значение для параметра `-b -1`:

```
$ journalctl -b -1
```

ПРИМЕЧАНИЕ

Вы можете быстро проверить, правильно ли выключилась система в последний раз, объединив параметры -b и -r (reverse). Если вывод выглядит так, как в приведенном далее примере, значит, система завершила работу правильно:

```
$ journalctl -r -b -1
-- Logs begin at Wed 2019-04-03 12:29:31 EDT, end at Fri 2019-08-02 19:10:14
EDT. --
Jul 18 12:19:52 mymachine systemd-journald[602]: Journal stopped
Jul 18 12:19:52 mymachine systemd-shutdown[1]: Sending SIGTERM to remaining
processes...
Jul 18 12:19:51 mymachine systemd-shutdown[1]: Syncing filesystems and block
devices.
```

Вместо смещения сортировки на `-1` вы можете просматривать загрузки по идентификаторам. Используйте следующую команду:

```
$ journalctl --list-boots
-1 e598bd09e5c046838012ba61075dccbb Fri 2019-03-22 17:20:01 EDT—Fri 2019-04-12
08:13:52 EDT
 0 5696e69b1c0b42d58b9c57c31d8c89cc Fri 2019-04-12 08:15:39 EDT—Fri 2019-08-02
19:17:01 EDT
```

Наконец, можно отображать сообщения ядра (kernel) (с выбором конкретной загрузки или без нее) с помощью команды `journalctl -k`.

Фильтрация по важности

Некоторые программы выдают множество диагностических сообщений, которые могут скрывать важные журналы. Их можно отфильтровать по уровню важности (severity, priority), указав значение от `0` (наиболее важное) до `7` (наименее важное) рядом с параметром `-p`. Например, чтобы получить журналы с 0-го по 3-й уровень, выполните команду:

```
$ journalctl -p 3
```

Если вам нужны только журналы с определенным набором уровней важности, используйте синтаксис с точками (`..`) и указанием значений «от» и «до»:

```
$ journalctl -p 2..3
```

Фильтрация по степени важности кажется эффективной, но для нее нет особо полезного применения. Большинство приложений по умолчанию не генерируют большие объемы информационных данных, хотя некоторые включают параметры конфигурации, позволяющие вести журнал более подробно.

Простой мониторинг журнала

Один из традиционных способов мониторинга журналов — использование команды `tail -f` или `less` (`less +F`) в файле журнала, чтобы просматривать сообщения по мере их поступления. Практика регулярного мониторинга системы не очень эффективна (легко что-то упустить), но она полезна для изучения службы при попытке найти проблему или более подробно изучить запуск и работу в режиме реального времени.

Команда `tail -f` не работает с утилитой `journald`, поскольку в ней не используются текстовые файлы, вместо этого вы можете применить параметр `-f` для команды `journalctl`, чтобы получить тот же результат — вывод сообщений журнала по мере их поступления:

```
$ journalctl -f
```

Эта команда достаточно эффективна для большинства задач. Однако вам могут понадобиться и другие параметры фильтрации и сортировки, если в вашем

журнале наблюдается постоянный поток сообщений, не связанных с тем, что вы хотите найти.

7.1.3. Ротация файлов журнала

Во время использования демона `syslog` любое сообщение, которое записывает ваша система, записывается также где-то в файл журнала, поэтому время от времени необходимо удалять старые сообщения, чтобы они в конце концов не заняли все пространство для хранения. Разные дистрибутивы выполняют эту задачу по-разному, но большинство из них — с помощью утилиты `logrotate`.

Этот механизм называется *ротацией журнала (log rotation)*. Поскольку традиционный текстовый файл журнала содержит самые старые сообщения в начале, а самые новые в конце, довольно сложно удалить из файла только старые сообщения, чтобы очистить память. Вместо этого журнал, обслуживаемый утилитой `logrotate`, разделяется на множество блоков (chunks).

Допустим, у вас в каталоге `/var/log` есть файл журнала с именем `auth.log`, содержащий самые последние сообщения. Также есть файлы `auth.log.1`, `auth.log.2` и `auth.log.3`, каждый из которых содержит все более старые данные. Когда утилита `logrotate` решает, что пришло время удалить часть устаревших данных, она делает следующее:

1. Удаляет самый старый файл, `auth.log.3`.
2. Переименовывает `auth.log.2` в `auth.log.3`.
3. Переименовывает `auth.log.1` в `auth.log.2`.
4. Переименовывает `auth.log` в `auth.log.1`.

Имена и детали в разных дистрибутивах могут различаться. Например, конфигурация Ubuntu указывает, что `logrotate` должна сжимать файл, перемещенный с позиции `1` на позицию `2`, поэтому в предыдущем примере было бы `auth.log.2.gz` и `auth.log.3.gz`. В других дистрибутивах `logrotate` переименовывает файлы журналов с помощью суффикса даты, например `-20200529`. Одним из преимуществ этого способа является то, что файл журнала за определенное время проще найти.

А что произойдет, если `logrotate` выполнит ротацию в то же время, когда другая утилита (например, `rsyslogd`) захочет добавить сообщения в файл журнала? Скажем, программа ведения журнала открывает файл журнала для записи, но не закрывает его до того, как `logrotate` выполнит переименование. В этом несколько необычном случае сообщение журнала будет записано, потому что в Linux, когда файл открыт, система ввода-вывода не может знать, что он уже переименован. Но обратите внимание на то, что файл, в котором появится сообщение, будет файлом с новым именем, например `auth.log.1`.

Если утилита `logrotate` уже переименовала файл до того, как программа ведения журнала попытается его открыть, системный вызов `open()` создаст новый файл журнала (например, `auth.log`), как если бы `logrotate` не была запущена.

7.1.4. Обслуживание журналов journald

Журналы, хранящиеся в каталоге `/var/log/journal`, не нуждаются в ротации, поскольку `journald` сам может идентифицировать и удалять старые сообщения. В отличие от традиционного управления журналами, `journald` принимает решение об удалении сообщений на основании того, сколько места осталось в файловой системе, на которой находится журнал, сколько журнал должен занимать в процентах и каков его максимальный размер. Существуют и другие параметры управления, например максимально допустимое время существования сообщения журнала. Описание настроек по умолчанию, а также других настроек можно найти на странице руководства `journald.conf(5)`.

7.1.5. Детали системного журналирования

Теперь, когда вы ознакомились с подробностями работы `syslog` и journal, пришло время немного отступить назад и рассмотреть причины, по которым журналирование работает именно таким образом. Эта тема носит скорее теоретический, чем практический характер, так что можете сразу перейти к следующей теме книги.

В 1980-х годах возникла проблема: серверам Unix требовался способ записи диагностической информации, но для решения этой задачи еще не существовало определенного стандарта действий. Когда появился syslog с почтовым сервером sendmail, разработчики других служб охотно начали его использовать. Протокол RFC 3164 описывает процесс развития `syslog`.

Механизм работы довольно прост. Традиционный `syslogd` прослушивает и ожидает сообщений в доменном сокете Unix `/dev/log`. Еще одной мощной функцией `syslogd` является возможность прослушивания сетевого сокета в дополнение к `/dev/log`, что позволяет клиентским машинам отправлять сообщения по сети. Это позволяет объединять все сообщения системного журнала по всей сети на одном сервере, предназначенном для ведения журнала, и по этой причине `syslog` стал очень популярным у сетевых администраторов. Многие сетевые устройства, такие как маршрутизаторы и встроенные устройства, могут выступать в качестве клиентов `syslog`, отправляя свои диагностические сообщения на сервер.

`syslog` имеет классическую архитектуру «клиент — сервер», включая собственный протокол (в настоящее время это RFC 5424). Однако протокол не всегда был стандартом, и его более ранние версии были мало структурированы. Программисты, использующие `syslog`, должны были разработать описательный, но ясный и краткий формат сообщений журнала для собственных приложений. Со временем протокол приобрел новые функции, все еще пытаясь поддерживать как можно большую обратную совместимость.

Объекты, важность и другие поля

Поскольку `syslog` отправляет сообщения различных типов из разных служб в разные пункты назначения, для эффективной работы ему необходимо классифицировать

каждое сообщение. Традиционный метод заключается в применении закодированных значений объектов и важности, которые обычно (но не всегда) включались в сообщение. В дополнение к выводу файлов даже очень старые версии `syslogd` были способны отправлять важные сообщения на консоли и непосредственно определенным пользователям, вошедшим в систему, в зависимости от объекта и важности сообщений.

Объект (facility) — это общая категория служб, определяющая, какая служба отправила сообщение. Объекты включают в себя службы и системные компоненты, такие как ядро, почтовая система и принтер.

Важность (severity) — это срочность сообщения журнала. Существует восемь уровней, от 0 до 7. Обычно у них есть названия, хотя они не очень логичны и различаются в разных дистрибутивах:

```
0: emerg       4: warning
1: alert       5: notice
2: crit        6: info
3: err         7: debug
```

Объект и важность вместе составляют *приоритет (priority)* — определенный как одно число в протоколе `syslog`. Вы можете прочитать все об этих полях в протоколе RFC 5424, узнать, как указать их в приложениях, на странице руководства `syslog(3)` и как сопоставить — на странице руководства `rsyslog.conf(5)`. Однако вы можете запутаться, работая с ними в `journald`, где важность упоминается также в качестве приоритета (например, при вводе команды `journalctl -o json` для получения машиночитаемых выходных данных журнала).

К сожалению, при изучении деталей приоритетной части протокола вы обнаружите, что она не успевает за изменениями и требованиями, фиксирующимися в остальной части ОС. Определение важности по-прежнему работает эффективно, но доступные объекты уже жестко определены и включают в себя редко используемые службы, такие как UUCP, без возможности определения новых (только ряд общих слотов `local0` через `local7`).

Мы уже говорили о других полях в данных журнала, но RFC 5424 также включает в себя положение о структурированных данных — наборах произвольных пар ключей, которые программисты могут применять для определения собственных полей. Они могут быть использованы и с утилитой `journald`, но гораздо чаще их отправляют в другие типы баз данных.

Взаимосвязь между syslog и journald

Если `journald` полностью вытеснил `syslog` в некоторых системах, остается вопрос: почему `syslog` все еще используется в других? Есть две основные причины:

- `syslog` имеет четко определенные средства объединения журналов на многих компьютерах. Гораздо проще отслеживать журналы, когда они находятся только на одной машине.

- Версии `syslog`, такие как `rsyslogd`, являются модульными и могут выводиться в различные форматы и базы данных, включая формат journal. Это облегчает их подключение к инструментам анализа и мониторинга.

А утилита `journald` подчеркивает важность сбора и организации выходных данных журнала одной машины в одном формате.

Способность `journald` передавать свои журналы в другой регистратор журналов обеспечивает высокую степень универсальности при попытке выполнить более сложные задачи. Особенно если учесть, что `systemd` может собирать выходные данные серверных устройств и отправлять их в `journald`, предоставляя вам доступ к еще большему количеству данных журнала, чем приложения шлют в `syslog`.

И еще немного о журналировании

Журналирование в системах Linux значительно изменилось за время своего существования и будет продолжать развиваться. На данный момент процесс сбора, хранения и извлечения журналов на одной машине четко определен, но есть и другие аспекты ведения журнала, которые все еще не стандартизированы.

Во-первых, существует огромный набор параметров объединения и хранения журналов через сеть компьютеров. Вместо централизованного сервера журналов, просто хранящего их в текстовых файлах, журналы теперь могут храниться в базах данных, и часто сам централизованный сервер заменяется интернет-сервисом.

Во-вторых, изменился характер использования журналов. Одно время они не считались реальными данными, их основной целью было создание ресурса, который администратор (человек) мог прочитать, если что-то пошло не так. Однако по мере усложнения приложений потребности в ведении журнала увеличивались. Новые требования включают возможность поиска, извлечения, отображения и анализа данных, содержащихся в журналах. Существует множество способов хранения журналов в базах данных, но инструменты для применения журналов в приложениях все еще находятся в зачаточном состоянии.

И наконец, остается нерешенным вопрос обеспечения надежности журналов. В начальном `syslog` не было никакой аутентификации — приходилось просто доверять любому приложению и/или машине, отправляющим данные журнала. Кроме того, журналы не были зашифрованы, что делало их уязвимыми в сети, и довольно рискованно было применять их в сетях, которые требовали высокой безопасности. Современные серверы `syslog` имеют стандартные методы шифрования сообщения журнала и проверки подлинности компьютера, с которого оно было отправлено. Однако при переходе к отдельным приложениям общая картина безопасности становится менее ясной. Например, можете ли вы быть уверены, что то, что называет себя вашим веб-сервером, на самом деле им является?

Мы рассмотрим несколько углубленных тем, относящихся к аутентификации, позже в этой главе. А сейчас перейдем к основам организации файлов конфигурации в системе.

7.2. Структура каталога /etc

Большинство файлов конфигурации системы в дистрибутивах Linux находятся в каталоге `/etc`. Исторически сложилось так, что у каждой программы или системной службы был один или несколько файлов конфигурации и из-за большого количества компонентов в системе Unix каталог `/etc` быстро накапливал файлы.

При таком подходе существовали две проблемы: трудно было найти определенные файлы конфигурации в работающей системе, а также поддерживать систему, настроенную таким образом. Например, если вы хотите изменить конфигурацию `sudo`, вам придется отредактировать файл `/etc/sudoers`. Но после изменения обновление вашего дистрибутива может стереть все существовавшие настройки, потому что перезапишет файл `/etc`.

В течение многих лет существовала тенденция размещать файлы конфигурации системы в подкаталогах под `/etc`, например как для `systemd`, которая использует `/etc/systemd`. В `/etc` все еще есть несколько отдельных файлов конфигурации, но если вы запустите команду `ls -F /etc`, то увидите, что большинство элементов теперь находятся в подкаталогах.

Чтобы решить проблему перезаписи файлов конфигурации, появилась возможность размещать настройки в отдельных файлах в подкаталогах конфигурации, например в `/etc/grub.d`.

Какие же файлы конфигурации находятся в каталоге `/etc`? Основное правило заключается в том, что настраиваемые конфигурации для одной машины, такие как информация о пользователе (`/etc/passwd`) и сведения о сети (`/etc/network`), входят в `/etc`. Однако общие сведения о приложении, например значения по умолчанию для пользовательского интерфейса дистрибутива, не относятся к `/etc`. Файлы конфигурации системы по умолчанию, не предназначенные для настройки, также обычно находятся в другом месте, как и в случае с предварительно упакованными юнит-файлами `systemd` в `/usr/lib/systemd`.

Вам уже встречались некоторые файлы конфигурации, относящиеся к загрузке. Давайте рассмотрим, какие настройки могут быть у пользователей в системе.

7.3. Файлы управления пользователями

Системы Unix допускают существование нескольких независимых пользователей. На уровне ядра пользователи — это просто числа (идентификаторы пользователей, user IDs), но поскольку запомнить имя намного проще, чем номер, при управлении Linux обычно применяются *имена пользователей* (usernames, login names, *логины*). Они существуют только в пользовательском пространстве, поэтому любая программа, работающая с именем пользователя, должна находить соответствующий идентификатор пользователя при взаимодействии с ядром.

7.3.1. Файл /etc/passwd

Текстовый файл `/etc/passwd` сопоставляет имена пользователей с идентификаторами пользователей. Пример приведен в листинге 7.1.

Листинг 7.1. Список пользователей в `/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 содержит семь полей, разделенных двоеточиями. Первое — это имя пользователя.

Далее идет зашифрованный пароль пользователя или, по крайней мере, нечто похожее на поле для пароля. В большинстве систем Linux пароль на самом деле хранится не в файле `passwd`, а в *теневом* (shadow) файле (см. подраздел 7.3.3). Формат файла `shadow` аналогичен формату файла `passwd`, но у обычных пользователей нет разрешения на чтение для `shadow`. Второе поле в файлах `passwd` или `shadow` — это зашифрованный пароль, и он выглядит как куча несвязанных символов, например `d1CVEWiB/oppc`. Пароли Unix никогда не хранятся в открытом виде, на самом деле поле — это не сам пароль, а его производная. В большинстве случаев получить исходный пароль из этого поля максимально сложно (при условии, что пароль нелегко угадать).

Символ `x` во втором поле файла `passwd` говорит о том, что зашифрованный пароль хранится в файле `shadow`, который должен находиться в вашей системе. Звездочка (`*`) указывает, что пользователь не может войти в систему. Если в поле пароля пусто (то есть отображаются два двоеточия подряд — `::`), то для входа в систему пароль не требуется. Не оставляйте поля паролей пустыми. В системе не должно быть пользователя, который может войти в систему без пароля.

Остальные поля файла `passwd`:

- *Идентификатор пользователя (user ID, UID)*, который является представлением пользователя в ядре. У вас могут существовать две записи с одинаковым идентификатором пользователя, но это только запутает вас и систему, поэтому следите за уникальностью идентификаторов пользователей.
- *Идентификатор группы (group ID, GID)*, который должен быть одной из пронумерованных записей в файле `/etc/group`. Группы определяют права доступа к файлам и др. Эта группа называется также *основной группой* (primary group) пользователя.
- Настоящее имя пользователя (чаще всего оно имеет сокращенное название *GECOS*). Иногда в этом поле вы найдете запятые, обозначающие номера комнат и телефонов.

- Домашний каталог пользователя.
- Оболочка пользователя (программа, которая запускается, когда пользователь запускает сеанс терминала).

На рис. 7.1 показаны различные поля для одной из строк из листинга 7.1.

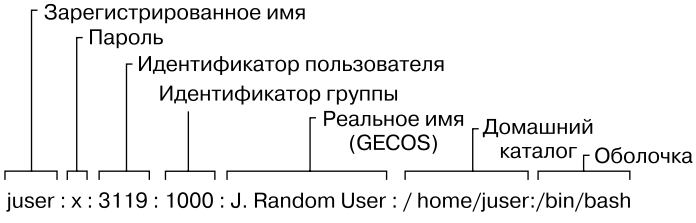


Рис. 7.1. Поля записи в файле паролей

Синтаксис файла `/etc/passwd` довольно строг и не допускает комментариев или пустых строк.

ПРИМЕЧАНИЕ

Пользователь в /etc/passwd и соответствующий ему домашний каталог в совокупности называются учетной записью (account). Однако помните, что это еще и соглашение о пользовательском пространстве. Записи в файле passwd обычно достаточно — домашний каталог не обязательно должен существовать, чтобы большинство программ смогли распознать учетную запись. Кроме того, существуют способы добавления пользователей в систему без явного включения их в файл passwd, например добавление пользователей с сетевого сервера с помощью различных служб, таких как NIS (Network Information Service — информационная служба сети) или LDAP (Lightweight Directory Access Protocol — легковесный протокол доступа к каталогам).

7.3.2. Особые пользователи

В файле `/etc/passwd` можно найти несколько особых пользователей. Суперпользователь (superuser, `root`) всегда имеет UID 0 и GID 0, как в листинге 7.1.

Некоторые пользователи, например демоны, не имеют привилегий при входе в систему. Пользователь `nobody` — непривилегированный, некоторые процессы выполняются от его имени, потому что он обычно не может ничего записывать в системе. Пользователи, которые не могут войти в систему, называются *псевдопользователями* (pseudo-users). Хотя они не могут попасть в систему, сама она может запускать процессы с их идентификаторами пользователей. Псевдопользователи, такие как `nobody`, обычно создаются по соображениям безопасности.

Все это соглашения о пользовательском пространстве. Эти пользователи не имеют особого значения для ядра, единственный идентификатор пользователя, который

влияет на ядро, — это идентификатор суперпользователя, `0`. Пользователю `nobody` можно предоставить доступ ко всему в системе, как и любому другому.

7.3.3. Файл /etc/shadow

Файл теневого пароля (`/etc/shadow`) в системе Linux обычно содержит данные об аутентификации пользователя, включая зашифрованные пароли и информацию об истечении срока действия пароля, которые соответствуют пользователям в `/etc/passwd`.

Теневой файл был введен для обеспечения более гибкого (и, возможно, более безопасного) способа хранения паролей. Он включал набор библиотек и утилит, многие из которых вскоре были заменены частями PAM (Pluggable Authentication Modules, подключаемые модули аутентификации, подробнее мы рассмотрим эту тему в разделе 7.10). Вместо того чтобы вводить совершенно новый набор файлов для Linux, PAM использует файл `/etc/shadow`, но не задействует определенные файлы конфигурации, например `/etc/login.defs.e`.

7.3.4. Управление пользователями и паролями

Обычные пользователи взаимодействуют с файлом `/etc/passwd` с помощью команды `passwd` и нескольких других инструментов. Применяйте команду `passwd`, чтобы изменить свой пароль. Также можете использовать команды `chfn` и `chsh` для изменения реального имени и оболочки соответственно (оболочка должна быть указана в `/etc/shells`). Все это исполняемые команды с правами suid-root суперпользователя, так как только он может изменить файл `/etc/passwd`.

Изменение файла /etc/passwd от имени суперпользователя

Поскольку `/etc/passwd` — это обычный текстовый файл, суперпользователю технически разрешено задействовать любой текстовый редактор для внесения изменений. Чтобы добавить пользователя, можно просто добавить соответствующую строку и создать домашний каталог для него, чтобы удалить — сделать обратное.

Однако прямое редактирование файла `passwd`, как описано ранее, — плохая идея. Мало того что легко ошибиться в файле, к тому же можно столкнуться с проблемой параллельного выполнения, когда изменения в файл `passwd` вносятся одновременно из разных источников. Гораздо проще и безопаснее вносить изменения в данные пользователей с помощью отдельных команд, доступных из терминала или через графический интерфейс. Например, чтобы установить пароль пользователя *`user`*, запустите команду `passwd` *`user`* от имени суперпользователя. Задействуйте команды `adduser` и `userdel` для добавления и удаления пользователей соответственно.

Но если вам действительно необходимо отредактировать файл напрямую (например, если он поврежден), примените программу `vipw`, которая создает резервные

копии и в качестве дополнительной меры предосторожности блокирует `/etc/passwd` во время его редактирования.

Чтобы отредактировать `/etc/shadow` вместо `/etc/passwd`, используйте команду `vipw -s`. (Надеемся, вам никогда не придется делать ни того, ни другого.)

7.3.5. Работа с группами пользователей

Группы (groups) в Unix — это способ организации совместного доступа к файлам между определенными пользователями. Суть его заключается в том, что вы можете установить биты разрешений на чтение или запись для определенной группы, исключая всех остальных. Раньше эта функция была необходима, потому что многие пользователи совместно работали на одном компьютере или в сети, но в последние годы она стала менее значимой, поскольку общие рабочие станции применяются все реже.

Файл `/etc/group` определяет идентификаторы групп (например, содержащиеся в файле `/etc/passwd`), как в листинге 7.2.

Листинг 7.2. Пример файла `/etc/group`

```
root:*:0:juser
daemon:*:1:
bin:*:2:
sys:*:3:
adm:*:4:
disk:*:6:juser,beazley
nogroup:*:65534:
user:*:1000:
```

Как и в файле `/etc/passwd`, каждая строка в `/etc/group` представляет собой набор полей, разделенных двоеточиями. Перечислим поля в каждой строке слева направо.

- **Название группы.** Появляется при выполнении такой команды, как `ls -l`.
- **Пароль группы.** Пароли групп Unix почти никогда не используются, и их в целом не стоит применять (хорошая альтернатива в большинстве случаев — команда `sudo`). Задействуйте символ `*` или любое другое значение по умолчанию. Символ `x` здесь означает, что в `/etc/gshadow` есть соответствующая запись, где описан почти всегда отключенный пароль, обозначенный символами `*` или `!`.
- **Идентификатор группы (Group ID, GID, номер).** Идентификатор GID должен быть уникальным в файле группы. Этот номер вводится в поле группы пользователя в его записи `/etc/passwd`.
- **Необязательный список пользователей, входящих в группу.** Вдобавок к перечисленным здесь пользователям все, кто имеет соответствующий идентификатор группы в своих записях файла `passwd`, также принадлежат к группе.

На рис. 7.2 показаны поля для записей в файле `/etc/group`.

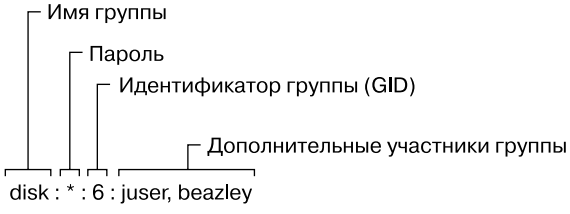


Рис. 7.2. Запись в файле /etc/group

Чтобы просмотреть группы, к которым вы принадлежите, запустите команду `groups`.

ПРИМЕЧАНИЕ

Дистрибутивы Linux часто создают для каждого нового пользователя новую группу, имя которой совпадает с именем пользователя.

7.4. Команды getty и login

Утилита `getty` подключается к терминалам и отображает приглашение для входа в систему. В большинстве систем Linux `getty` легко использовать, поскольку они применяют ее только для входа в систему на виртуальных терминалах. В списке процессов она выглядит примерно так (например, при запуске в файле `/dev/tty1`):

```
$ ps ao args | grep getty
/sbin/agetty -o -p -- \u --noclear tty1 linux
```

Во многих системах вы можете даже не увидеть процесс `getty`, пока не получите доступ к виртуальному терминалу с помощью, например, ввода комбинации клавиш Ctrl+Alt+F1. В этом примере показана утилита `agetty` — версия, которую многие дистрибутивы Linux включают по умолчанию.

После того как вы введете свое имя для входа, `getty` заменит себя программой входа `login`, которая запросит ваш пароль. Если вы введете правильный пароль, `login` заменит себя (с помощью `exec()`) вашей оболочкой. В противном случае вы получите сообщение Login incorrect («Неверный логин»). Большая часть аутентификации в систему `login` выполняется с помощью модулей PAM (см. раздел 7.10).

ПРИМЕЧАНИЕ

Изучая работу утилиты getty, вы можете наткнуться на понятие скорости передачи бодов, например 38400. Эта устаревшая настройка. Виртуальные терминалы игнорируют скорость передачи данных в бодах, она используется только для подключения к реальным последовательным каналам.

Теперь вы знакомы с командами `getty` и `login`, но вам, скорее всего, не придется их настраивать или изменять. На самом деле вы даже редко будете их применять, потому что большинство пользователей теперь входят либо через графический интерфейс, такой как `gdm`, либо удаленно с помощью SSH, и ни один из них не использует `getty` или `login`.

7.5. Установка времени

Все системы Unix зависят от точного времени. Ядро поддерживает системные часы (system clock), то есть часы, с которыми вы сверяетесь при выполнении таких команд, как `date`. Вы также можете установить системные часы с помощью команды `date`, но лучше так не делать, потому что таким образом сложно установить максимально точное время. Ваши системные часы должны быть настроены как можно ближе к правильному времени.

Оборудование для систем имеет *часы реального времени с питанием от батареи (RTC, Real-Time Clock)*. Это не самые точные часы в мире, но, как говорится, лучше, чем ничего. Ядро обычно устанавливает свое время на основе RTC во время загрузки, и вы можете сбросить системные часы до текущего аппаратного времени с помощью команды `hwclock`.

Сохраняйте системные часы во всеобщем скоординированном времени (UTC, Universal Coordinated Time), чтобы избежать проблем с настройкой часового пояса или переходом на летнее время. Вы можете установить RTC на часы UTC для ядра, используя команду

```
# hwclock --systohc --utc
```

К сожалению, ядро еще хуже справляется со временем, чем RTC, и поскольку машины Unix часто работают в течение нескольких месяцев или даже лет при одной загрузке, то, как правило, появляется сдвиг во времени. *Сдвиг во времени (time drift)* — это разница между временем ядра и истинным временем (определяется атомными часами или другими очень точными часами).

Не пытайтесь исправить сдвиг во времени с помощью команды `hwclock`, потому что системные события, основанные на времени, могут быть потеряны или искажены. Вы можете запустить утилиту, например `adjtimex`, чтобы плавно обновлять часы на основе RTC, но лучше корректировать системное время с помощью демона сетевого времени (см. подраздел 7.5.2).

7.5.1. Представление времени ядра и часовые пояса

Системные часы ядра представляют текущее время как количество секунд с 12:00 полуночи 1 января 1970 года по UTC. Чтобы увидеть это число в данный момент, выполните следующую команду:

```
$ date +%s
```

Чтобы преобразовать это число в читаемое значение, пользовательские программы меняют его на местное время и компенсируют переход на летнее время и любые другие обстоятельства, например проживание в Индиане. Локальный часовой пояс управляется файлом `/etc/localtime`. (Его можно не рассматривать, так как это двоичный файл.)

Файлы часовых поясов в вашей системе находятся в `/usr/share/zoneinfo`. Этот каталог содержит множество часовых поясов и псевдонимов для них. Чтобы установить часовой пояс (time zone) вашей системы вручную, скопируйте один из файлов из `/usr/share/zoneinfo` в `/etc/localtime` либо создайте символическую ссылку, еще ее можно изменить с помощью инструмента часового пояса вашего дистрибутива. Команда `tzselect` может помочь вам определить файл часового пояса.

Чтобы использовать часовой пояс, отличный от системного по умолчанию, только для одного сеанса оболочки, установите переменную среды `TZ` в качестве имени файла в `/usr/share/zoneinfo` и протестируйте, например, так:

```
$ export TZ=US/Central
$ date
```

Как и в случае с другими переменными окружения, вы можете задать часовой пояс лишь на время выполнения одной команды, например:

```
$ TZ=US/Central date
```

7.5.2. Сетевое время

Если ваша система постоянно подключена к интернету, можете запустить демон протокола сетевого времени (NTP, Network Time Protocol), чтобы установить и поддерживать время с помощью удаленного сервера. Раньше эта задача обрабатывалась демоном `ntpd`, но, как во многом другом, `systemd` заменила его собственным пакетом с именем `timesyncd`. Большинство дистрибутивов Linux используют `timesyncd`, и он включен по умолчанию. Вам не нужно настраивать его, но если вас интересует, как это сделать, изучите страницу руководства `timesyncd.conf(5)`. Наиболее распространенное переопределение конфигурации — изменение удаленного сервера (серверов) времени.

Если вы хотите вместо этого запустить службу `ntpd`, нужно отключить `timesyncd`. Перейдите на сайт www.ntppool.org, чтобы ознакомиться с инструкцией, рассказывающей, как это сделать. На этом сайте также можно узнать, как использовать `timesyncd` с разными серверами.

Если на вашем компьютере нет постоянного подключения к интернету, можете задействовать демон `chronyd` для поддержания времени, когда система отключена от интернет-сети.

Можете также настроить системные аппаратные часы на основе сетевого времени, чтобы помочь своей системе поддерживать точное время при перезагрузках. Многие

дистрибутивы выполняют это автоматически, но чтобы сделать все вручную, убедитесь, что системное время установлено из сети, а затем выполните команду

```
# hwclock --systohc --utc
```

7.6. Планирование повторяющихся задач с помощью команды cron и юнитов таймера

Существует два способа запуска повторяющихся задач: применение команды `cron` и юниты таймера `systemd`. Это важная функция для задач автоматизации обслуживания системы. Одним из примеров являются утилиты ротации файлов журналов, которые гарантируют, что ваш жесткий диск не заполнится старыми файлами журналов (как обсуждалось ранее в этой главе). Служба `cron` давно является стандартом планирования, и мы рассмотрим ее подробно. Таймеры `systemd` в некоторых случаях имеют преимущества по сравнению с утилитой `cron`, поэтому мы также поговорим о том, как их использовать.

С помощью `cron` вы можете запускать любую программу в удобное время. Программа, запущенная через `cron`, называется *задачей* `cron`. Чтобы установить задачу `cron`, добавьте строку ввода в файле `crontab`, выполнив команду `crontab`. Например, следующая запись файла `crontab` запускает по расписанию команду `/home/juser/bin/make` ежедневно в 9:15 (по местному времени):

```
15 09 * * * /home/juser/bin/spmake
```

Пять полей в начале этой строки, разделенные пробелами, указывают запланированное время (рис. 7.3). Поля расположены следующим образом:

- минута (с 0 до 59) — задача `cron` из примера установлена на минуту 15;
- час (с 0 до 23) — задача в примере назначена на 9 часов;
- день месяца (с 1 до 31);
- месяц (с 1 до 12);
- день недели (с 0 до 7) — числа 0 и 7 означают воскресенье.

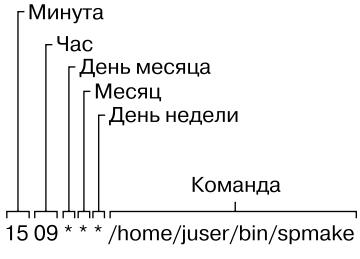


Рис. 7.3. Запись в файле crontab

Символ звездочки (*) в любом поле означает соответствие всем значениям. Команда spmake из приведенного примера запускается ежедневно, потому что поля «День месяца», «Месяц» и «День недели» заполнены звездочками, которые cron читает как «Выполнять эту задачу каждый день, каждый месяц, каждый день недели».

Чтобы запускать команду spmake только 14-го числа каждого месяца, используйте следующую команду:

```
15 09 14 * * /home/juser/bin/spmake
```

Для каждого поля можно выбрать несколько значений времени. Например, чтобы запустить программу 5-го и 14-го числа каждого месяца, нужно ввести 5,14 в третье поле:

```
15 09 5,14 * * /home/juser/bin/spmake
```

ПРИМЕЧАНИЕ

Если задача cron выдает стандартный вывод, ошибку или завершается некорректно, команда cron должна отправить эту информацию по электронной почте владельцу задачи (при условии, что электронная почта указана и работает в системе). Перенаправьте вывод в /dev/null или какой-либо другой файл журнала, если вас раздражает передача через электронную почту.

Страница руководства crontab(5) содержит полную информацию о формате crontab.

7.6.1. Установка файлов Crontab

У каждого пользователя может быть собственный файл crontab, что означает: в каждой системе в каталоге /var/spool/cron/crontabs их может быть несколько. Обычные пользователи не могут писать в этот каталог — команда crontab устанавливает, перечисляет, редактирует и удаляет crontab пользователей.

Самый простой способ установить crontab — это поместить записи crontab в файл *file*, а затем задействовать команду crontab *file* для установки файла *file* в качестве текущего crontab. Команда crontab проверяет формат файла, чтобы убедиться, что вы не допустили никаких ошибок. Для просмотра списка заданий cron запустите команду crontab -l. Чтобы удалить crontab, примените команду crontab -r.

После создания первоначального файла crontab может быть немного неудобно использовать временные файлы для внесения дальнейших изменений. Вместо этого вы можете сразу отредактировать и установить свой crontab с помощью команды crontab -e. Если допущена ошибка, crontab сообщит вам, где она, и спросит, хотите ли вы снова отредактировать файл.

7.6.2. Системные файлы Crontab

Многие распространенные системные задачи, активируемые cron, выполняются от имени суперпользователя. Однако вместо того, чтобы редактировать

и поддерживать `crontab` суперпользователя, дистрибутивы Linux задействуют файл `/etc/crontab` для всей системы. Для редактирования этого файла не нужно применять команду `crontab`, и в любом случае она немного отличается по формату: перед полем запуска команды есть дополнительное поле, указывающее пользователя, который должен выполнить задание. (Это дает возможность группировать системные задачи, даже если не все они выполняются одним и тем же пользователем.) Например, следующая задача `cron` из файла `/etc/crontab` выполняется в 6:42 утра от имени суперпользователя (`root` ❶):

```
42 6 * * * root❶ /usr/local/bin/cleansystem > /dev/null 2>&1
```

ПРИМЕЧАНИЕ

Некоторые дистрибутивы хранят дополнительные системные файлы crontab в каталоге /etc/cron.d. У них может быть любое имя, но тот же формат, что и у /etc/crontab. В этом каталоге могут присутствовать и подкаталоги, такие как /etc/cron.daily, но файлы здесь обычно являются сценариями, выполняемыми конкретными задачами cron в /etc/crontab или /etc/cron.d. Иногда бывает сложно отследить, где находятся задачи и когда они выполняются.

7.6.3. Юниты таймера

Помимо `cron` для повторяющийся задачи можно создать юнит таймера `systemd timer`. Для совершенно новой задачи необходимы два юнита: юнит таймера и служебный юнит. Их нужно создать потому, что юнит таймера не содержит никаких сведений о выполняемой задаче — это просто механизм активации для запуска служебного юнита (или другого типа юнита, но чаще всего именно служебного).

В качестве примера рассмотрим типичную пару «таймер/служебный юнит», начиная с юнита таймера. Назовем его `loggertest.timer` и, как и в случае с другими файлами пользовательских юнитов, поместим его в `/etc/systemd/system` (листинг 7.3).

Листинг 7.3. Таймер loggertest.timer

```
[Unit]
Description=Example timer unit

[Timer]
OnCalendar=*-*-* *:00,20,40
Unit=loggertest.service

[Install]
WantedBy=timers.target
```

Этот таймер запускается каждые 20 минут, а параметр `OnCalendar` напоминает синтаксис `cron`. В этом примере он находится в начале каждого часа, а также через 20 и 40 минут после каждого часа.

Формат времени по календарю `OnCalendar` — «год-месяц-день, час:минута:секунда». Поле для секунд необязательное.

Как и в случае с `cron`, символ `*` представляет собой своего рода подстановочный знак, а запятые допускают указание нескольких значений. Синтаксис периода `/` тоже допустим — в предыдущем примере можно было бы изменить значение `*:00,20,40` на `*:00/20` (каждые 20 минут), чтобы получить тот же результат.

ПРИМЕЧАНИЕ

Синтаксис для указания времени в поле OnCalendar имеет множество ярлыков и вариантов. Полный список см. в разделе «События календаря» на странице руководства systemd.time(7).

Связанный служебный юнит называется `loggertest.service` (листинг 7.4). Мы специально дали ему название в таймере с параметром `Unit`, но это необязательно, потому что `systemd` ищет файл `.service` с тем же базовым именем, что и юнит-файл таймера. Этот служебный юнит также входит в каталог `/etc/systemd/system` и похож на служебные юниты из главы 6.

Листинг 7.4. Таймер loggertest.service

```
[Unit]
Description=Example Test Service

[Service]
Type=oneshot
ExecStart=/usr/bin/logger -p local3.debug I\'m a logger
```

Суть заключается в строке `ExecStart`, которая представляет собой команду, выполняемую службой при активации. В примере она отправляет сообщение в системный журнал.

Обратите внимание на то, что `oneshot` используется в качестве типа службы, указывающего на то, что ожидаются запуск и выход службы, и `systemd` не будет считать службу запущенной до завершения команды, указанной в `ExecStart`. Благодаря этому у таймеров появляется несколько преимуществ:

- В юнит-файле можно указать несколько команд `ExecStart`. Другие служебные юниты, которые мы рассматривали в главе 6, не допускают этого.
- Проще контролировать строгий порядок зависимостей при активации других устройств с помощью директив зависимостей `Wants` и `Before`.
- После работы таймера в журнале остаются более точные записи о времени начала и окончания работы устройства.

ПРИМЕЧАНИЕ

В примере устройства мы используем регистратор logger для отправки записи в системный журнал syslog и journal. Как известно из подраздела 7.1.2, вы можете просматривать сообщения журнала по юнитам. Однако юнит может закончить работу до того, как journald получит сообщение. Это вызывает эффект гонок, и если юнит завершается слишком быстро, journald не сможет найти юнит, связанный с сообщением системного

журнала syslog (это делается по идентификатору процесса). Следовательно, сообщение, которое записывается в журнал, может не содержать поля юнита, из-за чего команда фильтрации, такая как journalctl -f -u loggertest.service, не сможет отобразить сообщение системного журнала syslog. У более длительных служб такой проблемы нет.

7.6.4. Утилита cron против юнитов таймера

Утилита `cron` — один из старейших компонентов системы Linux, она существует уже несколько десятилетий (возникла до появления самой Linux), и за это время ее формат конфигурации изменился незначительно. А как говорится, если что-то устаревает настолько сильно, жди замены.

Юниты таймера `systemd`, которые мы рассмотрели ранее, могут показаться логичной заменой, и действительно, многие дистрибутивы уже перенесли задачи периодического обслуживания системного уровня на юниты таймера. Тем не менее у `cron` есть свои преимущества:

- более простая конфигурация;
- совместимость со многими сторонними службами;
- упрощенная настройка пользовательских задач.

Преимущества юнитов таймера:

- эффективное отслеживание процессов, связанных с задачами/подразделениями с использованием cgroups;
- эффективное отслеживание диагностической информации в журнале;
- дополнительные параметры для времени и частоты активации;
- возможность использования зависимостей `systemd` и механизмов активации.

Возможно, когда-нибудь появится уровень совместимости для задач `cron`, схожий с механизмом, который есть у юнитов монтирования (mount units) и `/etc/fstab`. Однако одна только возможность упрощенной конфигурации говорит о том, что формат `cron` вряд ли исчезнет в ближайшее время. Как вы увидите в следующем разделе, утилита под названием `systemd-run` позволяет создавать юниты таймера и связанные с ними службы без создания юнит-файлов, но управление и реализация различаются настолько, что многие пользователи, скорее всего, предпочтут `cron`.

7.7. Планирование разовых задач с помощью службы at

Чтобы выполнить задачу один раз в будущем без использования `cron`, задействуйте службу `at`. Например, чтобы запустить задание `myjob` в 22:30, введите команду

```
$ at 22:30
at> myjob
```

Завершите ввод с помощью сочетания клавиш Ctrl+D. (Утилита `at` считывает команды со стандартного ввода.)

Чтобы проверить, что задание было запланировано, используйте команду `atq`. Чтобы удалить его, примените команду `atrm`. Вы также можете запланировать для этой задачи дни в будущем, добавив дату в формате *ДД.ММ.ГГ*, например `at 22:30 30.09.15`.

Команда `at` выполняет только эти функции. Она применяется не слишком часто, но очень эффективна, когда в ней возникает необходимость.

7.7.1. Аналоги таймера

Вы можете заменить службу `at` юнитами таймера `systemd`. Создавать их гораздо проще, чем периодические юниты таймера, которые мы рассмотрели ранее, и можно запускать в командной строке следующим образом:

```
# systemd-run --on-calendar='2022-08-14 18:00' /bin/echo this is a test
Running timer as unit: run-rbd000cc6ee6f45b69cb87ca0839c12de.timer
Will run service as unit: run-rbd000cc6ee6f45b69cb87ca0839c12de.service
```

Команда `systemd-run` создает временный юнит таймера, который можно просмотреть с помощью обычной команды `systemctl list-timers`. Если вас не волнует конкретное время, укажите вместо него смещение по времени с помощью параметра `--on-active` (например, `--on-active=30m` — действие будет выполнено в течение 30 минут в будущем).

ПРИМЕЧАНИЕ

При использовании параметра --on-calendar важно указать календарную дату в будущем, а также время. В противном случае юниты таймера и службы останутся, а таймер будет запускать службу каждый день в указанное время, как было бы, если бы вы создали обычный юнит таймера, описанный ранее. Синтаксис для этого параметра такой же, как и для параметра OnCalendar.

7.8. Юниты таймера обычных пользователей

Все юниты таймера `systemd`, которые мы встречали до сих пор, были запущены от имени суперпользователя. Юнит таймера можно создать и от имени обычного пользователя. Для этого добавьте параметр `--user` в `systemd-run`.

Но если вы выйдете из системы до запуска юнита, он не запустится, а если выйдете из нее до завершения работы юнита, его работа завершится. Это происходит потому, что в systemd есть менеджер пользователей, связанный с каждым пользователем, зашедшим в систему, и это необходимо для запуска юнитов таймера. Вы можете указать systemd, чтобы он сохранял менеджер пользователя после выхода пользователя из системы, с помощью команды

```
$ loginctl enable-linger
```

Как суперпользователь, вы можете включить менеджер для другого пользователя *user*:

```
# loginctl enable-linger user
```

7.9. Доступ пользователя

Оставшаяся часть этой главы посвящена нескольким темам: как пользователи получают разрешение на вход в систему, как происходит переключение пользователей и выполнение других связанных задач. Это материал продвинутого уровня, и вы можете перейти к следующей главе, если готовы пропустить изучение глубоких внутренних процессов.

7.9.1. ID пользователей и переключение пользователей

Мы обсудили, как программы `setuid`, такие как `sudo` и `su`, позволяют вам временно сменить пользователя, а также рассмотрели системные компоненты, например `login`, которые контролируют доступ пользователей. Возможно, вас заинтересовало, как работают эти части и какую роль ядро играет в переключении пользователей.

Когда вы временно переключаетесь на другого пользователя, реально происходит смена идентификатора пользователя. Есть два способа сделать это, и ядро обрабатывает оба. Первый — с помощью исполняемого файла `setuid`, который был описан в разделе 2.17. Второй — с помощью семейства системных вызовов `setuid()`. Существует несколько версий этого системного вызова для охвата различных идентификаторов пользователей, связанных с процессом, о чем вы узнаете в подразделе 7.9.2.

В ядре есть основные правила, регламентирующие, что может и чего не может делать процесс, и вот три основных принципа, которых придерживаются исполняемые файлы `setuid` и `setuid()`:

- Процесс может запускать исполняемый файл `setuid`, если у него есть соответствующие права доступа к файлам.
- Процесс, запущенный от имени суперпользователя (ID пользователя 0), может задействовать `setuid ()`, чтобы стать любым другим пользователем.
- Процесс, запущенный не от имени суперпользователя, имеет ограничения на то, как он может применять `setuid()`, — в большинстве случаев это запрещено.

Как следствие из этих правил: если вы хотите переключить идентификаторы пользователей с обычного пользователя на другого пользователя, потребуется применить несколько методов. Например, исполняемый файл `sudo` имеет права доступа root для `setuid`, и после запуска он может вызвать `setuid()`, чтобы стать другим пользователем.

ПРИМЕЧАНИЕ

По сути, переключение пользователей не имеет ничего общего с паролями или именами пользователей. Это исключительно концепции пользовательского пространства, как вы впервые увидели в подразделе 7.3.1, когда рассказывалось о файле /etc/passwd. Более подробно о том, как это работает, говорится в подразделе 7.9.4.

7.9.2. Владельцы процессов, действующий UID, реальный UID и сохраненный UID

До сих пор мы поверхностно касались темы ID пользователей. На самом деле каждый процесс имеет более одного ID пользователя. Ранее мы изучили *действующие идентификаторы* (действующий UID, effective user ID, или `euid`), которые определяют права доступа для процесса (что наиболее важно, права доступа к файлам). Второй идентификатор пользователя — *реальный идентификатор пользователя* (реальный UID, real user ID, или `ruid`) указывает, кто инициировал процесс. Обычно эти идентификаторы идентичны, но, когда вы запускаете программу `setuid`, Linux устанавливает `euid` для владельца программы во время выполнения и при этом сохраняет ваш исходный идентификатор пользователя в `ruid`.

Разница между действующим и реальными UID настолько запутанна, что множество документации, касающейся владения процессом, содержит неверную информацию.

Представьте, что `euid` — это исполнитель, а `ruid` — владелец. `ruid` определяет пользователя, который может взаимодействовать с запущенным процессом и, что наиболее важно, завершать процессы и отправлять сигналы процессу. Например, если пользователь А запускает новый процесс, который выполняется от имени пользователя Б (на основе разрешений `setuid`), то пользователь А по-прежнему владеет процессом и может его принудительно завершить.

Мы видели, что большинство процессов имеют одинаковые `euid` и `ruid`. В результате выходные данные по умолчанию для `ps` и других программ диагностики системы показывают только идентификатор `euid`. Вы можете просмотреть оба идентификатора пользователей в своей системе с помощью команды

```
$ ps -eo pid,euser,ruser,comm
```

но оба столбца идентификаторов пользователей могут быть идентичны для всех процессов в ней, и это не должно вас удивлять.

Чтобы создать исключение и увидеть разные значения в столбцах, поэкспериментируйте, создав копию `setuid` команды `sleep`, запустив ее в течение нескольких секунд, а затем выполнив предыдущую команду `ps` в другом окне до завершения выполнения копии.

Еще большую путаницу создает то, что помимо реальных и действующих идентификаторов пользователей существует также *сохраненный идентификатор пользователя* (saved user ID, который обычно не сокращается). Процесс во время выполнения может переключить свой идентификатор `euid` на идентификатор `ruid` или сохраненный идентификатор пользователя. (Еще больше усложняет ситуацию наличие в Linux еще одного, к счастью, редко применяемого идентификатора пользователя — *идентификатора пользователя файловой системы (file system user ID)*, или `fsuid`, который определяет пользователя, получающего доступ к файловой системе.)

Типичное поведение программы setuid

Концепция идентификатора `ruid` может отличаться от того, что вы знаете о работе с идентификаторами. Почему же вам прежде не приходилось иметь дело с другими идентификаторами пользователей? Например, если нужно принудительно завершить процесс, запущенный с помощью команды `sudo`, вы все равно должны применять `sudo` и не можете убить процесс от имени обычного пользователя. И разве в этом случае обычный пользователь не должен иметь идентификатора `ruid`, который дает подходящие разрешения и права?

Причина в том, что `sudo` и многие другие программы `setuid` явно изменяют `euid` *и* `ruid` с помощью одного из системных вызовов `setuid()`. Это происходит потому, что часто возникают непредвиденные побочные эффекты и проблемы с доступом, когда все идентификаторы пользователей не совпадают.

> **ПРИМЕЧАНИЕ**
>
> Если вас интересуют подробности и правила, касающиеся переключения идентификаторов пользователей, прочитайте страницу руководства setuid(2) и проверьте другие страницы руководства, перечисленные в разделе SEE ALSO («См. также»). Существует множество вариантов системных вызовов для различных ситуаций.

Некоторые программы не любят идентификатор `ruid` суперпользователя. Чтобы предотвратить изменение правил `sudo`, добавьте в свой файл `/etc/sudoers` следующую строку (остерегайтесь побочных эффектов для других программ, которые будете запускать от имени суперпользователя!):

```
Defaults    stay_setuid
```

Последствия для безопасности

Поскольку ядро Linux обрабатывает все пользовательские переключения (и как следствие, права доступа к файлам) с помощью программ `setuid` и последующих системных вызовов, разработчики и администраторы систем должны быть предельно осторожны в том, что касается:

- количества и качества программ, имеющих разрешения `setuid`;
- действия этих программ.

Если вы создадите копию оболочки `bash` с root `setuid`, любой локальный пользователь сможет выполнить ее и полностью запустить систему. Да, это действительно так просто. Однако даже специальная программа с root `setuid` может представлять опасность, если в ней есть ошибки. Слабые места в программах, запущенных от имени суперпользователя, — основной путь вторжения в системы, а вторжения случаются часто.

Поскольку существует множество способов взлома системы, предотвращение этого — дело многогранное и масштабное. Один из наиболее важных способов избежать нежелательной активности в вашей системе — принудительно идентифицировать пользователей с помощью имен и надежных паролей.

7.9.3. Идентификация пользователя, аутентификация и авторизация

Многопользовательская система должна обеспечивать базовую поддержку безопасности пользователей в трех областях: идентификации, аутентификации и авторизации. *Идентификация* отвечает на вопрос о том, *кем* являются пользователи. *Аутентификация* просит их *доказать*, что они именно те, кем назвались. Наконец, *авторизация* применяется для определения и ограничения разрешенной деятельности пользователей.

В процессе идентификации пользователя ядро Linux задействует только числовые идентификаторы пользователей для процесса и владельца файла. Ядро знает правила авторизации для запуска исполняемых файлов `setuid` и для того, как идентификаторы пользователей могут запускать семейство системных вызовов `setuid()` для перехода от одного пользователя к другому. Однако ядро ничего не знает об аутентификации — именах пользователей, паролях и пр. Практически все, что связано с аутентификацией, происходит уже в пространстве пользователя.

Мы обсудили сопоставление идентификаторов пользователей и паролей в подразделе 7.3.1, теперь же рассмотрим, как пользовательские процессы получают доступ к этой процедуре. Начнем с упрощенного варианта, когда пользовательский процесс хочет знать свое имя пользователя *`username`* (имя, соответствующее идентификатору `euid`). В традиционной системе Unix процесс, чтобы получить свое имя пользователя, может выполнить следующие действия:

1. Запросить у ядра свой идентификатор `euid` с помощью системного вызова `getuid()`.
2. Открыть файл `/etc/passwd` и начать чтение с начала.
3. Считать строку файла `/etc/passwd`. Если считывать больше нечего, процесс не смог найти имя пользователя.
4. Разобрать строку на поля, разбивая их двоеточиями. Третье поле — это идентификатор пользователя для текущей строки.

5. Сравнить идентификатор шага 4 с идентификатором шага 1. Если они идентичны, значит, первое поле на шаге 4 — это желаемое имя пользователя, и процесс может прекратить поиск и применить его.
6. Перейти к следующей строке в файле `/etc/passwd` и возвратиться к шагу 3.

Это длительная процедура, а ее реализация, как правило, еще более сложна.

7.9.4. Применение библиотек для получения информации о пользователе

Если бы каждый разработчик, которому нужно знать текущее имя пользователя, должен был написать весь код из приведенного ранее примера, система была бы ужасно распределенной, раздутой и непригодной для работы. К счастью, существуют стандартные библиотеки, которые мы можем использовать для выполнения повторяющихся задач, подобных данной. В этом случае для получения имени пользователя нужно всего лишь вызвать функцию `getpwuid()` в стандартной библиотеке после получения ответа от `geteuid()`. (Подробнее о том, как они работают, см. на страницах руководства для этих вызовов.)

Стандартная библиотека для исполняемых файлов в вашей системе разделяемая, таким образом, вы можете внести существенные изменения в реализацию аутентификации без изменения какой-либо программы. Например, можете отказаться от применения `/etc/passwd` для своих пользователей, а вместо этого взять сетевую службу, такую как LDAP, изменив только конфигурацию системы.

Этот подход хорошо работает для идентификации имен пользователей, связанных с идентификаторами пользователей, но для паролей неэффективен. В подразделе 7.3.1 описывается, как традиционно зашифрованный пароль был частью `/etc/passwd`, поэтому, если вы хотите проверить пароль, введенный пользователем, то должны зашифровать все, что ввел пользователь, и сравнить с содержимым файла `/etc/passwd`.

Подобная традиционная реализация имеет множество ограничений, в том числе:

- не позволяет установить общесистемный стандарт для протокола шифрования;
- предполагает, что у вас есть доступ к зашифрованному паролю;
- предполагает, что система будет запрашивать у пользователя пароль каждый раз, когда он хочет получить доступ к чему-то, что требует аутентификации (а это очень раздражает);
- предполагает, что вы хотите использовать пароли. Если вы захотите применять одноразовые токены, смарт-карты, биометрические данные или какую-либо другую форму аутентификации пользователя, потребуется добавить эту функцию самостоятельно.

Некоторые из этих ограничений способствовали разработке пакета *теневых (shadow)* паролей, описанного в подразделе 7.3.3, который стал первым шагом

в обеспечении возможности настройки паролей в масштабах всей системы. Но решение основной массы проблем пришло с разработкой и внедрением модулей PAM.

7.10. Подключаемые модули аутентификации (PAM)

Чтобы обеспечить гибкость аутентификации пользователей, в 1995 году компания Sun Microsystems предложила новый стандарт под названием *«подключаемые модули аутентификации» (Pluggable Authentication Module, PAM)* — систему разделяемых библиотек аутентификации (Open Software Foundation RFC 86.0, октябрь 1995 года). Для аутентификации пользователя приложение передает пользователя утилите PAM, чтобы определить, может ли он успешно идентифицировать себя. Таким образом, получилось довольно легко добавить поддержку дополнительных методов аутентификации, таких как двухфакторные и физические ключи. В дополнение к поддержке механизма аутентификации PAM предоставляет ограниченный объем контроля авторизации для служб (например, если вы хотите запретить определенную службу, такую как `cron`, определенным пользователям).

Поскольку существует множество сценариев аутентификации, PAM задействует ряд динамически загружаемых *модулей аутентификации*. Каждый модуль выполняет определенную задачу и является разделяемым объектом, который процессы могут динамически загружать и запускать в своем исполняемом пространстве. Например, `pam_unix.so` — это модуль, который может проверить пароль пользователя.

На самом деле эта технология довольно сложна. Интерфейс программирования непрост, и до сих пор неясно, действительно ли PAM решает все существующие проблемы. Тем не менее PAM поддерживается почти в каждой программе, требующей аутентификации в системе Linux, большинство дистрибутивов также применяют PAM. И поскольку PAM работает поверх существующего API аутентификации Unix, интеграция поддержки в клиент требует небольшой дополнительной работы (если вообще потребует).

7.10.1. Конфигурация PAM

Изучать основы работы модулей PAM начнем с их конфигурации. Файлы конфигурации приложений PAM находятся в каталоге `/etc/pam.d` (более старые системы могут использовать один файл `/etc/pam.conf`). Большинство установок содержат много файлов, поэтому может быть неясно, с чего начать. Некоторые имена файлов, такие как `cron` и `passwd`, соответствуют частям системы, с которыми вы уже знакомы.

Поскольку конкретная конфигурация этих файлов значительно варьируется в разных дистрибутивах, довольно трудно найти общий пример. Рассмотрим пример строки конфигурации, которая работает для `chsh` (программа изменения оболочки — CHange SHell):

```
auth            requisite     pam_shells.so
```

В этой строке говорится, что оболочка пользователя должна быть указана в файле `/etc/shells`, чтобы пользователь мог успешно пройти аутентификацию в программе `chsh`. Давайте рассмотрим, как это сделать. Каждая строка конфигурации содержит три поля в следующем порядке: тип функции, аргумент управления и модуль. Их значения для этого примера таковы:

- **Тип функции.** Функция, которую пользовательское приложение запрашивает у PAM. В этом случае `auth` — аутентификация пользователя.
- **Аргумент управления.** Этот параметр определяет, что делает PAM после успешного или неудачного выполнения своего действия для текущей строки (requisite в примере). Мы еще вернемся к этой теме.
- **Модуль.** Модуль аутентификации, который запускается для строки, определяет, что она делает. В примере модуль `pam_shells.so` проверяет, указана ли текущая оболочка пользователя в файле `/etc/shells`.

Конфигурация PAM подробно описана на странице руководства `pam.conf(5)`. Рассмотрим несколько основных моментов.

Типы функции

Пользовательское приложение может попросить PAM выполнить одну из следующих четырех функций:

- `auth` — аутентифицировать пользователя (узнать, является ли он тем, за кого себя выдает);
- `account` — проверить статус учетной записи пользователя (например, имеет ли он право что-то делать);
- `session` — выполнить что-то только для текущего сеанса пользователя (например, отобразить «сообщение дня» (message of the day, motd));
- `password` — изменить пароль пользователя или другие учетные данные.

Для любой строки конфигурации модуль и функция вместе определяют действие модулей PAM. Модуль может иметь несколько типов функций, поэтому при определении назначения строки конфигурации не забывайте, что функция и модуль работают в паре. Например, модуль `pam_unix.so` проверяет пароль при выполнении функции `auth`, но устанавливает пароль при выполнении функции `password`.

Управляющие аргументы и стек правил

Одной из важных особенностей PAM является то, что правила, указанные в строках конфигурации модулей, образуют *стек*, что означает: вы можете применять множество правил при выполнении функции. Вот почему важен аргумент управления: успех или неудача действия в одной строке может повлиять на последующие строки или привести к успешному или неудачному выполнению всей функции.

Существует два вида управляющих аргументов: с простым синтаксисом и более сложным. Вот три основных управляющих аргумента с простым синтаксисом, которые вы можете найти в правиле:

- `sufficient` — если это правило выполняется, аутентификация проходит успешно и PAM больше не нужно просматривать какие-либо правила. Если правило не выполняется, PAM переходит к дополнительным правилам;
- `requisite` — если это правило выполняется успешно, PAM переходит к дополнительным правилам. Если правило не выполняется, аутентификация завершается неудачно и PAM больше не нужно просматривать какие-либо правила;
- `required` — если это правило выполняется успешно, PAM переходит к дополнительным правилам. Если правило не выполняется, PAM переходит к дополнительным правилам, но всегда возвращает неудачную аутентификацию независимо от конечного результата применения дополнительных правил.

Пример 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
```

При такой конфигурации, когда команда `chsh` запрашивает у PAM функцию аутентификации, PAM выполняет следующее (см. блок-схему на рис. 7.4):

1. Модуль `pam_rootok.so` проверяет, не пытается ли суперпользователь пройти аутентификацию. Если это так, процесс немедленно выполнится успешно и не будет пытаться выполнить дальнейшую аутентификацию. Это сработает, потому что аргумент управления имеет значение `sufficient`, что означает: выполнения этого действия достаточно для того, чтобы модуль PAM немедленно сообщил об успехе `chsh`. В противном случае он переходит к шагу 2.
2. Модуль `pam_shells.so` проверяет, указана ли оболочка пользователя в файле `/etc/shells`. Если ее там нет, модуль возвращает ошибку и аргумент управления `requisite` указывает, что модуль PAM должен немедленно сообщить об этом сбое `chsh` и не пытаться выполнить аутентификацию. В противном случае модуль успешно завершит выполнение и в соответствии с требованиями аргумента управления `requisite` перейдет к шагу 3.
3. Модуль `pam_unix.so` запрашивает у пользователя пароль и проверяет его. Аргументу управления присвоено значение `sufficient`, поэтому успеха данного модуля (правильного пароля) достаточно, чтобы PAM сообщил об этом оболочке `chsh`. Если пароль неверен, PAM переходит к шагу 4.
4. Модуль `pam_deny.so` никогда не выполняется, и поскольку аргумент управления установлен в значение `required`, PAM сообщает об ошибке в оболоч-

ку `chsh`. Это значение используется по умолчанию для случаев, когда больше нет вариантов действий. (Обратите внимание, что аргумент управления `required` не приводит к немедленному сбою функции PAM: он будет запускать все строки, оставшиеся в его стеке, но при этом модуль всегда будет сообщать приложению о сбое.)

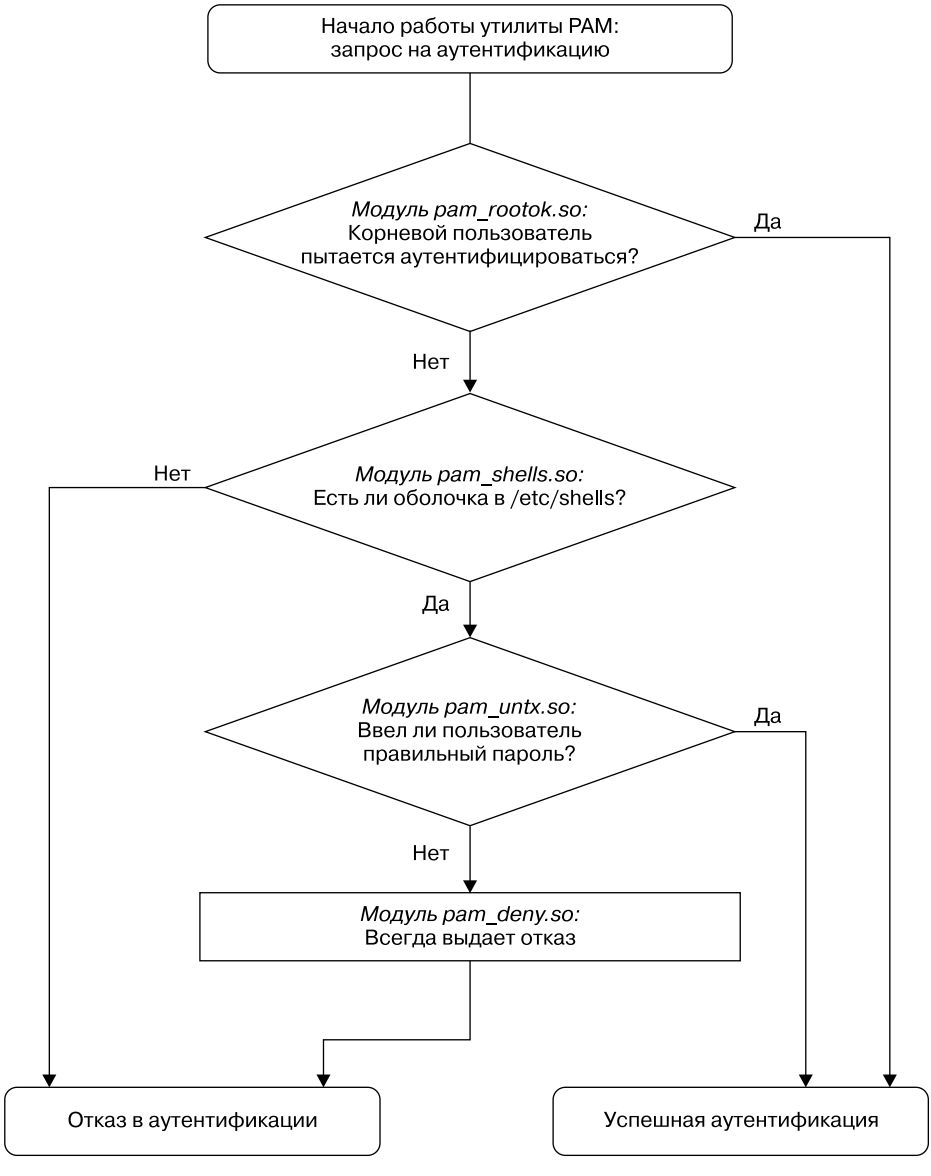


Рис. 7.4. Диаграмма выполнения правила PAM

ПРИМЕЧАНИЕ

Не путайте термины «функция» и «действие», работая с модулями PAM. Функция является целью высокого уровня — тем, чего требует пользовательское приложение (например, аутентифицировать пользователя). Действие — это конкретный шаг, который PAM предпринимает для достижения этой цели. Просто помните, что сначала пользовательское приложение вызывает функцию, а потом PAM применяет действия для достижения нужной цели.

Расширенный синтаксис аргументов управления, указанный в квадратных скобках (`[]`), позволяет вручную управлять реакцией на основе конкретного возвращаемого значения модуля (а не только успеха или неудачи). Дополнительные сведения см. на странице руководства `pam.conf(5)`: когда вы поймете, как работает простой синтаксис, у вас не возникнет проблем с расширенным синтаксисом.

Аргументы модуля

Модули PAM могут принимать аргументы, стоящие после имени модуля. Вы часто будете сталкиваться с подобным вариантом для модуля `pam_unix.so`:

```
auth       sufficient    pam_unix.so    nullok
```

Аргумент `nullok` в примере сообщает, что у пользователя может не быть пароля (по умолчанию то, что у него нет пароля, будет провоцировать ошибку).

7.10.2. Советы по синтаксису конфигурации PAM

Благодаря возможностям потока управления и синтаксису аргументов модуля синтаксис конфигурации PAM обладает многими функциями языка программирования и определенной степенью мощности. Ранее мы лишь слегка коснулись этой темы, рассмотрим еще несколько советов по поводу конфигурации PAM.

- Чтобы узнать, какие модули PAM присутствуют в вашей системе, введите команду `man -k pam_` (обратите внимание на подчеркивание). Могут возникнуть трудности при поиске местоположения модулей. Введите команду `locate pam_unix.so` и посмотрите, какой результат она выведет.
- Страницы руководства содержат функции и аргументы для каждого модуля.
- Многие дистрибутивы автоматически генерируют определенные файлы конфигурации PAM, поэтому не стоит изменять их непосредственно в файле `/etc/pam.d`. Прочитайте комментарии в ваших файлах `/etc/pam.d` перед их редактированием: если файлы сгенерированы, комментарии в них сообщат, откуда они взялись.
- Файл конфигурации `/etc/pam.d/other` определяет конфигурацию по умолчанию для любого приложения, в котором нет собственного файла конфигурации. По умолчанию чаще всего конфигурация запрещает любые действия.

- Существуют различные способы включения дополнительных файлов конфигурации в файл конфигурации PAM. Синтаксис `@include` загружает весь файл конфигурации, но вы также можете задействовать аргумент управления для загрузки только конфигурации для определенной функции. Использование варьируется в зависимости от дистрибутивов.
- Конфигурация PAM не ограничивается аргументами модуля. Некоторые модули могут получать доступ к дополнительным файлам в `/etc/security`, обычно для настройки ограничений для каждого пользователя.

7.10.3. Модули PAM и пароли

В связи с развитием проверки паролей Linux на протяжении многих лет существует ряд артефактов конфигурации паролей, которые могут вызывать путаницу. Во-первых, файл `/etc/login.defs`. Это файл конфигурации для исходного набора shadow-паролей. Он содержит информацию об алгоритме шифрования, используемом для файла пароля `/etc/shadow`, но редко применяется в системе с установленным PAM, поскольку конфигурация PAM уже содержит эту информацию. Тем не менее алгоритм шифрования в `/etc/login.defs` должен соответствовать конфигурации PAM в тех редких случаях, когда вы запускаете приложение, не поддерживающее PAM.

Откуда PAM получает информацию о схеме шифрования паролей? Помните, что у PAM есть два способа взаимодействия с паролями: функция `auth` (для проверки пароля) и функция `password` (для установки пароля). Проще всего отследить параметр установки пароля с помощью команды `grep`:

```
$ grep password.*unix /etc/pam.d/*
```

Соответствующие строки должны содержать `pam_unix.so` и выглядеть примерно так:

```
password        sufficient    pam_unix.so obscure sha512
```

Аргументы `obscure` и `sha512` указывают PAM, что делать при установке пароля. Сначала PAM проверяет, достаточно ли надежен пароль (среди прочего — не похож ли новый пароль на старый), а затем применяет алгоритм `SHA512` для шифрования нового пароля.

Но это происходит *только* тогда, когда пользователь *устанавливает* пароль, а не когда PAM *проверяет* пароль. Итак, как PAM узнает, какой алгоритм применять при аутентификации? К сожалению, конфигурация вам ничего не сообщит, для этого нет аргументов шифрования `pam_unix.so` для функции `auth`. Страницы руководства также ничего не расскажут.

Оказывается, модуль `pam_unix.so` просто пытается угадать алгоритм, обычно передавая библиотеке `libcrypt` грязную работу, перепробовав целую кучу вещей, пока что-то не сработает или не останется ничего, что можно было бы попробовать (так

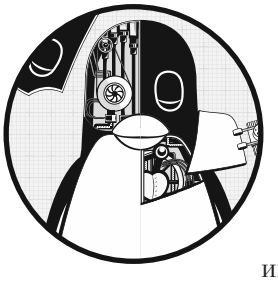
было на момент написания книги). Таким образом, вам не нужно беспокоиться об алгоритме шифрования проверки.

7.11. Что дальше?

Сейчас мы находимся примерно в середине книги и уже успели рассмотреть многие жизненно важные блоки системы Linux. Обсуждение журналирования и пользователей в системе Linux показало, как можно разделить службы и задачи на небольшие независимые фрагменты, которые все еще могут в определенной степени взаимодействовать друг с другом. В этой главе речь шла почти исключительно о пользовательском пространстве, и теперь нам необходимо конкретизировать свое представление о процессах пользовательского пространства и потребляемых ими ресурсах. Для этого в главе 8 мы вернемся к обсуждению ядра.

8

Процессы и использование ресурсов



В этой главе мы глубже изучим взаимосвязи между процессами, ядром и системными ресурсами. Существуют три основных вида аппаратных ресурсов: процессор, память и устройства ввода-вывода. Процессы соперничают за эти ресурсы, и задача ядра заключается в справедливом их распределении. Само ядро также является ресурсом — программным ресурсом, который используется для выполнения таких задач, как создание новых процессов и взаимодействие с другими процессами.

Многие из инструментов, которые встретятся нам в этой главе, считаются инструментами мониторинга производительности. Они особенно полезны, если работа вашей системы замедлилась и вы хотите понять почему. Однако особо отвлекаться на производительность не стоит. Попытка оптимизировать систему, которая уже работает правильно, — пустая трата времени. Настройки по умолчанию в большинстве систем оптимальны, поэтому вы должны изменять их только в том случае, если в этом появилась крайняя необходимость. Вместо этого давайте сосредоточимся на том, что на самом деле измеряют инструменты, как ядро работает и взаимодействует с процессами.

8.1. Отслеживание процессов

В разделе 2.16 мы узнали, как с помощью команды `ps` получить список процессов, запущенных в вашей системе в определенное время. Команда `ps` выводит текущие процессы и статистику их применения, но мало что рассказывает о том, как

процессы меняются с течением времени, потому она не сразу поможет вам определить, какой процесс использует слишком много процессорного времени или памяти.

Команда `top` предоставляет интерактивный интерфейс отображения информации из команды `ps`. Она показывает текущее состояние системы, а также поля из списка `ps` и обновляется каждую секунду. Возможно, самое важное заключается в том, что `top` выводит наиболее активные процессы (по умолчанию те, которые в настоящее время занимают больше всего процессорного времени) в верхней части экрана.

Вы можете отправлять команды в интерфейс `top` с помощью горячих клавиш. Наиболее часто используемые команды относятся к изменению порядка сортировки или фильтрации списка процессов. Вот соответствующие им клавиши:

- `Пробел` — немедленно обновляет отображение;
- `M` — сортирует по текущему использованию памяти;
- `T` — сортирует по общей (совокупной) загрузке процессора;
- `P` — сортирует по текущей загрузке процессора (по умолчанию);
- `u` — отображает только процессы одного пользователя;
- `f` — выбирает другую статистику для отображения;
- `?` — отображает краткую инструкцию по применению для всех основных команд `top`.

ПРИМЕЧАНИЕ

Горячие клавиши команды top чувствительны к регистру.

У двух аналогичных утилит, `atop` и `htop`, расширенный набор отображений и функций. Большинство дополнительных их функций добавляют функциональность из других инструментов. Например, `htop` использует многие возможности команды `lsof`, описанные в следующем разделе.

8.2. Поиск открытых файлов с помощью команды lsof

Команда `lsof` выводит список открытых файлов и использующих их процессов. Поскольку Unix уделяет большое внимание файлам, `lsof` является одним из наиболее полезных инструментов для поиска проблемных мест. Но `lsof` не ограничивается обычными файлами, она может выводить сетевые ресурсы, динамические библиотеки, конвейеры и многое другое.

8.2.1. Считывание вывода команды lsof

Запуск команды `lsof` в командной строке обычно приводит к огромному количеству выходных данных. Далее приведен фрагмент того, что вы можете увидеть.

Этот вывод (слегка скорректированный для удобства чтения) включает открытые файлы из процесса `systemd` (`init`), а также запущенный процесс `vi`:

```
# lsof

COMMAND PID  USER   FD TYPE DEVICE SIZE/OFF  NODE NAME
systemd   1  root  cwd  DIR 8,1     4096        2 /
systemd   1  root  rtd  DIR 8,1     4096        2 /
systemd   1  root  txt  REG 8,1  1595792  9961784 /lib/systemd/systemd
systemd   1  root  mem  REG 8,1  1700792  9961570 /lib/x86_64-linux-gnu/libm-2.27.so
systemd   1  root  mem  REG 8,1   121016  9961695 /lib/x86_64-linux-gnu/libudev.so.1

--пропуск--
vi     1994  juser cwd  DIR 8,1     4096  4587522 /home/juser
vi     1994  juser  3u  REG 8,1    12288   786440 /tmp/.ff.swp

--пропуск--
```

В выводе в верхней строке перечислены следующие поля:

- `COMMAND` — имя команды для процесса, содержащего файловый дескриптор;
- `PID` — идентификатор процесса;
- `USER` — пользователь, выполняющий процесс;
- `FD` — это поле может содержать два вида элементов. В большей части представленного вывода команды столбец `FD` показывает назначение файла. В поле `FD` также может быть указан *файловый дескриптор (File Descriptor)* открытого файла — число, которое процесс использует вместе с системными библиотеками и ядром для идентификации файла и управления им. В последней строке показан файловый дескриптор 3;
- `TYPE` — тип файла (обычный файл, каталог, сокет и т. д.);
- `DEVICE` — основной и второстепенный номера устройства, на котором хранится файл;
- `SIZE/OFF` — размер файла;
- `NODE` — номер `inode` файла;
- `NAME` — имя файла.

Страница руководства `lsof(1)` содержит полный список того, что можно встретить в каждом из полей, но вывод должен быть очевидным. Например, посмотрите на записи с `cwd` в поле `FD`. Эти строки указывают текущие рабочие каталоги процессов.

Другой пример — самая последняя строка, в которой показан временный файл, используемый процессом `vi` пользователя (`PID 1994`).

ПРИМЕЧАНИЕ

Вы можете запустить команду lsof и от имени суперпользователя, и от имени обычного пользователя, но больше информации получите, заходя от имени суперпользователя.

8.2.2. Использование команды lsof

Существует два основных способа использования команды `lsof`:

- перечислить все и передать вывод в команду типа `less`, а затем найти то, что нужно. Это может занять некоторое время из-за большого объема данных;
- сузить список, выводимый командой `lsof`, с помощью параметров командной строки.

Вы можете использовать параметры командной строки, чтобы указать имя файла в качестве аргумента, и в `lsof` будут перечислены только те записи, которые соответствуют аргументу. Например, следующая команда отображает записи для файлов, открытых в `/usr` и всех его подкаталогах:

```
$ lsof +D /usr
```

Чтобы просмотреть список открытых файлов для определенного идентификатора процесса, запустите следующую команду:

```
$ lsof -p pid
```

Чтобы увидеть краткое описание множества параметров `lsof`, запустите команду `lsof -h`. Большинство параметров относятся к формату вывода (см. главу 10, в которой будет рассказано о сетевых функциях `lsof`).

ПРИМЕЧАНИЕ

Команда lsof сильно зависит от информации о ядре. Если вы выполните обновление дистрибутива как для ядра, так и для команды lsof, обновленная lsof может не работать до тех пор, пока вы не перезагрузите ядро.

8.3. Отслеживание выполнения программ и системных вызовов

Инструменты, которые мы встречали ранее, исследуют активные процессы. Однако если вы понятия не имеете, почему программа завершается почти сразу после запуска, команда `lsof` вам не поможет. На самом деле будет трудно даже запустить `lsof` одновременно с неудачной командой.

Команды `strace` (system call trace — отслеживание системных вызовов) и `ltrace` (library trace — отслеживание библиотек) помогут определить, что пытается сделать программа. Эти инструменты выдают чрезвычайно большой вывод, но как только вы узнаете, что искать, в вашем распоряжении будет больше информации для отслеживания проблем.

8.3.1. Команда strace

Напомним, что *системный вызов* — это привилегированная операция, которую процесс пользовательского пространства запрашивает у ядра, например открытие

файла и чтение данных из него. Утилита `strace` выводит все системные вызовы, выполняемые процессом. Чтобы увидеть ее в действии, выполните команду

```
$ strace cat /dev/null
```

По умолчанию `strace` отправляет свой вывод в стандартную ошибку `stderr`. Если хотите сохранить выходные данные в файле, используйте параметр `-o` *`save_file`*, где *`save_file`* — имя файла. Вы можете перенаправить данные, добавив параметр `2> save_file` в командную строку, а также зафиксировать любую стандартную ошибку в исследуемой команде.

В главе 1 мы узнали, что когда один процесс хочет запустить другой процесс, он делает системный вызов `fork()`, чтобы создать свою копию, а затем она использует один из вызовов `exec()` для запуска новой программы. Команда `strace` начинает работу над новым процессом (копией исходного процесса) сразу после вызова функции `fork()`. Поэтому в первых строках вывода этой команды должна отображаться функция `execve()` в действии, за которой следует вызов инициализации памяти `brk()`, как показано далее:

```
execve("/bin/cat", ["cat", "/dev/null"], 0x7ffef0be0248 /* 59 vars */) = 0
brk(NULL)                               = 0x561e83127000
```

Следующая часть вывода посвящена в основном загрузке разделяемых библиотек. Эту информацию можно пропустить, если только вы действительно не хотите углубляться в систему разделяемых библиотек:

```
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=119531, ...}) = 0
mmap(NULL, 119531, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fa9db241000
close(3)                                = 0

--пропуск--
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"...,
832) = 832
```

Кроме того, пропустите вывод `mmap`, пока не дойдете до строк в конце вывода, которые выглядят следующим образом:

```
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 1), ...}) = 0
openat(AT_FDCWD, "/dev/null", O_RDONLY) = 3
fstat(3, {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 3), ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7fa9db21b000
read(3, "", 131072)                     = 0
munmap(0x7fa9db21b000, 139264)          = 0
close(3)                                = 0
close(1)                                = 0
```

```
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Эта часть вывода отображает команду в действии. Сначала посмотрите на вызов `openat()` (еще один вариант — `open()`), который открывает файл. Значение `3` — это результат, означающий, что задача успешно выполнена (`3` — это файловый дескриптор, который ядро возвращает после открытия файла). Потом вы можете увидеть, где `cat` считывает данные из `/dev/null` (вызов `read()`, который также содержит `3` в качестве файлового дескриптора). Далее считывать больше нечего, поэтому программа закрывает файловый дескриптор и завершает работу с помощью `exit_group()`.

Что происходит, когда команда обнаруживает ошибку? Используйте команду `strace cat` *`not_a_file`* и изучите вызов `open()` в следующем выводе:

```
openat(AT_FDCWD, "not_a_file", O_RDONLY) = -1 ENOENT (No such file or directory)
```

Поскольку вызов `open()` не смог открыть файл, он вернул значение `-1`, чтобы сообщить об ошибке. Из примера видно, что `strace` сообщает о точной ошибке и выдает ее краткое описание (`No such file or directory` — нет такого файла или каталога).

Наиболее распространенной проблемой в программах Unix являются отсутствие файлов, поэтому, если системный журнал и другая информация в журнале не помогли и вы не находите отсутствующий файл, команда `strace` может быть очень полезна. Вы даже можете использовать ее на демонах, которые разветвляются или отсоединяются от процессов. Например, чтобы отследить системные вызовы вымышленного демона под названием `crummyd`, введите

```
$ strace -o crummyd_strace -ff crummyd
```

В этом примере параметр `-o` для `strace` регистрирует действия любого дочернего процесса, который `crummyd` породил в `crummyd_strace.pid`, где `pid` — идентификатор дочернего процесса.

8.3.2. Команда ltrace

Команда `ltrace` отслеживает вызовы разделяемых библиотек. Вывод аналогичен выводу команды `strace`, но `ltrace` ничего не отслеживает на уровне ядра. Имейте в виду, что вызовов разделяемых библиотек намного больше, чем системных вызовов. Вам определенно нужно будет отфильтровать вывод, и в самом `ltrace` есть множество встроенных параметров, которые в этом вам помогут.

ПРИМЕЧАНИЕ

Дополнительные сведения о разделяемых библиотеках см. в подразделе 15.1.3. Команда ltrace не работает со статически связанными двоичными файлами.

8.4. Потоки

В Linux некоторые процессы разделены на части, называемые *потоками*. Поток очень похож на процесс: у него есть *идентификатор (thread ID, TID — идентификатор потока)*, и ядро планирует и запускает потоки точно так же, как процессы. Однако, в отличие от отдельных процессов, которые обычно независимо используют системные ресурсы, такие как память и подключения к вводу-выводу, все потоки внутри одного процесса совместно задействуют свои системные ресурсы и часть памяти.

8.4.1. Однопоточные и многопоточные процессы

Многие процессы имеют только один поток. Процесс с одним потоком является *однопоточным*, а процесс с более чем одним потоком — *многопоточным*. Все процессы начинаются с однопоточных. Этот начальный поток обычно называется *основным потоком*. Он может запускать новые потоки, делая процесс многопоточным, подобно тому как процесс может вызывать `fork()` для запуска нового процесса.

ПРИМЕЧАНИЕ

О потоках обычно не упоминается, когда процесс однопоточный. В этой книге не упоминаются потоки, если только многопоточные процессы не влияют на то, что вы видите или делаете.

Основное преимущество многопоточного процесса заключается в следующем: когда у процесса много задач, потоки могут выполняться одновременно на нескольких процессорах, что потенциально ускоряет вычисления. Хотя вы также можете выполнять одновременные вычисления с несколькими процессами, потоки запускаются быстрее, чем процессы, и часто потокам проще или эффективнее взаимодействовать через их общую память, чем процессам — по каналу (сетевое соединение или конвейер).

Некоторые программы используют потоки для решения проблем, связанных с управлением несколькими ресурсами ввода-вывода. Традиционно процесс иногда использует `fork()` для запуска нового подпроцесса, чтобы начать работу с новым входным или выходным потоком. Потоки предлагают аналогичный механизм без дополнительных затрат на запуск нового процесса.

8.4.2. Просмотр потоков

По умолчанию в выводе команд `ps` и `top` отображаются только процессы. Чтобы отобразить информацию о потоке в `ps`, добавьте параметр `m`. В листинге 8.1 показан пример вывода.

Листинг 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 -
12534 tty7    -    668:30 /usr/lib/xorg/Xorg -core :0❸
    - -       Ssl+ 659:55 -
    - -       Ssl+   0:00 -
    - -       Ssl+   0:00 -
    - -       Ssl+   8:35 -
```

В листинге показаны процессы вместе с потоками. Каждая строка с номером в столбце `PID` (❶, ❷ и ❸) представляет процесс, как и в обычном выводе `ps`. Строки со знаками «дефис» в столбце `PID` представляют потоки, связанные с процессом. В этом выводе процессы ❶ и ❷ имеют по одному потоку каждый, а процесс `12534` в ❸ — многопоточный (с четырьмя потоками).

Если вы хотите просматривать идентификаторы потоков с помощью команды `ps`, используйте специальный формат вывода. В листинге 8.2 показаны только идентификаторы PID, TID и команды.

Листинг 8.2. Отображение идентификаторов PID и TID с помощью команды `ps m`

```
$ ps m -o pid,tid,command
  PID   TID    COMMAND
 3587     -    bash
    -  3587    -
 3592     -    bash
    -  3592    -
12534     -    /usr/lib/xorg/Xorg -core :0
    - 12534    -
    - 13227    -
    - 14443    -
    - 14448    -
```

Пример вывода здесь соответствует потокам, показанным в листинге 8.1. Обратите внимание на то, что потоки однопоточных процессов идентичны PIDS — это основной поток. Для многопоточного процесса `12534` поток `12534` также является основным потоком.

ПРИМЕЧАНИЕ

Скорее всего, вы не будете взаимодействовать с отдельными потоками, как с процессами. Чтобы влиять на отдельный поток в какой-то момент, необходимо много знать о том, как была написана многопоточная программа, и даже тогда такое действие — не самая хорошая идея.

Потоки могут сбивать с толку, когда дело доходит до мониторинга ресурсов, поскольку отдельные потоки в многопоточном процессе могут потреблять ресурсы одновременно. Например, `top` по умолчанию не отображает потоки, вам нужно будет нажать клавишу `H`, чтобы включить отображение. Для большинства инструментов мониторинга ресурсов, которые мы рассмотрим в дальнейшем, придется проделать небольшую дополнительную работу, чтобы включить отображение потока.

8.5. Введение в мониторинг ресурсов

Теперь обсудим некоторые темы мониторинга ресурсов, включая процессорное (CPU) время, память и дисковый ввод-вывод. Мы рассмотрим его применение в масштабах всей системы, а также на основе каждого процесса.

Многие пользователи начинают вникать во внутреннюю работу ядра Linux в интересах повышения производительности. Однако большинство систем Linux хорошо работают в соответствии с настройками дистрибутива по умолчанию, и вы можете потратить часы и даже дни, пытаясь повысить производительность своей машины, особенно если не знаете, что именно настраивать. Поэтому вместо того, чтобы думать о производительности, экспериментируя с инструментами, описанными в этой главе, изучите действия ядра при распределении ресурсов между процессами.

8.5.1. Измерение процессорного времени

Чтобы отслеживать один или несколько конкретных процессов с течением времени, используйте параметр `-p` в команде `top`, как показано далее:

```
$ top -p pid1 [-p pid2 ...]
```

Чтобы узнать, сколько процессорного времени команде требуется в течение выполнения, задействуйте команду `time`. К сожалению, здесь возникает некоторая путаница, потому что в большинстве оболочек есть встроенная команда `time`, которая не предоставляет обширной статистики, а в `/usr/bin/time` есть своя системная утилита. Вероятно, вы сначала столкнетесь со встроенной оболочкой `bash`, поэтому попробуйте выполнить `time` с помощью команды `ls`:

```
$ time ls
```

После завершения `ls` команда `time` отобразит вывод, как показано далее:

```
real    0m0.442s
user    0m0.052s
sys     0m0.091s
```

Пользовательское время (user time, user) — это количество секунд, потраченных процессором на выполнение *собственного* кода программы. Некоторые команды выполняются так быстро, что процессорное время приближается к нулю. *Системное время* (system time, sys или system) — это количество времени, которое ядро тратит на выполнение работы процесса, например чтение файлов и каталогов. Наконец, *реальное время* (real time, real) (также называется *затраченным временем*) — это общее время, затраченное на запуск процесса от начала до конца, включая время, затраченное процессором на выполнение других задач. Это число не очень помогает измерить производительность, но если вычесть пользовательское и системное время из реального, вы получите общее представление о том, сколько времени процесс тратит на ожидание системных и внешних ресурсов. Например, время, затраченное

на ожидание ответа сетевого сервера на запрос, будет отображаться как реальное время, а не как пользовательское или системное.

8.5.2. Настройка приоритетов процесса

Вы можете изменить распорядок процесса в ядре, чтобы предоставить определенному процессу больше или меньше процессорного времени, чем другим. Ядро запускает каждый процесс в соответствии с назначенным ему приоритетом, который обозначен числом от –20 до 20, причем –20 — это высший приоритет. (Да, это может сбить с толку.)

Команда `ps -l` отображает текущий приоритет процесса, но с помощью команды `top` проще увидеть приоритеты в действии, как показано далее:

```
$ 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 xfce4-session
 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 xfce4-panel
```

В выводе команды `top` в столбце `PR` (priority — приоритет) указан текущий приоритет ядра для процесса. Чем больше число, тем меньше вероятность того, что ядро запланирует процесс, если другим потребуется процессорное время. Однако приоритет запуска сам по себе не определяет решение ядра выделить процессорное время процессу, и ядро также может изменять приоритет во время выполнения программы в зависимости от количества процессорного времени, потребляемого процессом.

Рядом со столбцом приоритета находится столбец `NI` *(nice, значение относительного приоритета)*, который дает подсказку планировщику ядра. Именно через это значение можно повлиять на решение ядра. Ядро добавляет значение `nice` к текущему приоритету, чтобы определить следующий временной интервал для процесса. Устанавливая более высокое значение `nice`, вы становитесь «добрее» к другим процессам, потому что ядро повышает их приоритет.

По умолчанию значение `nice` равно 0. Теперь предположим, что вы выполняете большие вычисления в фоновом режиме и не хотите затягивать свой сеанс работы. Чтобы этот процесс отошел на второй план по сравнению с другими процессами и выполнялся только тогда, когда другим задачам нечего делать, можете изменить значение `nice` на 20 с помощью команды `renice`, где `pid` — идентификатор процесса, который вы хотите изменить:

```
$ renice 20 pid
```

Если вы суперпользователь, то можете сделать значение `nice` отрицательным числом, но лучше так не поступать, потому что системные процессы могут перестать получать достаточно процессорного времени. На самом деле вам не придется сильно изменять значения `nice`, потому что во многих системах Linux есть только один пользователь и количество выполняемых им реальных вычислений невелико. (Значение `nice` было гораздо важнее в те времена, когда на одной машине было много пользователей.)

8.5.3. Измерение производительности процессора с помощью среднего значения загрузки

Среднее значение загрузки — это среднее количество процессов, готовых к запуску в данный момент. То есть это оценка количества процессов, которые способны использовать процессор в любой момент времени: сюда входят запущенные процессы и ожидающие возможности задействовать процессор. Имейте в виду, что большинство процессов в вашей системе обычно ждут ввода (например, с клавиатуры, мыши или из сети), что означает: они не готовы к запуску и не должны вносить какие-либо данные в среднее значение нагрузки. На среднюю нагрузку влияют только активные в данный момент процессы.

Команда uptime

Команда `uptime` сообщает три средних значения загрузки дополнительно ко времени всей работы ядра:

```
$ uptime
... up 91 days, ... load average: 0.08, 0.03, 0.01
```

Три цифры, выделенные жирным шрифтом, являются средними значениями загрузки за последние 1 минуту, 5 и 15 минут соответственно. Как вы можете видеть, эта система не очень загружена: в среднем за последние 15 минут на всех процессорах выполнялось всего 0,01 процесса. Другими словами, если бы у вас был только один процессор, он запускал бы приложения для пользовательского пространства только в течение 1 % от последних 15 минут.

Раньше на большинстве ПК средняя загрузка составляла значение, близкое к нулю, если в работе находились не компиляция программы или игра. Среднее значение

загрузки, равное нулю, — хороший знак, так как это означает, что процессор не перегружен и вы экономите энергию.

Однако компоненты пользовательского интерфейса в современных ПК, как правило, занимают больше ресурсов процессора, чем в прошлом. В частности, некоторые веб-сайты (особенно их реклама) делают браузеры ресурсоемкими.

Если средняя загрузка достигает значения 1, это значит, что один процесс, скорее всего, задействует процессор практически постоянно. С помощью команды `top` идентифицируйте этот процесс — обычно он находится в верхней части вывода.

Большинство современных систем имеют несколько процессорных ядер или процессоров, поэтому несколько процессов могут легко выполняться одновременно. Если у вас два ядра, среднее значение загрузки 1 означает, что в любой момент времени, вероятно, активно только одно из ядер, а среднее значение загрузки 2 — что оба ядра заняты.

Управление высокими нагрузками

Высокая средняя загрузка не всегда означает, что у вашей системы возникли проблемы. Система с достаточным объемом памяти и количеством ресурсов ввода-вывода может легко обрабатывать множество запущенных процессов. Если средняя загрузка высока, а система по-прежнему хорошо откликается, не паникуйте: в ней просто много процессов, совместно задействующих процессор. Процессы должны конкурировать друг с другом за процессорное время, в результате им потребуется больше времени для выполнения своих вычислений, чем если бы каждому из них было разрешено постоянно использовать процессор. Другой случай, когда среднее значение высокой загрузки считается нормальным, связан с веб-сервером или вычислительным сервером, где процессы могут запускаться и завершаться так быстро, что механизм измерения среднего значения нагрузки не может эффективно функционировать.

Однако, если средняя загрузка очень высока и вы чувствуете, что система замедляется, у вас могут возникнуть проблемы с производительностью памяти. Если в системе мало памяти, ядро может начать быстро подкачивать память с диска. Когда это произойдет, многие процессы будут готовы к запуску, но для них память может оказаться недоступной, поэтому они останутся в состоянии готовности к запуску (способствуя увеличению значения средней загрузки) гораздо дольше, чем обычно. Далее мы рассмотрим, почему это может произойти, подробнее изучив оперативную память.

8.5.4. Мониторинг состояния памяти

Один из простейших способов проверить состояние памяти системы в целом — это запустить команду `free` или просмотреть каталог `/proc/meminfo`, чтобы узнать, сколько реальной памяти затрачивается для кэширования и буферизации. Как упоминалось ранее, проблемы с производительностью могут возникать из-за нехватки памяти. Если используется не так много кэш-памяти/буферной памяти

(а остальная часть реальной памяти занята), вам может потребоваться больше памяти. Однако слишком легко считать нехватку памяти причиной всех проблем с производительностью на вашей машине.

Как работает память

Как объяснялось в главе 1, центральный процессор имеет блок управления памятью (MMU) для обеспечения гибкости доступа к ней. Ядро помогает MMU, разбивая память, используемую процессами, на более мелкие фрагменты, называемые *страницами*. Ядро поддерживает структуру данных, называемую *таблицей страниц*, которая сопоставляет адреса виртуальных страниц процесса с реальными адресами страниц в памяти. Когда процесс обращается к памяти, MMU преобразует виртуальные адреса, задействуемые процессом, в реальные адреса на основе таблицы страниц ядра.

На самом деле пользовательскому процессу не нужно, чтобы все его страницы памяти немедленно оказывались доступными для запуска. Ядро обычно загружает и выделяет страницы по мере необходимости процесса, эта система известна как загрузка *страниц по требованию*, или просто загрузка по требованию. Чтобы понять, как это работает, рассмотрим, как программа стартует и запускается как новый процесс.

1. Ядро загружает начало кода инструкции программы на страницы памяти.
2. Ядро может выделить несколько страниц рабочей памяти для нового процесса.
3. По мере выполнения процесс может достичь точки, когда следующая инструкция в его коде не будет содержаться ни на одной из страниц, изначально загруженных ядром. В этот момент ядро берет на себя управление, загружает необходимую страницу в память, а затем позволяет программе возобновить выполнение.
4. Аналогично, если программе требуется больше рабочей памяти, чем было выделено изначально, ядро обрабатывает ее, находя свободные страницы (или выделяя место) и назначая их процессу.

Вы можете узнать размер страницы системы, просмотрев конфигурацию ядра:

```
$ getconf PAGE_SIZE
4096
```

Это число выражено в байтах, и значение 4K типично для большинства систем Linux.

Ядро неявно сопоставляет страницы реальной памяти с виртуальными адресами, то есть оно не помещает все доступные страницы в один большой пул и не выделяет их оттуда. Реальная память имеет множество разделов, которые зависят от аппаратных

ограничений, оптимизации ядра смежных страниц и других факторов. Однако вам не следует беспокоиться ни о чем этом, начиная работать с Linux.

Ошибки страницы

Если страница памяти не готова, а процесс уже хочет ее использовать, он вызывает *ошибку страницы*. В этом случае ядро берет на себя управление процессором из процесса, чтобы подготовить страницу. Существует два вида ошибок страницы — незначительные и серьезные.

- **Незначительная ошибка страницы** *(minor page fault)* возникает, когда нужная страница действительно находится в основной памяти, но MMU не знает, где именно. Это может произойти, когда процесс запрашивает больше памяти или когда в MMU недостаточно места для хранения всех расположений страниц для процесса (внутренняя таблица сопоставления MMU обычно довольно мала). В этом случае ядро сообщает MMU о странице и разрешает продолжить процесс. О таких ошибках страницы беспокоиться не стоит, многие из них возникают во время выполнения процесса.
- **Серьезная ошибка страницы** *(major page fault)* возникает, когда нужной страницы вообще нет в основной памяти. Это означает, что ядро должно загрузить ее с диска или из какого-то другого медленного места хранения. Множество серьезных сбоев страниц приведет к сбою системы, потому что ядро должно выполнить значительный объем работы для предоставления страниц, лишая обычные процессы возможности запуска.

  Некоторые серьезные ошибки страницы неизбежны, например возникающие при загрузке кода с диска при первом запуске программы. Самые большие проблемы появляются, если у вас близка к исчерпанию память (Out of Memory), из-за чего ядро начинает переносить страницы рабочей памяти на диск, чтобы освободить место для новых страниц, а это может привести к сбоям.

Вы можете перейти к ошибкам страницы для отдельных процессов с помощью команд `ps`, `top` и `time`. Используйте системную версию `time` (`/usr/bin/time`) вместо встроенной в оболочку. Далее приведен простой пример того, как команда `time` отображает ошибки страницы (вывод команды `cal` сейчас не имеет значения, поэтому мы отбрасываем его, перенаправляя в `/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 видно из выделенного жирным шрифтом текста, при запуске этой программы были две серьезные ошибки страницы и 254 незначительных. Серьезные ошибки страницы возникли, когда ядру потребовалось впервые загрузить программу с диска. Если бы вы снова выполнили эту команду, то, вероятно, не получили бы никаких серьезных ошибок страницы, потому что ядро кэшировало бы страницы с диска.

Если вы предпочитаете видеть ошибки страниц процессов по мере их выполнения, используйте команды `top` или `ps`. При запуске `top` примените параметр `f` для изменения отображаемых полей и выберите `nMaj` в качестве одного из столбцов, чтобы отобразить количество серьезных ошибок страницы. Столбец `vMj` (количество серьезных ошибок страницы с момента последнего обновления) полезен, если вы пытаетесь отследить процесс, который, предположительно, работает неправильно.

Для команды `ps` можете использовать специальный формат вывода, чтобы просмотреть ошибки страницы для определенного процесса. Вот пример для процесса `PID 20365`:

```
$ ps -o pid,min_flt,maj_flt 20365
  PID MINFL MAJFL
20365 834182   23
```

В столбцах `MINFL` и `MAJFL` показано количество незначительных и серьезных ошибок страницы. Конечно, вы можете объединить это с любыми другими вариантами выбора процесса, как описано на странице руководства `ps(1)`.

Просмотр ошибок страниц по процессам может помочь вам сосредоточиться на определенных проблемных компонентах. Но если вас интересует производительность системы в целом, потребуется инструмент для обобщения действий процессора и памяти во всех процессах.

8.5.5. Мониторинг производительности процессора и памяти с помощью команды vmstat

Среди множества инструментов, доступных для мониторинга производительности системы, команда `vmstat` является одной из старейших, и ее применение влечет за собой минимальные накладные расходы. Она позволяет получить высокоуровневое представление о том, как часто ядро подкачивает страницы, насколько занят процессор и как используются ресурсы ввода-вывода.

Чтобы раскрыть возможности команды `vmstat`, требуется понимать ее вывод. Например, вот вывод `vmstat 2`, который сообщает статистику каждые 2 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
```

Вывод подразделяется на категории: `procs` для процессов, `memory` для использования памяти, `swap` для страниц, перемещаемых в область подкачки и обратно, `io` для

применения диска, `system` для количества переключений ядра на код ядра и `cpu` для времени, затраченного различными частями системы.

Приведенный вывод типичен для системы, которая выполняет небольшое количество задач. Стоит начинать изучение с просмотра второй строки вывода — в первой приведены средние значения за все время работы системы. Например, в примере система имеет 320 416 Кбайт памяти, перенесенной на диск (`swpd`), и около 3 027 000 Кбайт (3 Гбайт) реальной памяти `free`. Несмотря на то что используется некоторое пространство подкачки, столбцы `si` с нулевым значением (swap-in, подкачка в) и `so` (swap-out, подкачка из) сообщают, что ядро в настоящее время ничего не подкачивает с диска или на диск. Столбец `buff` указывает объем памяти, который ядро использует для дисковых буферов (см. подраздел 4.2.5).

В крайнем правом углу, под заголовком `CPU`, вы можете увидеть распределение процессорного времени в столбцах `us`, `sy`, `id` и `wa`. Соответственно, в них указан процент времени, которое процессор тратит на пользовательские задачи, системные задачи (ядра), время простоя (idle) и ожидание (waiting) ввода-вывода. В предыдущем примере запущено не так уж много пользовательских процессов (они используют максимум 1 % времени процессора), ядро практически ничего не делает, а процессор бездельничает 99 % времени.

В листинге 8.3 показано, что происходит при запуске большой программы.

Листинг 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 0 ❶
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 18 ❷
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 здесь показано, процессор ❶ начинает применяться в течение длительного периода, особенно в пользовательских процессах. Поскольку свободной памяти достаточно, объем используемого кэша и буферного пространства начинает расти по мере того, как ядро все больше задействует диск.

Позже мы увидим кое-что интересное: обратите внимание на то, что ядро ❷ извлекает в память некоторые страницы из области подкачки (столбец `si`). Это означает, что только что запущенная программа, вероятно, получила доступ к некоторым страницам, используемым совместно с другим процессом, что является обычным явлением, так как многие процессы задействуют код в определенных разделяемых библиотеках только при запуске.

Также обратите внимание: в столбце `b` несколько процессов *заблокированы* (blocked, запрещены к запуску) во время ожидания страниц памяти. В целом объем свободной

памяти уменьшается, но она еще далеко не исчерпана. Также наблюдается значительная активность на диске, о чем свидетельствуют увеличивающиеся числа в столбцах `bi` (blocks in, входящие блоки) и `bo` (blocks out, выходящие блоки).

Вывод получается совсем другим, когда у вас заканчивается память. По мере истощения свободного пространства размеры буфера и кэша уменьшаются, поскольку ядру требуется все больше пространства для пользовательских процессов. Как только ничего не останется, вы увидите активность в столбце `so` (swapped-out, выходящая подкачка), когда ядро начнет перемещать страницы на диск, и в этот момент почти все остальные столбцы вывода изменятся, чтобы отразить объем работы, выполняемой ядром. Вы увидите больше системного времени, больше данных, поступающих на диск и выходящих с него, и больше заблокированных процессов, потому что память, которую они хотят задействовать, недоступна (она была перемещена в область подкачки).

Мы не изучили все столбцы вывода команды `vmstat`. Можете углубиться в эту тему на странице руководства `vmstat(8)`, но сначала стоит узнать больше об управлении памятью ядра из книги Зильбершаца, Гагне и Гэлвина (Silberschatz, Gagne and Galvin's) *Operating System Concepts*, 10-е издание (Wiley, 2018), чтобы понять их.

8.5.6. Мониторинг ввода-вывода I/O

По умолчанию команда `vmstat` предоставляет общую статистику ввода-вывода I/O (Input/Output). Хотя вы можете получить очень подробную информацию об использовании ресурсов для каждого дискового раздела с помощью команды `vmstat -d`, вас может поразить размер вывода, полученного в результате применения этого параметра. Вместо этого попробуйте инструмент только для мониторинга ввода-вывода I/O под названием `iostat`.

ПРИМЕЧАНИЕ

Многие утилиты ввода-вывода, которые мы здесь обсудим, по умолчанию не встроены в большинство дистрибутивов, но их легко можно установить.

Использование команды iostat

Как и в случае с командой `vmstat`, при запуске без каких-либо параметров `iostat` отображает статистику за время работы вашей машины:

```
$ iostat
[информация о ядре]
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`, расположенная вверху, сообщает ту же информацию об использовании процессора, что и другие утилиты, которые вы видели ранее в этой главе, поэтому обратите внимание на нижнюю часть, где отображена следующая информация для каждого устройства:

- `tps` — среднее количество передач данных в секунду;
- `kB_read/c` — среднее количество считываемых килобайт в секунду;
- `kB_wrtn/c` — среднее количество записываемых килобайт в секунду;
- `kB_read` — общее количество прочитанных килобайт;
- `kB_wrtn` — общее количество записанных килобайт.

Сходство с `vmstat` заключается еще и в том, что вы можете предоставить аргумент интервала, например `iostat 2`, для обновления каждые 2 секунды. При использовании интервала может потребоваться отобразить только отчет об устройстве с помощью параметра `-d` (например, `iostat -d 2`).

По умолчанию в выводе `iostat` отсутствует информация о разделах диска. Чтобы отобразить ее, добавьте параметр `-p ALL`. Поскольку в типичной системе много разделов, вы получите много выходных данных. Вот часть того, что можно увидеть:

```
$ iostat -p ALL
--пропуск--
Device:            tps   kB_read/s   kB_wrtn/s   kB_read    kB_wrtn
--пропуск--
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
--пропуск--
sde               0.00         0.00        0.00      1230          0
```

В этом примере `sda1`, `sda2` и `sda5` являются разделами диска `sda`, поэтому столбцы чтения и записи отчасти пересекаются. Однако сумма столбцов разделов не обязательно будет равна значению в столбце диска. Хотя чтение из `sda1` также считается чтением из `sda`, имейте в виду, что можно читать из `sda` напрямую, например при чтении таблицы разделов.

Мониторинг использования ввода-вывода для каждого процесса с помощью команды iotop

Если вы хотите копнуть еще глубже, чтобы увидеть ресурсы ввода-вывода, задействуемые отдельными процессами, изучите инструмент `iotop`. Команда `iotop` используется аналогично команде `top`. Она генерирует постоянно обновляемый дисплей, на котором отображаются процессы, задействующие наибольшее количество ресурсов ввода-вывода, с общей сводкой 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, что наряду со столбцами `user`, `command` и `read/write` вместо столбца `PID` появился столбец `TID`. Инструмент `iotop` — одна из немногих утилит, которая отображает потоки вместо процессов.

Столбец `PRIO` (приоритет) показывает приоритет ввода-вывода. Он похож на приоритет процессора, который мы встречали ранее, но влияет на то, как быстро ядро планирует для процесса операции ввода-вывода для чтения и записи. В приоритете, таком как `be/4`, часть `be` является классом планирования, а номер — уровнем приоритета. Как и в случае с приоритетами CPU, меньшие числа более важны, например, ядро позволяет дать процессу с приоритетом `be/3` больше времени ввода-вывода, чем процессу с приоритетом `be/4`.

Ядро использует класс планирования, чтобы обеспечить больший контроль планирования ввода-вывода. Существует три класса планирования из `iotop`:

- `be` — наибольшая эффективность (Best Effort). Ядро делает все возможное, чтобы справедливо распланировать ввод-вывод для этого класса. Большинство процессов выполняются в этом классе планирования ввода-вывода;
- `rt` — в реальном времени (Real Time). Ядро планирует любой ввод-вывод в реальном времени перед любым другим классом ввода-вывода, несмотря ни на что;
- `idle` — ядро выполняет ввод-вывод для этого класса только тогда, когда нет других операций ввода-вывода, которые необходимо выполнить. Этот класс планирования не имеет уровня приоритета.

Вы можете проверить и изменить приоритет ввода-вывода для процесса с помощью утилиты `ionice`; подробную информацию см. на странице руководства `ionice(1)`. Однако вам, вероятно, никогда не придется беспокоиться о приоритете ввода-вывода.

8.5.7. Мониторинг каждого процесса с помощью команды pidstat

Вы видели, как можно отслеживать определенные процессы с помощью утилит `top` и `iotop`. Однако со временем дисплей вывода обновляется, и каждое обновление удаляет предыдущий вывод. Утилита `pidstat` позволяет видеть потребление ресурсов процессом с течением времени в стиле `vmstat`. Вот простой пример для процесса мониторинга `1329`, обновляемого каждую секунду:

```
$ pidstat -p 1329 1
Linux 5.4.0-48-generic (duplex)       11/09/2020       _x86_64_       (4 CPU)

09:26:55 PM   UID  PID    %usr %system %guest    %CPU    CPU Command
09:27:03 PM  1000  1329    8.00   0.00   0.00    8.00      1 myprocess
09:27:04 PM  1000  1329    0.00   0.00   0.00    0.00      3 myprocess
09:27:05 PM  1000  1329    3.00   0.00   0.00    3.00      1 myprocess
09:27:06 PM  1000  1329    8.00   0.00   0.00    8.00      3 myprocess
09:27:07 PM  1000  1329    2.00   0.00   0.00    2.00      3 myprocess
09:27:08 PM  1000  1329    6.00   0.00   0.00    6.00      2 myprocess
```

Вывод по умолчанию показывает процентное соотношение пользовательского и системного времени, общий процент времени процессора и даже сообщает, на каком процессоре выполнялся процесс. (Столбец `%guest` здесь несколько странный — это процент времени, в течение которого процесс выполнял что-то внутри виртуальной машины. Если вы не работаете с виртуальной машиной, не обращайте на него внимания.)

Хотя `pidstat` по умолчанию показывает использование процессора, он способен отобразить гораздо больше полезной информации. Например, вы можете использовать параметр `-r` для мониторинга памяти и параметр `-d` для включения мониторинга диска. Попробуйте применить их, а затем просмотрите страницу руководства `pidstat(1)`, чтобы увидеть еще больше возможностей для потоков, переключения контекста или всего того, о чем мы говорили в этой главе.

8.6. Группы управления (cgroups)

До сих пор вы видели, как просматривать и отслеживать использование ресурсов. А если вы хотите ограничить количество ресурсов, которые могут потребляться сверх того, что было показано с помощью команды `nice`? Для этого существует несколько традиционных систем, таких как интерфейс POSIX rlimit, но наиболее гибким вариантом для большинства типов ограничений ресурсов в системах Linux в настоящее время является функция ядра `cgroup` (control group — группа управления).

Основная идея заключается в следующем: вы помещаете несколько процессов в группу, что позволяет управлять ресурсами, которые они потребляют, в масштабах всей группы. Например, ограничить объем памяти, который совокупно может потреблять набор процессов, способна `cgroup`.

После создания `cgroup` вы можете добавить в нее процессы, а затем использовать контроллер, чтобы изменить поведение этих процессов. Например, существует контроллер `cpu`, позволяющий ограничивать процессорное время, контроллер `memory` и т. д.

ПРИМЕЧАНИЕ

Хотя systemd широко применяет функцию cgroup и большинство (если не все) групп в вашей системе могут управляться systemd, группы находятся в пространстве ядра и не зависят от systemd.

8.6.1. Различие между версиями cgroup

Существуют две версии `cgroup`, 1 и 2, к сожалению, в настоящее время обе они используются и могут быть настроены в системе одновременно, что может привести к путанице. Версии имеют различный набор функций, и структурные различия между ними можно суммировать следующим образом:

- В `cgroups v1` каждый тип контроллера (`cpu`, `memory` и т. д.) имеет собственный набор `cgroups`. Процесс может принадлежать одной группе на контроллер, то есть он может принадлежать нескольким группам. Например, в `v1` процесс может принадлежать группам `cpu` и `memory`.
- В `cgroups v2` процесс может принадлежать только одной группе `cgroup`. Вы можете настроить различные типы контроллеров для каждой группы.

Чтобы визуализировать разницу, рассмотрим три набора процессов: `A`, `B` и `C`. Мы хотим использовать контроллеры `cpu` и `memory` для каждого из них. На рис. 8.1 показана схема для `cgroups v1`. Всего нам нужны шесть групп, потому что каждая ограничена одним контроллером.

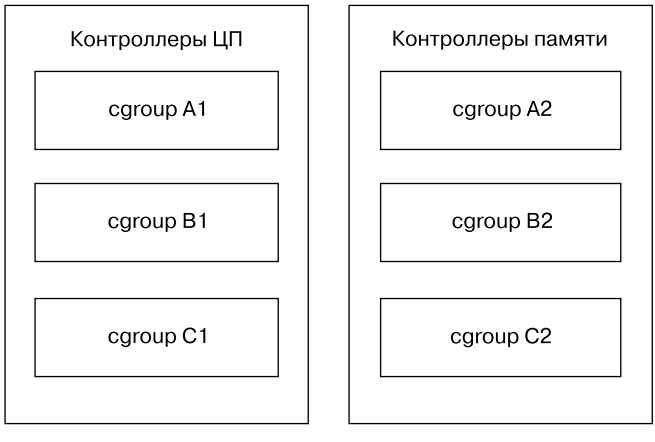


Рис. 8.1. cgroups v1. Процесс может принадлежать одной группе на контроллер

На рис. 8.2 показано, как это происходит в `cgroups v2`. Нам нужны только три группы, потому что для каждой из них можно настроить несколько контроллеров.

Вы можете перечислить контрольные группы `v1` и `v2` для любого процесса, просмотрев его файл `cgroup` в `/proc/<pid>`. Начните с просмотра `cgroups` вашей оболочки с помощью следующей команды:

```
$ cat /proc/self/cgroup
12:rdma:/
11:net_cls,net_prio:/
```

```
10:perf_event:/
9:cpuset:/
8:cpu,cpuacct:/user.slice
7:blkio:/user.slice
6:memory:/user.slice
5:pids:/user.slice/user-1000.slice/session-2.scope
4:devices:/user.slice
3:freezer:/
2:hugetlb:/testcgroup ❶
1:name=systemd:/user.slice/user-1000.slice/session-2.scope
0::/user.slice/user-1000.slice/session-2.scope
```

Рис. 8.2. cgroups v2. Процесс может принадлежать только одной группе

Не пугайтесь, если вывод в вашей системе значительно короче, — это просто означает, что у вас, вероятно, есть только группы управления `cgroups v2`. Каждая строка вывода в примере начинается с номера и представляет собой отдельную группу. Вот несколько советов по поводу того, как читать такой вывод:

- Числа `2-12` относятся к `cgroups v1`. Контроллеры для них перечислены рядом с номером.
- Номер `1` также относится к версии 1, но в нем нет контроллера. Эта группа предназначена только для целей управления (в данном случае ее настроил `systemd`).
- Последняя строка, номер `0`, предназначена для `cgroup v2`. Здесь нет никаких контроллеров. В системе, в которой нет групп управления `v1`, это будет единственная строка вывода.
- Имена являются иерархическими и выглядят как части путей к файлам. В этом примере вы можете видеть, что некоторые группы называются `/user.slice`, а другие — `/user.slice/user-1000.slice/session-2.scope`.
- Имя `/testcgroup` (❶) было создано, чтобы показать, что в группах управления версии 1 группы управления для процесса могут быть полностью независимыми.

- Имена в разделе `user.slice`, включающие сеанс `session`, являются сеансами входа в систему, назначенными `systemd`. Вы увидите их, когда будете просматривать группы управления оболочки. Группы управления для ваших системных служб будут находиться в разделе `system.slice`.

Можно предположить, что `cgroups v1` по сравнению с `v2` более гибки лишь в том, что позволяют назначать процессам различные комбинации контрольных групп. Но на самом деле никто не использует их таким образом, и этот подход был более сложным в настройке и реализации, чем просто наличие одной группы на процесс.

Поскольку `cgroups v1` постепенно выходят из употребления, с этого момента обсуждение будет сосредоточено на `cgroups v2`. Имейте в виду, что, если контроллер задействуется в контрольных группах `v1`, он не может одновременно использоваться в `v2` из-за потенциальных конфликтов. Это означает, что части, зависящие от контроллера, которые мы собираемся обсудить, не будут работать правильно, если ваша система все еще применяет версию 1, но вы все равно сможете попробовать использовать ее, изучив похожие настройки.

8.6.2. Просмотр cgroups

В отличие от традиционного интерфейса системных вызовов Unix для взаимодействия с ядром, доступ к `cgroups` осуществляется полностью через файловую систему, которая обычно монтируется как файловая система `cgroup2` в `/sys/fs/cgroup`. (Если вы используете также `cgroups v1`, она, вероятно, будет находиться в `/sys/fs/cgroup/unified`.)

Давайте рассмотрим настройку `cgroup` оболочки. Откройте оболочку и найдите ее `cgroup` в `/proc/self/cgroup`, как показано ранее. Затем загляните в `/sys/fs/cgroup` или `/sys/fs/cgroup/unified`. Найдя каталог с таким именем, перейдите в него и изучите:

```
$ cat /proc/self/cgroup
0::/user.slice/user-1000.slice/session-2.scope
$ cd /sys/fs/cgroup/user.slice/user-1000.slice/session-2.scope/
$ ls
```

ПРИМЕЧАНИЕ

Имя cgroup может быть довольно длинным в средах рабочего стола, которые любят создавать новую группу для каждого вновь запущенного приложения.

Среди множества файлов, которые могут находиться в этом каталоге, основные файлы интерфейса `cgroup` выделяются тем, что начинаются с `cgroup`. Начните с просмотра `cgroup.procs` (можно использовать команду `cat`), в котором перечислены процессы в `cgroup`. Аналогичный файл, `cgroup.threads`, содержит также потоки.

Чтобы увидеть контроллеры, применяемые в настоящее время для группы, посмотрите на `cgroup.controllers`:

```
$ cat cgroup.controllers
memory pids
```

Большинство групп управления, применяемых для оболочек, имеют эти два контроллера, которые могут управлять объемом используемой памяти и общим количеством процессов в группе. Чтобы взаимодействовать с контроллером, найдите файлы, соответствующие префиксу контроллера. Например, если хотите увидеть количество потоков, запущенных в контрольной группе, обратитесь к `pids.current`:

```
$ cat pids.current
4
```

Чтобы увидеть максимальный объем памяти, который может потреблять группа, взгляните на `memory.max`:

```
$ cat memory.max
max
```

Значение `max` говорит о том, что у этой группы нет определенного ограничения, но поскольку группы являются иерархическими, `cgroup`, возвращающаяся вниз по цепочке подкаталогов, может ограничить ее.

8.6.3. Управление и создание cgroups

Хотя вам, вероятно, никогда не понадобится изменять группы управления, делать это легко. Чтобы поместить процесс в `cgroup`, запишите его `PID` в файл `cgroup.procs` от имени суперпользователя:

```
# echo pid > cgroup.procs
```

Именно так работают многие изменения в группах. Например, если вы хотите ограничить максимальное количество `PID` в группе (скажем, до 3000), сделайте это следующим образом:

```
# echo 3000 > pids.max
```

Создавать группы управления сложнее. Технически это так же просто, как создать подкаталог где-нибудь в дереве `cgroup`, при этом ядро автоматически создает файлы интерфейса. Если в `cgroup` нет процессов, вы можете удалить ее с помощью команды `rmdir` даже при наличии файлов интерфейса. Что может сбить вас с толку, так это правила, регулирующие группы.

- Вы можете помещать процессы только в группы управления внешнего уровня (конечные). Например, если у вас есть группы с именами `/my-cgroup` и `/my-cgroup/my-subgroup`, вы не можете поместить процессы в `/my-cgroup` —

подойдет вариант `/my-cgroup/my-subgroup`. (Исключение составляет случай, когда в контрольных группах нет контроллеров, но давайте не будем углубляться.)

- В группе не может быть контроллера, который не входит в ее родительскую группу.
- Вы должны явно указать контроллеры для дочерних контрольных групп. Это делается с помощью файла `cgroup.subtree_control`: например, если хотите, чтобы дочерняя группа имела контроллеры `cpu` и `pid`, напишите `+cpu +pid` в этот файл.

Исключением из этих правил является корневая контрольная группа, расположенная в нижней части иерархии. Можно поместить в нее процессы. Одна из причин, по которой вы захотите это сделать, — возможность отключить процесс от управления `systemd`.

8.6.4. Отображение использования ресурсов

В дополнение к возможности ограничения ресурсов по группам вы можете видеть, какие ресурсы используются в данный момент всеми процессами в их группах. Даже без включенных контроллеров можно увидеть нагрузку CPU группы, просмотрев ее файл `cpu.stat`:

```
$ cat cpu.stat
usage_usec 4617481
user_usec 2170266
system_usec 2447215
```

Поскольку это накопленная нагрузка процессора за весь срок работы `cgroup`, можно видеть, как служба потребляет процессорное время, даже если она порождает множество подпроцессов, которые в конечном итоге завершаются.

Вы можете просмотреть другие типы использования, если включены соответствующие контроллеры. Например, контроллер `memory` предоставляет доступ к файлу `memory.current` для текущего использования памяти и файлу `memory.stat`, содержащему подробные данные о памяти за весь срок службы группы. Эти файлы недоступны в корневой контрольной группе.

И это еще не все. Полная информация о том, как применять каждый отдельный контроллер, а также все правила создания контрольных групп имеются в документации ядра: поискав в интернете «документацию cgroups2», вы найдете множество дополнительной информации.

На данный момент, однако, вы должны хорошо представлять, как работают группы. Понимание основ их работы помогает объяснить, как системы организуют процессы. Позже, прочитав о контейнерах, вы увидите, как они используются для совершенно других целей.

8.7. Дополнительно

Одна из причин, по которой существует так много инструментов для измерения ресурсов и управления их использованием, заключается в том, что различные типы ресурсов потребляются по-разному. В этой главе мы рассмотрели процессор, память и ввод-вывод как системные ресурсы, потребляемые процессами, потоками внутри процессов и ядром.

Другая причина заключается в том, что ресурсы ограничены, и для того чтобы система работала хорошо, ее компоненты должны стремиться потреблять меньше ресурсов. В прошлом многие пользователи работали на общей машине, поэтому необходимо было убедиться, что ресурсы разделены между ними поровну. На современном настольном компьютере может и не быть нескольких пользователей, однако на нем все еще существует множество процессов, конкурирующих за ресурсы. Аналогично, высокопроизводительные сетевые серверы требуют интенсивного мониторинга системных ресурсов, поскольку на них выполняется множество процессов для одновременной обработки нескольких запросов.

Дополнительные темы в области мониторинга ресурсов и анализа производительности, которые вы, возможно, захотите изучить:

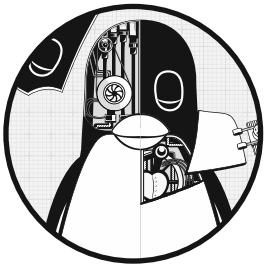
- `sar` (System Activity Reporter). Пакет `sar` обладает множеством возможностей непрерывного мониторинга `vmstat`, а также регистрирует использование ресурсов с течением времени. С помощью `sar` вы можете вернуться назад в определенное время, чтобы увидеть, что делала ваша система. Это удобно, когда нужно проанализировать системное событие, произошедшее в прошлом;
- `acct` (process accounting, учет процессов). Пакет `acct` может записывать процессы и использование ими ресурсов;
- `Quotas`. Вы можете ограничить объем дискового пространства, которое пользователь может задействовать с системой `quota`.

Если вас интересует настройка систем и, в частности, производительность, в книге Брендана Грегга (Brendan Gregg) *Systems Performance: Enterprise and the Cloud*, 2-е издание (Addison-Wesley, 2020), вы найдете гораздо больше деталей.

Мы также еще не коснулись множества инструментов, с помощью которых можно выполнять мониторинг использования сетевых ресурсов. Однако, прежде чем делать это, вам нужно понять, как работает сеть. Об этом мы поговорим далее.

9

Сеть и ее конфигурация



Сеть — это способ подключения компьютеров и передачи данных между ними. Звучит довольно просто, но, чтобы понять, как это работает, нужно задать два фундаментальных вопроса:

- Как компьютер, отправляющий данные, узнает, *куда* отправлять свои данные?
- Когда конечный компьютер получает данные, как он узнает, *что* он только что получил?

Компьютер решает эти вопросы с помощью ряда компонентов, каждый из которых отвечает за определенный аспект отправки, получения и идентификации данных. Компоненты входят в группы, образующие *сетевые уровни (network layers)*, расположенные один над другим и формирующие целостную систему. Ядро Linux обрабатывает сеть аналогично подсистеме SCSI, описанной в главе 3.

Поскольку каждый уровень, как правило, независим, можно создавать сети с множеством различных комбинаций компонентов. Именно здесь конфигурация сети может стать очень сложной. По этой причине мы начнем эту главу с рассмотрения уровней в очень простых сетях. Вы узнаете, как просматривать свои сетевые настройки, и, когда поймете основные принципы работы каждого уровня, будете готовы научиться настраивать их самостоятельно. Наконец, мы перейдем к более сложным темам, таким как создание собственных сетей и настройка брандмауэров. (Пропустите эту информацию, если перестанете что-либо понимать, — вы всегда можете вернуться к ней позднее.)

9.1. Основы сети

Прежде чем перейти к теории сетевых уровней, взгляните на простую сеть, показанную на рис. 9.1.

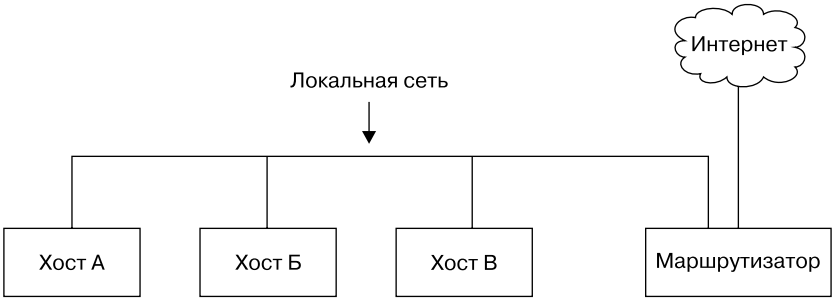


Рис. 9.1. Типичная локальная сеть LAN с маршрутизатором, обеспечивающим доступ в интернет

Этот тип сети распространен повсеместно, большинство домашних и небольших офисных сетей настроены подобным образом. Каждая машина, подключенная к сети, называется *хостом*. Одним из них является *маршрутизатор*, который может перемещать данные из одной сети в другую. Здесь четыре узла — хосты *А*, *Б*, *В* и маршрутизатор — образуют *локальную сеть* (Local Area Network, LAN). Подключения в такой сети могут быть проводными или беспроводными. Строгого определения локальной сети не существует, машины, входящие в нее, обычно физически близки и имеют почти одинаковые конфигурацию и права доступа. Вскоре мы рассмотрим конкретный пример.

Маршрутизатор подключен к интернету — облачку на рисунке. Это соединение называется *восходящим каналом* или *подключением к глобальной сети* (Wide Area Network, WAN), так как оно соединяет гораздо меньшую локальную сеть LAN с более крупной сетью. Поскольку маршрутизатор подключен как к LAN, так и к интернету, через него все компьютеры, входящие в сеть, также имеют доступ к интернету. Одна из целей главы — изучить, как маршрутизатор обеспечивает этот доступ.

Мы будем рассматривать процесс с компьютера на базе Linux, такого как *хост А* в локальной сети на рис. 9.1.

9.2. Пакеты

Компьютер передает данные по сети небольшими порциями, называемыми *пакетами*, которые состоят из двух частей — *заголовка* и *полезной нагрузки*. Заголовок содержит идентифицирующую информацию, такую как исходная и конечная хост-машины и базовый протокол. А полезная нагрузка — это фактические данные приложения, которые компьютер хочет отправить, например данные HTML или изображения.

Хост может отправлять, получать и обрабатывать пакеты в любом порядке независимо от того, откуда они пришли или куда направляются, что позволяет нескольким хостам обмениваться данными одновременно. Например, если хосту необходимо передать данные двум другим в одно время, он может чередовать пункты назначения в исходящих пакетах. Разбиение сообщений на более мелкие блоки также облегчает обнаружение и компенсацию ошибок при передаче.

По большей части, вам не нужно беспокоиться о переводе между пакетами и данными, которые использует приложение, потому что операционная система делает это сама. Однако полезно знать о роли пакетов в сетевых уровнях, которые мы будем изучать далее.

9.3. Сетевые уровни

Полностью функционирующая сеть включает в себя набор сетевых уровней, называемых *сетевым стеком*. Любая функциональная сеть имеет стек. Типичный интернет-стек, от верхнего до нижнего уровня, выглядит следующим образом.

- **Прикладной уровень (application layer).** Содержит язык, который приложения и серверы используют для связи, обычно это какой-то протокол высокого уровня. Общие протоколы прикладного уровня включают HTTP (Hypertext Transfer Protocol, применяемый для интернета), протоколы шифрования, такие как TLS, и протокол передачи файлов (File Transfer Protocol, FTP). Протоколы прикладного уровня часто могут быть объединены. Например, TLS обычно задействуется в сочетании с HTTP для формирования HTTPS.

  Обработка прикладного уровня происходит в пространстве пользователя.

- **Транспортный уровень (transport layer).** Определяет характеристики передачи данных прикладного уровня. Включает проверку целостности данных, порты источника и назначения, а также спецификации для разбиения данных приложения на пакеты на стороне хоста (если прикладной уровень еще этого не сделал) и их повторной сборки в месте назначения. Протокол управления передачей (TCP, Transmission Control Protocol) и протокол пользовательских датаграмм (UDP, User Datagram Protocol) — наиболее распространенные протоколы транспортного уровня. Этот уровень иногда называют *уровнем протокола*.

  В Linux транспортный уровень и все нижележащие уровни в основном обрабатываются ядром, но есть некоторые исключения, когда пакеты отправляются в пространство пользователя для обработки.

- **Сетевой или межсетевой уровень (networkor internet layer).** Определяет, как перемещать пакеты с исходного хоста на конечный. Конкретный набор правил межсетевой передачи пакетов известен как *интернет-протокол (Internet Protocol, IP)*. Так как в этой книге мы будем говорить только о межсетевых взаимодействиях, то будем иметь в виду только межсетевой уровень. Но поскольку сетевые уровни должны быть аппаратно независимыми, вы

можете одновременно настроить на одном хосте несколько независимых сетевых уровней, таких как IP (IPv4), IPv6, IPX и AppleTalk.

- **Физический уровень.** Определяет способ отправки необработанных данных через физический носитель, такой как Ethernet или модем. Его иногда называют *уровнем сетевого доступа* или *уровнем хост-сети*.

Важно понимать структуру сетевого стека, потому что ваши данные должны пройти через его уровни по крайней мере дважды, прежде чем достигнут программы в пункте назначения. Например, если вы отправляете данные с *хоста А* на *хост Б* (см. рис. 9.1), байты покидают прикладной уровень на *хосте А*, проходят через его транспортный и сетевой уровни, затем спускаются на физический носитель, пересекают его и почти таким же образом поднимаются через различные нижние уровни на прикладной уровень на *хосте Б*. Если вы отправляете что-то хосту в другой сети через маршрутизатор, посланное пройдет через некоторые (обычно не все) уровни маршрутизатора и все расположенное между ними.

Уровни иногда перетекают друг в друга странным образом, потому что не всегда эффективно обрабатывать их по порядку. Например, устройства, которые исторически имели дело только с физическим уровнем, теперь иногда просматривают данные транспортного и сетевого уровней одновременно, чтобы быстро фильтровать и маршрутизировать данные. Кроме того, сама терминология может сбивать с толку. Например, TLS означает «безопасность транспортного уровня» (Transport Layer Security), но на самом деле он находится на один уровень выше, на прикладном уровне. (На первоначальном этапе изучения такие детали не важны.)

Мы начнем с рассмотрения того, как ваша машина Linux подключается к сети, чтобы ответить на вопрос «где?» из начала главы. Это нижняя часть стека — физический и сетевой уровни. Позже рассмотрим два верхних слоя, которые отвечают на вопрос «что?».

ПРИМЕЧАНИЕ

Возможно, вы слышали о другом наборе уровней, известном как модель взаимодействия открытых систем (Open Systems Interconnection, OSI). Это семиуровневая сетевая модель, часто используемая в обучении и проектировании сетей, но мы не станем ее рассматривать, потому что будем работать непосредственно с четырьмя уровнями, описанными в книге. Чтобы узнать гораздо больше об уровнях и сетях в целом, см. книгу Эндрю С. Таненбаума и Дэвида Дж. Уэзералла «Компьютерные сети», 5-е издание (Питер, 2019).

9.4. Сетевой уровень

Вместо того чтобы начать с самого низкого уровня стека, вначале займемся сетевым уровнем, потому что он проще для понимания. Межсетевое взаимодействие, каким мы его знаем в настоящее время, основано на версиях интернет-протокола 4 (IPv4) и 6 (IPv6). Одним из наиболее важных аспектов сетевого уровня является то, что он должен быть программной сетью, которая не предъявляет особых требований

к оборудованию или операционным системам. Суть этого заключается в том, что вы можете отправлять и получать сетевые пакеты через любое оборудование, используя любую операционную систему.

Начнем обсуждение с IPv4, потому что с ним немного легче читать адреса (и понимать их ограничения), а далее рассмотрим основные отличия от него IPv6.

Топология интернета децентрализована, он состоит из небольших сетей, называемых *подсетями*. Все подсети каким-то образом взаимосвязаны. Например, локальная сеть (см. рис. 9.1) обычно представляет собой одну подсеть.

Хост может быть подключен к нескольким подсетям. Как сказано ранее, он называется маршрутизатором, если может передавать данные из одной подсети в другую (еще одно название маршрутизатора — *шлюз*). Рисунок 9.2 уточняет рис. 9.1, идентифицируя LAN как подсеть, а также интернет-адреса для каждого хоста и маршрутизатора. Маршрутизатор на рисунке имеет два адреса, локальную подсеть 10.23.2.1 и канал связи с интернетом (его адрес сейчас неважен, поэтому он просто помечен как адрес Uplink). Сначала рассмотрим адреса, а затем обозначения подсети.

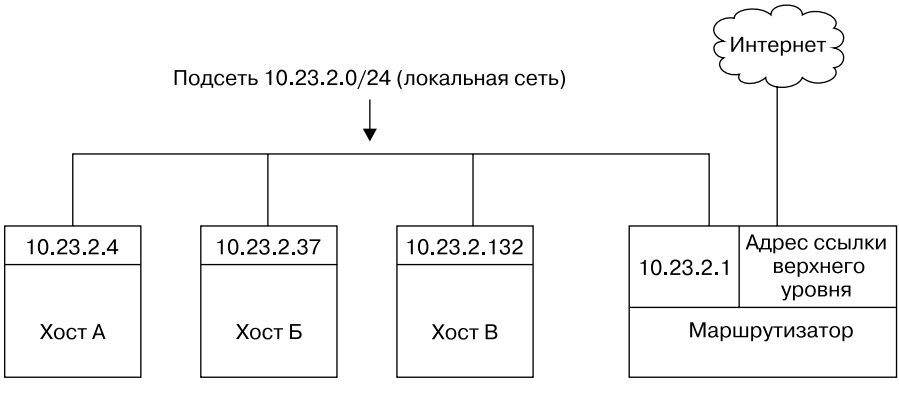


Рис. 9.2. Сеть с IP-адресами

У каждого интернет-хоста есть по крайней мере один цифровой IP-адрес, для IPv4 — в форме `a.b.c.d`, например 10.23.2.37. Адресом в этой записи называется *последовательность из четырех чисел, разделенных точками*. Если хост подключен к нескольким подсетям, у него есть по крайней мере один IP-адрес на подсеть. IP-адрес каждого хоста должен быть уникальным во всем интернете, но как вы увидите позже, частные сети и преобразование сетевых адресов (Network Address Translation, NAT) могут запутать процесс.

Пока не беспокойтесь об обозначении подсети на рис. 9.2, мы обсудим его в ближайшее время.

ПРИМЕЧАНИЕ

Технически IP-адрес состоит из 4 байт (или 32 бит), abcd. Байты a и d — это числа от 1 до 254, а b и c — числа от 0 до 255. Компьютер обрабатывает IP-адреса в виде необработанных байтов. Однако человеку гораздо проще читать и записывать адрес с точками, такой как 10.23.2.37, вместо чего-то наподобие шестнадцатеричного адреса 0x0A170225.

IP-адреса в некотором смысле похожи на почтовые адреса. Для связи с другим хостом ваша машина должна знать его IP-адрес.

Давайте взглянем на адрес вашего компьютера.

9.4.1. Просмотр IP-адреса

Одна машина может иметь множество IP-адресов, включающих несколько физических интерфейсов, виртуальных внутренних сетей и многое другое. Чтобы просмотреть адреса, активные на вашем компьютере с Linux, запустите следующую команду:

```
$ ip address show
```

Вероятно, она выведет много данных, сгруппированных по физическому интерфейсу, описанному в разделе 9.10, но они должны включать что-то вроде этого:

```
2: enp0s31f6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state
UP group default qlen 1000
    link/ether 40:8d:5c:fc:24:1f brd ff:ff:ff:ff:ff:ff
    inet 10.23.2.4/24 brd 10.23.2.255 scope global noprefixroute enp0s31f6
       valid_lft forever preferred_lft forever
```

Вывод команды `ip` включает в себя множество деталей из сетевого уровня(-ей) и физического уровня. (Иногда в нем даже нет интернет-адреса!) Подробнее мы обсудим вывод позже, а сейчас сосредоточимся на четвертой строке, в которой сообщается, что хост имеет IPv4-адрес (обозначается `inet`) `10.23.2.4`. Значение `/24` после адреса помогает определить подсеть, к которой принадлежит IP-адрес. Давайте рассмотрим, как это работает.

ПРИМЕЧАНИЕ

Команда ip — стандартный инструмент настройки сети. В другой документации вы можете встретить ifconfig. Эта устаревшая команда, которая использовалась в других версиях Unix в течение десятилетий, но теперь она менее эффективна. Чтобы соответствовать современной рекомендуемой практике (и дистрибутивам, которые могут даже не включать ifconfig по умолчанию), мы будем применять команду ip. Также существуют команды, которые могут заменить ip, — это route и arp.

9.4.2. Подсети

Подсеть представляет собой подключенную группу хостов с IP-адресами в определенном диапазоне. Например, хосты в диапазоне от 10.23.2.1 до 10.23.2.254 могут

включать подсеть, как и все хосты в диапазоне от 10.23.1.1 до 10.23.255.254. Обычно хосты подсети находятся в одной физической сети, как показано на рис. 9.2.

Подсеть определяется на основе двух частей: *префикса сети* (также называемого *префиксом маршрутизации*) и *маски подсети* (иногда называемой *маской сети* или *маской маршрутизации*). Допустим, вы хотите создать подсеть, содержащую IP-адреса между 10.23.2.1 и 10.23.2.254. Сетевой префикс — это часть, *общая* для всех адресов в подсети, в данном примере это 10.23.2.0 с маской подсети 255.255.255.0. Давайте посмотрим, откуда взялись эти цифры.

Чтобы увидеть, как префикс и маска работают вместе, предоставляя вам все возможные IP-адреса в подсети, рассмотрим двоичную форму. Маска отмечает битовые местоположения в IP-адресе, которые являются общими для подсети. Например, вот двоичные формы 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 расположение битов в 10.23.2.0, которые равны 1 в 255.255.255.0:

```
10.23.2.0:       00001010 00010111 00000010 00000000
```

Любой адрес, содержащий конфигурацию битов, выделенную жирным шрифтом, находится в подсети. Для битов, *не выделенных* жирным шрифтом (последний набор из восьми нулей), установка 1 на любую позицию приводит к получению действительного IP-адреса в этой подсети, за исключением случаев, когда все биты являются 0 или 1.

Собрав все это вместе, вы можете увидеть, что хост с IP-адресом 10.23.2.1 и маской подсети 255.255.255.0 находится в той же подсети, к которой относится любой другой компьютер, IP-адрес которого начинается с 10.23.2. Можно обозначить эту подсеть 10.23.2.0/255.255.255.0.

Теперь посмотрим, как эти данные перевести в сокращенную нотацию (например, в /24), которую мы встречали в выводе команды `ip`.

9.4.3. Распространенные маски подсети и нотация CIDR

В большинстве сетевых инструментов вы столкнетесь с другой формой представления подсети, называемой *бесклассовой междоменной маршрутизацией* (Classless Inter-Domain Routing, CIDR), где подсеть, такая как 10.23.2.0/255.255.255.0, обозначается 10.23.2.0/24. Это сокращение использует преимущества простого шаблона, которому соответствуют маски подсети.

Посмотрите на маску в двоичной форме из примера, приведенного в предыдущем разделе. Все маски подсети являются (или должны быть согласно RFC 1812) только одним блоком из единиц, за которым следует один блок из нулей. Например, мы

знаем, что 255.255.255.0 в двоичной форме состоит из 24 бит единиц, за которыми следуют 8 бит нулей. Адресация CIDR идентифицирует маску подсети по количеству ведущих единиц в ней. Поэтому комбинация 10.23.2.0/24 включает как префикс подсети, так и ее маску подсети.

В табл. 9.1 приведены несколько примеров масок подсети и их форм CIDR. Маска подсети /24 наиболее широко распространена в локальных сетях конечных пользователей, она часто применяется в сочетании с одной из частных сетей, которые вы увидите в разделе 9.22.

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 знакомы с преобразованием между десятичным, двоичным и шестнадцатеричным форматами, используйте утилиту калькулятора bc или dc для преобразования между различными представлениями систем счисления. Например, в bc можно выполнить команду obase=2; 240, чтобы набрать число 240 в двоичной форме (основание 2).

Вы, возможно, уже заметили, что если у вас есть IP-адрес и маска подсети, то даже не нужно беспокоиться об отдельном определении сети. Можете объединить их, как показано в подразделе 9.4.1, вывод `ip address show` включает 10.23.2.4/24.

Определение подсетей и их хостов — это первый строительный блок для понимания того, как работает интернет. Однако вам все равно необходимо соединить подсети.

9.5. Маршруты и таблица маршрутизации ядра

Подключение интернет-подсетей — это в основном процесс отправки данных через хосты, подключенные к нескольким подсетям. Возвращаясь к рис. 9.2, рассмотрите *хост A* по IP-адресу 10.23.2.4.

Этот хост подключен к локальной сети 10.23.2.0/24 и может напрямую связываться с хостами в ней. Чтобы связаться с хостами в остальной части интернета, он должен взаимодействовать через маршрутизатор в 10.23.2.1.

Ядро Linux различает эти два разных типа назначений, задействуя *таблицу маршрутизации* для определения поведения маршрутизации. Чтобы отобразить таблицу

маршрутизации, используйте команду `ip route show`. Вот что вы можете увидеть для простого хоста, такого как 10.23.2.4:

```
$ ip route show
default via 10.23.2.1 dev enp0s31f6 proto static metric 100
10.23.2.0/24 dev enp0s31f6 proto kernel scope link src 10.23.2.4 metric 100
```

ПРИМЕЧАНИЕ

Традиционным инструментом для просмотра маршрутов является команда route, выполняемая как route -n. Параметр -n указывает route показывать IP-адреса, вместо того, чтобы пытаться показывать имена хостов и сетей. Это важный параметр, который следует запомнить, потому что его можно использовать и в других командах, связанных с сетью, таких как netstat.

Этот вывод нелегко прочитать. Каждая строка является правилом маршрутизации; давайте начнем со второй строки примера и разобьем ее на части.

Первая часть 10.23.2.0/24 является сетью назначения. Как и в предыдущих примерах, это локальная подсеть хоста. Данное правило гласит, что хост может связаться с локальной подсетью напрямую через свой сетевой интерфейс, обозначенный меткой механизма `dev enp0s31f6` после пункта назначения. (После этого поля приводится более подробная информация о маршруте, в том числе о том, как он был создан. Вам не нужно беспокоиться об этом сейчас.)

Затем мы можем вернуться к первой строке вывода, в которой используется *сеть назначения* `default`. Это правило, которое соответствует любому хосту вообще, также называется *маршрутом по умолчанию* (изучим в следующем разделе). Механизм `via 10.23.2.1` указывает, что трафик, использующий маршрут по умолчанию, должен быть отправлен на адрес 10.23.2.1 (в нашем примере сети это маршрутизатор), `dev enp0s31f6` указывает, что физическая передача будет происходить по этому сетевому интерфейсу.

9.6. Шлюз по умолчанию

Запись `default` в таблице маршрутизации имеет особое значение, поскольку она соответствует любому адресу в интернете. В адресации CIDR это 0.0.0.0/0 для IPv4. Это маршрут по умолчанию, и адрес, настроенный в нем в качестве посредника, является *шлюзом по умолчанию*. Когда никакие другие правила не подходят, маршрут по умолчанию всегда подходит, и шлюз по умолчанию — это место, откуда вы отправляете сообщения, если нет другого варианта. Можете настроить хост без шлюза по умолчанию, но он не сможет связаться с хостами за пределами пунктов назначения в таблице маршрутизации.

В большинстве сетей с маской сети /24 (255.255.255.0) маршрутизатор обычно находится по адресу 1 подсети (например, 10.23.2.1 в 10.23.2.0/24). Это просто договоренность, и могут быть исключения.

КАК ЯДРО ВЫБИРАЕТ МАРШРУТ

В маршрутизации есть одна хитрая деталь. Допустим, хост хочет отправить что-то в 10.23.2.132, что соответствует обоим правилам в таблице маршрутизации: маршруту по умолчанию и 10.23.2.0/24. Как ядро узнает, что нужно использовать второй маршрут? Порядок в таблице маршрутизации не имеет значения, ядро выбирает самый длинный соответствующий префикс назначения. Вот где адресация CIDR особенно полезна: 10.23.2.0/24 соответствует, и его префикс имеет длину 24 бита; 0.0.0.0/0 также соответствует, но его префикс имеет длину 0 бит (то есть у него нет префикса), поэтому правило для 10.23.2.0/24 становится приоритетным.

9.7. IPv6-адреса и сети

Если вы взглянете на раздел 9.4, то увидите, что IPv4-адреса состоят из 32 бит, или 4 байтов. Это дает в общей сложности примерно 4,3 млрд адресов, что недостаточно для нынешних масштабов интернета. Появилось несколько проблем, вызванных отсутствием адресов в IPv4, поэтому Инженерный совет интернета (Internet Engineering Task Force, IETF) разработал версию IPv6. Прежде чем перейти к другим сетевым инструментам, обсудим адресное пространство IPv6.

Адрес IPv6 имеет 128 бит — 32 байта, расположенных в восьми наборах по 4 байта. В длинной форме адрес записывается следующим образом:

```
2001:0db8:0a0b:12f0:0000:0000:0000:8b6e
```

Представление шестнадцатеричное, каждая цифра относится к диапазону от 0 до f. Существует несколько широко распространенных методов сокращения представления. Во-первых, можно опустить все ведущие нули (например, 0db8 превращается в db8), и один и только один набор смежных нулевых групп может быть представлен как `::` (два двоеточия). Поэтому предыдущий адрес записывается как

```
2001:db8:a0b:12f0::8b6e
```

Подсети по-прежнему обозначаются с помощью адресации CIDR. Для конечного пользователя они часто охватывают половину доступных битов в адресном пространстве (/64), но бывают случаи, когда используется меньше. Часть адресного пространства, уникальная для каждого хоста, называется *идентификатором интерфейса*. На рис. 9.3 показан пример адреса с 64-разрядной подсетью.

ПРИМЕЧАНИЕ

В этой книге мы, как правило, рассматриваем работу системы с точки зрения среднестатистического пользователя. Этот взгляд немного отличается от взгляда поставщика услуг, где подсеть дополнительно разделяется на префикс маршрутизации и другой идентификатор сети, иногда также называемый подсетью. Сейчас об этом беспокоиться не стоит.

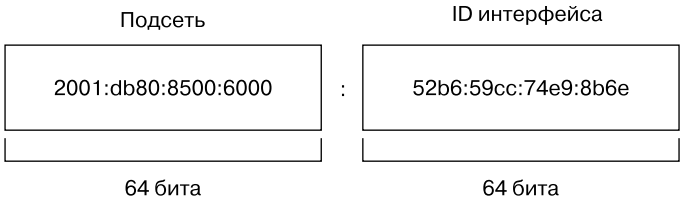


Рис. 9.3. Идентификатор подсети и интерфейса типичного IPv6-адреса

Последнее, что нужно знать на данный момент об IPv6, — это то, что хосты обычно имеют по крайней мере два адреса. Первый, который действителен в интернете, называется *глобальным индивидуальным адресом*. Второй, для локальной сети, называется *локальным адресом канала*. Локальные адреса канала всегда имеют префикс fe80::/10, за которым следует 54-разрядный сетевой идентификатор с нулевым значением, и заканчиваются 64-разрядным идентификатором интерфейса. То есть локальный адрес канала в вашей системе будет находиться в подсети fe80::/64.

ПРИМЕЧАНИЕ

Глобальные индивидуальные адреса имеют префикс 2000::/3. Поскольку первый байт начинается с 001 с этим префиксом, он может быть завершен как 0010 или 0011. Поэтому глобальный индивидуальный адрес всегда начинается с 2 или 3.

9.7.1. Просмотр конфигурации IPv6 в системе

Если у системы есть конфигурация IPv6, вы можете получить некоторую информацию об IPv6 из команды `ip`, которую мы запускали ранее. Чтобы выделить IPv6, используйте параметр `-6`:

```
$ ip -6 address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 state UNKNOWN qlen 1000
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s31f6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP qlen 1000
    inet6 2001:db8:8500:e:52b6:59cc:74e9:8b6e/64 scope global dynamic noprefixroute
       valid_lft 86136sec preferred_lft 86136sec
    inet6 fe80::d05c:97f9:7be8:bca/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

В дополнение к интерфейсу loopback (о нем мы поговорим позже) вы можете увидеть еще два адреса. Глобальный индивидуальный адрес обозначается как `scope global`, а локальный адрес канала получает метку `scope link`.

Просмотр маршрутов аналогичен:

```
  $ ip -6 route show
  ::1 dev lo proto kernel metric 256 pref medium
❶ 2001:db8:8500:e::/64 dev enp0s31f6 proto ra metric 100 pref medium
```

```
❷ fe80::/64 dev enp0s31f6 proto kernel metric 100 pref medium
❸ default via fe80::800d:7bff:feb8:14a0 dev enp0s31f6 proto ra metric 100 pref
   medium
```

Это немного сложнее, чем настройка IPv4, поскольку настроены как локальные, так и глобальные подсети. Строка ❶ предназначена для адресатов в локально подключенных глобальных индивидуальных адресных подсетях: хост знает, что он может связаться с ними напрямую и локальный канал связи ❷ аналогичен. Для маршрута ❸ по умолчанию (также записывается как ::/0 в IPv6, помните, что это все, что не связано напрямую) эта конфигурация организует передачу трафика через маршрутизатор по локальному адресу канала fe80::800d:7bff:feb8:14a0 вместо его адреса в глобальной подсети. Позже вы увидите, что маршрутизатор обычно не заботится о том, как он получает трафик, — только о том, куда должен идти. Использование локального адреса канала в качестве шлюза по умолчанию имеет то преимущество, что его не нужно менять при изменении глобального пространства IP-адресов.

9.7.2. Настройка сетей с двумя стеками

Как вы, возможно, уже догадались, можно настроить хосты и сети для применения как IPv4, так и IPv6. Эту возможность называют *сетью с двумя стеками (dual-stack network)*, хотя в данном случае использовать слово «стек» не совсем точно, поскольку один уровень типичного сетевого стека попросту дублируется (истинный двойной стек был бы чем-то вроде IP + IPX). Если отбросить условности, протоколы IPv4 и IPv6 независимы друг от друга и могут работать одновременно. На таком хосте приложение (например, веб-браузер) может выбрать IPv4 или IPv6 для подключения к другому хосту.

Приложение, изначально написанное для IPv4, не поддерживает IPv6 автоматически. К счастью, поскольку в стеке уровни, расположенные поверх сетевого уровня, не изменились, можно задействовать простой код, необходимый для коммуникации по IPv6. Большинство важных приложений и серверов теперь поддерживают IPv6.

9.8. Основные инструменты ICMP и DNS

Теперь пришло время взглянуть на базовые практические утилиты, которые помогут вам взаимодействовать с хостами. Эти инструменты применяют два протокола, представляющих особый интерес: протокол управления сообщениями в интернете (Internet Control Message Protocol, ICMP), который может помочь устранить проблемы с подключением и маршрутизацией, и систему доменных имен (Domain Name Service, DNS), которая сопоставляет имена с IP-адресами, чтобы пользователю не приходилось запоминать кучу номеров.

ICMP — это протокол транспортного уровня, применяемый для настройки и диагностики интернет-сетей, он отличается от других протоколов транспортного уровня тем, что не содержит никаких истинных пользовательских данных, и, следовательно,

над ним нет прикладного уровня. Для сравнения: DNS — это протокол прикладного уровня для сопоставления удобочитаемых имен с интернет-адресами.

9.8.1. Команда ping

Команда `ping` (подробнее на ftp.arl.army.mil/~mike/ping.html) — один из самых простых инструментов отладки сети. Она отправляет эхо-запросы ICMP хосту, который просит хост-получатель вернуть пакет отправителю. Если хост-получатель получает пакет и сконфигурирован для ответа, он отправляет пакет эхо-ответа ICMP.

К примеру, запустим команду `ping 10.23.2.1` и получим следующий вывод:

```
$ 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
```

В первой строке говорится, что вы отправляете 56-байтовые пакеты (84 байта, если включены заголовки) в 10.23.2.1 (по умолчанию один пакет в секунду), а остальные строки указывают на ответы из 10.23.2.1. Наиболее важными частями вывода являются порядковый номер (`icmp_req`) и время прохождения в оба конца (`time`). Количество возвращаемых байтов равно размеру отправленного пакета плюс 8. (Содержимое пакетов для нас неважно.)

Разрыв в порядковых номерах, например, между 2 и 4, обычно означает, что появилась какая-то проблема с подключением. Пакеты не должны поступать не по порядку, потому что `ping` отправляет только один пакет в секунду. Если для получения ответа требуется более секунды (1000 мс), подключение крайне медленное.

Время прохождения в оба конца — это общее время с момента отправки пакета запроса до момента прибытия пакета ответа. Если нет способа добраться до места назначения, последний маршрутизатор, который увидит пакет, возвращает `ping` пакет ICMP о том, что хост недоступен (host unreachable).

В проводной локальной сети пакеты не должны теряться, а время прохождения должно быть очень малым. (В предыдущем примере вывод осуществляется из беспроводной сети.) Также не должны теряться пакеты из вашей сети к интернет-провайдеру и от него, а время прохождения в оба конца должно быть довольно стабильным.

ПРИМЕЧАНИЕ

По соображениям безопасности некоторые хосты в интернете отключают ответ на пакеты эхо-запросов ICMP, поэтому вы можете обнаружить, что способны подключиться к веб-сайту на хосте, но не получить ответ ping.

Вы можете заставить `ping` использовать IPv4 или IPv6 с параметрами `-4` и `-6` соответственно.

9.8.2. Служба DNS и команда host

IP-адреса трудно запомнить, и они могут быть изменены, поэтому мы обычно берем такие имена, как www.example.com. Библиотека службы доменных имен (Domain Name System, DNS) в вашей системе обычно переводит имя в IP-адрес автоматически, но иногда может потребоваться выполнить это вручную. Чтобы найти IP-адрес, скрывающийся за доменным именем, используйте команду `host`:

```
$ host www.example.com
example.com has address 172.17.216.34
example.com has IPv6 address 2001:db8:220:1:248:1893:25c8:1946
```

Обратите внимание на то, что в этом примере есть как IPv4-адрес 172.17.216.34, так и гораздо более длинный IPv6-адрес. Для имени хоста могут существовать несколько адресов, и вывод может содержать дополнительную информацию, такую как почтовые обменники.

Можно использовать команду `host` и в обратном порядке: введите IP-адрес вместо имени хоста, чтобы попытаться обнаружить имя хоста, скрытое за IP-адресом. Однако не ожидайте, что этот способ сработает надежно. Один IP-адрес может быть связан с несколькими именами хостов, и DNS не знает, как определить, какое из них должно соответствовать IP-адресу. Кроме того, администратору хоста необходимо настроить обратный поиск вручную, а делают это нечасто.

В DNS есть возможности гораздо бóльшие, чем просто команда `host`. Мы рассмотрим базовую конфигурацию клиента в разделе 9.15.

С утилитой `host` используются параметры `-4` и `-6`, но они работают иначе, чем можно ожидать. Они заставляют команду `host` получать информацию через IPv4 или IPv6, но поскольку она должна быть одинаковой вне зависимости от сетевого протокола, вывод потенциально будет включать как IPv4, так и IPv6.

9.9. Физический уровень и сеть Ethernet

Один из ключевых моментов, который необходимо знать об интернете, заключается в том, что это *программная* сеть. Ничто из того, что мы обсуждали до сих пор, не относится к конкретному оборудованию, и действительно, одна из причин популярности интернета заключается в том, что он работает практически на любых компьютере, операционной системе и физической сети. Но если вы действительно хотите поговорить с другим компьютером, вам придется разместить сетевой уровень поверх какого-либо оборудования. Этот интерфейс является физическим уровнем.

В этой книге мы рассмотрим наиболее распространенный вид физического уровня — сеть Ethernet. Документация по стандартам семейства IEEE 802 определяет множество различных типов сетей Ethernet, от проводных до беспроводных, но у всех них есть несколько общих черт:

- Все устройства в сети Ethernet имеют адрес управления доступом к среде (Media Access Control, MAC), иногда называемый *аппаратным адресом*. Он не зависит от IP-адреса хоста и уникален для сети Ethernet хоста (но не обязательно для более крупной программной сети, такой как интернет). Пример MAC-адреса — 10:78:d2:eb:76:97.
- Устройства в сети Ethernet отправляют сообщения в виде *фреймов*, или *кадров*, которые являются обертками вокруг отправленных данных. Кадр содержит MAC-адреса источника и назначения.

Ethernet на самом деле не пытается выйти за рамки аппаратного обеспечения в одной сети. Например, если у вас есть две разные сети Ethernet с одним хостом, подключенным к обеим (и два разных устройства сетевого интерфейса), вы не сможете напрямую передавать кадр из одной сети Ethernet в другую, если не настроите мост Ethernet bridge. Именно здесь вступают в действие более высокие сетевые уровни (такие, как интернет-уровень). По соглашению каждая сеть Ethernet также обычно является подсетью интернета. Даже если кадр не может покинуть одну физическую сеть, маршрутизатор может извлечь из него данные, переупаковать их и отправить на хост в другой физической сети, что и происходит в интернете.

9.10. Сетевые интерфейсы ядра

Физический и сетевой уровни должны быть соединены таким образом, чтобы сетевой уровень мог сохранять свою не зависящую от аппаратного обеспечения гибкость. Ядро Linux поддерживает собственное разделение между двумя уровнями и обеспечивает стандарт связи для их соединения, называемый *сетевым интерфейсом (ядра)*. При настройке сетевого интерфейса вы связываете параметры IP-адреса со стороны сети с идентификацией оборудования на стороне физического устройства. Сетевые интерфейсы обычно имеют имена, указывающие на тип используемого оборудования, например enp0s31f6 (интерфейс в слоте PCI). Такое имя называется *предсказуемым именем сетевого интерфейса*, потому что оно остается неизменным после перезагрузки. Во время загрузки интерфейсы имеют традиционные имена, такие как eth0 (первая карта Ethernet в компьютере) и wlan0 (беспроводной интерфейс), но на большинстве машин под управлением `systemd` они быстро переименовываются.

В подразделе 9.4.1 мы узнали, как просмотреть настройки сетевого интерфейса с помощью команды `ip address show`. Вывод сгруппирован по интерфейсам, как встречалось ранее:

```
2: enp0s31f6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state \
❶ UP group default qlen 1000
  ❷ link/ether 40:8d:5c:fc:24:1f brd ff:ff:ff:ff:ff:ff
    inet 10.23.2.4/24 brd 10.23.2.255 scope global noprefixroute enp0s31f6
       valid_lft forever preferred_lft forever
       inet6 2001:db8:8500:e:52b6:59cc:74e9:8b6e/64 scope global dynamic
```

```
noprefixroute
       valid_lft 86054sec preferred_lft 86054sec
    inet6 fe80::d05c:97f9:7be8:bca/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

Каждый сетевой интерфейс получает номер, в данном случае 2. Интерфейс 1 почти всегда является интерфейсом loopback, описанным в разделе 9.16. Флаг `UP` сообщает, что интерфейс работает ❶. В дополнение к частям сетевого уровня, которые мы уже рассмотрели, вы также видите MAC-адрес на физическом уровне, `link/ether` ❷.

Хотя команда `ip` показывает часть информации об оборудовании, она предназначена в основном для просмотра и настройки программных уровней, подключенных к интерфейсам. Чтобы глубже изучить аппаратный и физический уровень сетевого интерфейса, используйте команду `ethtool` для отображения или изменения настроек на картах Ethernet. (Мы кратко рассмотрим беспроводные сети в разделе 9.27.)

9.11. Введение в настройки сетевого интерфейса

Мы изучили все основные элементы, которые входят в нижние уровни сетевого стека: физический уровень, сетевой (межсетевой) уровень и сетевые интерфейсы ядра Linux. Чтобы объединить эти части для подключения компьютера Linux к интернету, вы или часть программного обеспечения должны выполнить следующие действия:

1. Подключить сетевое оборудование и убедиться, что в ядре есть драйвер для него. Если он имеется, вывод команды `ip address show` включает запись для устройства, даже если оно не было настроено.
2. Выполнить любые дополнительные настройки физического уровня, такие как выбор имени сети или введение пароля.
3. Назначить IP-адреса и подсети сетевому интерфейсу ядра, чтобы драйверы устройств ядра (физический уровень) и подсистемы интернета (сетевой уровень) могли взаимодействовать друг с другом.
4. Добавить любые дополнительные маршруты, включая шлюз по умолчанию.

Когда компьютеры были большими стационарными коробками, соединенными вместе, все это было относительно просто: ядро выполняло шаг 1, шаг 2 не требовался, и вы выполняли шаг 3 старой командой `ifconfig` и шаг 4 старой командой `route`. Мы кратко рассмотрим, как это сделать с помощью команды `ip`.

9.11.1. Ручная настройка интерфейсов

Сейчас рассмотрим, как настроить интерфейсы вручную, но не будем вдаваться в подробности, потому что это редко требуется и может привести к ошибкам. Лучше выполнять эти действия только на экспериментальной системе. Даже при

настройке для создания конфигурации в текстовом файле вы можете использовать инструмент Netplan вместо ряда команд, как показано далее.

Можете привязать интерфейс к сетевому уровню с помощью команды `ip`. Чтобы добавить IP-адрес *`address`* и подсеть *`subnet`* для сетевого интерфейса ядра, выполните следующее:

```
# ip address add address/subnet dev interface
```

Здесь *`interface`* — это имя интерфейса, например `enp0s31f6` или `eth0`. Эта же команда работает для IPv6, за исключением того, что нужно добавить параметры (например, указать статус локального канала). Если хотите просмотреть все параметры, используйте страницу руководства по `ip-address(8)`.

9.11.2. Добавление и удаление маршрутов вручную

С настроенным интерфейсом вы можете добавлять маршруты, что обычно сводится к настройке шлюза по умолчанию, например:

```
# ip route add default via gw-address dev interface
```

Параметр *`gw-address`* — это IP-адрес шлюза по умолчанию. Это должен быть адрес в локально подключенной подсети, назначенный одному из ваших сетевых интерфейсов. Чтобы удалить шлюз по умолчанию, запустите следующую команду:

```
# ip route del default
```

Можно легко переопределить шлюз по умолчанию с помощью других маршрутов. Предположим, компьютер находится в подсети 10.23.2.0/24, вы хотите подключиться к подсети в 192.168.45.0/24 и знаете, что хост в 10.23.2.44 может выступать для нее маршрутизатором. Чтобы отправить трафик, привязанный к 192.168.45.0, на этот маршрутизатор, выполните следующую команду:

```
# ip route add 192.168.45.0/24 via 10.23.2.44
```

Вам не нужно указывать маршрутизатор, чтобы удалить маршрут:

```
# ip route del 192.168.45.0/24
```

Прежде чем сойти с ума от количества маршрутов, вы должны знать, что настройка маршрутов часто сложнее, чем кажется. В этом конкретном примере необходимо убедиться также, что маршрутизация для всех хостов на 192.163.45.0/24 может привести к 10.23.2.0/24, или же первый добавленный вами маршрут в основном бесполезен.

Обычно все нужно делать как можно проще, настроив локальные сети так, чтобы их хостам требовался только маршрут по умолчанию. Если вам нужны несколько подсетей и возможность маршрутизации между ними, обычно стоит настроить

маршрутизаторы, действующие как шлюзы по умолчанию, для выполнения всей работы по маршрутизации между различными локальными подсетями. (Вы увидите пример в разделе 9.21.)

9.12. Конфигурация сети, активируемая при загрузке

Мы обсудили способы ручной настройки сети и выяснили, что традиционный способ обеспечения правильности сетевой конфигурации машины — это выполнение сценария init для запуска ручной настройки во время загрузки. Все сводится к запуску утилиты `ip` где-то в цепочке событий загрузки.

В Linux было много попыток стандартизировать файлы конфигурации для работы сети во время загрузки. Среди них инструменты `ifup` и `ifdown`: например, загрузочный скрипт может (теоретически) запускать `ifup eth0` для запуска правильных `ip`-команд для настройки интерфейса. К сожалению, разные дистрибутивы имеют совершенно разные реализации `ifup` и `ifdown`, и в результате их файлы конфигурации также различаются.

Существует еще более глубокое различие из-за того, что элементы конфигурации сети присутствуют на каждом из сетевых уровней. Следствием этого является то, что программное обеспечение, ответственное за создание сетей, находится в нескольких частях ядра и инструментах пользовательского пространства, написанных и поддерживаемых разными разработчиками. В Linux существует общее соглашение не обмениваться файлами конфигурации между отдельными наборами инструментов или библиотеками, поскольку изменения, внесенные для одного инструмента, могут нарушить работу другого.

Работа с конфигурацией сети в нескольких разных местах затрудняет управление системами. В результате появилось несколько различных инструментов управления сетью, у каждого из которых собственный подход к проблеме конфигурации. Однако они, как правило, специализированы для определенного рода ролей, которые может выполнять машина Linux. Инструмент способен работать на ПК, но не подходит для сервера.

Инструмент Netplan предлагает другой подход к проблеме конфигурации. Netplan — не столько управление сетью, сколько унифицированный стандарт конфигурации сети и инструмент для преобразования этой конфигурации в файлы, используемые существующими сетевыми менеджерами. В настоящее время Netplan поддерживает NetworkManager и systemd-networkd, о которых мы поговорим позже в этой главе. Файлы Netplan представлены в формате YAML и находятся в `/etc/netplan`.

Прежде чем говорить о менеджерах конфигурации сети, давайте немного подробнее рассмотрим проблемы, с которыми они сталкиваются.

9.13. Проблемы с настройкой сети вручную и при загрузке

Несмотря на то что в большинстве систем настройка сети заложена в механизме загрузки (многие серверы поддерживают эту традицию), динамическая природа современных сетей означает, что большинство машин не имеют статического (неизменного) IP-адреса. В IPv4 компьютер не хранит IP-адрес и другую сетевую информацию, а получает ее откуда-то из локальной физической сети при первом подключении к ней. Большинству обычных сетевых клиентских приложений не особенно важно, какой IP-адрес использует машина, если он рабочий. Инструменты протокола динамической настройки узла (Dynamic Host Configuration Protocol, DHCP, см. раздел 9.19) выполняют базовую настройку сетевого уровня на типичных клиентах IPv4. В IPv6 клиенты могут в определенной степени настраивать себя, мы кратко рассмотрим эту способность в разделе 9.20.

И это еще не вся история. Например, беспроводные сети добавляют в конфигурацию интерфейса дополнительные параметры, такие как имена сетей, методы аутентификации и шифрования. Взглянув на общую картину, можно увидеть, что системе необходимо ответить на следующие вопросы:

- Если устройство имеет несколько физических сетевых интерфейсов (например, ноутбук с проводным и беспроводным Ethernet), как выбрать, какой из них использовать?
- Как машина должна настроить физический интерфейс? Для беспроводных сетей этот процесс включает сканирование имен сетей, выбор имени и аутентификацию.
- Как только физический сетевой интерфейс будет подключен, как машина должна настроить сетевые уровни программного обеспечения, например интернет-уровень?
- Как позволить пользователю выбирать параметры подключения, например беспроводную сеть?
- Что должен делать компьютер, если подключение к сетевому интерфейсу будет нарушено?

Ответы на эти вопросы обычно глубже, чем могут дать простые загрузочные сценарии, и все это очень сложно сделать вручную. Ответ заключается в применении системной службы, которая может отслеживать физические сети и выбирать (и автоматически настраивать) сетевые интерфейсы ядра на основе набора правил, имеющих смысл для пользователя. Служба также должна иметь возможность отвечать на запросы пользователей, которые, в свою очередь, должны иметь возможность изменять беспроводную сеть, в которой работают, не становясь суперпользователем.

9.14. Менеджеры настройки сети

Существует несколько способов автоматической настройки сетей в системах на базе Linux. Наиболее широко используемым вариантом на ПК и ноутбуках является NetworkManager. Существует дополнение к systemd, называемое systemd-networkd, способное выполнять базовую настройку сети и полезное для машин, которым не требуется большая гибкость (например, серверов), но не обладающее динамическими возможностями NetworkManager. Другие системы управления конфигурацией сети предназначены в основном для небольших встроенных систем, например `netifd` OpenWRT, служба ConnectivityManager для Android, ConnMan и Wicd.

Мы кратко рассмотрим NetworkManager, потому что именно эту утилиту вы, скорее всего, будете использовать. Однако не будем вдаваться в подробности, потому что после того, как вы ознакомитесь с основными концепциями, понять NetworkManager и другие системы конфигурации будет намного проще. Если вас интересует systemd-networkd, изучите страницу руководства `systemd.network(5)`, на которой описаны настройки, или каталог конфигурации `/etc/systemd/network`.

9.14.1. Работа NetworkManager

NetworkManager — это демон, который система запускает при загрузке. Как и большинство демонов, он не зависит от работающего компонента рабочего стола. Его задача состоит в том, чтобы прослушивать события, инициируемые системой и пользователями, и изменять конфигурацию сети на основе набора правил.

Во время работы NetworkManager поддерживает два основных уровня конфигурации. Первый — это сбор информации о доступных аппаратных устройствах, которую он обычно получает из ядра и поддерживает путем мониторинга `udev` по шине рабочего стола Desktop Bus (D-Bus). Второй уровень конфигурации представляет собой более конкретный список *подключений*: аппаратные устройства и дополнительные параметры конфигурации физического и сетевого уровней. Например, беспроводная сеть может быть представлена в виде подключения.

Чтобы активировать подключение, NetworkManager часто делегирует задачи другим специализированным сетевым инструментам и демонам, таким как `dhclient`, для получения конфигурации интернет-уровня из локально подключенной физической сети. Поскольку инструменты и схемы настройки сети различаются в разных дистрибутивах, NetworkManager использует плагины для взаимодействия с ними, а не навязывает собственный стандарт. Например, существуют плагины как для конфигурации интерфейса Debian/Ubuntu, так и для интерфейса Red Hat.

При запуске NetworkManager собирает всю доступную информацию о сетевом устройстве, просматривает список подключений, а затем пробует активировать одно из них. Вот как он принимает это решение для интерфейсов Ethernet:

1. Если доступно проводное подключение, то подключается с его помощью. В противном случае использует беспроводные подключения.
2. Просматривает список доступных беспроводных сетей. Если доступна сеть, к которой система ранее подключалась, NetworkManager повторит попытку.
3. Если доступны несколько беспроводных сетей, к которым демон подключался ранее, он выбирает ту, с которой соединялся в предыдущий раз.

После установления соединения NetworkManager поддерживает его до тех пор, пока оно не нарушится, не станет доступной улучшенная сеть (например, во время подключения по беспроводной сети вы подсоединяете сетевой кабель) или пользователь не внесет изменения.

9.14.2. Взаимодействие с NetworkManager

Большинство пользователей взаимодействуют с NetworkManager через приложение на рабочем столе (обычно это значок в правом верхнем или нижнем углу, который указывает состояние подключения — проводное, беспроводное или не подключено). При нажатии на значок вы получаете ряд вариантов подключения, например выбор беспроводных сетей и возможность отключения от текущей сети. Каждая среда рабочего стола имеет собственную версию этого приложения, поэтому на каждом из них оно выглядит немного по-разному.

В дополнение к приложению есть несколько инструментов, которые можно использовать для выполнения запросов и управления NetworkManager из своей оболочки. Для быстрого получения сводки о состоянии текущего подключения примените команду `nmcli` без аргументов. Вы получите список интерфейсов и параметров конфигурации. В некотором смысле это похоже на работу команды `ip`, за исключением того, что в этом выводе больше деталей, особенно при просмотре беспроводных подключений.

Команда `nmcli` позволяет управлять демоном NetworkManager из командной строки. Это довольно обширная команда: в дополнение к обычной странице руководства `nmcli(1)` есть страница руководства с примерами использования `nmcli-examples(5)`.

Наконец, утилита `nm-online` сообщит, работает сеть или нет. Если она подключена, команда возвращает `0` в качестве кода возврата, в противном случае он не равен нулю. (Подробнее о том, как применять код выхода в сценарии командной оболочки, читайте в главе 11.)

9.14.3. Настройка NetworkManager

Обычно каталог общей конфигурации NetworkManager — это `/etc/NetworkManager`, существует несколько различных типов его конфигурации. Общий файл конфигурации — `NetworkManager.conf`. Формат аналогичен формату файлов `XDG-style.desktop` и `Microsoft.ini` с параметрами «ключ — значение», попадающими в разные секции.

Почти в каждом файле конфигурации есть секция `[main]`, которая определяет задействуемые плагины (подключаемые модули). Вот простой пример, который активирует подключаемый модуль `ifupdown`, используемый системами Ubuntu и Debian:

```
[main]
plugins=ifupdown,keyfile
```

Другими специфичными для конкретного дистрибутива плагинами являются `ifcfg-rh` (для семейства Red Hat) и `ifcfg-suse` (для SuSE). Подключаемый модуль `keyfile` поддерживает встроенную поддержку файлов конфигурации NetworkManager. При использовании подключаемого модуля вы можете просмотреть все известные подключения системы в `/etc/NetworkManager/system-connections`.

По большей части вам не нужно будет изменять файл `NetworkManager.conf`, потому что более конкретные параметры конфигурации находятся в других файлах.

Неуправляемые интерфейсы

Хотя вы можете захотеть, чтобы NetworkManager управлял большинством ваших сетевых интерфейсов, будут моменты, когда потребуется, чтобы он игнорировал интерфейсы. Например, большинству пользователей не понадобится динамическая конфигурация интерфейса localhost (`lo`; см. раздел 9.16), поскольку его конфигурация никогда не меняется. К тому же лучше настроить этот интерфейс на ранней стадии загрузки, потому что от него часто зависят базовые системные службы. Большинство дистрибутивов держат NetworkManager подальше от localhost.

Вы можете указать NetworkManager игнорировать интерфейс с помощью подключаемых модулей. Если используете подключаемый модуль `ifupdown` (например, в Ubuntu и Debian), добавьте конфигурацию интерфейса в файл `/etc/network/interfaces`, а затем установите значение `managed` в `false` в секции `ifupdown` в файле `NetworkManager.conf`:

```
[ifupdown]
managed=false
```

Для плагина `ifcfg-rh`, который применяют Fedora и Red Hat, найдите в каталоге `/etc/sysconfig/network-scripts`, содержащем файлы конфигурации `ifcfg-*`, строку

```
NM_CONTROLLED=yes
```

Если она отсутствует или значение равно `no`, NetworkManager игнорирует интерфейс. В случае локального хоста значение будет деактивировано в файле `ifcfg-lo`. Вы также можете указать аппаратный адрес, который нужно игнорировать, например:

```
HWADDR=10:78:d2:eb:76:97
```

Если вы не задействуете ни одну из этих схем конфигурации сети, все равно можете применить подключаемый модуль `keyfile`, чтобы указать неуправляемое

устройство непосредственно в файле `NetworkManager.conf` с помощью его MAC-адреса. Вот пример, показывающий два неуправляемых устройства:

```
[keyfile]
unmanaged-devices=mac:10:78:d2:eb:76:97;mac:1c:65:9d:cc:ff:b9
```

Диспетчеризация

Последняя деталь конфигурации NetworkManager связана с определением дополнительных системных действий при включении или выключении сетевого интерфейса. В частности, некоторым сетевым демонам (например, демону безопасной оболочки, обсуждаемому в следующей главе) для правильной работы необходимо знать, когда начинать или прекращать прослушивание интерфейса.

Когда состояние сетевого интерфейса системы изменяется, NetworkManager запускает все в `/etc/NetworkManager/dispatcher.d` с аргументами `up` или `down`. Все довольно просто, но многие дистрибутивы имеют собственные сценарии управления сетью, поэтому не помещают отдельные сценарии диспетчера в этот каталог. В Ubuntu, например, есть только один скрипт с именем `01ifupdown`, который запускает все в соответствующем подкаталоге `/etc/network` — `/etc/network/if-up.d`.

Как и в случае с остальной конфигурацией NetworkManager, детали этих сценариев не особо важны: вам нужно лишь знать, как отследить подходящее местоположение, если требуется что-то добавить или изменить (или использовать Netplan и позволить ему определить местоположение самостоятельно). Как всегда, не забывайте просматривать сценарии в своей системе.

9.15. Разрешения сетевых имен

Одной из последних основных задач в любой конфигурации сети является разрешение имен хостов (hostname resolution) с помощью DNS. Мы уже видели инструмент host, который переводит такое имя, как www.example.com, в IP-адрес 10.23.2.132.

DNS отличается от сетевых элементов, которые мы рассматривали до сих пор, потому что находится на прикладном уровне, полностью в пространстве пользователя. Из-за этого технически он немного неуместен в этой главе, как и обсуждение сетевого и физического уровней. Однако без надлежащей настройки DNS подключение к интернету практически бесполезно. Никто в здравом уме не показывает IP-адреса (тем более IPv6-адреса) для веб-сайтов и адресов электронной почты, потому что IP-адрес хоста может быть изменен, а запомнить кучу цифр непросто.

Практически все сетевые приложения в системе Linux выполняют поиск DNS. Процесс разрешения имен обычно протекает следующим образом.

1. Приложение вызывает функцию для поиска IP-адреса, скрытого за именем хоста. Функция находится в разделяемой библиотеке системы, поэтому при-

ложению не нужно знать подробности о том, как она работает или изменится ли реализация.

2. Когда функция в разделяемой библиотеке запускается, она действует в соответствии с набором правил, найденных в `/etc/nsswitch.conf` (см. раздел 9.15.4), чтобы определить план действий при поиске. Например, в правилах обычно говорится, что еще до обращения к DNS необходимо проверить, нет ли ручного переопределения в файле `/etc/hosts`.
3. Когда функция решает использовать DNS для поиска имени, она обращается к дополнительному файлу конфигурации для поиска сервера имен DNS. Сервер имен задается в качестве IP-адреса.
4. Функция отправляет DNS-запрос (по сети) на сервер имен.
5. Сервер имен отвечает IP-адресом для имени хоста, и функция возвращает этот IP-адрес приложению.

Это упрощенная версия. В типичной современной системе больше участников пытаются ускорить транзакцию или повысить гибкость. Давайте пока проигнорируем это и рассмотрим основные элементы. Как и в случае с другими типами конфигурации сети, вам, вероятно, не потребуется изменять разрешение имени хоста, но знать, как это работает, полезно.

9.15.1. Файл /etc/hosts

В большинстве систем вы можете переопределить поиск имени 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
::1               localhost ip6-localhost
```

Запись (или записи) для `localhost` почти всегда появляется именно здесь (см. раздел 9.16). Другие записи в примере иллюстрируют простой способ добавления хостов в локальную подсеть.

ПРИМЕЧАНИЕ

В старые добрые времена существовал один центральный файл hosts, который каждый копировал на свою машину, чтобы иметь актуальную информацию (см. RFC 606, 608, 623 и 625), но по мере роста ARPANET/интернета это быстро вышло из-под контроля.

9.15.2. Файл resolv.conf

Традиционным файлом конфигурации DNS-серверов является файл `/etc/resolv.conf`. Раньше, когда все работало проще, вот как мог бы выглядеть адрес сервера имен провайдера 10.32.45.23 и 10.3.2.3:

```
search mydomain.example.com example.com
nameserver 10.32.45.23
nameserver 10.3.2.3
```

Строка `search` определяет правила для неполных имен хостов (только для первой часть имени хоста, например `myserver` вместо `myserver.example.com`). В данном случае библиотека разрешения имен resolver попытается найти `host.mydomain.example.com` и `host.example.com`.

Сейчас поиск имен уже не так прост. В конфигурацию DNS было внесено множество улучшений и изменений.

9.15.3. Кэширование и DNS с нулевой конфигурацией

Существуют две основные проблемы с традиционной конфигурацией DNS. Во-первых, локальная машина не кэширует ответы сервера имен, поэтому повторный доступ к сети может неоправданно замедлиться из-за запросов к серверу имен. Чтобы решить эту проблему, многие компьютеры (и маршрутизаторы, если они действуют как серверы имен) запускают промежуточный демон для перехвата запросов сервера имен и кэширования ответа, а затем по возможности используют эти ответы. Наиболее распространенным из этих демонов является `systemd-resolved`, также в системах встречаются `dnsmasq` или `nscd`. Вы можете настроить BIND (стандартный демон сервера имен Unix) в качестве кэша. Можно сказать, что в системе применяется демон кэширования сервера имен, если встречаются адреса 127.0.0.53 или 127.0.0.1 либо в файле `/etc/resolv.conf`, либо в списке серверов при запуске `nslookup -debug` *`host`* (где *`host`* — имя хоста). Однако давайте посмотрим внимательнее. Если вы используете `systemd-resolved`, то можете заметить, что `resolv.conf` даже не является файлом в `/etc` — это ссылка на автоматически созданный файл в `/run`.

Утилита `systemd-resolved` таит в себе гораздо больше, чем кажется на первый взгляд, поскольку она может объединять несколько служб поиска имен и предоставлять их по-разному для каждого интерфейса. А это решает вторую проблему с традиционной настройкой сервера имен: она может быть особенно негибкой, если вы хотите просматривать имена в своей локальной сети, не путаясь с большим количеством настроек. Например, если вы настроили сетевое устройство в своей сети, то захотите немедленно вызвать его по имени. Это часть идеи, лежащей в основе систем службы имен с нулевой конфигурацией, таких как многоадресная DNS (Multicast DNS, mDNS) и протокола широковещательного разрешения имен (Link-Local Multicast Name Resolution, LLMNR). Если процесс хочет найти хост по имени в локальной сети, он просто передает широковещательный запрос по сети, и целевой хост отвечает своим адресом. Эти протоколы выходят за рамки разрешения имен хостов, предоставляя также информацию о доступных службах.

Можно проверить текущие настройки DNS с помощью команды `resolvectl status` (обратите внимание на то, что в старых системах она может называться

`systemd-resolve`). Вы получите список глобальных настроек (обычно редко используемых), а затем увидите настройки для каждого отдельного интерфейса. Это будет выглядеть так:

```
Link 2 (enp0s31f6)
      Current Scopes: DNS
       LLMNR setting: yes
MulticastDNS setting: no
      DNSSEC setting: no
    DNSSEC supported: no
         DNS Servers: 8.8.8.8
          DNS Domain: ~.
```

В примере отображены различные поддерживаемые протоколы имен, а также сервер имен, у которого демон systemd-resolved запрашивает неизвестное ему имя.

Мы не будем углубляться в DNS или systemd-resolved, потому что это очень обширная тема. Если вы хотите изменить свои настройки, загляните на страницу руководства `resolved.conf(5)` и перейдите к внесению изменений в `/etc/systemd/resolved.conf`. Однако вам, вероятно, потребуется ознакомиться с документацией systemd-resolved, а также с DNS в целом, например в книге Крикета Ли и Пола Альбица «DNS и BIND», 5-е издание (Символ-Плюс, 2008).

9.15.4. Файл /etc/nsswitch.conf

Прежде чем закончить с темой поиска имен, рассмотрим одну последнюю настройку, о которой необходимо знать. Файл `/etc/nsswitch.conf` — это традиционный интерфейс для управления несколькими настройками приоритета, связанными с именами в системе (например, информацией о пользователе и пароле), и у него есть настройка поиска хоста. Файл в вашей системе должен содержать такую строку:

```
hosts:          files dns
```

Добавление `files` перед `dns` гарантирует, что при поиске хостов система проверяет файл `/etc/hosts` на поиск хоста, прежде чем запрашивать любой DNS-сервер, в том числе с systemd-resolved. Чаще всего это хорошая затея (особенно для поиска `localhost`, как обсуждается далее), но файл `/etc/hosts` должен быть как можно *меньше*. Не добавляйте в него ничего, чтобы повысить производительность. Вы можете разместить хосты в небольшой частной локальной сети в `/etc/hosts`, но общее правило заключается в том, что, если у конкретного хоста есть запись в DNS, ей нет места в `/etc/hosts`. (Файл `/etc/hosts` полезен также для разрешения имен хостов на ранних стадиях загрузки, когда сеть может быть недоступна.)

Все это работает при помощи стандартных вызовов, имеющихся в системной библиотеке. Может оказаться сложно запомнить все места, в которых может происходить поиск имен, но если вам когда-нибудь понадобится отследить их, начните с `/etc/nsswitch.conf`.

9.16. Localhost

При запуске команды `ip address show` вы встретите интерфейс `lo`:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
```

Интерфейс `lo` — это виртуальный сетевой интерфейс, называемый интерфейсом *возвратной петли (loopback)*, потому что он закольцован сам на себя. В таком случае подключение к 127.0.0.1 (или ::1 в IPv6) означает подключение к компьютеру, который вы используете в данный момент. Когда данные, уходящие на локальный хост, достигают сетевого интерфейса ядра для `lo`, ядро просто переупаковывает их как входящие данные и отправляет обратно через `lo` для применения любой прослушивающей этот интерфейс серверной программой (по умолчанию большинство программ делают это).

Интерфейс `lo` часто является единственным местом, где можно увидеть статическую конфигурацию сети в сценариях во время загрузки. Например, команда `ifup` в системе Ubuntu читает файл `/etc/network/interfaces`. Однако это может быть излишним, поскольку systemd настраивает интерфейс loopback при запуске.

Интерфейс возвратной петли (loopback) имеет одну особенность, которую вы, возможно, заметили. Маска сети равна /8, и все, что начинается со 127, назначается для обратной связи. Это позволяет запускать разные серверы на разных IPv4-адресах в пространстве loopback без настройки дополнительных интерфейсов. Одним из серверов, который использует это преимущество, является systemd-resolved, применяющий 127.0.0.53. Таким образом, это не мешает процессу другого сервера имен, работающего на 127.0.0.1. Пока что IPv6 определяет только один адрес обратной связи, но в будущем, возможно, это изменится.

9.17. Транспортный уровень: TCP, UDP и службы

До сих пор мы видели только, как пакеты перемещаются от хоста к хосту в сети, другими словами, это ответ на вопрос «куда?», заданный в начале главы. Теперь начнем отвечать на вопрос о том, *что* именно передается. Важно знать, как ваш компьютер представляет пакетные данные, которые получает от других хостов, запущенным у себя процессам. Программам пользовательского пространства было бы сложно и неудобно работать с кучей необработанных пакетов так, как это делает ядро. Гибкость особенно важна: несколько приложений должны иметь возможность подключаться к сети одновременно (например, у вас может работать электронная почта и запущено несколько веб-клиентов).

Протоколы *транспортного уровня* устраняют разрыв между необработанными пакетами сетевого уровня и уточненными потребностями приложений. Двумя наиболее популярными транспортными протоколами являются протокол управления передачей (Transmission Control Protocol, TCP) и протокол пользовательских датаграмм (User Datagram Protocol, UDP). Мы сосредоточимся на TCP, потому что это, безусловно, самый распространенный протокол, но коротко рассмотрим и UDP.

9.17.1. TCP-порты и соединения

Протокол TCP обеспечивает работу нескольких сетевых приложений на одном компьютере с помощью *сетевых портов* (network ports), которые являются просто номерами, применяемыми в сочетании с IP-адресом. Если IP-адрес хоста похож на почтовый адрес многоквартирного дома, то номер порта похож на номер почтового ящика — это еще одно уточнение.

При использовании TCP приложение открывает *соединение* (не путать с подключениями NetworkManager) между одним портом на своем компьютере и портом на удаленном хосте. Например, браузер может открыть соединение между портом 36406 на собственном компьютере и портом 80 на удаленном хосте. С точки зрения приложения порт 36406 является локальным, а порт 80 — удаленным.

Вы можете идентифицировать соединение, используя пару IP-адресов и номеров портов. Для просмотра соединений, открытых в данный момент на вашем компьютере, задействуйте команду `netstat`. Приведем пример, который показывает TCP-соединения, параметр `-n` отключает разрешение имен хостов (DNS), а параметр `-t` ограничивает вывод только выводом 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` (Локальный адрес) и `Foreign Address` (Внешний адрес) относятся к подключениям с точки зрения вашего компьютера, поэтому на данном компьютере интерфейс настроен на 10.23.2.4, и все порты 47626, 41475 и 57132 на локальной стороне подключены. Первое соединение в примере — от порта 47626 к порту 5222 на 10.194.79.125.

Чтобы отображать только подключения IPv6, добавьте параметр `-6` в команду `netstat`.

Установка TCP-соединений

Чтобы установить соединение транспортного уровня, процесс на одном хосте инициирует соединение одного из своих локальных портов с портом на втором хосте

с помощью специальной серии пакетов. Чтобы распознать входящее соединение и ответить, у второго хоста должен быть процесс, *прослушивающий* правильный порт. Обычно подключающийся процесс называется *клиентом*, а прослушивающий — *сервером* (подробнее об этом в главе 10).

Важно знать, что клиент выбирает на своей стороне порт, который в настоящее время не используется, и почти всегда подключается к какому-либо известному порту на стороне сервера. Давайте снова рассмотрим вывод 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
```

Зная о соглашениях о нумерации портов, вы можете увидеть, что это соединение, вероятно, было инициировано локальным клиентом `r` удаленному серверу, потому что порт на локальной стороне (47626) выглядит как динамически назначаемый номер, в то время как удаленный порт (5222) является хорошо известной службой, указанной в `/etc/services` (служба обмена сообщениями Jabber, или XMPP, если быть точным). Существует множество подключений к порту 443 (по умолчанию для HTTPS) на большинстве ПК.

ПРИМЕЧАНИЕ

Динамически назначаемый порт называется эфемерным портом.

Однако если локальный порт в выводе хорошо известен, удаленный хост, вероятно, инициировал соединение. В этом примере удаленный хост 172.24.54.234 подключился к порту 443 на локальном хосте:

```
Proto Recv-Q Send-Q Local Address       Foreign Address         State
tcp        0      0 10.23.2.4:443       172.24.54.234:43035     ESTABLISHED
```

Удаленный хост, подключающийся к вашей машине через хорошо известный порт, подразумевает, что сервер на вашей локальной машине прослушивает этот порт. Чтобы подтвердить это, перечислите все TCP-порты, прослушиваемые вашей машиной, с помощью `netstat`, на этот раз с параметром `-l`, который показывает прослушиваемые процессами порты:

```
$ netstat -ntl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address       Foreign Address         State
❶ tcp        0      0 0.0.0.0:80          0.0.0.0:*               LISTEN
❷ tcp        0      0 0.0.0.0:443         0.0.0.0:*               LISTEN
❸ tcp        0      0 127.0.0.53:53       0.0.0.0:*               LISTEN
--пропуск--
```

Строка ❶ с локальным адресом 0.0.0.0:80 показывает, что локальная машина прослушивает порт 80 для подключений с любой удаленной машины, то же самое верно для порта 443 (строка ❷). Сервер может ограничить доступ к определенным

интерфейсам, как показано в строке ❸, где что-то прослушивает соединения только в интерфейсе localhost. В этом случае используется systemd-resolved — мы говорили о том, почему он прослушивает с адреса 127.0.0.53 вместо 127.0.0.1, еще в разделе 9.16. Чтобы узнать больше, примените `lsof` для определения конкретного процесса, выполняющего прослушивание (как описано в подразделе 10.5.1).

Номера портов и файл /etc/services

Как узнать, хорошо ли известен порт? Одного способа определить это не существует, но для начала стоит заглянуть в `/etc/services`, который переводит известные номера портов в имена. Это обычный текстовый файл. В нем вы встретите записи, подобные этой:

```
ssh          22/tcp         # SSH Remote Login Protocol
smtp         25/tcp
domain       53/udp
```

Первый столбец — это имя, второй — номер порта и конкретный протокол транспортного уровня, который может отличаться от TCP.

ПРИМЕЧАНИЕ

Помимо /etc/services существует онлайн-реестр портов на www.iana.org, который регулируется документом по сетевым стандартам RFC 6335.

В Linux только процессы, работающие от имени суперпользователя, могут задействовать порты с 1 по 1023, известные также как системные или привилегированные порты. Все пользовательские процессы могут прослушивать и создавать соединения с портов 1024 и выше.

Характеристики TCP

TCP популярен как протокол транспортного уровня, потому что требует не так уж много действий со стороны приложения. Приложение должно знать только, как открывать (или прослушивать), считывать, записывать и закрывать соединение. Для приложения это входящие и исходящие потоки данных, процесс почти так же прост, как работа с файлом.

Тем не менее много действий происходит за кулисами. Во-первых, реализация TCP должна знать, как разбить исходящий поток данных из процесса на пакеты. Сложнее всего понять, как преобразовать серию входящих пакетов во входной поток данных, чтобы процессы могли их считывать, особенно когда входящие пакеты поступают в случайном порядке. Кроме того, хост, использующий TCP, должен проверять наличие ошибок: пакеты могут теряться или искажаться при отправке по сети, и реализация TCP должна обнаруживать и исправлять эти ситуации. На рис. 9.4 показан упрощенный пример того, как хост может применять TCP для отправки сообщения.

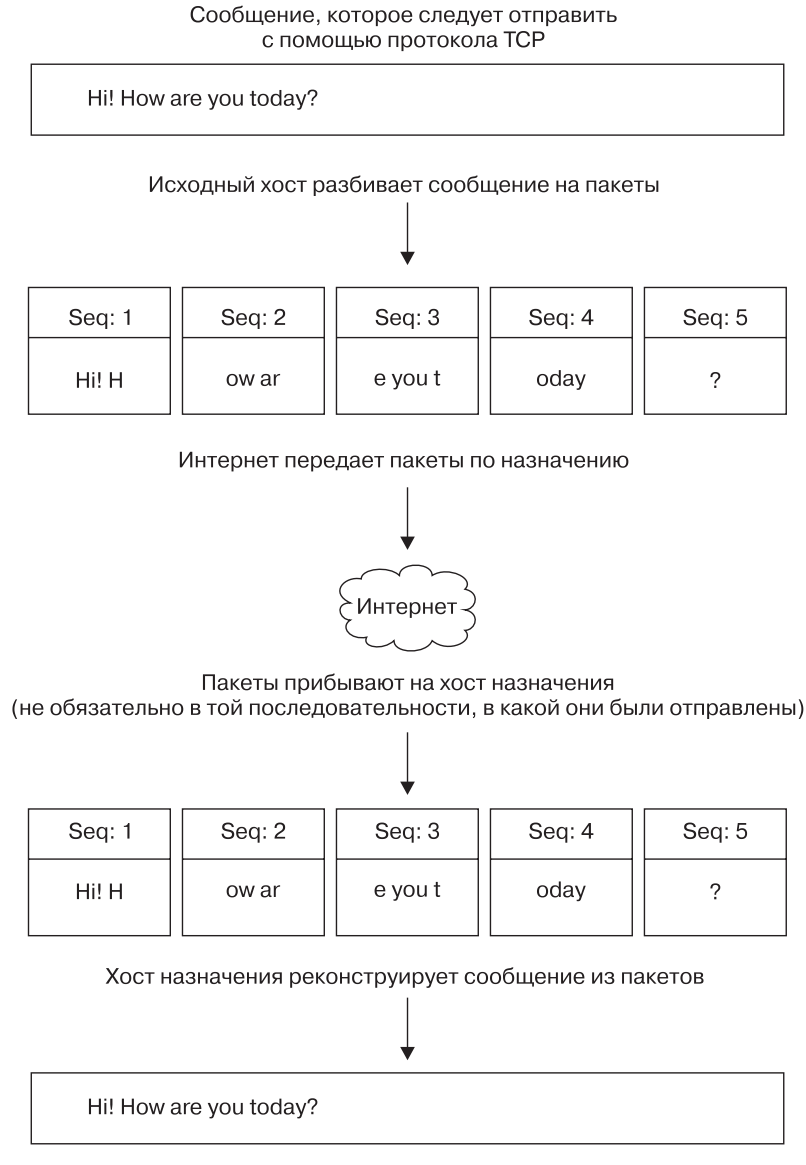


Рис. 9.4. Отправка сообщения по протоколу TCP

К счастью, пользователю не нужно с этим взаимодействовать, стоит только знать, что реализация TCP Linux в основном находится в ядре и что утилиты, работающие с транспортным уровнем, как правило, манипулируют структурами данных ядра. Одним из примеров является система фильтрации пакетов `iptables`, обсуждаемая в разделе 9.25.

9.17.2. Протокол UDP

UDP — это более простой, чем TCP, транспортный уровень. Он определяет передачу лишь отдельных сообщений, поток данных отсутствует. В то же время, в отличие от TCP, UDP не будет исправлять потерянные или неупорядоченные пакеты. На самом деле, хотя UDP имеет порты, у него даже нет соединений! Один хост просто отправляет сообщение с одного из своих портов на порт сервера, и сервер при желании отправляет что-то обратно. Однако в UDP есть функция обнаружения ошибок для данных внутри пакета: хост может обнаружить, что пакет поврежден, но не будет ничего с этим делать.

Если работа TCP похожа на телефонный разговор, то работа UDP подобна отправке письма, телеграммы или мгновенного сообщения (за исключением того, что мгновенные сообщения более надежны). Приложения, использующие UDP, часто стремятся отправлять сообщения как можно быстрее. Они не хотят применять усложненный TCP, потому что предполагают, что сеть между двумя хостами в целом надежна. Они не нуждаются в исправлении ошибок, выполняемом TCP, потому что у них либо есть собственные системы обнаружения ошибок, либо они не обращают на них внимания.

Одним из примеров приложения, использующего UDP, является *протокол сетевого времени* (Network Time Protocol, NTP). Клиент отправляет короткий простой запрос на сервер, чтобы узнать текущее время, и получает от сервера такой же короткий ответ. Поскольку клиент хочет получить ответ как можно быстрее, UDP подходит для приложения: если ответ с сервера затеряется где-то в сети, клиент может просто отправить запрос повторно. Другой пример — видеочат. В этом случае картинки отправляются с помощью UDP, и если некоторые фрагменты теряются по пути, клиент на принимающей стороне компенсирует это как может.

ПРИМЕЧАНИЕ

Остальная часть этой главы посвящена более сложным сетевым темам, таким как сетевая фильтрация и маршрутизаторы, поскольку они относятся к нижним сетевым уровням, которые мы уже встречали: физическому, сетевому и транспортному. Если хотите, переходите к следующей главе, чтобы изучить уровень приложений, где все части объединяются в пользовательском пространстве. Вы увидите процессы, которые на самом деле задействуют сеть, а не просто перекидывают кучу адресов и пакетов.

9.18. Возврат к простой локальной сети

Сейчас мы рассмотрим дополнительные компоненты простой сети, представленные в разделе 9.4. Эта сеть состоит из одной LAN в качестве подсети и маршрутизатора, который соединяет ее с остальной частью интернета. Мы рассмотрим:

- как хост в подсети автоматически получает свою сетевую конфигурацию;
- как настроить маршрутизацию;

- что такое маршрутизатор на самом деле;
- как узнать, какие IP-адреса использовать для подсети;
- как настроить брандмауэры для фильтрации нежелательного трафика из интернета.

По большей части мы сосредоточимся на IPv4 (хотя бы по той причине, что такие адреса легче читать), но рассмотрим и отличия IPv6.

Начнем с того, как хост в подсети автоматически получает свою сетевую конфигурацию.

9.19. Протокол DHCP

В соответствии с IPv4 при настройке сетевого хоста на автоматическое получение конфигурации из сети пользователь указывает ему задействовать протокол динамической настройки хоста (Dynamic Host Configuration Protocol, DHCP) для получения IP-адреса, маски подсети, шлюза по умолчанию и DNS-серверов. Применение DHCP не только избавляет сетевых администраторов от необходимости вводить эти параметры вручную, но и дает им другие преимущества, такие как предотвращение конфликтов IP-адресов и минимизация последствий при изменениях сети. Редко встречается такая сеть, которая не использует DHCP.

Чтобы хост мог настроить свою конфигурацию с помощью DHCP, он должен иметь возможность отправлять сообщения на DHCP-сервер в своей подключенной сети. Поэтому каждая физическая сеть должна иметь собственный DHCP-сервер, и в простой сети (например, как в разделе 9.1) маршрутизатор обычно действует как DHCP-сервер.

ПРИМЕЧАНИЕ

При отправке первоначального запроса DHCP хост даже не знает адреса DHCP-сервера, поэтому он передает широковещательный запрос всем хостам (обычно всем хостам в своей физической сети).

Когда компьютер просит DHCP-сервер назначить ему IP-адрес, на самом деле он запрашивает аренду адреса на определенное время. Когда оно истекает, клиент может попросить продлить договор аренды.

9.19.1. DHCP-клиенты Linux

Хотя существует множество различных типов систем управления сетями, только два DHCP-клиента могут фактически арендовать адреса. Традиционным типовым клиентом является программа `dhclient`, работающая по стандартам Internet Software Consortium (ISC, Консорциум по разработке ПО для сети интернет). Однако у systemd-networkd теперь также есть встроенный DHCP-клиент.

При запуске `dhclient` сохраняет свой идентификатор процесса в `/var/run/dhclient.pid` и информацию об аренде в `/var/lib/dhcp/dhclient.leases`.

Вы можете протестировать `dhclient` вручную в командной строке, но перед этим *должны* удалить любой маршрут шлюза (см. подраздел 9.11.2). Чтобы запустить тестирование, просто укажите имя сетевого интерфейса (в примере это `enp0s31f6`):

```
# dhclient enp0s31f6
```

В отличие от `dhclient`, DHCP-клиент systemd-networkd не может быть запущен вручную в командной строке. Конфигурация, описанная на странице руководства `systemd.network(5)`, находится в каталоге `/etc/systemd/network`, но, как и другие виды конфигурации сети, может быть автоматически сгенерирована с помощью Netplan.

9.19.2. DHCP-серверы Linux

Поставьте системе Linux задачу запустить DHCP-сервер, который обеспечивает хороший контроль над выдаваемыми им адресами. Но если вы не управляете большой сетью со множеством подсетей, вам, вероятно, стоит использовать специализированное оборудование маршрутизатора, которое включает встроенные DHCP-серверы.

Вероятно, важнее всего знать о DHCP-серверах следующее: чтобы избежать проблем с конфликтующими IP-адресами или неправильными конфигурациями, необходимо, чтобы в одной подсети работал только один сервер.

9.20. Автоматическая настройка сети IPv6

DHCP работает хорошо, но он основывается на определенных предположениях, в том числе на том, что DHCP-сервер будет доступен, он правильно реализован и стабилен и может отслеживать и поддерживать аренду адресов. Хотя есть версия DHCP для IPv6 под названием DHCPv6, существует и более распространенная альтернатива.

IETF воспользовалась большим адресным пространством IPv6 для разработки нового способа настройки сети, который не требует центрального сервера. Он называется *конфигурацией без состояния*, поскольку клиентам не нужно хранить какие-либо данные, к примеру назначения аренды.

Конфигурация сети IPv6 без сохранения состояния начинается с сети локального канала. Напомним, что эта сеть включает адреса с префиксом fe80::/64. Поскольку в сети локального канала связи имеется так много доступных адресов, хост может сгенерировать адрес, который вряд ли будет дублироваться где-либо в сети. Кроме того, сетевой префикс уже зафиксирован, поэтому хост может транслировать в сеть, спрашивая, использует ли какой-либо другой хост в сети этот адрес.

Как только у хоста появится локальный адрес канала, он может определить глобальный адрес. Это делается прослушиванием сообщения об *объявлении маршрутизатора* (router advertisement, RA), которое маршрутизаторы иногда отправляют

по локальной сети канала. RA содержит префикс глобальной сети, IP-адрес маршрутизатора и, возможно, информацию DNS. С помощью этих данных хост может попытаться заполнить часть идентификатора интерфейса глобального адреса аналогично тому, что он сделал с локальным адресом канала.

Конфигурация без сохранения состояния зависит от префикса глобальной сети длиной не более 64 бит (другими словами, ее маска сети равна /64 или ниже).

ПРИМЕЧАНИЕ

Маршрутизаторы отправляют RA также в ответ на запросы маршрутизаторов от хостов. Эти и некоторые другие сообщения являются частью протокола ICMP для IPv6 (ICMPv6).

9.21. Настройка Linux в качестве маршрутизатора

Маршрутизаторы — это просто компьютеры с несколькими физическими сетевыми интерфейсами. Вы можете легко настроить компьютер Linux в качестве маршрутизатора.

Рассмотрим пример. Допустим, у вас есть две подсети локальной сети, 10.23.2.0/24 и 192.168.45.0/24. Для их соединения имеется маршрутизатор Linux с тремя сетевыми интерфейсами: два для подсетей локальной сети и один для uplink-связи с интернетом (рис. 9.5).

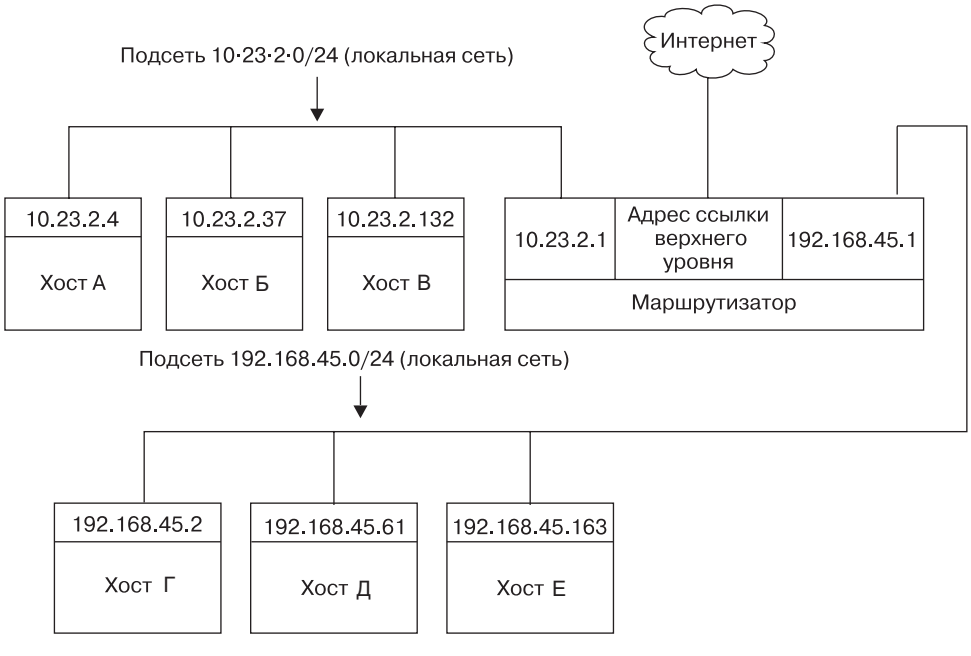


Рис. 9.5. Две подсети, соединенные маршрутизатором

Видно, что этот пример не сильно отличается от простого примера сети, рассматриваемого в предыдущей части главы. IP-адреса маршрутизатора для подсетей локальной сети — 10.23.2.1 и 192.168.45.1. Когда они настроены, таблица маршрутизации выглядит примерно так (на практике имена интерфейсов могут быть иными; пока игнорируйте uplink-канал к интернету):

```
# ip route show
10.23.2.0/24 dev enp0s31f6 proto kernel scope link src 10.23.2.1 metric 100
192.168.45.0/24 dev enp0s1 proto kernel scope link src 192.168.45.1 metric 100
```

Теперь предположим, что хосты в каждой подсети имеют маршрутизатор в качестве шлюза по умолчанию: 10.23.2.1 для 10.23.2.0/24 и 192.168.45.1 для 192.168.45.0/24. Если 10.23.2.4 хочет отправить пакет чему-либо за пределами 10.23.2.0/24, он передает пакет в 10.23.2.1. Например, чтобы пакет попал с 10.23.2.4 (хост *A*) на 192.168.45.61 (хост *Д*), он отправляется в 10.23.2.1 (маршрутизатор) через интерфейс `enp0s31f6`, а затем уходит с маршрутизатора через интерфейс `enp0s1`.

Однако в некоторых базовых конфигурациях ядро Linux не перемещает пакеты автоматически из одной подсети в другую. Чтобы включить эту базовую функцию маршрутизации, вам необходимо задать *перенаправление IP-адресов* в ядре маршрутизатора с помощью команды

```
# sysctl -w net.ipv4.ip_forward=1
```

Как только вы введете эту команду, машина должна начать маршрутизацию пакетов между подсетями, предполагая, что хосты в них знают, как отправлять свои пакеты на только что созданный вами маршрутизатор.

ПРИМЕЧАНИЕ

Вы можете проверить состояние переадресации IP-адресов с помощью команды sysctl net.ipv4.ip_forward.

Чтобы сделать это изменение постоянным после перезагрузки, добавьте его в свой файл `/etc/sysctl.conf`. В зависимости от дистрибутива может существовать возможность поместить его в файл `/etc/sysctl.d`, чтобы обновления дистрибутива не перезаписывали изменения.

Когда у маршрутизатора есть третий сетевой интерфейс с каналом uplink, эта настройка обеспечивает доступ в интернет для всех хостов в обеих подсетях, поскольку они настроены на использование маршрутизатора в качестве шлюза по умолчанию.

Вот тут все становится еще сложнее. Проблема в том, что некоторые IPv4-адреса, например 10.23.2.4, не видны всему интернету — они находятся в так называемых *частных сетях*. Чтобы обеспечить подключение к интернету, необходимо настроить функцию *преобразования сетевых адресов* (Network Address Translation, NAT) на маршрутизаторе. Программное обеспечение почти на всех специализированных маршрутизаторах делает это, здесь нет ничего необычного, но давайте рассмотрим проблему частных сетей немного подробнее.

9.22. Частные сети (IPv4)

Допустим, вы решили создать собственную сеть. У вас есть компьютеры, маршрутизатор и сетевое оборудование. Учитывая то, что вы уже знаете о простой сети, ваш следующий вопрос: «Какую IP-подсеть следует использовать?»

Если вам нужен блок интернет-адресов, которые может видеть каждый хост в интернете, вы можете купить его у своего интернет-провайдера. Но поскольку диапазон IPv4-адресов очень ограничен, эта затея становится дорогой и бесполезной для чего-то большего, чем работа с одним сервером, который может видеть весь интернет. Большинству пользователей на самом деле не нужны такого рода услуги, потому что они выходят в интернет как клиенты.

Обычной недорогой альтернативой является выбор частной подсети из адресов, указанных в документах по стандартам интернета RFC 1918/6761 (табл. 9.2).

Таблица 9.2. Частные сети, определенные 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 создавать частные подсети по своему усмотрению. Если не планируете более 254 хостов в одной сети, выберите небольшую подсеть, например 10.23.2.0/24, которую мы использовали на протяжении всей этой главы. (Сети с этой маской сети иногда называют *подсетями класса C*. Хотя этот термин технически устарел, он все еще полезен.)

В чем же подвох? Хосты в реальном интернете ничего не знают о частных подсетях и не будут отправлять им пакеты, поэтому без некоторой помощи хосты, составляющие частную подсеть, не могут общаться с внешним миром. Маршрутизатор, подключенный к интернету (с глобальным, не частным адресом), должен иметь какой-то способ заполнить пробел между этим соединением и хостами в частной сети.

9.23. Преобразование сетевых адресов NAT (маскарадинг IP-адресов)

NAT — это наиболее часто применяемый способ совместного использования одного IP-адреса частной сетью, он почти универсален для домашних и небольших офисных сетей. В Linux вариант NAT, с которым работают большинство людей, известен как *маскарадинг IP (IP masquerading)*.

Основная суть NAT заключается в том, что маршрутизатор не просто перемещает пакеты из одной подсети в другую — он преобразует их по мере перемещения. Хосты в интернете знают, как подключиться к маршрутизатору, но им ничего не известно о частной сети, стоящей за ним. Хосты в частной сети не нуждаются в специальной настройке, маршрутизатор является их шлюзом по умолчанию.

Система работает примерно так:

1. Хост во внутренней частной сети хочет установить соединение с внешним миром, поэтому он отправляет свои пакеты запросов на соединение через маршрутизатор.
2. Маршрутизатор перехватывает пакет запроса на соединение, а не передает его в интернет, где он будет потерян, потому что в общедоступном интернете нет частных сетей.
3. Маршрутизатор определяет пункт назначения пакета-запроса и открывает собственное соединение с пунктом назначения.
4. Установив соединение, маршрутизатор подделывает сообщение «соединение установлено» и отправляет его на исходный внутренний хост.
5. Маршрутизатор теперь является посредником между внутренним хостом и пунктом назначения. Адресат ничего не знает о внутреннем хосте — соединение на удаленном хосте выглядит так, как будто оно пришло от маршрутизатора.

Это не так просто, как кажется. Обычная IP-маршрутизация знает только IP-адреса источника и назначения на сетевом интернет-уровне. Но если бы маршрутизатор работал только с интернет-уровнем, каждый хост во внутренней сети мог бы одновременно устанавливать только одно соединение с одним пунктом назначения (среди прочих ограничений), поскольку в части пакета интернет-уровня нет информации, позволяющей различать несколько запросов от одного и того же хоста к одному и тому же пункту назначения. Следовательно, NAT должен выходить за рамки интернет-уровня и анализировать пакеты, чтобы извлекать больше идентифицирующей информации, в частности номера портов UDP и TCP транспортных уровней. UDP довольно прост, потому что есть порты, но нет соединений, а транспортный уровень TCP сложнее.

Чтобы настроить компьютер Linux для работы в качестве маршрутизатора NAT, вы должны активировать следующие функции в конфигурации ядра: фильтрацию сетевых пакетов (поддержку брандмауэра), отслеживание соединений, поддержку `iptables`, полную поддержку NAT и целевую поддержку `MASQUERADE`. Большинство ядер дистрибутива поставляются с ними.

Затем нужно реализовать несколько сложно выглядящих команд `iptables`, чтобы маршрутизатор выполнял NAT для своей частной подсети. Вот пример, который применим к внутренней сети Ethernet на `enp0s2`, совместно использующей внешнее соединение на `enp0s31f6` (вы узнаете больше о синтаксисе `iptables` в разделе 9.25):

```
# sysctl -w net.ipv4.ip_forward=1
# iptables -P FORWARD DROP
# iptables -t nat -A POSTROUTING -o enp0s31f6 -j MASQUERADE
# iptables -A FORWARD -i enp0s31f6 -o enp0s2 -m state --state
ESTABLISHED,RELATED -j ACCEPT
# iptables -A FORWARD -i enp0s2 -o enp0s31f6 -j ACCEPT
```

Скорее всего, вам никогда не понадобится вручную вводить эти команды, если вы не разрабатываете собственное программное обеспечение, особенно при наличии такого большого количества специального оборудования маршрутизаторов. Однако различные программы виртуализации могут настраивать NAT для использования в сети для виртуальных машин и контейнеров.

Хотя NAT хорошо работает на практике, помните, что, по сути, это обходной маневр, который продлевает срок службы адресного пространства IPv4.

IPv6 не нуждается в NAT благодаря большему и более сложному адресному пространству, описанному в разделе 9.7.

9.24. Linux и маршрутизаторы

Сразу после появления широкополосной связи пользователи с не слишком большими потребностями просто подключали свои устройства непосредственно к интернету. Но через некоторое время многие захотели применять для собственных сетей одно широкополосное соединение, и пользователи Linux, в частности, нередко устанавливали дополнительную машину для применения в качестве маршрутизатора с NAT.

Производители отреагировали на новый рынок, предложив специализированное оборудование маршрутизатора, состоящее из эффективного процессора, флеш-памяти и нескольких сетевых портов, чьей мощности было достаточно для управления типичной простой сетью, запуска важного программного обеспечения, такого как DHCP-сервер, и использования NAT. Когда дело дошло до программного обеспечения, многие производители обратились к Linux, чтобы с его помощью приводить в действие свои маршрутизаторы. Они добавили необходимые функции ядра, сократили программное обеспечение пользовательского пространства и создали интерфейсы администрирования на основе графического интерфейса.

Почти сразу же после появления первого из этих маршрутизаторов многие пользователи заинтересовались более глубоким изучением аппаратного обеспечения. Один производитель, Linksys, должен был выпустить исходный код своего программного обеспечения в соответствии с условиями лицензии одного из его компонентов, и вскоре для маршрутизаторов появились специализированные дистрибутивы Linux, такие как OpenWRT. (WRT в названии произошло от номера модели Linksys.)

Стоит ли устанавливать эти дистрибутивы на маршрутизаторы? Да, ведь они более стабильны, чем прошивка производителя, особенно на более старом оборудовании маршрутизатора, и обычно предлагают дополнительные функции. Например, многие производители требуют, чтобы для создания моста между сетью и беспроводным подключением вы покупали соответствующее оборудование, но при установленном OpenWRT производитель и возраст оборудования уже не имеют значения. Это связано с тем, что на маршрутизаторе используется действительно открытая операционная система, которой все равно, какое оборудование применяется, лишь бы оно поддерживалось.

Вы можете использовать большую часть сведений, изложенных в этой книге, для изучения внутренних компонентов встроенного программного обеспечения Linux, хотя, конечно, столкнетесь с различиями, особенно после входа в систему. Как и во многих встроенных системах, открытая прошивка, как правило, задействует BusyBox для предоставления многих функций оболочки. BusyBox — это одна исполняемая программа, которая предлагает ограниченную функциональность для многих команд Unix, таких как `shell`, `ls`, `grep`, `cat` и др., что экономит значительный объем памяти. Кроме того, инициализация во встроенных системах во время загрузки обычно очень проста. Однако, скорее всего, вы не обратите на эти ограничения внимания, потому что пользовательская прошивка Linux часто включает интерфейс веб-администрирования, аналогичный имеющемуся у производителя.

9.25. Брандмауэры

Маршрутизаторы должны иметь какой-то брандмауэр, чтобы не пропускать нежелательный трафик из вашей сети. *Брандмауэр* (firewall) — это конфигурация программного и/или аппаратного обеспечения, которая обычно находится на маршрутизаторе между интернетом и сетью меньшего размера, пытаясь гарантировать, что ничто плохое из интернета не повредит сети меньшего размера. Вы можете настроить функции брандмауэра на любом хосте для отображения всех входящих и исходящих данных на уровне пакетов, в отличие от уровня приложений, где серверные программы обычно пытаются самостоятельно контролировать доступ. Брандмауэр на отдельных компьютерах иногда называют *IP-фильтрацией (IP filtering)*.

Система может фильтровать, получать, отправлять пакет или пересылать (маршрутизировать) пакеты другому хосту или шлюзу.

Без брандмауэра система просто обрабатывает пакеты и отправляет их дальше. Брандмауэры устанавливают контрольные точки для пакетов в только что перечисленных пунктах передачи данных. Контрольные точки отклоняют или принимают пакеты, как правило, на основе таких критериев:

- IP-адрес источника или места назначения либо подсеть;
- порт источника или места назначения (в информации транспортного уровня);
- сетевой интерфейс брандмауэра.

Брандмауэры позволяют работать с подсистемой ядра Linux, которая обрабатывает IP-пакеты. Давайте это рассмотрим.

9.25.1. Основы брандмауэров Linux

В Linux вы создаете правила брандмауэра в последовательности под названием *цепочка (chain)*. Набор цепочек составляет *таблицу (table)*. Когда пакет проходит через различные части сетевой подсистемы Linux, ядро применяет к пакетам правила, входящие в определенные цепочки. Например, новый пакет, поступающий с физического уровня, классифицируется ядром как ввод (input), поэтому он активирует правила, включенные в цепочки, соответствующие вводу.

Все эти структуры данных поддерживаются ядром. Вся система называется *iptables*. Для работы с ней предназначена команда пользовательского пространства `iptables` — она позволяет создавать правила и управлять ими.

ПРИМЕЧАНИЕ

Существует система под названием nftables, предназначенная для замены iptables, но на момент написания книги iptables по-прежнему широко используется. Команда для администрирования nftables — это nft, существует также переводчик iptables в nftables, называемый iptables-translate, для команд iptables, которые здесь встречаются. Еще больше все усложнила недавно внедренная система под названием bpfilter с другим подходом. Постарайтесь не увязнуть в специфике команд, ведь главное — это действия, которые они совершают.

Поскольку таблиц может быть много — и каждая со своими наборами цепочек, которые, в свою очередь, могут содержать множество правил, — движение пакетов может стать довольно сложным. Однако в основном вы будете работать с одной таблицей с именем `filter`, которая управляет основным потоком пакетов. В таблице фильтров есть три основные цепочки: `INPUT` для входящих пакетов, `OUTPUT` для исходящих пакетов и `FORWARD` для маршрутизируемых пакетов.

На рис. 9.6 и 9.7 показаны упрощенные блок-схемы, в которых правила применяются к пакетам в таблице фильтров. Приведены две иллюстрации, потому что пакеты могут либо поступать в систему из сетевого интерфейса (рис. 9.6), либо генерироваться локальным процессом (рис. 9.7).

Как видно из рисунков, входящий пакет из сети может быть задействован пользовательским процессом и не дойти цепочек `FORWARD` и `OUTPUT`. Пакеты, сгенерированные пользовательскими процессами, не достигнут цепочек `INPUT` или `FORWARD`.

Процесс усложняется, потому что на этом пути появляется много дополнительных шагов, помимо этих трех цепочек. Изучите всю диаграмму происходящего, найдя в интернете статьи на тему «Поток пакетов Linux в сетевом фильтре netfilter», но помните, что в эти диаграммы попытались включить все возможные сценарии поступления и передачи пакетов. Часто помочь разобраться может разбиение диаграммы по источникам пакетов, как показано на рис. 9.6 и 9.7.

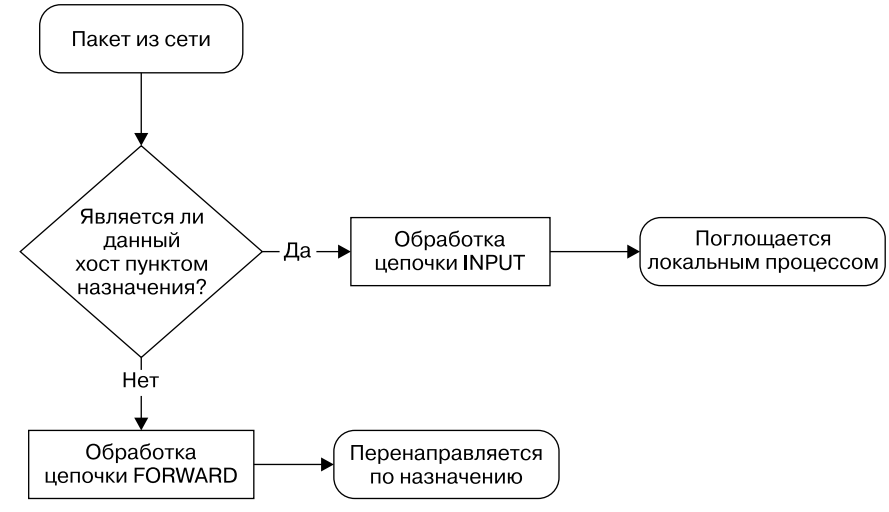


Рис. 9.6. Последовательность обработки входящих пакетов из сети

Рис. 9.7. Последовательность обработки цепочек для входящих пакетов от локального процесса

9.25.2. Настройка правил брандмауэра

Посмотрим, как система `iptables` работает на практике. Начнем с изучения текущей конфигурации с помощью команды

```
# iptables -L
```

Вывод обычно представляет собой пустой набор цепочек, как показано далее:

```
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
```

Каждая цепочка брандмауэров обладает политикой по умолчанию, которая определяет, что делать с пакетом, если он не соответствует ни одному из правил. Политика для всех трех цепочек в этом примере — это политика `ACCEPT`, что означает: ядро позволяет пакету проходить через систему фильтрации пакетов. Политика

`DROP` указывает ядру отбросить пакет. Чтобы установить политику в цепочке, используйте команду `iptables -P` следующим образом:

```
# iptables -P FORWARD DROP
```

ВНИМАНИЕ

Не изменяйте политику на своем компьютере, пока не прочтете этот раздел целиком.

Допустим, что кто-то, находящийся по адресу 192.168.34.63, докучает вам. Чтобы запретить ему общаться с вашей машиной, выполните команду

```
# iptables -A INPUT -s 192.168.34.63 -j DROP
```

Параметр `-A INPUT` добавляет правило к цепочке `INPUT`. Часть `-s` 192.168.34.63 указывает IP-адрес источника в правиле, и часть `-j DROP` приказывает ядру отбросить любой пакет, соответствующий правилу. Таким образом, ваш компьютер отбросит любой пакет, поступающий от 192.168.34.63.

Чтобы увидеть, что правило действует, снова запустите команду `iptables -L`:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all -- 192.168.34.63         anywhere
```

К сожалению, ваш «друг» в 192.168.34.63 велел всем в своей подсети подключаться к вашему SMTP-порту (TCP-порт 25). Чтобы избавиться и от этого трафика, запустите команду

```
# iptables -A INPUT -s 192.168.34.0/24 -p tcp --destination-port 25 -j DROP
```

В этом примере к исходному адресу добавляется квалификатор маски сети, а также параметр `-p tcp` для выбора только пакетов TCP. Еще одно ограничение, `--destination-port 25`, гласит, что правило должно применяться только к трафику на порт 25. Список таблиц IP для `INPUT` теперь будет выглядеть 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
```

Выясняется, что кто-то, кого вы знаете по адресу 192.168.34.37, не может отправить вам электронное письмо, потому что вы заблокировали этот компьютер. Думая, что это быстрое решение, вы запускаете команду

```
# iptables -A INPUT -s 192.168.34.37 -j ACCEPT
```

Однако это не сработало. Чтобы понять почему, посмотрим 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 цепочку сверху вниз, используя первое подходящее правило.

Первое правило не соответствует 192.168.34.37, но соответствует второе, потому что оно применяется ко всем хостам с 192.168.34.1 по 192.168.34.254, и это правило предписывает отбрасывать пакеты. Когда правило подходит, ядро выполняет действие и не смотрит дальше вниз по цепочке. (Обратите внимание, что 192.168.34.37 может отправлять пакеты на любой порт вашего компьютера, кроме порта 25, потому что второе правило применяется только к порту 25.)

Решение состоит в том, чтобы переместить третье правило наверх. Вначале удалите третье правило с помощью команды

```
# iptables -D INPUT 3
```

Затем *вставьте* его в начало цепочки с помощью команды `iptables -I`:

```
# iptables -I INPUT -s 192.168.34.37 -j ACCEPT
```

Чтобы вставить правило в другое место цепочки, поместите его номер после имени цепочки (например, `iptables -I INPUT 4 ...`).

9.25.3. Стратегии брандмауэра

Хотя ранее мы рассмотрели, как вставлять правила и как ядро обрабатывает цепочки IP-адресов, но еще не изучили стратегий брандмауэра, которые действительно работают. Поговорим об этом.

Существуют два основных типа сценариев брандмауэра: один для защиты отдельных машин, где вы устанавливаете правила в цепочке `INPUT` каждой машины, другой для защиты сети машин, где устанавливаете правила в цепочке `FORWARD` маршрутизатора. В обоих случаях речь о серьезной безопасности не идет, если вы используете политику по умолчанию `ACCEPT` и постоянно вставляете правила для отбрасывания пакетов от источников, которые начинают отправлять ненужные или вредоносные данные. Необходимо разрешать только те пакеты, которым доверяете, и запрещать все остальное.

Предположим, что на вашем компьютере установлен SSH-сервер на TCP-порте 22. У любого случайного хоста нет причин инициировать подключение к любому другому порту на вашем компьютере, и вы не должны давать ни одному подобному хосту такой возможности. Чтобы это настроить, сначала установите политику цепочки `INPUT` в `DROP`:

```
# iptables -P INPUT DROP
```

Чтобы включить ICMP-трафик (для `ping` и других утилит), используйте строку

```
# iptables -A INPUT -p icmp -j ACCEPT
```

Убедитесь, что можете получать пакеты, которые отправляете как на собственный сетевой IP-адрес, так и на 127.0.0.1 (localhost). Предположим, что IP-адрес вашего хоста — *my_addr*. Выполните следующую команду:

```
# iptables -A INPUT -s 127.0.0.1 -j ACCEPT
# iptables -A INPUT -s my_addr -j ACCEPT
```

ВНИМАНИЕ

Не выполняйте эти команды одну за другой на машине, к которой у вас есть только удаленный доступ. Самая первая команда DROP мгновенно заблокирует доступ, и вы не сможете восстановить его, пока не вмешаетесь в работу всей системы (например, перезагрузив компьютер).

Если вы контролируете всю свою подсеть (и доверяете всему в ней), то можете заменить *my_addr* своим адресом подсети и маской подсети, например 10.23.2.0/24.

Теперь, даже если мы хотим запретить входящие TCP-соединения, все равно нужно убедиться, что хост может устанавливать TCP-соединения с внешним миром. Поскольку все TCP-соединения начинаются с пакета SYN (запроса соединения), то если вы пропускаете все TCP-пакеты, которые не являются SYN-пакетами, все будет в порядке:

```
# iptables -A INPUT -p tcp '!' --syn -j ACCEPT
```

Символ `!` указывает на отрицание, так что `! --syn` соответствует любому не-SYN пакету.

Далее, если вы используете удаленный DNS на основе UDP, необходимо принимать трафик с сервера имен, чтобы ваша машина могла искать имена с помощью DNS. Выполните это для всех DNS-серверов в `/etc/resolv.conf`. Примените следующую команду, где адрес сервера имен — это *ns_addr*:

```
# iptables -A INPUT -p udp --source-port 53 -s ns_addr -j ACCEPT
```

И наконец, разрешите SSH-соединения с кем угодно:

```
# iptables -A INPUT -p tcp --destination-port 22 -j ACCEPT
```

Приведенные настройки `iptables` работают во многих ситуациях, включая любое прямое подключение, особенно широкополосное, когда злоумышленник с гораздо большей вероятностью просканирует порты вашего компьютера. Вы также можете адаптировать эти настройки для маршрутизатора брандмауэра, используя цепочку `FORWARD` вместо `INPUT` и подсети источника и назначения, где это уместно. Для более сложных конфигураций вам может пригодиться инструмент настройки Shorewall.

В этом обсуждении мы лишь коснулись политики безопасности. Помните, что ее суть заключается в том, чтобы разрешать только то, что вы считаете приемлемым, а не пытаться найти и исключить все плохое. Кроме того, IP-брандмауэр — это

лишь часть общей картины безопасности (в следующей главе мы рассмотрим и другие).

9.26. Ethernet, IP, ARP и NDP

Рассмотрим еще одну важную деталь реализации IP через Ethernet. Напомним, что хост должен поместить IP-пакет внутрь фрейма Ethernet, чтобы передать его через физический уровень другому хосту. Помните также, что сами фреймы не содержат информации об IP-адресе — они используют MAC-адреса (аппаратные). Вопрос заключается в следующем: при построении фрейма Ethernet для IP-пакета как хост узнает, какой MAC-адрес соответствует IP-адресу пункта назначения?

Обычно мы не задумываемся над этим вопросом, потому что сетевое программное обеспечение включает автоматическую систему поиска MAC-адресов. В IPv4 это называется *протоколом определения адресов* (Address Resolution Protocol, ARP). Хост, использующий Ethernet в качестве физического уровня и IP в качестве сетевого уровня, поддерживает небольшую таблицу, называемую кэшем ARP, которая сопоставляет IP-адреса с MAC-адресами. В Linux кэш ARP находится в ядре. Чтобы просмотреть кэш ARP вашей системы, возьмите команду `ip neigh` (часть `neigh` будет иметь смысл, когда вы увидите эквивалент IPv6. Старая команда для работы с кэшем ARP — `arp`):

```
$ ip -4 neigh
10.1.2.57 dev enp0s31f6 lladdr 1c:f2:9a:1e:88:fb REACHABLE
10.1.2.141 dev enp0s31f6 lladdr 00:11:32:0d:ca:82 STALE
10.1.2.1 dev enp0s31f6 lladdr 24:05:88:00:ca:a5 REACHABLE
```

Мы используем параметр `-4`, чтобы ограничить вывод IPv4. В выводе видны IP-адреса и аппаратные адреса хостов, о которых известно ядру. Последнее поле указывает статус записи в кэше. `REACHABLE` означает, что общение с хостом произошло недавно, а `STALE` — что прошло какое-то время и запись должна быть обновлена.

Когда машина загружается, ее кэш ARP пуст. Так как же MAC-адреса попадают в кэш? Все начинается, когда машина хочет отправить пакет другому хосту. Если целевой IP-адрес отсутствует в кэше ARP, выполняются следующие действия:

1. Исходный хост создает специальный фрейм Ethernet, содержащий пакет запроса ARP для MAC-адреса, соответствующего целевому IP-адресу.
2. Исходный хост передает этот фрейм во всю физическую сеть для подсети цели.
3. Если один из прочих хостов в подсети знает правильный MAC-адрес, он создает ответный пакет и фрейм, содержащий адрес, и отправляет его исходному хосту. Чаще всего отвечающий хост является целевым, и он просто сообщает свой MAC-адрес.
4. Исходный хост добавляет пару «IP-адрес — MAC-адрес» в кэш ARP и может продолжать.

ПРИМЕЧАНИЕ

Помните, что ARP применяется только к системам в локальных подсетях. Чтобы добраться до пунктов назначения за пределами вашей подсети, ваш хост отправляет пакет на маршрутизатор, и дальнейшее перестает быть вашей проблемой. Конечно, вашему хосту все равно нужно знать MAC-адрес маршрутизатора, и он может использовать ARP для его поиска.

Единственная реальная проблема, с которой вы можете столкнуться в ходе работы с ARP, заключается в том, что кэш вашей системы может устареть, если вы перемещаете IP-адрес с одной карты сетевого интерфейса на другую, потому что карты имеют разные MAC-адреса (например, при тестировании машины). Системы Unix аннулируют записи кэша ARP, если через некоторое время не происходит никаких действий, поэтому не должно быть никаких проблем, кроме небольшой задержки для недействительных данных, но вы можете немедленно удалить запись кэша ARP с помощью команды

```
# ip neigh del host dev interface
```

На странице руководства `ip-neighbour(8)` объясняется, как вручную настроить записи кэша ARP, но вам этого делать не нужно. Обратите внимание на название страницы.

ПРИМЕЧАНИЕ

Не путайте ARP с обратным протоколом преобразования адресов (RARP). RARP преобразует MAC-адрес обратно в имя хоста или IP-адрес. До того как DHCP стал популярным, некоторые бездисковые рабочие станции и другие устройства использовали RARP для получения своей конфигурации, но сегодня он встречается довольно редко.

IPV6: NDP

Вам может быть интересно, почему команды, управляющие кэшем ARP, не содержат слова «arp» (или, если вы смутно знакомы с этой темой, можете задаться вопросом, почему мы не применяем `arp`). В IPv6 есть новый механизм — *протокол обнаружения соседей* (Neighbor Discovery Protocol, NDP), используемый в локальной сети канала. Команда `ip` объединяет ARP с IPv4 и UDP с IPv6. NDP включает в себя два вида сообщений:

- **запрос доступных соседей (Neighbor solicitation).** С его помощью получают информацию о локальном канале хоста, включая его аппаратный адрес;
- **перенаправление (Neighbor advertisement).** Используется для ответа на запрос доступных соседей.

Существует несколько других компонентов NDP, включая сообщения RA, которые мы рассматривали в разделе 9.20.

9.27. Беспроводная сеть Ethernet

В принципе, беспроводные сети Ethernet (Wi-Fi) незначительно отличаются от проводных. Как и любое проводное оборудование, они имеют MAC-адреса и используют фреймы Ethernet для передачи и приема данных, и в результате ядро Linux может взаимодействовать с беспроводным сетевым интерфейсом так же, как и с проводным. Все на сетевом уровне и выше одинаково, основные различия заключаются в дополнительных компонентах на физическом уровне, таких как частоты, сетевые идентификаторы и функции безопасности.

В отличие от проводного сетевого оборудования, которое очень хорошо автоматически приспосабливается к нюансам физической настройки, конфигурация беспроводной сети гораздо более вариативна. Для правильной работы беспроводного интерфейса Linux требуются дополнительные инструменты настройки.

Взглянем на дополнительные компоненты беспроводных сетей.

- **Детали передачи.** Это физические характеристики, например радиочастота.
- **Сетевая идентификация.** Поскольку несколько беспроводных сетей могут совместно использовать один и тот же базовый носитель, вы должны уметь различать их. Параметр SSID (Service Set Identifier, или сетевое имя) является идентификатором беспроводной сети.
- **Управление.** Хотя и возможно настроить беспроводную сеть так, чтобы хосты напрямую общались друг с другом, однако большинство беспроводных сетей управляются одной или несколькими *точками доступа*, через которые проходит весь трафик. Точки доступа часто являются мостами между беспроводной и проводной сетями, в результате чего обе они выглядят как одна сеть.
- **Аутентификация.** Возможно, вам захочется ограничить доступ к беспроводной сети. Для этого можно настроить точки доступа так, чтобы им требовался пароль или другой ключ аутентификации еще до того, как они начнут взаимодействовать с клиентом.
- **Шифрование.** В дополнение к ограничению начального доступа к беспроводной сети обычно требуется зашифровать весь трафик, передаваемый по радиоволнам.

Конфигурация Linux и утилиты, которые обрабатывают эти компоненты, распределены по нескольким областям. Некоторые из них находятся в ядре, Linux имеет набор беспроводных расширений, которые стандартизируют доступ пользователей к оборудованию. Что касается пользовательского пространства, то конфигурация беспроводной сети может усложниться, поэтому большинство людей предпочитают применять инструменты с графическим интерфейсом, такие как настольный апплет для NetworkManager. Тем не менее стоит взглянуть на то, что скрыто от глаз пользователя.

9.27.1. Утилита iw

Можно просматривать и изменять устройство пространства ядра и конфигурацию сети с помощью утилиты `iw`. Для работы с `iw` обычно необходимо знать имя сетевого интерфейса устройства, например `wlp1s0` (предсказуемое имя устройства) или `wlan0` (традиционное имя). Рассмотрим пример, который выводит данные сканирования доступных беспроводных сетей (если вы находитесь в городе, вывод информации будет большим):

```
# iw dev wlp1s0 scan
```

ПРИМЕЧАНИЕ

Для работы этой команды сетевой интерфейс должен быть включен (если это не так, запустите команду ifconfig wlp1s0 up), но поскольку он все еще находится на физическом уровне, не нужно настраивать параметры сетевого уровня, такие как IP-адрес.

Если сетевой интерфейс подключен к беспроводной сети, можете просмотреть сведения о сети следующим образом:

```
# iw dev wlp1s0 link
```

MAC-адрес в выводе этой команды относится к точке доступа, к которой вы в данный момент подключены.

ПРИМЕЧАНИЕ

Команда iw различает имена физических устройств (например, phy0) и сетевых интерфейсов (например, wlp1s0) и позволяет изменять различные настройки для каждого из них. Вы даже можете создать несколько сетевых интерфейсов для одного физического устройства. Однако почти всегда будете просто использовать имя сетевого интерфейса.

Примените команду `iw` для подключения сетевого интерфейса `wlp1s0` к незащищенной беспроводной сети *`network_name`* следующим образом:

```
# iw wlp1s0 connect network_name
```

Подключение к защищенным сетям — это совсем другая история. Для довольно небезопасной системы, использующей протокол защиты, эквивалентной проводной WEP (Wired Equivalent Privacy), можно задействовать параметр `keys` с командой `iw connect`. Однако применять WEP не следует, так как она небезопасна и немногие сети ее поддерживают.

9.27.2. Безопасность беспроводной сети

Для большинства настроек безопасности беспроводной сети Linux использует демон *wpa_supplicant*, управляющий как аутентификацией, так и шифрованием интерфейса беспроводной сети. Он может обрабатывать схемы аутентификации WPA2

и WPA3 (Wi-Fi Protected Access — защищенный доступ; не применяйте старые небезопасные WPA), а также практически любые методы шифрования, используемые в беспроводных сетях. При первом запуске демон считывает файл конфигурации (по умолчанию `/etc/wpa_supplicant.conf`) и пытается идентифицировать себя с точкой доступа и установить связь на основе заданного имени сети. Система хорошо задокументирована, в частности страница руководства `wpa_supplicant(8)` очень подробна и ее стоит изучить.

Запуск демона вручную каждый раз, когда вы хотите установить соединение, — это та еще работа. Да и простое создание файла конфигурации оказывается утомительным из-за большого количества возможных параметров. Что еще хуже, вся работа по запуску `iw` и wpa_supplicant просто позволяет вашей системе подключаться к беспроводной физической сети — даже не настраивается сетевой уровень. Именно в таком случае автоматические менеджеры конфигурации сети, такие как NetworkManager, значительно облегчают процесс. Хотя они ничего не делают самостоятельно, но знают правильную последовательность и необходимую конфигурацию для каждого шага, обеспечивающие работу беспроводной сети.

9.28. Выводы

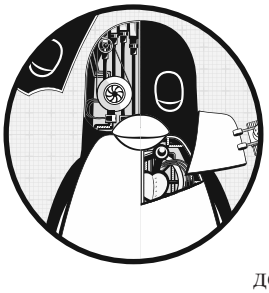
Знание позиций и ролей различных сетевых уровней имеет решающее значение для понимания того, как работает сеть Linux и как выполнять ее настройку. Хотя мы рассмотрели только основы, более сложные темы на физическом, сетевом и транспортном уровнях аналогичны тому, что вы видели в этой главе. Сами уровни часто также делятся из-за различных компонентов физического уровня в беспроводной сети.

Значительная часть действий, описанных в этой главе, выполняется в ядре с помощью некоторых базовых утилит управления пользовательским пространством для управления внутренними структурами данных ядра, такими как таблицы маршрутизации. Это традиционный способ работы с сетью. Однако, как и во многих других темах, обсуждаемых в этой книге, некоторые задачи не подходят для ядра из-за их сложности и необходимости обеспечивать гибкость, и именно здесь начинают превалировать утилиты пользовательского пространства. В частности, NetworkManager отслеживает и отправляет запросы в ядро, а затем управляет его конфигурацией. Другим примером является поддержка протоколов динамической маршрутизации, таких как BGP (Border Gateway Protocol — протокол пограничного шлюза), который применяется в крупных интернет-маршрутизаторах.

Но вам, вероятно, уже наскучила настройка сети. Давайте перейдем к ее использованию — на прикладной уровень.

10

Сетевые приложения и службы



В этой главе рассматриваются основные сетевые приложения — клиенты и серверы, работающие в пользовательском пространстве, которые находятся на прикладном уровне. Поскольку этот слой расположен в верхней части стека, ближе к конечным пользователям, понять этот материал проще, чем приведенный в главе 9, ведь они каждый день взаимодействуют с сетевыми клиентскими приложениями, такими как браузеры.

Для выполнения своей работы сетевые клиенты подключаются к соответствующим сетевым серверам. Сетевые серверы Unix бывают разными. Серверная программа может прослушивать порт самостоятельно или через дополнительный сервер. Мы рассмотрим некоторые распространенные серверы, а также инструменты, которые помогут вам понять и отладить работу сервера.

Сетевые клиенты задействуют протоколы и интерфейсы транспортного уровня операционной системы, поэтому важно понимать основы транспортных уровней TCP и UDP. Давайте изучим сетевые приложения, экспериментируя с сетевым клиентом, использующим TCP.

10.1. Основы работы служб

Понять службы TCP легче, чем многие другие, поскольку они основаны на простых, непрерывных, двусторонних потоках данных. Возможно, лучший способ увидеть, как они работают, — напрямую связаться с незашифрованным веб-сервером через

TCP-порт 80, чтобы получить представление о том, как данные проходят через это соединение. Например, для подключения к веб-серверу примера документации IANA выполните следующую команду:

```
$ telnet example.org 80
```

Вы получите подобный ответ, говорящий об успешном подключении к серверу:

```
Trying some address... some address – IP адрес сервера example.org
Connected to example.org.
Escape character is '^]'.
```

Теперь введите следующие две строки:

```
GET / HTTP/1.1
Host: example.org
```

ПРИМЕЧАНИЕ

HTTP 1.1, как и его предшественник HTTP 1.0, серьезно устарел — сейчас используются более новые протоколы, такие как HTTP/2, QUICK и разрабатываемый HTTP/3.

После последней строки дважды нажмите клавишу Enter. Сервер должен отправить большое количество HTML-текста в качестве ответа. Чтобы прервать соединение, нажмите сочетание клавиш Ctrl+D.

Это упражнение показывает:

- что удаленный хост имеет процесс веб-сервера, прослушивающий TCP-порт 80;
- `telnet` — это клиент, который инициировал соединение.

Причина, по которой вы должны прервать соединение с помощью комбинации клавиш Ctrl+D, заключается в том, что в противном случае соединение может оставаться открытым, поскольку для загрузки большинства веб-страниц требуется несколько запросов. Изучив веб-серверы на уровне протокола, вы не всегда сможете обнаружить такое поведение. Например, многие серверы быстро отключаются, если не получают запрос вскоре после открытия соединения.

ПРИМЕЧАНИЕ

telnet изначально предназначался для выполнения входа на удаленные хосты. Клиентской программы telnet может не оказаться в вашем дистрибутиве по умолчанию, но ее легко установить в качестве дополнительного пакета. Хотя вход на удаленный сервер с помощью telnet не совсем безопасен (это мы узнаем позже), клиент telnet может быть полезен для отладки удаленных служб. telnet не работает с UDP или любым другим транспортным уровнем, кроме TCP. Если вы ищете сетевой клиент общего назначения, рассмотрите netcat, описанный в разделе 10.5.3.

10.2. Взглянем глубже

В предыдущем примере мы вручную взаимодействовали с веб-сервером по сети с помощью `telnet`, задействуя протокол прикладного уровня HTTP. Хотя пользователи обычно для такого рода соединения задействуют веб-браузер, давайте отойдем от `telnet` и применим программу командной строки, которая знает, как обращаться к прикладному уровню HTTP. Мы используем утилиту `curl` со специальным параметром для записи сведений о выполненном взаимодействии в файл `trace_file`:

```
$ curl --trace-ascii trace_file http://www.example.org/
```

ПРИМЕЧАНИЕ

В вашем дистрибутиве может не быть предустановленного пакета curl, но проблем с его установкой нет.

Вы получите большой HTML-вывод. Проигнорируйте его или перенаправьте в `/dev/null` и вместо этого посмотрите на только что созданный файл `trace_file`. Если соединение было успешным, первая его часть должна выглядеть примерно так, как показано далее, в точке, где `curl` пытается установить TCP-соединение с сервером:

```
== Info:   Trying 93.184.216.34...
== Info: TCP_NODELAY set
== Info: Connected to www.example.org (93.184.216.34) port 80 (#0)
```

Все, что вы видели до сих пор, происходит на транспортном уровне или ниже. Однако если это соединение завершится успешно, `curl` затем попытается отправить запрос (header — заголовок) — именно здесь начинается прикладной уровень:

```
❶ => Send header, 79 bytes (0x4f)
❷ 0000: GET / HTTP/1.1
   0010: Host: www.example.org
   0027: User-Agent: curl/7.58.0
   0040: Accept: */*
   004d:
```

Строка ❶ — это вывод отладки `curl`, сообщающий, что команда будет делать дальше. Остальные строки показывают, что `curl` отправляет на сервер. Это текст, выделенный жирным шрифтом, шестнадцатеричные числа в начале — просто отладочные смещения, которые `curl` добавляет, чтобы помочь отслеживать, сколько данных было отправлено или получено.

В строке ❷ видно, что `curl` начинает с отправки команды `GET` серверу (как вы делали с `telnet`), за которой следует некая дополнительная информация для сервера и пустая строка. Затем сервер отправляет ответ, сначала с собственным заголовком, выделенным здесь жирным шрифтом:

```
<= Recv header, 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header, 22 bytes (0x16)
0000: Accept-Ranges: bytes
<= Recv header, 12 bytes (0xc)
0000: Age: 17629
--пропуск—
```

Как и в предыдущем выводе, строки `<=` являются выводом отладки, а `0000:` предшествует строкам вывода, чтобы сообщить вам смещения (в `curl` заголовок не учитывается в рамках смещения, вот почему все эти строки начинаются с 0).

Заголовок в ответе сервера может быть довольно длинным, но в какой-то момент сервер переходит от передачи заголовков к отправке фактического запрошенного документа, например:

```
   <= Recv header, 22 bytes (0x16)
   0000: Content-Length: 1256
   <= Recv header, 2 bytes (0x2)
❶  0000:
   <= Recv data, 1256 bytes (0x4e8)
   0000: <!doctype html>.<html>.<head>.     <title>Example Domain</title>.
   0040: .    <meta charset="utf-8" />.     <meta http-equiv="Content-type
   --пропуск--
```

Этот вывод иллюстрирует также важное свойство прикладного уровня. Несмотря на то что в выводе отладки говорится о заголовке `Recv header` и данных `Recv data`, причем подразумевается, что это два разных вида сообщений с сервера, нет никакой разницы в том, как команда `curl` общалась с операционной системой, чтобы получить эти два сообщения, и в том, как операционная система обрабатывала их, и в том, как сеть обрабатывала пакеты. Разница заключается в приложении `curl` пользовательского пространства. Команда `curl` знала, что до этого момента она получала заголовки, но когда получила пустую строку ❶, которая означает конец заголовков в HTTP, то все, что следует за ней, стала интерпретировать как запрошенный документ.

То же самое относится и к серверу, отправляющему эти данные. При отправке ответа операционная система сервера не проводила различий между данными заголовка и документа — различия заключаются внутри серверной программы пользовательского пространства.

10.3. Сетевые серверы

Большинство сетевых серверов похожи на другие серверные демоны в вашей системе, такие как cron, за исключением того, что они взаимодействуют с сетевыми портами. Фактически демон syslogd, который мы рассматривали в главе 7, принимает UDP-пакеты на порте 514 при запуске с параметром `-r`.

Перечислим другие распространенные сетевые серверы, которые могут быть запущены в вашей системе:

- **httpd**, **apache**, **apache2, nginx** — веб-серверы;
- **sshd** — демон безопасной оболочки;
- **postfix**, **qmail**, **sendmail** — почтовые серверы;
- **cupsd** — сервер печати;
- **nfsd**, **mountd** — демоны сетевой файловой системы (общий доступ к файлам);
- **smbd**, **nmbd** — демоны общего доступа к файлам Windows (см. главу 12);
- **rpcbind** — демон службы сопоставления портов удаленного вызова процедур (Remote Procedure Call, RPC).

Особенность большинства сетевых серверов заключается в том, что они обычно работают как несколько процессов. По крайней мере один процесс прослушивает сетевой порт, и при получении нового входящего соединения процесс прослушивания использует `fork()` для создания дочернего процесса, который затем отвечает за новое соединение. Дочерний процесс, часто называемый *рабочим* процессом, завершается при закрытии соединения. Тем временем исходный процесс продолжает прослушивать сетевой порт. Этот процесс позволяет серверу легко и без особых проблем обрабатывать множество подключений.

Однако в этой модели есть некоторые исключения. Вызов `fork()` добавляет значительный объем системных накладных расходов. Чтобы избежать этого, высокопроизводительные TCP-серверы, такие как веб-сервер Apache, могут при запуске создавать несколько рабочих процессов, чтобы по мере необходимости использовать их для обработки подключений. Серверам, которые принимают UDP-пакеты, вообще не нужно разветвляться, так как у них нет подключений для прослушивания — они просто получают данные и реагируют на них.

10.3.1. Служба защищенной оболочки SSH

Каждая программа сетевого сервера работает немного по-разному. Чтобы на практике изучить настройку и работу сервера, давайте внимательно рассмотрим автономный сервер защищенной оболочки (SSH). Одно из наиболее распространенных приложений сетевых служб, SSH является стандартом для удаленного доступа к машине Unix. SSH предназначен для обеспечения безопасного входа в оболочку, удаленного выполнения программ, простого обмена файлами и многого другого как замена старых небезопасных систем удаленного доступа `telnet` и `rlogin` криптографией с открытым ключом для аутентификации и более простыми шифрами для данных сеанса. Большинству интернет-провайдеров и облачных провайдеров SSH требуется для доступа к своим службам через оболочку, и многие сетевые устройства на базе Linux, такие как сетевые хранилища или устройства NAS, также предоставляют доступ через SSH. OpenSSH (www.openssh.com) — популярная

бесплатная реализация SSH для Unix, и почти все дистрибутивы Linux поставляются с ее предустановленной версией. Клиентская программа OpenSSH — это `ssh`, а сервер — `sshd`.

Существует две основные версии протокола SSH — 1 и 2. OpenSSH поддерживает только версию 2, отказавшись от поддержки версии 1 из-за уязвимостей и непопулярности.

Среди множества полезных возможностей и функций SSH реализует следующие:

- шифрует пароль и все другие данные сеанса, защищая систему от шпионов;
- туннелирует другие сетевые подключения, в том числе от клиентов X Window System (рассмотрим X в главе 14);
- имеет клиентов практически для любой операционной системы;
- использует ключи для аутентификации хоста.

ПРИМЕЧАНИЕ

Туннелирование (tunneling) — это процесс упаковки и транспортировки одного сетевого соединения внутри другого. Преимущества использования SSH для туннелирования соединений X Window System заключаются в том, что он настраивает среду отображения и шифрует данные X внутри туннеля.

У SSH есть несколько недостатков. Во-первых, для настройки SSH-соединения вам нужен открытый ключ удаленного хоста, а его получение не всегда безопасно (хотя вы можете проверить его вручную, чтобы убедиться, что вас не обманывают).

КРИПТОГРАФИЯ С ОТКРЫТЫМ КЛЮЧОМ

Мы использовали термин *«открытый ключ»* без объяснения, поэтому давайте вернемся назад и кратко обсудим его на случай, если вы с ним не знакомы. До 1970-х годов алгоритмы шифрования были *симметричными*, требуя, чтобы отправитель и получатель сообщения имели один и тот же ключ. Взлом кода сводился к краже ключа, и чем больше людей им владело, тем больше было возможностей для его взлома. Но в криптографии с открытым ключом есть два ключа: открытый и закрытый. Открытый ключ может зашифровать сообщение, но не расшифровать его, поэтому не имеет значения, у кого есть доступ к нему. Только *закрытый ключ* может расшифровать сообщение, зашифрованное с помощью открытого ключа. В большинстве случаев защитить закрытый ключ несложно, так как необходима лишь одна его копия, которую не стоит никому передавать.

Другим применением помимо шифрования является аутентификация: существуют способы проверить, что у кого-то есть закрытый ключ для данного открытого ключа без передачи каких-либо ключей.

Если хотите узнать, как работают некоторые методы криптографии, ознакомьтесь с книгой *Serious Cryptography: A Practical Introduction to Modern Encryption* (No Starch Press, 2017) Жан-Филиппа Омассона (Jean-Philippe Aumasson). Две подробные книги по SSH — это книги *SSH Mastery: OpenSSH, PuTTY, Tunnels, and Keys*, 2-е издание (Tilted Windmill Press, 2018), Майкла У. Лукаса (Michael W. Lucas) и *SSH, The Secure Shell: The Definitive Guide*, 2-е издание (O'Reilly, 2005), Дэниела Дж. Барретта, Ричарда Э. Сильвермана и Роберта Г. Бирнса (Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes).

10.3.2. Сервер sshd

Запуск сервера `sshd` для разрешения удаленных подключений к вашей системе требует файла конфигурации и ключей хоста. Большинство дистрибутивов хранят конфигурации в каталоге конфигурации `/etc/ssh` и пытаются настроить все для пользователя, если установить пакет `sshd`. (Файл конфигурации сервера `sshd_config` легко спутать с файлом настройки клиента `ssh_config`, поэтому будьте осторожны.)

Вам не нужно ничего менять в `sshd_config`, но можно его проверить. Файл состоит из пар «ключ — значение», как показано в этом фрагменте:

```
Port 22
#AddressFamily any

#ListenAddress 0.0.0.0
#ListenAddress ::
#HostKey /etc/ssh/ssh_host_rsa_key
#HostKey /etc/ssh/ssh_host_ecdsa_key
#HostKey /etc/ssh/ssh_host_ed25519_key
```

Строки, начинающиеся с символа `#`, — это комментарии, в файле `sshd_config` многие комментарии указывают значения по умолчанию для различных параметров, как видно из примера. Страница руководства `sshd_config(5)` содержит описания параметров и возможных значений, следующие — одни из наиболее важных:

- `HostKey` *`file`* — использует *`file`* в качестве ключа хоста (ключи хоста описаны далее).
- `PermitRootLogin` *`value`* — разрешает суперпользователю входить в систему по SSH, если *`value`* установлено в значение `yes`. Установите *`value`* в значение `no`, чтобы предотвратить это.
- `LogLevel` *`level`* — регистрирует сообщения с уровнем системного журнала *`level`* (по умолчанию берется значение `INFO`).
- `SyslogFacility` *`name`* — регистрирует сообщения с именем объекта системного журнала *`name`* (по умолчанию используется значение `AUTH`).
- `X11Forwarding` *`value`* — включает туннелирование клиента X Window System, если *`value`* установлено в значение `yes`.

- `XAuthLocation` *`path`* — указывает расположение утилиты `xauth` в вашей системе. X-туннелирование не будет работать без этого пути. Если `xauth` отсутствует в `/usr/bin`, задайте путь *`path`* к полному имени пути для `xauth`.

Создание ключей хоста

OpenSSH имеет несколько наборов ключей хоста. Каждый набор имеет открытый ключ (с расширением файла `.pub`) и закрытый ключ (без расширения).

ВНИМАНИЕ

Не позволяйте никому увидеть закрытый ключ даже в вашей собственной системе: вы рискуете, что злоумышленники, получив его, смогут проникнуть в систему.

SSH версии 2 имеет ключи RSA и DSA. RSA и DSA — это алгоритмы криптографии с открытым ключом. Имена файлов ключей приведены в табл. 10.1.

Таблица 10.1. Файлы ключей OpenSSH

Имя файла	Тип ключа
ssh_host_rsa_key	Закрытый ключ RSA
ssh_host_rsa_key.pub	Открытый ключ RSA
ssh_host_dsa_key	Закрытый ключ DSA
ssh_host_dsa_key.pub	Открытый ключ DSA

Создание ключа включает в себя численное вычисление, которое генерирует как открытые, так и закрытые ключи. Вам не нужно создавать ключи, потому что программа установки OpenSSH или сценарий установки вашего дистрибутива справятся с этим сами, но вам нужно знать, как это сделать, если вы планируете использовать такие программы, как `ssh-agent`, которые предоставляют услуги аутентификации без пароля. Чтобы создать ключи протокола SSH версии 2, воспользуйтесь программой `ssh-keygen`, которая поставляется с 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
```

Сервер SSH и клиенты используют также файл ключей, называемый `ssh_known_hosts`, для хранения открытых ключей от других хостов. Если вы собираетесь применять аутентификацию на основе удостоверения удаленного клиента, файл `ssh_known_hosts` сервера должен содержать открытые ключи хоста всех доверенных клиентов. Знать о файле ключей удобно, если вы заменяете компьютер. При установке новой системы с нуля можете импортировать файлы ключей со старой машины, чтобы гарантировать, что у пользователей не будет несоответствия ключей при подключении к новой системе.

Запуск SSH-сервера

Хотя большинство дистрибутивов поставляются с SSH, они не запускают сервер `sshd` по умолчанию. В Ubuntu и Debian SSH-сервер не устанавливается в новой системе — установка его пакета создает ключи, запускает сервер и добавляет запуск сервера в конфигурацию загрузки.

В Fedora `sshd` установлен по умолчанию, но отключен. Чтобы запустить `sshd` при загрузке, используйте команду `systemctl` следующим образом:

```
# systemctl enable sshd
```

Если вы хотите запустить сервер немедленно, без перезагрузки, задействуйте команду

```
# systemctl start sshd
```

Fedora обычно создает все отсутствующие файлы ключей хоста при первом запуске `sshd`.

Если используется другой дистрибутив, вам, скорее всего, не потребуется вручную настраивать запуск `sshd`. Однако необходимо знать, что существует два режима запуска: автономный и по требованию. Автономный сервер распространен гораздо шире, и запускается он от имени суперпользователя. Серверный процесс `sshd` записывает свой `PID` в `/var/run/sshd.pid` (конечно, при запуске `systemd` отслеживается также его `cgroup`, как мы видели в главе 6).

В качестве альтернативы `systemd` может запускать `sshd` по требованию через юнит сокета. Это не очень хорошая идея, так как серверу иногда требуется создавать файлы ключей и этот процесс может занять много времени.

10.3.3. Утилита fail2ban

Если вы настроите SSH-сервер на своем компьютере и откроете его для доступа в интернет, очень скоро начнутся постоянные попытки вторжения. Эти атаки «грубой силой» (с полным перебором вариантов ключей) не увенчаются успехом, если система правильно настроена и пароли достаточно сложны. Однако они будут раздражать, потреблять процессорное время и излишне загромождать журналы данными.

Чтобы это предотвратить, необходимо настроить механизм блокировки повторных попыток входа в систему. На момент написания книги наиболее популярным способом является пакет `fail2ban` — скрипт, который просматривает сообщения журнала. Увидев определенное количество неудачных запросов от одного хоста в течение определенного времени, `fail2ban` использует `iptables` для создания правила, запрещающего трафик с этого хоста. По истечении указанного периода, в течение которого хост, вероятно, откажется от попыток подключения, `fail2ban` удаляет правило.

Большинство дистрибутивов Linux предлагают пакет `fail2ban` с предварительно настроенными значениями по умолчанию для SSH.

10.3.4. SSH-клиент

Чтобы войти на удаленный хост, запустите следующую команду:

```
$ ssh remote_username@remote_host
```

Вы можете не указывать имя пользователя на удаленном хосте *`remote_username@`*, если ваше локальное имя пользователя совпадает с именем пользователя на удаленном хосте *`remote_host`*. Также можете запускать конвейеры командой `ssh`, как показано в следующем примере, где каталог `dir` копируется на другой хост:

```
$ tar zcvf - dir | ssh remote_host tar zxvf -
```

Файл конфигурации глобального SSH-клиента `ssh_config` должен находиться в `/etc/ssh`, в том же месте, что и ваш файл `sshd_config`. Как и в файле конфигурации сервера, в файле конфигурации клиента есть пары «ключ — значение», но их не нужно изменять.

Наиболее часто проблема с использованием SSH-клиентов возникает, когда открытый ключ SSH в вашем локальном файле `ssh_known_hosts` или `.ssh/known_hosts` не совпадает с ключом на удаленном хосте. Неверные ключи вызывают ошибки или предупреждения, подобные следующему:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ 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.
❶ Offending key in /home/user/.ssh/known_hosts:12
RSA host key for host has changed and you have requested
strict checking.
Host key verification failed.
```

Это означает, что администратор удаленного хоста изменил ключи (что часто происходит при обновлении оборудования или облачного сервера), но лучше, чтобы он сам это подтвердил. В любом случае, в сообщении говорится, что неверный ключ находится в строке 12 пользовательского файла `known_hosts` ❶.

Если все в порядке, просто удалите неправильную строку или замените ее правильным открытым ключом.

Клиенты передачи файлов SSH

OpenSSH включает в себя программы передачи файлов `scp` и `sftp`, которые предназначены для замены старых небезопасных программ `rcp` и `ftp`. Вы можете использовать `scp` для передачи файлов на удаленную машину, или с удаленной машины на свою, или с одного хоста на другой. Эта команда работает так же, как и команда `cp`. Рассмотрим несколько примеров.

Скопируйте файл с удаленного хоста в текущий каталог:

```
$ scp user@host:file .
```

Скопируйте файл с локального компьютера на удаленный хост:

```
$ scp file user@host:dir
```

Скопируйте файл с одного удаленного хоста на другой удаленный хост:

```
$ scp user1@host1:file user2@host2:dir
```

Утилита `sftp` работает как устаревший ftp-клиент командной строки, используя команды `get` и `put`. На удаленном хосте должна быть установлена программа `sftp`-сервера, наличия которой можно ожидать, если удаленный хост также применяет OpenSSH.

ПРИМЕЧАНИЕ

Если вам нужно больше возможностей и требуется бóльшая гибкость, чем предлагают scp и sftp (например, вы часто передаете большое количество файлов), ознакомьтесь с утилитой rsync, описанной в главе 12.

SSH-клиенты для платформ, отличных от Unix

Существуют SSH-клиенты для всех популярных операционных систем.

Какой из них лучше выбрать? PuTTY — это хороший базовый клиент Windows, который включает в себя безопасную программу для копирования файлов. MacOS основана на Unix и включает в себя OpenSSH.

10.4. Сетевые серверы до systemd: inetd/xinetd

До широкого использования `systemd` и сокет-юнитов, которые мы рассматривали в подразделе 6.3.7, существовало несколько серверов, которые предоставляли стандартные средства построения сетевой службы. Многие второстепенные сетевые службы очень схожи по своим требованиям к соединению, поэтому внедрение автономных серверов для каждой службы может оказаться неэффективным. Каждый сервер должен быть настроен отдельно для обработки прослушивания портов, контроля доступа и настройки портов.

Эти действия выполняются одинаково для большинства служб; лишь когда сервер принимает соединение, связь обрабатывается по-разному.

Одним из традиционных способов упрощения использования серверов является демон `inetd` — своего рода *суперсервер*, предназначенный для стандартизации доступа к сетевым портам и интерфейсов между серверными программами и сетевыми портами. После запуска `inetd` считывает свой файл конфигурации, а затем прослушивает сетевые порты, определенные в этом файле. По мере появления новых сетевых соединений `inetd` подключает к соединению только что запущенный процесс.

Более новая версия `inetd` под названием `xinetd` предлагает более простую конфигурацию и улучшенный контроль доступа. К сожалению, проект `xinetd` почти полностью был свернут в пользу `systemd`, однако вы можете встретить его в более старых системах или в тех, которые не применяют `systemd`.

ОБЕРТКИ TCP: TCPD, /ETC/HOSTS.ALLOW И /ETC/HOSTS.DENY

До того как низкоуровневые брандмауэры, такие как `iptables`, стали популярными, многие администраторы использовали библиотеку-обертку *TCP wrapper* и демон для управления доступом к сетевым службам. В этих реализациях `inetd` запускает программу `tcpd`, которая сначала просматривает входящее соединение, а также списки управления доступом в файлах `/etc/hosts.allow` и `/etc/hosts.deny`.

Программа `tcpd` регистрирует соединение, и если она решает, что с входящим соединением все в порядке, то передает его в программу финальной службы. Вы можете столкнуться с системами, которые все еще используют систему-обертку TCP, но мы не будем подробно ее описывать, потому что она в основном вышла из употребления.

10.5. Средства диагностики

Рассмотрим несколько инструментов диагностики, которые полезны для изучения *прикладного уровня*. Некоторые проникают на транспортные и сетевые уровни, потому что все находящееся на прикладном уровне в итоге сопоставляется с чем-то с более нижних уровней.

Как обсуждалось в главе 9, `netstat` — это базовый инструмент отладки сетевых служб, который может отображать ряд статистических данных транспортного и сетевого уровней. В табл. 10.2 рассматриваются несколько полезных параметров для просмотра соединений.

Таблица 10.2. Полезные параметры отчетов о соединениях для netstat

Параметр	Описание
`-t`	Выводит информацию о TCP-порте
`-u`	Выводит информацию о порте UDP
`-l`	Выводит прослушиваемые порты
`-a`	Выводит информацию о каждом активном порте
`-n`	Отключает поиск имен (ускоряет процесс, полезен, если DNS не работает)
`-4, -6`	Ограничивает вывод до IP версии 4 или 6

10.5.1. Утилита lsof

В главе 8 мы узнали, что утилита `lsof` может не только отслеживать открытые файлы, но и составлять список программ, которые в настоящее время используют или прослушивают порты. Для получения полного списка таких программ запустите команду

```
# lsof -i
```

Когда вы запускаете эту команду от имени обычного пользователя, она показывает только его процессы. А когда запускаете ее от имени суперпользователя, вывод будет выглядеть примерно так, отображая различные процессы и пользователей:

```
COMMAND     PID  USER    FD  TYPE   DEVICE SIZE/OFF NODE NAME
rpcbind     700  root    6u  IPv4    10492      0t0  UDP *:sunrpc ❶
rpcbind     700  root    8u  IPv4    10508      0t0  TCP *:sunrpc (LISTEN)
avahi-dae   872  avahi  13u  IPv4 21736375      0t0  UDP *:mdns ❷
cupsd      1010  root    9u  IPv6 42321174      0t0  TCP ip6-localhost:ipp (LISTEN) ❸
ssh       14366  juser   3u  IPv4 38995911      0t0  TCP thishost.local:55457-> ❹
    somehost.example.com:ssh (ESTABLISHED)
chromium- 26534 juser   8r  IPv4 42525253      0t0  TCP thishost.local:41551-> ❺
    anotherhost.example.com:https (ESTABLISHED)
```

В этом примере показаны идентификаторы пользователей и процессов для серверных и клиентских программ от служб RPC старого образца вверху ❶ до многоадресной службы DNS, предоставляемой `avahi` ❷, и даже службы печати с поддержкой IPv6, `cupsd` ❸. Последние две строки показывают клиентские соединения: SSH-соединение ❹ и безопасное веб-соединение из веб-браузера Chromium ❺. Поскольку вывод команды может быть обширным, стоит применять фильтр, как описано в следующем разделе.

Утилита `lsof` похожа на `netstat` тем, что она пытается преобразовать каждый IP-адрес, который находит, в имя хоста, что замедляет вывод. Используйте параметр `-n`, чтобы отключить разрешение имен:

```
# lsof -n -i
```

Вы также можете указать параметр `-P`, чтобы отключить поиск имени портов в `/etc/services`.

Фильтрация по протоколу и порту

Если вы ищете определенный порт (скажем, вам известно, что процесс использует определенный порт, и вы хотите знать, что это за процесс), примените команду

```
# lsof -i:port
```

Полный синтаксис выглядит следующим образом:

```
# lsof -iprotocol@host:port
```

Параметры `protocol`, `@host` и `:port` необязательные и будут соответствующим образом фильтровать вывод `lsof`. Как и в большинстве сетевых утилит, `host` и `port` могут быть либо именами, либо номерами. Например, если вы хотите видеть соединения только на TCP-порте 443 (порт HTTPS), используйте:

```
# lsof -iTCP:443
```

Для фильтрации на основе версии IP возьмите `-i4` (IPv4) или `-i6` (IPv6). Можно задействовать их как отдельный параметр или просто добавить номер с более сложными фильтрами (например, `-i6 TCP:443`).

Можно также указать имена служб из `/etc/services` (как в `-tcp:ssh`) вместо номеров.

Фильтрация по состоянию соединения

Одним из особенно удобных фильтров `lsof` является состояние соединения. Например, чтобы отобразить только процессы, прослушивающие TCP-порты, введите:

```
# lsof -iTCP -sTCP:LISTEN
```

Эта команда дает хороший обзор процессов сетевого сервера, запущенных в настоящее время в системе. Но поскольку UDP-серверы не прослушивают и не имеют подключений, вам придется использовать параметр `-iUDP` для просмотра запущенных клиентов, а также серверов. Обычно это не проблема, потому что в системе не так уж много UDP-серверов.

10.5.2. Утилита tcpdump

Система обычно не беспокоится о сетевом трафике, который не адресован ни одному из ее MAC-адресов. Если нужно точно увидеть, что пересекает вашу сеть, `tcpdump` переводит карту сетевого интерфейса в *«неразборчивый» режим* и сообщает о каждом встречающемся пакете. Команда `tcpdump` без аргументов приводит к выводу, как в примере далее, который включает 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
```

Чтобы добавить конкретики в вывод команды `tcpdump`, используйте фильтры. Можно фильтровать на основе хостов источника и назначения, сетей, адресов Ethernet, протоколов на многих уровнях сетевой модели и многого другого. Среди множества пакетных протоколов, которые распознает `tcpdump`, есть ARP, RARP, ICMP, TCP, UDP, IP, IPv6, пакеты AppleTalk и IPX. Например, чтобы указать `tcpdump` выводить только TCP-пакеты, выполните следующую команду:

```
# tcpdump tcp
```

Чтобы просмотреть веб-пакеты и UDP-пакеты, введите

```
# tcpdump udp or port 80 or port 443
```

Ключевое слово `or` указывает, что условие слева или справа может быть истинным для прохождения фильтра. Аналогично, ключевое слово `and` требует, чтобы оба условия были истинными.

> **ПРИМЕЧАНИЕ**
>
> Если хотите часто просматривать пакеты, используйте графический интерфейс, альтернативный tcpdump, например Wireshark.

Примитивы

В предыдущих примерах `tcp`, `udp` и `port 80` — это базовые элементы фильтров, называемые *примитивами*. Наиболее важные примитивы перечислены в табл. 10.3.

Таблица 10.3. Примитивы tcpdump

Примитив	Описание
tcp	TCP-пакеты
udp	UDP-пакеты
ip	IPv4-пакеты
ip6	IPv6-пакеты
port *port* (порт)	TCP- и/или UDP-пакеты, направляемые в порт *port* или из него
host *host* (хост)	Пакеты на хост *host* или с него
net *network* (сеть)	Пакеты в сеть *network* или из нее

Операторы

Ключевое слово `or`, использованное ранее, является *оператором*. В команде `tcpdump` можно указать несколько операторов (например, `and` и `!`), которые группируют в скобках. Если вы планируете серьезную работу с командой `tcpdump`, обязательно прочитайте страницу руководства `pcap-filter(7)`, особенно раздел, в котором описываются примитивы.

ПРИМЕЧАНИЕ

Будьте осторожны при использовании команды tcpdump. Вывод tcpdump, показанный ранее в этом разделе, включает только информацию о заголовке TCP-пакета (транспортный уровень) и IP (уровень интернета), но можно заставить tcpdump вывести все содержимое пакета. Несмотря на то что наиболее важный сетевой трафик теперь зашифрован по протоколу TLS, вы не должны шпионить в сетях, если не являетесь их владельцем или не имеете на это разрешения.

10.5.3. Утилита netcat

Если вам требуется бóльшая гибкость при подключении к удаленному хосту, чем дает команда `telnet host port`, используйте команду `netcat` (или `nc`). Утилита `netcat` может подключаться к удаленным портам TCP/UDP, указывать локальный порт, прослушивать порты, сканировать их, перенаправлять стандартный ввод-вывод в сетевые соединения и из них и многое другое. Чтобы открыть TCP-соединение с портом с помощью `netcat`, запустите

```
$ netcat host port
```

Команда `netcat` завершается, когда другая сторона разрывает соединение. Это может привести к путанице, если перенаправить стандартный ввод в `netcat`, так как вы можете не получить приглашение командной строки обратно после отправки

данных (в отличие от почти любого другого конвейера команд). Можете прервать соединение в любое время, нажав сочетание клавиш Ctrl+C. (Если хотите, чтобы программа и сетевое подключение завершились на основе стандартного входного потока, попробуйте использовать вместо этого утилиту `sock`.)

Чтобы прослушать определенный порт, запустите

```
$ netcat -l port_number
```

Если команда `netcat` успешно прослушивает порт, она будет ждать соединения, а после его установления выведет выходные данные соединения и отправит в него все со стандартного ввода.

Дополнительные особенности `netcat`:

- По умолчанию вывод данных отладки невелик. Если что-то не удается, `netcat` молча завершает работу, но устанавливает соответствующий код выхода. Если вам нужна дополнительная информация, добавьте параметр `-v` (verbose — «подробный»).
- По умолчанию клиент `netcat` пытается подключиться с помощью IPv4 и IPv6. Однако в режиме сервера `netcat` по умолчанию использует IPv4. Чтобы принудительно включить протокол, примените `-4` для IPv4 и `-6` для IPv6.
- Параметр `-u` указывает UDP вместо TCP.

10.5.4. Сканирование портов

Иногда пользователь может не знать, какие службы предлагают компьютеры в его сетях или даже какие IP-адреса задействуются. Программа Network Mapper (Nmap) сканирует все порты на компьютере или сети компьютеров в поисках открытых портов и перечисляет найденные. В большинстве дистрибутивов есть пакет Nmap, также его можно загрузить по адресу www.insecure.org. (Информация о возможностях Nmap собрана на странице руководства Nmap и в онлайн-ресурсах.)

При перечислении портов на компьютере полезно запустить сканирование Nmap по крайней мере из двух точек: с собственного компьютера и с другого, возможно, за пределами вашей локальной сети. Это даст представление о том, что блокирует ваш брандмауэр.

> **ВНИМАНИЕ**
>
> Если кто-то другой контролирует сеть, которую вы хотите сканировать с помощью Nmap, попросите разрешения. Сетевые администраторы следят за сканированием портов и обычно отключают доступ к машинам, с которых оно выполняется.

Запустите `nmap` *`host`* (*`host`* — адрес хоста), чтобы выполнить обычное сканирование на хосте, например:

```
$ 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
```

В примере открыт ряд служб, многие из которых по умолчанию не включены в большинстве дистрибутивов. В данном случае единственная служба, которая обычно включена по умолчанию, — это порт 111, порт `rpcbind`.

Nmap может сканировать порты по протоколу IPv6, если вы добавите параметр `-6`. Это удобный способ определения служб, которые не поддерживают IPv6.

10.6. Удаленный вызов процедур

А что же насчет службы rpcbind из вывода сканирования из предыдущего раздела? *RPC* (remote procedure call) расшифровывается как *удаленный вызов процедур*, это система, расположенная в нижних частях прикладного уровня. Он разработан для того, чтобы программистам было проще создавать сетевые приложения «клиент — сервер», в которых клиентская программа вызывает функции, выполняемые на удаленном сервере. Каждый тип программы удаленного сервера идентифицируется присвоенным номером программы.

Реализации RPC используют транспортные протоколы, такие как TCP и UDP, и им требуется специальная промежуточная служба для сопоставления номеров программ с портами TCP и UDP. Сервер называется rpcbind, он должен быть запущен на любой машине, которая задействует службы RPC.

Чтобы узнать, какие службы RPC есть на вашем компьютере, запустите следующую команду:

```
$ rpcinfo -p localhost
```

RPC — один из протоколов, которые до сих пор не выходят из употребления. Протокол сетевого доступа к файловым системам (Network File System, NFS) и информационная служба сети (Network Information Service, NIS) используют RPC, но они совершенно не нужны на автономных компьютерах. Однако всякий раз, когда кажется, что в rpcbind больше нет необходимости, появляется что-то новое, например поддержка монитора доступа к файлам (FAM) в GNOME.

10.7. Безопасность сети

Поскольку Linux — это очень популярный вариант Unix для ПК, и особенно потому, что он широко применяется для веб-серверов, он привлекает множество неприятных пользователей, которые пытаются проникнуть в компьютерные системы. В разделе 9.25 уже обсуждались брандмауэры, но это далеко не все.

Сетевая безопасность привлекает экстремистов — тех, кому нравится взламывать системы ради удовольствия или денег, и тех, кто придумывает сложные схемы защиты и действительно любит отваживать злоумышленников, пытающихся проникнуть в их системы (это тоже может быть очень выгодно). К счастью, нет необходимости знать всю тему, чтобы обеспечить безопасность своей системы. Вот несколько основных правил:

- **Запускайте как можно меньше служб.** Злоумышленники не могут проникнуть в службы, которых не существует в вашей системе. Если вы знаете, что делает служба, но не пользуетесь ею, не включайте ее просто так, по единственной причине, что можете захотеть применить ее когда-нибудь.
- **Блокируйте как можно больше служб с помощью брандмауэра.** Системы Unix имеют ряд внутренних служб, о которых вы можете не знать (например, TCP-порт 111 для сервера сопоставления портов RPC), и ни одна система в мире не должна знать о них. Может быть довольно сложно отслеживать и регулировать службы в вашей системе, потому что различные виды программ прослушивают различные порты. Чтобы злоумышленники не могли обнаружить внутренние службы в системе, используйте эффективные правила брандмауэра и установите последний на своем маршрутизаторе.
- **Отслеживайте службы, которыми вы делитесь в интернете.** Если применяете SSH-сервер, Postfix или аналогичные службы, обновляйте программное обеспечение и получайте соответствующие предупреждения о безопасности. (Некоторые онлайн-ресурсы перечислены в подразделе 10.7.2.)
- **Используйте выпуски дистрибутива с долгосрочной поддержкой (long-term support, LTS) для серверов.** Организации разработчиков систем безопасности обычно сосредоточиваются на стабильных поддерживаемых выпусках дистрибутивов. Версиям систем для разработки и тестирования, таким как Debian Unstable и Fedora Rawhide, уделяется гораздо меньше внимания в плане безопасности.
- **Не предоставляйте учетную запись в своей системе тому, кому она не нужна.** Гораздо проще получить доступ суперпользователя с локальной учетной записи, чем проникнуть в нее удаленно. На самом деле, учитывая огромную базу программного обеспечения (и вытекающие из этого ошибки и недостатки дизайна), доступную в большинстве систем, может быть легко получить доступ суперпользователя к системе, после того как вы получите приглашение оболочки. Не думайте, что ваши друзья знают, как защитить свои пароли, или в первую очередь выбирают хорошие пароли.

- **Избегайте установки сомнительных двоичных пакетов.** Они могут содержать вирусы — троянских коней.

Это практическая часть защиты своей системы. Но почему так важно это сделать? Существует три основных вида сетевых атак, которые могут быть направлены на компьютер с системой Linux:

- **Полная компрометация (Full compromise).** Это означает получение доступа суперпользователя (полный контроль) к машине. Злоумышленник может этим воспользоваться и атаковать службу, например, с помощью эксплойта переполнения буфера, или завладеть слабо защищенной учетной записью пользователя, а затем попытаться применить плохо написанную программу `setuid`.
- **Атака с отказом в обслуживании (Denial-of-service, DoS).** Это не позволяет машине выполнять свои сетевые службы или заставляет компьютер работать со сбоями каким-то другим способом без использования какого-либо специального доступа. Обычно DoS-атака — это просто поток сетевых запросов, но она может быть также использованием ошибки в серверной программе, которая вызывает сбой. Эти атаки труднее предотвратить, но на них легче реагировать.
- **Вредоносные программы (Malware).** Пользователи Linux в основном защищены от вредоносных программ, таких как почтовые черви и вирусы, просто потому что их почтовые клиенты не настолько глупы, чтобы запускать программы, отправленные во вложениях сообщений. Но вредоносное ПО Linux существует. Избегайте загрузки и установки исполняемого программного обеспечения из небезопасных мест.

10.7.1. Типичные уязвимости

Существует два основных типа уязвимостей: прямые атаки и перехват паролей в открытом виде. В ходе *прямых атак* пытаются просто захватить машину, особо не хитря. Одним из наиболее распространенных способов является обнаружение в вашей системе незащищенной или уязвимой службы. Простой пример — служба, которая не прошла проверку подлинности по умолчанию, например учетная запись администратора без пароля. Как только злоумышленник получает доступ к одной службе в системе, он может использовать ее, чтобы попытаться скомпрометировать всю систему. Раньше обычной прямой атакой был эксплойт переполнения буфера, когда неосторожный программист не проверил границы буферного массива. Рандомизация размещения адресного пространства (Address Space Layout Randomization, ASLR) смягчила последствия данной атаки в ядре и других местах.

При *похищении пароля в открытом виде* захватывают пароли, отправленные в виде открытого текста, или используют базу данных паролей, которая пополняется благодаря утечкам данных. Как только ваш пароль станет известен злоумышленнику, игра закончится. Он неизбежно попробует получить доступ суперпользователя локально (как упоминалось ранее, это намного проще, чем удаленная атака), попытается задействовать машину в качестве посредника для атаки на другие хосты или и то и другое сразу.

ПРИМЕЧАНИЕ

Если вам нужно запустить службу, в которой не предусмотрена встроенная поддержка шифрования, используйте утилиту Stunnel (www.stunnel.org) — пакет-оболочка шифрования, очень похожий на TCP-оболочки. Stunnel особенно хорош для создания оболочки служб, которые активируются с помощью сокет-юнитов systemd или inetd.

Некоторые службы постоянно становятся объектами атак из-за плохих реализации и дизайна. Вы должны деактивировать следующие службы (на данный момент все они довольно сильно устарели и в большинстве систем редко активируются по умолчанию):

- **ftpd.** По какой-то причине все FTP-серверы страдают от уязвимостей. Кроме того, большинство FTP-серверов используют пароли в открытом виде. Если вам нужно переместить файлы с одной машины на другую, задействуйте решение SSH или `rsync`;
- **telnetd**, **rlogind**, **rexecd.** Все эти службы передают данные удаленного сеанса (включая пароли) в виде открытого текста. Избегайте их, если у вас нет версии с поддержкой Kerberos.

10.7.2. Ресурсы безопасности

Перечислим три хороших ресурса, посвященных вопросам безопасности.

- Организация SANS Institute (www.sans.org) предлагает обучение, сервисы, бесплатную еженедельную новостную рассылку с перечнем основных текущих уязвимостей, примеры политик безопасности и многое другое.
- Отдел сертификации Института разработки программного обеспечения Университета Карнеги — Меллона (www.cert.org) — хорошее место для поиска информации о наиболее серьезных проблемах.
- Insecure.org, проект хакера и создателя Nmap Гордона «Fyodor» Лиона (www.insecure.org) — здесь можно загрузить службу Nmap и найти указатели на всевозможные инструменты тестирования сетевых эксплойтов. Он гораздо более открыт и конкретен в отношении эксплойтов, чем многие другие сайты.

Если вас интересует сетевая безопасность, вам следует узнать все о безопасности транспортного уровня (Transport Layer Security, TLS) и его предшественника, уровня защищенных сокетов (Secure Socket Layer, SSL). Эти сетевые уровни пользовательского пространства обычно добавляют к сетевым клиентам и серверам для поддержки сетевых транзакций с помощью шифрования с открытым ключом и сертификатов. Хорошим руководством по безопасности является книга Дэвиса (Davies) *Implementing SSL/TLS Using Cryptography and PKI* (Wiley, 2011) и работа Жан-Филиппа Омассона (Jean-Philippe Aumasson) *Serious Cryptography: A Practical Introduction to Modern Encryption* (No Starch Press, 2017).

10.8. Что дальше?

Если вы хотите углубиться в изучение сложных сетевых серверов, то начните с наиболее распространенных из них — веб-серверов Apache или nginx и почтового сервера Postfix. В частности, веб-серверы просты в установке, и большинство дистрибутивов предоставляют соответствующие пакеты. Если ваша машина находится за брандмауэром или маршрутизатором с поддержкой NAT, можете экспериментировать с конфигурацией сколько угодно, не беспокоясь о безопасности.

В последних нескольких главах мы постепенно переходили из пространства ядра в пространство пользователя. Лишь несколько утилит, обсуждаемых в этой главе, например `tcpdump`, взаимодействуют с ядром. В оставшейся части главы описывается, как сокеты преодолевают разрыв между транспортным уровнем ядра и прикладным уровнем пользовательского пространства. Это более сложный материал, представляющий особый интерес для программистов, поэтому можете сразу перейти к следующей главе.

10.9. Сетевые сокеты

Рассмотрим, как процессы выполняют работу считывания данных и записи данных в сеть. Для процессов достаточно просто считывать и записывать данные в уже настроенные сетевые соединения: все, что вам нужно, — это несколько системных вызовов, о которых вы можете прочитать на страницах руководства `recv(2)` и `send(2)`. С точки зрения процесса самое важное, что нужно знать, — как получить доступ к сети при использовании этих системных вызовов. В системах Unix процесс задействует сокет для определения того, когда и как он взаимодействует с сетью. *Сокеты* (sockets) — это интерфейс, с помощью которого процессы получают доступ к сети через ядро, они представляют собой границу между пространством пользователя и пространством ядра. Их часто применяют для межпроцессного взаимодействия (Inter Process Communication, IPC).

Существуют разные типы сокетов, потому что процессы должны получать доступ к сети разными способами. Например, TCP-соединения представлены потоковыми сокетами (`SOCK_STREAM` с точки зрения программиста), а UDP-соединения — сокетами датаграмм (`SOCK_DGRAM`).

Настройка сетевого сокета может быть довольно сложной, поскольку в определенное время вам необходимо учитывать тип сокета, IP-адреса, порты и транспортный протокол. Однако после того, как все начальные детали будут уточнены, серверы начнут использовать определенные стандартные методы для обработки входящего трафика из сети. Блок-схема, приведенная на рис. 10.1, показывает, сколько серверов обрабатывают подключения для сокетов входящего потока.

Обратите внимание на то, что этот тип сервера включает в себя два вида сокетов: один для прослушивания и один для чтения и записи. Главный процесс задействует прослушивающий сокет для поиска подключений из сети. Когда появляется новое

соединение, главный процесс использует системный вызов `accept()`, чтобы принять соединение, которое создает для себя сокет для чтения/записи. Затем главный процесс применяет вызов `fork()` для создания нового дочернего процесса, обрабатывающего соединение. Наконец, исходный сокет как прослушиватель продолжает искать дополнительные соединения от имени главного процесса.

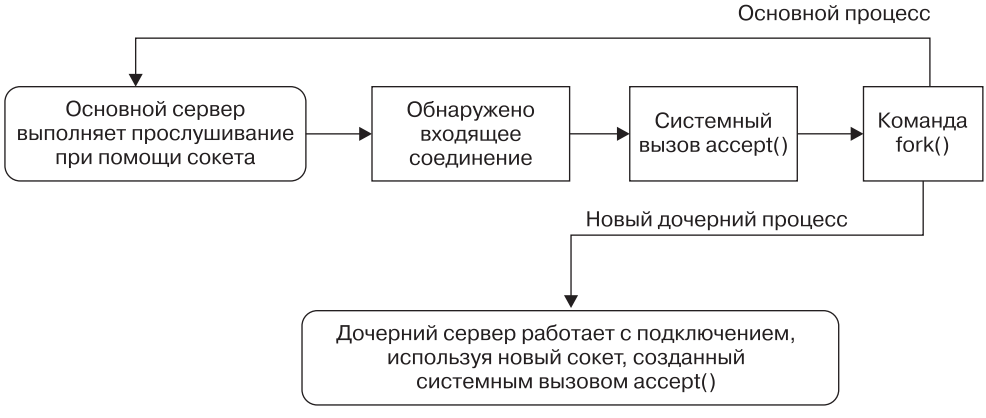


Рис. 10.1. Один из способов приема и обработки входящих подключений

После того как процесс настроил сокет определенного типа, он может взаимодействовать с ним способом, соответствующим этому типу. Именно это делает сокеты гибкими: если вам нужно изменить базовый транспортный уровень, нет необходимости переписывать все части, которые отправляют и получают данные, — в основном нужно изменить код инициализации.

Если вы программист и хотите научиться пользоваться интерфейсом сокетов, ознакомьтесь с книгой *Unix Network Programming*, том 1, 3-е издание (Addison-Wesley Professional, 2003) У. Ричарда Стивенса, Билла Феннера и Эндрю М. Рудоффа (W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff). Том 2 охватывает также межпроцессное взаимодействие.

10.10. Доменные сокеты Unix

Приложения, использующие сетевые средства, не обязательно должны включать два отдельных хоста. Многие из них построены как клиент-серверные или механизмы «узел — узел», где процессы, запущенные на одной машине, используют межпроцессную связь для согласования того, какую работу необходимо выполнить и кто ее станет делать. Например, вспомните, что демоны, такие как `systemd` и NetworkManager, применяли D-Bus для мониторинга и реагирования на системные события.

Процессы способны использовать обычную IP-сеть через локальный хост (127.0.0.1 или ::1) для связи друг с другом, но обычно они задействуют в качестве альтернативы специальный тип сокета, называемый *доменным сокетом Unix*. Когда процесс подключается к доменному сокету Unix, он ведет себя почти так же, как с сетевым сокетом: прослушивает и принимает соединения в сокете и позволяет выбирать между различными типами сокетов, чтобы действовать как TCP или UDP.

ПРИМЕЧАНИЕ

Имейте в виду, что доменный сокет Unix не является сетевым сокетом и за ним нет никакой сети. Вам даже не нужно настраивать сеть для его использования. Доменные сокеты Unix также не обязательно должны быть привязаны к файлам сокетов. Процесс может создать безымянный доменный сокет Unix и поделиться адресом с другим процессом.

Доменные сокеты Unix для IPC нравятся разработчикам по двум причинам. Во-первых, они позволяют использовать специальные файлы сокетов в файловой системе для управления доступом, поэтому любой процесс, у которого нет доступа к файлу сокета, не может его применять. И поскольку нет взаимодействия с сетью, она проще и менее подвержена обычным сетевым вторжениям. Например, файл сокета для D-Bus можно найти в `/var/run/dbus`:

```
$ 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 не нужно проходить через множество уровней своей сетевой подсистемы во время работы с доменными сокетами Unix, производительность, как правило, намного выше.

Создание кода для доменных сокетов Unix не сильно отличается от поддержки обычных сетевых сокетов. Поскольку преимущества могут быть значительными, некоторые сетевые серверы предлагают связь как через сетевые, так и через доменные сокеты Unix. Например, `mysqld` — сервер базы данных MySQL — может принимать клиентские подключения от удаленных хостов, но обычно он также предлагает доменный сокет Unix в `/var/run/mysqld/mysqld.sock`.

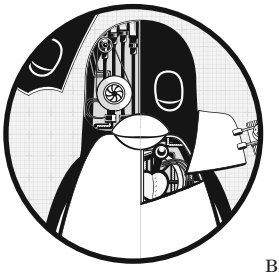
Вы можете просмотреть список доменных сокетов Unix, которые в данный момент используются в вашей системе, с помощью команды `lsof -U`:

```
# 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
--пропуск--
```

Список может быть довольно длинным, потому что многие приложения широко используют безымянные сокеты, которые обозначаются как `socket` в столбце `NAME`.

11

Сценарии оболочки



Если вы можете вводить команды в оболочке, то можете писать сценарии оболочки. *Сценарий*, или *скрипт оболочки* (известный также как сценарий Bourne shell), представляет собой серию команд, записанных в файл. Оболочка считывает команды из файла так же, как если бы вы ввели их в терминал.

11.1. Основы сценариев оболочки

Сценарии оболочки Bourne shell обычно начинаются со следующей строки, которая указывает, что программа `/bin/sh` должна выполнять команды в файле сценария (убедитесь, что в начале него нет пробелов):

```
#!/bin/sh
```

Часть `#!` называется *шебанг* (shebang), вы встретите ее и в других сценариях в этой книге. Можете перечислить любые команды, которые хотите, после строки `#!/bin/sh`, например:

```
#!/bin/sh
#
# Print something, then run ls

echo About to run the ls command.
ls
```

ПРИМЕЧАНИЕ

За исключением шебанга в верхней части сценария, символ # в начале строки указывает на комментарий, то есть оболочка игнорирует все, что находится в строке после #. Используйте комментарии, чтобы описать части сценариев, которые может быть трудно понять тем, кто станет читать ваш код, или чтобы освежить собственную память, когда вернетесь к коду позже.

Как и в любой программе в системах Unix, вам необходимо установить бит исполняемого файла для файла сценария оболочки, а также бит чтения, чтобы оболочка могла прочитать файл. Самый простой способ сделать это — ввести следующую команду:

```
$ chmod +rx script
```

Здесь *script* — имя сценария оболочки.

Команда `chmod` позволяет другим пользователям читать и выполнять сценарий `script`. Если вы этого не хотите, примените абсолютный режим 700 (и вернитесь к разделу 2.17, чтобы освежить информацию о правах доступа).

После создания сценария оболочки и настройки прав доступа на чтение и выполнение его можно запустить, поместив файл сценария в один из каталогов в пути к вашей команде, а затем запустив имя сценария в командной строке. Вы также можете запустить `./script`, если сценарий находится в текущем рабочем каталоге, или же использовать полный путь.

Запуск сценария с помощью шебанга почти (но не совсем) совпадает с запуском команды в оболочке: например, запуск сценария с именем `myscript` заставляет ядро запускать `/bin/sh myscript`.

Итак, мы разобрали основы, теперь рассмотрим некоторые ограничения сценариев оболочки.

ПРИМЕЧАНИЕ

Шебанг не обязательно должен иметь вид #!/bin/sh, он может быть создан для запуска в вашей системе чего угодно, что принимает ввод сценариев, например, #!/usr/bin/python для запуска программ на Python. Кроме того, вы можете столкнуться со сценариями с другим шаблоном, который включает /usr/bin/env. Например, можете встретить что-то вроде #!/usr/bin/env python в первой строке. Такой шебанг указывает утилите env запустить python. Причина этого довольно проста: env ищет команду для запуска в текущем пути к команде, поэтому вам не нужно стандартное расположение для исполняемого файла. Недостатком является то, что первый соответствующий исполняемый файл в пути к команде может оказаться неподходящим.

11.1.1. Ограничения скриптов оболочки

Оболочка относительно легко управляет командами и файлами. В разделе 2.14 вы видели, как оболочка может перенаправлять вывод, что является одним из важных

элементов программирования ее сценариев. Однако сценарий оболочки — это только один инструмент для программирования Unix, и хотя сценарии — крутая штука, у них также есть ограничения.

Одной из основных сильных сторон сценариев оболочки является то, что они могут упростить и автоматизировать задачи, которые в противном случае можно выполнять в командной строке, например манипулирование группами файлов. Но если вы пытаетесь анализировать строки, повторение арифметических вычислений или сложный доступ к базам данных либо вам нужны функции и сложные структуры управления, лучше использовать скриптовый язык, такой как Python, Perl или `awk`, или, возможно, даже компилируемый язык, такой как C. (Это важно, поэтому мы будем сталкиваться с этим на протяжении всей главы.)

Наконец, обратите внимание на размеры ваших сценариев оболочки — они должны оставаться короткими. Сценарии оболочки не должны быть большими, хотя вы, несомненно, еще столкнетесь с чудовищными размерами сценариев.

11.2. Кавычки и литералы

В ходе работы с оболочкой и сценариями сложнее всего разобраться в том, когда и почему следует использовать кавычки и другие знаки препинания. Допустим, вы хотите напечатать строку `$100`, выполнив следующие действия:

```
$ echo $100
00
```

Почему в выводе появилась строка с `00`? Потому что `$1` имеет префикс `$`, который оболочка интерпретирует как переменную оболочки (мы рассмотрим их в ближайшее время). Предположим, что нужно добавить двойные кавычки, чтобы оболочка не изменяла строку `$1`:

```
$ echo "$100"
00
```

Все равно не сработало. Давайте попробуем вместо двойных кавычек поставить одинарные:

```
$ echo '$100'
$100
```

Почему же этот вариант сработал?

11.2.1. Литералы

Используя кавычки, вы создаете литерал — строку, которую оболочка не должна анализировать или пытаться изменить перед передачей в командную строку. Помимо символа `$` из приведенного примера, иногда требуется передать символ *

в такую команду, как `grep`, без разворачивания ее оболочкой. То же касается и использования в команде точки с запятой `;`.

В ходе написания сценариев и работы в командной строке помните, что происходит, когда оболочка выполняет команду:

1. Перед выполнением команды оболочка ищет переменные, шаблоны поиска и другие замены и, если они появляются, выполняет их.
2. Оболочка передает результаты замен команде.

Проблемы, связанные с литералами, могут быть незаметными. Допустим, вы ищете в `/etc/passwd` все записи, соответствующие регулярному выражению `r.*t`, то есть строки, содержащие `r`, за которой следует `t`, что позволит находить имена пользователей, такие как `root`, `ruth` и `robot`. Для этого выполняете команду

```
$ grep r.*t /etc/passwd
```

Большую часть времени она работает, но иногда таинственным образом выходит из строя. Почему? Ответ, вероятно, кроется в текущем каталоге. Если он содержит файлы с такими именами, как `r.input` и `r.output`, то оболочка расширяет `r.*t` до `r.input r.output` и создает команду

```
$ grep r.input r.output /etc/passwd
```

Чтобы избежать подобных проблем, необходимо сначала распознать символы, которые могут доставить неприятности, а затем для их защиты применить правильные кавычки.

11.2.2. Одинарные кавычки

Самый простой способ создать литерал и заставить оболочку оставить строку в покое — заключить всю строку в одинарные кавычки (`'`), как в примере с `grep` и символом *:

```
$ grep 'r.*t' /etc/passwd
```

Что касается оболочки, то все символы между двумя одинарными кавычками, включая пробелы, составляют один параметр. Из-за этого следующая команда не сработает, потому что она запрашивает команду `grep` для поиска строки `r.*t /etc/passwd` в стандартном вводе, так как для `grep` передан только один параметр:

```
$ grep 'r.*t /etc/passwd'
```

Когда требуется использовать литерал, необходимо сначала взять одинарные кавычки, потому что так оболочка не будет пытаться выполнить какие-либо замены. В результате формируется в целом чистый синтаксис. Однако иногда для работы команды нужна немного бóльшая гибкость, поэтому можно применять двойные кавычки.

11.2.3. Двойные кавычки

Двойные кавычки (") работают так же, как одинарные, за исключением того, что оболочка расширяет любые переменные, которые в них появляются. Вы можете увидеть разницу, выполнив следующую команду, а затем заменив двойные кавычки одинарными и запустив ее снова:

```
echo "There is no * in my path: $PATH"
```

При выполнении команды обратите внимание на то, что оболочка подставляет значение $PATH, но не заменяет *.

ПРИМЕЧАНИЕ

Если вы используете двойные кавычки, работая с большими объемами текста, задействуйте here-документ, как описано в разделе 11.9.

11.2.4. Литерал с одинарными кавычками

Использование литералов в оболочке может усложниться, если передать команде литерал с одинарной кавычкой. Один из способов сделать это — поставить обратную косую черту перед символом одинарной кавычки:

```
$ echo I don\'t like contractions inside shell scripts.
```

Обратная косая черта и кавычка должны появляться за пределами любой пары одинарных кавычек. Строка 'don\'t приводит к синтаксической ошибке. Как ни странно, вы можете заключить одинарную кавычку в двойные кавычки, как показано в следующем примере (вывод идентичен выводу предыдущей команды):

```
$ echo "I don't like contractions inside shell scripts."
```

Если вы находитесь в затруднительном положении и вам нужно общее правило для цитирования всей строки без подстановок, выполните следующие действия:

1. Измените все символы ' (одинарная кавычка) на '\' ' (одинарная кавычка, обратная косая черта, одинарная кавычка, одинарная кавычка).
2. Заключите всю строку в одинарные кавычки.

Следовательно, вы можете процитировать неудобную строку, например this isn't a forward slash: \, таким образом:

```
$ echo 'this isn'\''t a forward slash: \'
```

ПРИМЕЧАНИЕ

Напомним: когда вы заключаете строку в кавычки, оболочка обрабатывает все, что находится внутри них, как один параметр. Следовательно, a b с считаются тремя параметрами, но a «b c» — это уже два параметра.

11.3. Специальные переменные

Большинство сценариев командной оболочки понимают параметры командной строки и взаимодействуют с командами, которые те выполняют. Чтобы превратить сценарии из простого списка команд в более гибкие программы сценариев оболочки, необходимо знать, как использовать специальные переменные оболочки. Они похожи на любую другую переменную оболочки, как описано в разделе 2.8, за исключением того, что вы не можете изменять значения некоторых из них.

ПРИМЕЧАНИЕ

Прочитав следующие несколько разделов, вы поймете, почему сценарии оболочки по мере написания накапливают множество специальных символов. Если вы пытаетесь изучить сценарий оболочки и натыкаетесь на строку, которая выглядит совершенно непонятной, разберите ее по частям.

11.3.1. Индивидуальные аргументы: $1, $2 и другие

`$1`, `$2` и все переменные, названные положительными ненулевыми целыми числами, содержат значения параметров скрипта, или *аргументов*. Например, рассмотрим следующий сценарий (назовем его `pshow`):

```
#!/bin/sh
echo First argument: $1
echo Third argument: $3
```

Попробуйте запустить сценарий следующим образом, чтобы увидеть, как он выводит аргументы:

```
$ ./pshow one two three
First argument: one
Third argument: three
```

Встроенную команду оболочки `shift` (сдвиг) можно использовать с переменными аргументов, чтобы удалить первый аргумент (`$1`) и продвинуть остальные аргументы, так чтобы `$2` стал `$1`, `$3` стал `$2` и т. д. Предположим, что имя следующего сценария — `shiftex`:

```
#!/bin/sh
echo Argument: $1
shift
echo Argument: $1
shift
echo Argument: $1
```

Запустите команду следующим образом:

```
$ ./shiftex one two three
Argument: one
```

```
Argument: two
Argument: three
```

Как видно из примера, `shiftex` выводит все три аргумента, печатая первый, сдвигая остальные аргументы и повторяя эти действия.

11.3.2. Количество аргументов: $#

Переменная `$#` содержит количество аргументов, переданных сценарию, и особенно важна, когда вы запускаете `shift` в цикле для выбора аргументов. Когда значение `$#` равно `0`, аргументов не остается, поэтому у `$1` значения нет. (Описание циклов см. в разделе 11.6.)

11.3.3. Все аргументы: $@

Переменная `$@` представляет все аргументы скрипта и очень полезна для передачи их команде внутри скрипта. Например, команды `Ghostscript (gs)` обычно длинные и сложные. Предположим, вам нужен ярлык для растеризации файла PostScript с разрешением 150 точек на дюйм, с использованием стандартного потока вывода, а также с возможностью передачи других параметров в `gs`. Можно написать такой сценарий, чтобы включить дополнительные параметры командной строки:

```
#!/bin/sh
gs -q -dBATCH -dNOPAUSE -dSAFER -sOutputFile=- -sDEVICE=pnmraw $@
```

ПРИМЕЧАНИЕ

Если строка в сценарии оболочки становится слишком длинной, что затрудняет чтение и обработку в текстовом редакторе, можете разделить ее с помощью обратной косой черты (\). Например, можно изменить предыдущий сценарий следующим образом:

```
#!/bin/sh
gs -q -dBATCH -dNOPAUSE -dSAFER \
-sOutputFile=- -sDEVICE=pnmraw $@
```

11.3.4. Имя сценария: $0

Переменная `$0` содержит имя сценария и полезна для генерации диагностических сообщений. Предположим, сценарий должен сообщить о недопустимом аргументе, который хранится в переменной `$BADPARM`. Можете вывести диагностическое сообщение со следующей строкой, чтобы имя сценария появилось в сообщении об ошибке:

```
echo $0: bad option $BADPARM
```

Все диагностические сообщения об ошибках должны быть переведены в стандартную ошибку. Как объясняется в разделе 2.14.1, `2>&1` перенаправляет стандартную ошибку на стандартный вывод. Для записи в стандартную ошибку вы можете

обратить этот процесс с помощью `1>&2`. Чтобы сделать это в предыдущем примере, используйте следующую команду:

```
echo $0: bad option $BADPARM 1>&2
```

11.3.5. ID процесса: $$

Переменная `$$` содержит идентификатор процесса оболочки PID.

11.3.6. Код возврата: $?

Переменная `$?` содержит код возврата последней команды, выполненной оболочкой. Далее мы рассмотрим коды выхода, которые имеют решающее значение для освоения сценариев оболочки.

11.4. Коды возврата

Когда программа Unix завершается, она оставляет для родительского процесса, запустившего программу, *код возврата* — числовое значение, известное также как *код ошибки* или *значение возврата*. Код возврата, равный нулю (0), означает, что программа работала без проблем. Однако если в программе возникает ошибка, она обычно завершается с числом, отличным от 0 (но не всегда, как вы увидите далее).

Оболочка содержит код возврата последней команды в специальной переменной `$?`, поэтому можно проверить его в командной строке:

```
$ ls / > /dev/null
$ echo $?
0
$ ls /asdfasdf > /dev/null
ls: /asdfasdf: No such file or directory
$ echo $?
1
```

Из примера видно, что успешная команда вернула `0`, а неудачная — `1` (конечно, при условии, что в системе нет каталога с именем `/asdfasdf`).

Если вы собираетесь использовать код возврата команды, то должны применить или сохранить этот код сразу после выполнения команды, потому что следующая выполненная команда перезапишет предыдущий код. Например, если запустить `echo $?` дважды подряд, то вывод второй команды всегда будет равен `0`, потому что первая команда `echo` успешно завершится.

При написании кода оболочки вы можете столкнуться с ситуациями, когда сценарий следует остановить из-за ошибки, например неправильного имени файла. Используйте `exit 1` в сценарии, чтобы завершить его и передать код возврата `1` любому родительскому процессу, запустившему сценарий. (Можете задействовать

разные ненулевые числа, если ваш сценарий имеет различные нестандартные условия возврата.)

Обратите внимание на то, что некоторые программы, например `diff` и `grep`, используют ненулевые коды возврата для обозначения нормальных условий. Например, `grep` возвращает `0`, если находит что-то, соответствующее шаблону поиска, и `1`, если это не так. Для этих программ код возврата `1` — не ошибка, поэтому `grep` и `diff` применяют код возврата `2`, если сталкиваются с реальной проблемой. Если вы считаете, что программа может использовать ненулевой код возврата для указания успешного действия, изучите ее справочную страницу. Коды возврата обычно рассматриваются в разделе `EXIT VALUE` или `DIAGNOSTICS`.

11.5. Условные операторы

У оболочки Bourne shell есть специальные конструкции для условных операторов, такие как `if`, `then`, `else` и `case`. Например, этот простой скрипт с условным оператором `if` проверяет, является ли значение первого аргумента скрипта равным `hi`:

```
#!/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 предыдущем сценарии являются ключевыми словами оболочки, все остальное — команды. Это различие чрезвычайно важно, потому что легко ошибочно принять условное `[ $1 = "hi" ]` за специальный синтаксис оболочки. На самом деле символ `[` является реальной программой в системе Unix. Во всех системах Unix есть команда под названием `[`, которая выполняет тесты или проверку условий условных операторов сценариев оболочки. Она известна также как `test`, страницы руководства для `test` и `[` одинаковы. (Скоро вы узнаете, что оболочка не всегда запускает `[`, но пока будем рассматривать ее как отдельную команду.)

Именно в подобном случае жизненно важно понимать коды возврата, как описано в разделе 11.4. Посмотрим, как на самом деле работает предыдущий сценарий:

1. Оболочка запускает команду после ключевого слова `if` и получает код возврата этой команды.
2. Если код возврата равен `0`, оболочка выполняет команды, следующие за ключевым словом `then`, останавливаясь при достижении ключевого слова `else` или `fi`.
3. Если код возврата не равен `0` и есть условие `else`, оболочка выполняет команды после ключевого слова `else`.
4. Условный оператор заканчивается после `fi`.

Мы установили, что тест, следующий за `if`, — это команда, поэтому давайте посмотрим на точку с запятой (`;`). Это обычный маркер оболочки для конца команды, и он применен в примере, потому что мы поместили ключевое слово `then` в ту же строку. Без точки с запятой оболочка передает `then` в качестве параметра команды `[`, что часто приводит к ошибке, которую нелегко отследить. Вы можете избежать использования точки с запятой, поместив ключевое слово `then` в отдельную строку следующим образом:

```
if [ $1 = hi ]
then
    echo 'The first argument was "hi"'
fi
```

11.5.1. Обходной путь для предотвращения ошибки Empty Parameter Lists

В предыдущем примере есть потенциальная проблема с условным оператором из-за часто упускаемого из виду сценария: `$1` может быть пустым, потому что пользователь может запустить сценарий без параметров. Если `$1` пуст, тест считывает `[ = hi ]` и команда `[` прервется с ошибкой. Это можно исправить, заключив параметр в кавычки одним из двух распространенных способов:

```
if [ "$1" = hi ]; then
if [ x"$1" = x"hi" ]; then
```

11.5.2. Другие команды для проверки условий

Существует множество возможностей использования команд, отличных от `[`, для проверки условий. Вот пример, в котором применяется `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

Существует также ключевое слово `elif`, которое позволяет объединять условные операторы `if`, как показано далее:

```
#!/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
```

Имейте в виду, что выполняется только первое истинное условие, поэтому, если вы запустите этот сценарий с аргументами `hi bye`, то получите только подтверждение аргумента `hi`.

ПРИМЕЧАНИЕ

Не слишком увлекайтесь ключевым словом elif, потому что чаще всего используется конструкция case, которую вы увидите в подразделе 11.5.6.

11.5.4. Логические конструкции

Существуют две быстрые однострочные конструкции условий, которые вы можете время от времени видеть, используя синтаксис `&&` (логическое И) и `||` (логическое ИЛИ). Конструкция `&&` работает следующим образом:

```
command1 && command2
```

В данном примере оболочка выполняет команду *`command1`*, и если код возврата равен `0`, оболочка выполняет также команду *`command2`*.

Конструкция `||` аналогична: если команда перед `||` возвращает ненулевой код возврата, оболочка выполняет вторую команду.

Конструкции `&&` и `||` часто используются при проверке условий `if`, и в обоих случаях код возврата последнего запуска команды определяет, как оболочка обрабатывает условие. Если в случае применения конструкции `&&` первая команда завершается неудачно, оболочка использует свой код возврата для оператора `if`, но если первая команда успешна, оболочка применяет код возврата второй команды для условия. В случае конструкции `||` оболочка задействует код возврата первой команды в случае успеха или код возврата второй, если первая не удалась, например:

```
#!/bin/sh
if [ "$1" = hi ] || [ "$1" = bye ]; then
    echo 'The first argument was "'$1'"'
fi
```

Если условия включают команду теста (`[`), как показано далее, можете использовать параметры `-a` и `-o` вместо `&&` и `||`, например:

```
#!/bin/sh
if [ "$1" = hi -o "$1" = bye ]; then
    echo 'The first argument was "'$1'"'
fi
```

Вы можете инвертировать тест (то есть создать логическое «нет»), поместив оператор `!` перед проверкой условия, например:

```
#!/bin/sh
if [ ! "$1" = hi ]; then
```

```
    echo 'The first argument was not hi'
fi
```

В этом конкретном случае сравнения видно, что `!=` используется в качестве альтернативы, а `!` может применяться с любой проверкой условий, описанных в следующем разделе.

11.5.5. Проверка условий

Мы разобрали, как работает `[`: код возврата равен `0`, если условие истинно, и ненулевой, когда тест завершается неудачей. Мы также знаем, как проверить равенство строк `str1` и `str2` с помощью `[ str1 = str2 ]`. Однако помните, что сценарии оболочки хорошо подходят для операций с целыми файлами, потому что многие полезные `[` тесты включают свойства файла. Например, следующая строка проверяет, является ли файл *`file`* обычным файлом, а не каталогом или специальным файлом:

```
[ -f file ]
```

В сценарии вы можете увидеть проверку условия `-f` в цикле, подобном следующему, который проверяет все элементы в текущем рабочем каталоге (подробнее о циклах вы узнаете в разделе 11.6):

```
for filename in *; do
    if [ -f $filename ]; then
        ls -l $filename
        file $filename
    else
        echo $filename is not a regular file.
    fi
done
```

ПРИМЕЧАНИЕ

Поскольку команда test так широко используется в сценариях, она встроена во многие версии оболочки Bourne shell (включая bash). Она ускоряет выполнение сценариев, поскольку оболочке не нужно запускать отдельную команду для каждого теста.

Существуют десятки операций проверки условий, которые делятся на три основные категории: операции проверки файлов, проверки строк и арифметической проверки. Руководство info содержит полную онлайн-документацию, а его страница `test(1)` может использоваться для получения быстрой справки. В следующих разделах описываются основные виды проверок. Мы рассмотрим наиболее распространенные.

Проверка файлов

Большинство проверок файлов, например `-f`, называются унарными операциями, потому что для них требуется только один аргумент — файл для проверки. Например, есть две важных проверки файлов:

- `-e` — возвращает значение `true`, если файл существует;
- `-s` — возвращает значение `true`, если файл не пустой.

Некоторые операции проверяют тип файла — это значит, что они могут определить, является ли файл обычным файлом, каталогом или каким-то специальным устройством, как показано в табл. 11.1. Существует также ряд унарных операций, которые проверяют права доступа к файлу, как показано в табл. 11.2. (В разделе 2.17 описаны разрешения.)

Таблица 11.1. Операторы проверки типов файлов

Оператор	Что проверяет
`-f`	Обычный файл
`-d`	Каталог
`-h`	Символическая ссылка
`-b`	Блочное устройство
`-c`	Символьное устройство
`-p`	Именованный канал
`-S`	Сокет

ПРИМЕЧАНИЕ

Если команда test используется для символической ссылки, она проверяет фактический объект, с которым связана ссылка, а не саму ссылку (за исключением проверки -h). То есть если ссылка link является символической ссылкой на обычный файл, то [-f link] возвращает код возврата true (0).

Таблица 11.2. Операторы проверки прав доступа

Оператор	Право доступа
`-r`	На чтение
`-w`	На запись
`-x`	На исполнение
`-u`	На Setuid
`-g`	На Setgid
`-k`	На Sticky

Наконец, при проверке файлов используются три бинарных оператора (проверки, для которых в качестве аргументов требуются два файла), но они не очень распространены. Рассмотрим команду, которая включает в себя `-nt` (newer than — младше):

```
[ file1 -nt file2 ]
```

Выражение верно, если файл *file1* имеет более новую дату изменения, чем файл *file2*.

Оператор `-ot` (older than — старше) делает обратное. Если же вам нужно обнаружить идентичные жесткие ссылки, `-ef` сравнивает два файла и возвращает `true`, если они имеют общие номера дескрипторов inode и устройства.

Проверка строк

Мы изучили бинарный строковый оператор `=`, который возвращает значение `true`, если его операнды равны, и оператор `!=`, который возвращает значение `true`, если его операнды не равны. Существуют две дополнительные операции с унарными строками:

- `-z` — возвращает значение `true`, если аргумент пуст (`[ -z "" ]` возвращает `0`);
- `-n` — возвращает значение `true`, если его аргумент не пуст (`[ -n "" ]` возвращает `1`).

Арифметическая проверка

Обратите внимание на то, что знак равенства (`=`) проверяет равенство *строк*, а не *числовое* равенство. Следовательно, `[ 1 = 1 ]` возвращает `0` (`true`), но `[ 01 = 1 ]` возвращает `false`. Работая с числами, используйте параметр `-eq` вместо знака равенства: `[ 01 -eq 1 ]` возвращает значение `true`. В табл. 11.3 приведен полный список операторов для числового сравнения.

Таблица 11.3. Операторы для числового сравнения

Оператор	Возвращает значение true, если первый аргумент...
`-eq`	равен второму
`-ne`	не равен второму
`-lt`	меньше, чем второй
`-gt`	больше, чем второй
`-le`	меньше или равен второму
`-ge`	больше или равен второму

11.5.6. Ключевое слово case

Ключевое слово `case` образует еще одну условную конструкцию, которая исключительно полезна для сопоставления строк. Она не выполняет никаких команд проверки и, следовательно, не оценивает коды возврата. Тем не менее она может выполнять сопоставление с образцом. На примере все выглядит более понятно:

```
#!/bin/sh
case $1 in
    bye)
        echo Fine, bye.
```

```
        ;;
    hi|hello)
        echo Nice to see you.
        ;;
    what*)
        echo Whatever.
        ;;
    *)
        echo 'Huh?'
        ;;
esac
```

Оболочка выполняет сценарий следующим образом:

1. Скрипт сопоставляет значение переменной `$1` с каждым из вариантов значений, отделенных символом `)`.
2. Если значение соответствует `$1`, оболочка выполняет команды, расположенные под этим вариантом, пока не встретит символ `;;`, после чего переходит к ключевому слову `esac`.
3. Условный оператор заканчивается на `esac`.

С каждым вариантом значения `case` можно сопоставить одну строку (например, `bye` в предыдущем примере) или несколько строк с `|` (`hi|hello` возвращает значение `true`, если `$1` равно `hi` или `hello`). Или можно использовать шаблоны `*` либо `?` (`what*`). Чтобы создать вариант, который по умолчанию отыскивает все возможные значения, отличные от указанных значений вариантов, возьмите один символ `*`, как показано в последнем случае в приведенном ранее примере.

ПРИМЕЧАНИЕ

Заканчивайте каждое условие case двойной точкой с запятой (;;), чтобы избежать возможной синтаксической ошибки.

11.6. Циклы

В командной оболочке Bourne shell есть два вида циклов, `for` и `while`.

11.6.1. Циклы for

Цикл `for` (который является циклом «для каждого») распространен наиболее широко, например:

```
#!/bin/sh
for str in one two three four; do
    echo $str
done
```

В этом сценарии `for`, `in`, `do` и `done` — ключевые слова оболочки. Оболочка выполняет следующие действия:

1. Присваивает переменной `str` первое (one) из четырех значений, разделенных пробелами и стоящих после ключевого слова `in`.
2. Запускает команду `echo` между `do` и `done`.
3. Возвращается к строке `for`, присваивает `str` следующее значение (`two`), выполняет команды, находящиеся между `do` и `done`, и повторяет процесс до тех пор, пока не закончатся значения, следующие за ключевым словом `in`.

Вывод этого скрипта выглядит следующим образом:

```
one
two
three
four
```

11.6.2. Циклы while

В цикле `while` оболочка Bourne shell использует коды возврата так же, как условный оператор `if`. Например, этот скрипт 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 возврата из `grep -q firstline`. Как только код возврата становится ненулевым (в данном случае, когда строка `firstline` больше не отображается в последних десяти строках файла `$FILE`), цикл завершается.

Вы можете выйти из цикла `while` с помощью выражения `break`. У оболочки Bourne shell есть также цикл `until`, который работает так же, как и `while`, за исключением того, что он прерывает цикл, когда встречает нулевой код возврата, а не ненулевой. Однако не стоит часто применять циклы `while` и `until`. Если вам понадобится использовать цикл `while`, вероятно, следует перейти на язык, более подходящий для задачи, например Python или `awk`.

11.7. Подстановка команд

Оболочка Bourne shell может перенаправлять стандартный вывод команды обратно в собственную командную строку оболочки. То есть вы можете использовать вывод

команды в качестве аргумента для другой команды или сохранить его в переменной оболочки, заключив команду в `$()`.

В следующем примере вывод команды хранится внутри переменной `FLAGS`. Подстановка команды выделена жирным шрифтом во второй строке.

```
#!/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
```

Этот пример довольно сложен, поскольку он демонстрирует, что вы можете использовать как одинарные кавычки, так и конвейеры внутри подстановки команды. Результат выполнения команды `grep` отправляется команде `sed` (подробнее о `sed` в подразделе 11.10.3), которая удаляет все, что соответствует выражению `.*:`. Результат работы `sed` передается в `head`.

С подстановкой команд легко переборщить. Например, не задействуйте `$(ls)` в сценарии, потому что быстрее сработает оболочка для расширения `*`. Кроме того, если вы хотите вызвать команду для нескольких имен файлов, которые получаете в результате работы команды `find`, используйте конвейер для `xargs`, а не подстановки команд, или примените параметр `-exec` (оба рассматриваются в подразделе 11.10.4).

ПРИМЕЧАНИЕ

Традиционный синтаксис подстановки команд заключается в том, чтобы заключить команду в обратные апострофы (`), их вы встретите во многих сценариях оболочки. Синтаксис $() новее, но он является стандартом POSIX, и его, как правило, проще (для людей) читать и писать.

11.8. Управление временными файлами

Бывает необходимо создать временный файл сбора вывода, чтобы использовать его в следующей команде. При этом убедитесь, что имя файла достаточно уникально, чтобы никакие другие программы случайно не записали в него данные. Можно применить нечто простое, например, PID оболочки (значение переменной `$$`)

в имени файла, но, чтобы убедиться, что конфликтов не возникнет, лучше всего использовать утилиту `mktemp`.

Приведем пример применения команды `mktemp` для создания временных имен файлов. Этот сценарий отображает прерывания устройства, которые произошли за последние 2 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 для `mktemp` — это шаблон. Команда `mktemp` преобразует `XXXXXX` в уникальный набор символов и создает пустой файл с этим именем. Обратите внимание на то, что этот сценарий использует имена переменных для хранения имен файлов, так что вам нужно изменить только одну строку, если вы хотите изменить имя файла.

ПРИМЕЧАНИЕ

Не все варианты систем Unix включают утилиту mktemp. Если у вас возникли проблемы с переносимостью, лучше всего установить пакет GNU coreutils для вашей операционной системы.

Распространенная проблема со сценариями, задействующими временные файлы, заключается в том, что в случае прерывания сценария временные файлы могут сохраниться в системе. В предыдущем примере, если нажать сочетание клавиш Ctrl+C перед второй командой `cat`, то временный файл останется в каталоге `/tmp`. По возможности стоит этого избегать. Вместо этого используйте команду `trap` для создания обработчика сигнала, чтобы перехватить сигнал, который генерирует Ctrl+C, и удалить временные файлы, как в следующем примере:

```
#!/bin/sh
TMPFILE1=$(mktemp /tmp/im1.XXXXXX)
TMPFILE2=$(mktemp /tmp/im2.XXXXXX)
trap "rm -f $TMPFILE1 $TMPFILE2; exit 1" INT
--пропуск--
```

Вы должны использовать `exit` в обработчике, чтобы явно завершить выполнение скрипта, иначе оболочка продолжит его выполнение после запуска обработчика сигналов.

ПРИМЕЧАНИЕ

Не обязательно указывать аргумент для команды mktemp: если этого не сделать, шаблон будет начинаться с префикса /tmp/tmp.

11.9. Here-документы

Допустим, вы хотите вывести большой фрагмент текста или передать много текста другой команде. Вместо того чтобы использовать несколько команд `echo`, можете применить функцию оболочки *here-документ*, как показано в следующем примере:

```
#!/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 here-документом.

`<<EOF` указывает оболочке перенаправить все последующие строки на стандартный ввод команды, предшествующей `<<EOF` (в данном случае это `cat`). Перенаправление прекращается, как только маркер `EOF` появляется в строке сам по себе. Маркер может быть любой строкой, но не забывайте использовать один и тот же маркер в начале и в конце here-документа. Кроме того, соглашение требует, чтобы маркер состоял только из прописных букв.

Обратите внимание на переменную оболочки `$DATE` в here-документе. Оболочка расширяет переменные оболочки внутри документов, что особенно полезно при выводе отчетов, содержащих много переменных.

11.10. Важные утилиты сценариев оболочки

Некоторые программы особенно полезны в сценариях оболочки. Такие утилиты, как `basename`, эффективны только при использовании с другими программами и поэтому редко применяются вне сценариев оболочки. Но и другие утилиты, например `awk`, могут быть весьма полезны для работы в командной строке.

11.10.1. Утилита basename

Если вам нужно удалить расширение из имени файла или избавиться от названия каталогов из полного пути, используйте команду `basename`. Попробуйте выполнить приведенные далее примеры в командной строке, чтобы увидеть, как работает команда:

```
$ basename example.html .html
$ basename /usr/local/bin/example
```

В обоих случаях `basename` возвращает `example`. Первая команда удаляет суффикс `.html` из `example.html`, а вторая удаляет каталоги из полного пути.

В следующем примере показано, как можно использовать basename в сценарии для преобразования файлов изображений GIF в формат PNG:

```
#!/bin/sh
for file in *.gif; do
    # выйти, если больше нет файлов
    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. Утилита awk

awk — не просто универсальная команда, это мощный язык программирования. К сожалению, использование awk в настоящее время является чем-то вроде утраченного искусства, его заменили более развитые языки, такие как Python.

Существуют полноценные книги по теме awk, в том числе книга Альфреда В. Ахо, Брайана У. Кернигана и Питера Дж. Вайнбергера (Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger) *The AWK Programming Language* (Addison-Wesley, 1988). Тем не менее многие пользователи применяют awk только для того, чтобы выбрать одно поле из входного потока, как в примере далее:

```
$ ls -l | awk '{print $5}'
```

Эта команда выводит пятое поле вывода ls (размер файла). В результате выводится список размеров файлов.

11.10.3. Утилита sed

Программа sed (stream editor — редактор потоков) — это автоматический текстовый редактор, который принимает входной поток (файл или стандартный ввод), изменяет его в соответствии с некоторым выражением и выводит результаты на стандартный вывод. Во многих отношениях sed похож на ed — оригинальный текстовый редактор Unix. В нем есть десятки операций, инструментов сопоставления и возможностей адресации. Как и об awk, о sed было написано множество книг, включая краткий справочник Арнольда Роббинса (Arnold Robbins), охватывающий как sed, так и awk, — *sed & awk Pocket Reference*, 2-е издание (O'Reilly, 2002).

Хотя sed — это большая программа, и ее углубленный анализ выходит за рамки книги, довольно легко понять, как она работает. В целом sed принимает адрес и операцию в качестве одного аргумента. Адрес представляет собой набор строк, и команда определяет, что делать с этими строками.

Широко распространенной задачей для `sed` является замена части текста с помощью регулярного выражения (см. подраздел 2.5.1), например:

```
$ sed 's/exp/text/'
```

Чтобы заменить первое двоеточие в каждой строке `/etc/passwd` на `%` и отправить результат в стандартный вывод, нужно сделать так:

```
$ sed 's/:/%/' /etc/passwd
```

Чтобы заменить все двоеточия в `/etc/passwd`, добавьте модификатор `g` (global) в конец операции, как показано далее:

```
$ sed 's/:/%/g' /etc/passwd
```

Так выглядит команда, которая работает для каждой строки, — она считывает `/etc/passwd`, удаляет строки с третьей по шестую и отправляет результат в стандартный вывод:

```
$ sed 3,6d /etc/passwd
```

В этом примере `3,6` — это адрес (диапазон строк), `d` — операция (delete — удаление). Если вы опустите адрес, `sed` будет работать со всеми строками в своем входном потоке. Двумя наиболее распространенными операциями `sed`, вероятно, являются `s` (поиск и замена) и `d`.

Вы также можете использовать регулярное выражение в качестве адреса. Следующая команда удаляет любую строку, соответствующую регулярному выражению *exp*:

```
$ sed '/exp/d'
```

Во всех этих примерах `sed` записывает данные в стандартный вывод, и этот вариант, безусловно, применяется чаще всего. Без аргументов файла `sed` считывает данные со стандартного ввода — таков шаблон, с которым вы часто сталкиваетесь в конвейерах оболочки.

11.10.4. Утилита xargs

Когда необходимо выполнить одну команду для огромного количества файлов, команда или оболочка могут сообщить, что не могут вместить все аргументы в буфер. Используйте команду `xargs`, чтобы обойти эту проблему, выполнив команду для каждого имени файла в его стандартном входном потоке.

Многие пользователи применяют `xargs` с командой `find`. Например, следующий сценарий помогает проверить, что каждый файл в текущем дереве каталогов, заканчивающийся на `.gif`, на самом деле является изображением GIF:

```
$ find . -name '*.gif' -print | xargs file
```

В этом примере xargs запускает команду file. Однако этот вызов может привести к ошибкам и проблемам с безопасностью, поскольку имена файлов могут содержать пробелы и новые строки. При написании сценария вместо этого используйте следующую форму, которая изменяет разделитель вывода find и разделитель аргументов xargs с новой строки на символ NULL:

```
$ find . -name '*.gif' -print0 | xargs -0 file
```

Команда xargs запускает множество процессов, поэтому не ожидайте от нее высокой производительности, если у вас большой список файлов.

Возможно, вам потребуется добавить два дефиса (--) в конец команды xargs, если есть вероятность, что любой из целевых файлов начнется с одного дефиса (-). Двойной дефис (--) сообщает программе, что все последующие аргументы являются именами файлов, а не параметрами. Однако имейте в виду, что не все программы поддерживают применение двойного дефиса.

При использовании команды find есть альтернатива xargs — параметр -exec. Однако его синтаксис довольно сложен, потому что нужно поставить фигурные скобки {}, чтобы заменить имя файла, и литерал ;, чтобы указать конец команды. Вот как можно выполнить предыдущую задачу с помощью только команды find:

```
$ find . -name '*.gif' -exec file {} \;
```

11.10.5. Утилита expr

Если вам нужно использовать арифметические операции в сценариях оболочки, можете делать это с помощью команды expr (и даже выполнить некоторые строковые операции). Например, команда expr 1 + 2 выводит 3. (Запустите expr --help для вывода полного списка операций.)

Команда expr — это неуклюжий и медленный способ выполнения математических задач. Если вы обнаружите, что часто пользуетесь этой командой, вам следует работать с языком Python вместо сценария оболочки.

11.10.6. Утилита exec

Команда exec — это встроенная функция оболочки, которая заменяет текущий процесс оболочки программой, которую вы указываете после команды exec. Она выполняет системный вызов exec(), описанный в главе 1. Эта функция предназначена для экономии системных ресурсов, но помните о ее необратимости: когда вы запускаете exec в сценарии оболочки, сценарий и оболочка, выполняющая его, исчезают, заменяясь новой командой.

Чтобы проверить работу команды в окне командной строки, запустите exec cat. После нажатия комбинации клавиш Ctrl+D или Ctrl+C для завершения программы cat окно должно исчезнуть, поскольку его дочернего процесса больше не существует.

11.11. Подоболочки

Допустим, вам нужно временно изменить среду в оболочке. Вы можете изменить и восстановить часть среды, например путь или рабочий каталог, задействуя переменные оболочки, но это довольно непрактичный способ решения проблем. Более простой вариант — использовать подоболочку: совершенно новый процесс оболочки, который вы можете создать просто для выполнения одной или двух команд. Новая оболочка содержит копию среды исходной оболочки, и при выходе из новой оболочки любые изменения, внесенные вами в ее среду оболочки, исчезают, оставляя исходную оболочку работать в обычном режиме.

Чтобы применить подоболочку, заключите команды, которые должны ею выполняться, в круглые скобки. Например, следующая строка выполняет команду `uglyprogram`, находясь в каталоге `uglydir`, и оставляет исходную оболочку нетронутой:

```
$ (cd uglydir; uglyprogram)
```

В этом примере показано, как добавить компонент в путь `$PATH`, который может вызвать проблемы, если сделать это изменение постоянным:

```
$ (PATH=/usr/confusing:$PATH; uglyprogram)
```

Подоболочка как способ внесения одноразового изменения в переменную среды используется так часто, что даже существует встроенный синтаксис, который позволяет избежать создания подоболочки:

```
$ PATH=/usr/confusing:$PATH uglyprogram
```

Каналы и фоновые процессы также работают с подоболочками. В следующем примере `tar` используется для архивирования всего дерева каталогов `orig`, а затем распаковывает архив в новый целевой каталог `target`, дублируя файлы и папки каталога `orig` (что полезно, поскольку сохраняет информацию о владельцах и правах доступа и, как правило, быстрее, чем выполнение команды `cp -r`):

```
$ tar cf - orig | (cd target; tar xvf -)
```

> **ВНИМАНИЕ**
>
> Дважды проверьте команду такого рода перед запуском, чтобы убедиться, что целевой каталог существует и полностью отделен от каталога orig (в сценарии вы можете проверить это с помощью [-d orig -a ! orig -ef target]).

11.12. Добавление файлов в скрипты

Если вам нужно включить код из другого файла в сценарий оболочки, используйте оператор точки (`.`). Например, следующая строка выполняет команды из файла `config.sh`:

```
. config.sh
```

Этот метод добавления называется также *сорсингом* (sourcing — включение) *файла* и полезен для чтения переменных (например, в общем файле конфигурации) и других определений. Это не то же самое, что выполнение другого сценария: когда вы запускаете сценарий как команду, он запускается в новой оболочке и не выдает ничего, кроме вывода и кода возврата.

11.13. Чтение пользовательского ввода

Команда `read` считывает строку текста из стандартного ввода и сохраняет текст в переменной. Например, следующая команда сохраняет ввод в переменную `$var`:

```
$ read var
```

Эта встроенная команда оболочки может оказаться полезной в сочетании с другими функциями оболочки, не упомянутыми в этой книге. С помощью `read` вы можете создавать простые взаимодействия, например запрашивать у пользователя ввод, вместо того чтобы требовать, чтобы он передавал все параметры в командной строке, и создавать подтверждения «Вы уверены?», предшествующие опасным операциям.

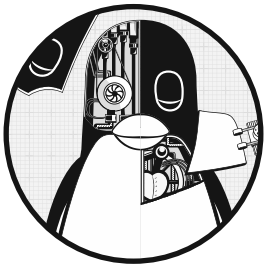
11.14. Когда не стоит использовать сценарии оболочки

Оболочка настолько многофункциональна, что трудно свести ее важные элементы в одну главу. Если вам интересно, что еще может оболочка, ознакомьтесь с книгами по программированию оболочки: *Unix Shell Programming*, 3-е издание (SAMS, 2003), Стивена Г. Кочана и Патрика Вуда (Stephen G. Kochan, Patrick Wood) или *The UNIX Programming Environment* (Prentice Hall, 1984) Брайана У. Кернигана и Роба Пайка (Brian W. Kernighan and Rob Pike).

Однако в определенный момент, особенно когда начинаете злоупотреблять встроенной функцией `read`, вы должны спросить себя, используете ли вы все еще правильный инструмент для этой работы. Помните, что сценарии оболочки лучше всего манипулируют простыми файлами и командами. Как говорилось ранее, если вы обнаружите, что пишете слишком запутанную команду, особенно если это связано со сложными строковыми или арифметическими операциями, не бойтесь обращаться к скриптовым языкам, например Python, Perl или `awk`.

12

Передача файлов по сети и доступ к ним



В этой главе мы рассмотрим варианты распространения файлов между компьютерами в сети и их совместного использования. Начнем с утилит для копирования файлов помимо утилит `scp` и `sftp`, которые рассматривали раньше. Затем обсудим общий доступ к файлам через подключение каталогов одного компьютера к другому.

Существует много способов распространения и совместного использования файлов, перечислим их с соответствующими утилитами (табл. 12.1).

Обратите внимание на то, что в таблице ничего не говорится о совместной реализации крупномасштабных решений для совместного доступа множества удаленных машин и пользователей.. Это тоже возможно, однако, как правило, требует изрядного объема работы и выходит за рамки данной книги. В конце главы мы обсудим почему.

В отличие от других глав книги, последняя часть данной главы не содержит материала для углубленного изучения, а разделы, из которых можно извлечь наибольшую пользу, — максимально теоретические. Из разделов 12.3 и 12.8 вы узнаете, почему в приведенной ранее таблице перечислено так много вариантов.

Таблица 12.1. Распространение и совместное использование файлов

Открывать временный доступ для файла или каталога из вашей системы Linux для других машин	Python SimpleHTTPServer (см. раздел 12.1)
Раздавать (копировать) файлы на разные компьютеры (регулярно)	rsync (см. раздел 12.2)
Регулярно предоставлять общий доступ к файлам с компьютера Linux на компьютер Windows	Samba (см. раздел 12.4)
Монтировать папки совместного доступа Windows на компьютере Linux	CIFS (см. раздел 12.4)
Внедрить маломасштабное решение для совместного доступа между машинами Linux с минимальной настройкой	SSHFS (см. раздел 12.5)
Подключать файловые системы большего размера с сетевого накопителя или другого сервера в надежной локальной сети	NFS (см. раздел 12.6)
Подключать к машине Linux облачное хранилище	Различные файловые системы на основе FUSE (см. раздел 12.7)

12.1. Быстрое копирование данных

Допустим, вы хотите скопировать файл (или файлы) со своего компьютера Linux на другой компьютер по личной сети и при этом обратно его копировать не нужно, то есть просто хотите быстро передать файл в одну сторону. Удобнее всего сделать это с помощью Python. Просто перейдите в каталог, содержащий файл(ы), и запустите следующую команду:

```
$ python -m SimpleHTTPServer
```

Команда запустит базовый веб-сервер, который откроет доступ к текущему каталогу для любого браузера в сети. По умолчанию он работает на порте 8000, поэтому, если адрес у машины, например, 10.1.2.4, добавьте это значение к браузеру в целевой системе в виде http://10.1.2.4:8000 и тогда сможете перекинуть файл.

ВНИМАНИЕ

В данном случае предполагается, что ваша локальная сеть защищена. Не запускайте эту команду в общедоступной или любой другой небезопасной сети.

12.2. Утилита rsync

Если вы хотите скопировать больше двух файлов, используйте инструменты, требующие поддержки сервера в месте назначения. Например, вы можете скопировать всю структуру каталогов в другое место с помощью команды `scp -r`, при условии что удаленное место назначения поддерживает SSH и сервер SCP (доступно для Windows и macOS). Мы уже рассматривали этот вариант в главе 10, где *`directory`* — копируемый каталог, *`user`* — учетная запись на удаленной машине, *`remote_host`* — адрес удаленной машины, *`dest_dir`* — целевой каталог на удаленной машине:

```
$ scp -r directory user@remote_host[:dest_dir]
```

Этот метод работает, но он не очень гибкий. В частности, после завершения передачи удаленный хост может не получить точную копию каталога. Если *`directory`* уже существует на удаленной машине и содержит посторонние файлы, все файлы объединяются в одном каталоге.

Если вы планируете регулярно копировать данные, особенно если хотите автоматизировать процесс, вам следует использовать специальную систему синхронизации, которая может выполнять также анализ и проверку. Для этого в Linux чаще всего задействуется утилита `rsync`. Это стандартное средство синхронизации систем, обеспечивающее хорошую производительность и множество полезных способов выполнения передач. В этом разделе мы рассмотрим часть основных режимов работы `rsync` и некоторые ее особенности.

12.2.1. Начало работы с rsync

Чтобы утилита `rsync` начала работать между двумя хостами, необходимо установить программу `rsync` как на исходном, так и на целевом компьютере, а также настроить доступ к одному компьютеру с другого. Самый простой способ передачи файлов — использовать учетную запись удаленной оболочки. Предположим, вы хотите передавать файлы с помощью доступа SSH. При этом не будем забывать, что утилита `rsync` подойдет и для копирования файлов и каталогов на одном компьютере, например из одной файловой системы в другую.

На первый взгляд команда `rsync` почти не отличается от команды `scp`, более того, команду `rsync` можно запускать с теми же аргументами. Например, чтобы скопировать группу файлов *`file1 file2`* ... в свой домашний каталог на *хосте* *`host`*, введите:

```
$ rsync file1 file2 ... host:
```

В любой современной системе утилита `rsync` предполагает, что вы используете SSH для подключения к удаленному хосту. Остерегайтесь следующего 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)
```

Сообщение об ошибке предупреждает, что удаленная оболочка не может найти `rsync` в системе. Если `rsync` находится в удаленной системе, но отсутствует в пути команд для пользователя в ней, примените команду `--rsync-path=`*`path`*, чтобы вручную указать местоположение утилиты (где *`path`* — абсолютный путь к `rsync` на удаленном хосте).

Если имя пользователя на двух хостах различается, добавьте *`user`*`@` к имени удаленного хоста в аргументах команды, где *`user`* — это ваше имя пользователя на хосте *`host`*:

```
$ rsync file1 file2 ... user@host:
```

Если вы не укажете дополнительные параметры, `rsync` скопирует только файлы. А если укажете только описанные ранее параметры и в качестве аргумента — каталог *`dir`*, в командной строке появится следующее сообщение:

```
skipping directory dir
```

Для передачи всей иерархии каталогов, включая символические ссылки, права доступа, режимы и устройства, используйте параметр `-a`. Если хотите скопировать данные в каталог, который отличается от вашего домашнего каталога на удаленном хосте, введите его имя *`dest_dir`* после удаленного хоста *`host`*, как показано далее:

```
$ rsync -a dir host:dest_dir
```

Процесс копирования каталогов может быть сложным, и, если вы не уверены в том, что именно произойдет при передаче файлов, используйте сочетание параметров `-nv`. Параметр `-n` позволяет `rsync` работать в режиме пробного запуска, то есть запускать пробную версию без фактического копирования каких-либо файлов. Параметр `-v` предназначен для режима, в котором отображаются подробные сведения о передаче и задействованных файлах:

```
$ rsync -nva dir host:dest_dir
```

Вывод команды выглядит 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 каталогов

По умолчанию `rsync` копирует файлы и каталоги без учета предыдущего содержимого целевого каталога. Например, если вы перенесли каталог `d`, содержащий файлы `a` и `b`, на машину, на которой уже был файл с именем `d/c`, то после копирования вторая машина будет содержать `d/a`, `d/b` и `d/c`.

Чтобы создать точную копию исходного каталога, необходимо из целевого каталога удалить не существующие в исходном каталоге файлы, например `d/c` в примере. Для этого используйте параметр `--delete`:

```
$ rsync -a --delete dir host:dest_dir
```

ВНИМАНИЕ

Это действие не всегда безопасно, поэтому перед копированием проверьте целевой каталог, чтобы случайно не удалить ничего лишнего. Помните: если вы не уверены в копировании, используйте параметр -nv, чтобы выполнить пробный запуск и точно знать, какие файлы утилита rsync может удалить.

12.2.3. Добавление косой черты

Будьте особенно осторожны при указании каталога в качестве источника для команды `rsync`. Рассмотрим базовую команду, с которой мы работали ранее:

```
$ rsync -a dir host:dest_dir
```

После копирования в системе появится каталог *dir* внутри *dest_dir* на хосте *host*. На рис. 12.1 показан пример того, как `rsync` обрабатывает каталог с файлами с именами `a` и `b`.

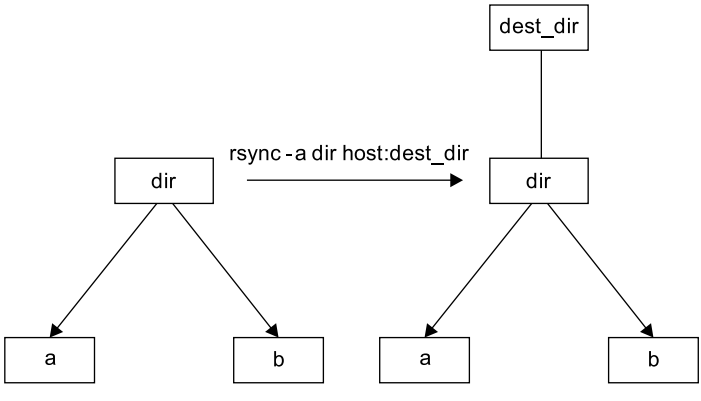


Рис. 12.1. Стандартное копирование rsync

Если добавить косую черту (`/`) к имени источника, она значительно изменит поведение команды:

```
$ rsync -a dir/ host:dest_dir
```

В данном случае `rsync` копирует *все*, что находится внутри *dir*, в каталог *dest_dir* на хосте *host*, фактически не создавая *dir* на целевом хосте. Поэтому можно

рассматривать передачу *dir/* как операцию, аналогичную *cp dir/* dest_dir* в локальной файловой системе.

Предположим, у вас есть каталог `dir`, содержащий файлы `a` и `b` (`dir/a` и `dir/b`). Вы запускаете версию команды с косой чертой, чтобы перенести их в каталог *dest_dir* на хосте *host*:

```
$ rsync -a dir/ host:dest_dir
```

Когда передача завершится, *dest_dir* будет содержать копии файлов `a` и `b`, но без каталога `dir`. Однако, если убрать косую черту `/` от каталога `dir`, в *dest_dir* появятся копии каталога `dir` с файлами `a` и `b` внутри. В результате переноса на удаленном хосте будут находиться файлы и каталоги с именами *dest_dir/dir/a* и *dest_dir/dir/b*. На рис. 12.2 показано, как `rsync` обрабатывает структуру каталогов с рис. 12.1 при использовании косой черты.

Если при передаче файлов и каталогов на удаленный хост случайно добавить косую черту `/` после пути, ошибки не произойдет — вы сможете перейти на удаленный хост, добавить каталог `dir` и поместить все переданные элементы обратно в каталог `dir`. К сожалению, если объединить косую черту `/` с параметром `--delete`, можно легко удалить несвязанные файлы, поэтому будьте с этим внимательнее.

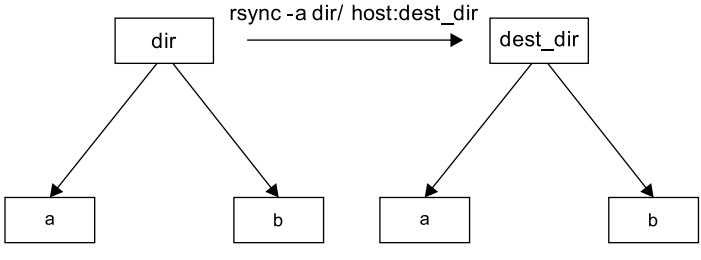


Рис. 12.2. Действие команды rsync с добавлением косой черты

ВНИМАНИЕ

Будьте осторожны с функцией автоматического заполнения имени файла вашей оболочки. Многие оболочки прикрепляют косую черту к именам каталогов после нажатия клавиши Tab.

12.2.4. Исключение файлов и каталогов

Одной из важных особенностей утилиты `rsync` является ее способность исключать файлы и каталоги из процесса передачи. Например, вы хотите перенести локальный каталог с именем `src` на хост и при этом исключить все данные с именем `.git`. Это можно сделать следующим образом:

```
$ rsync -a --exclude=.git src host:
```

Обратите внимание на то, что эта команда исключает все файлы и каталоги с именем `.git`, потому что параметр `--exclude` принимает шаблон, а не абсолютное имя файла. Чтобы исключить один конкретный файл, укажите абсолютный путь, начинающийся с косой черты `/`, как показано далее:

```
$ rsync -a --exclude=/src/.git src host:
```

ПРИМЕЧАНИЕ

Первая косая черта в строке /src/.git в приведенной ранее команде — не корневой каталог вашей системы, а скорее базовый каталог передачи.

Еще несколько советов по поводу того, как исключить определенные элементы (шаблоны) из переноса:

- Добавляйте любое количество параметров `--exclude` в команду.
- Если вы постоянно работаете с одними и теми же шаблонами, помещайте их в обычный текстовый файл (по одному шаблону на строку) и используйте параметр `--exclude-from=`*`file`*, где *`file`* — имя файла шаблонов.
- Чтобы исключить каталоги с именем *`item`*, но включить файлы с этим именем, применяйте косую черту: `--exclude=`*`item`*`/`.
- Шаблон исключения основывается на полном имени файла или каталога и может содержать простые шаблоны поиска (подстановочные знаки). Например, `t*s` соответствует имени `this`, но не соответствует имени `ethers`.
- Если вы исключили каталог или имя файла и обнаружили, что ваш шаблон слишком строг, используйте параметр `--include`, чтобы специально включить другой файл или каталог.

12.2.5. Проверка копирования, меры предосторожности и подробный режим

Для ускорения работы утилита `rsync` использует быструю проверку, чтобы узнать, находятся ли какие-либо файлы из источника передачи в целевом каталоге. При проверке применяется комбинация размера файла и даты его последнего изменения. При первой передаче всей иерархии каталогов на удаленный хост `rsync` видит, что ни одного из файлов нет в месте назначения, и передает все. Проверьте это действие с помощью команды `rsync -n`.

После первого запуска утилиты `rsync` запустите ее снова с параметром `-v`. На этот раз файлы не будут отображаться в списке передачи, потому что все они уже находятся на обоих компьютерах и имеют одинаковые даты изменения.

Если файлы на первом компьютере не идентичны файлам на втором, то `rsync` передает исходные файлы и перезаписывает любые файлы, существующие на втором компьютере. Однако подобное поведение по умолчанию может нанести вред, поскольку нужно убедиться в том, что файлы действительно одинаковы, прежде чем

передавать их. Могут понадобиться и дополнительные меры предосторожности. Перечислим несколько параметров, которые могут пригодиться.

- `--checksum` (сокращенно `-c`) — вычисляет контрольные суммы (в основном уникальные сигнатуры) файлов, чтобы проверить, совпадают ли они. Этот параметр во время передачи потребляет небольшое количество ресурсов ввода-вывода и процессора, но при передаче конфиденциальных данных или файлов с одинаковыми размерами это необходимо.
- `--ignore-existing` — не удаляет файлы, находящиеся на целевом компьютере.
- `--backup` (сокращенно `-b`) — не удаляет файлы, находящиеся на целевом компьютере, а переименовывает их, добавляя ~ к именам перед передачей новых файлов.
- `--suffix=`*`s`* — изменяет суффикс, используемый для `--backup`, с ~ на значение *`s`*.
- `--update` (сокращенно `-u`) — не перезаписывает ни один файл на целевом компьютере, который имеет более позднюю дату, чем соответствующий файл, находящийся в источнике.

Без каких-либо специальных параметров `rsync` выдает вывод только в случае возникновения проблем. Чтобы включить подробный режим, задействуйте команду `rsync -v` или `rsync -vv` для получения еще более подробного вывода. (Можно добавить сколько угодно параметров `v`, однако два — оптимальный вариант.) Чтобы получить полную сводку о передаче данных, используйте команду `rsync --stats`.

12.2.6. Сжатие данных

Для сжатия данных перед передачей многие пользователи предпочитают применять параметр `-z` в сочетании с параметром `-a`:

```
$ rsync -az dir host:dest_dir
```

Сжатие может повысить производительность в определенных ситуациях, например когда вы загружаете большой объем данных через медленное соединение (допустим, по медленному восходящему каналу) или когда между двумя хостами фиксируется продолжительное время ожидания. Однако в быстрой локальной сети две конечные машины могут быть ограничены процессорным временем, необходимым для сжатия и декомпрессии данных, поэтому несжатая передача может происходить быстрее.

12.2.7. Ограничение ширины полосы пропускания

Если выгружать большой объем данных на удаленный хост, довольно легко «засорить» восходящий канал интернет-соединения. Даже если вы не задействуете обычно большую пропускную способность нисходящего канала во время такой передачи, соединение все равно будет казаться довольно медленным, если позволить `rsync` работать максимально быстро. А все потому, что исходящие TCP-пакеты,

например запросы HTTP, будут конкурировать с вашими передачами за пропускную способность на восходящем канале.

Чтобы не попадать в подобную ситуацию, используйте параметр `--bwlimit` — это позволит восходящему каналу взять паузу. Например, для ограничения пропускной способности до 100 000 Кбит/с введите команду

```
$ rsync --bwlimit=100000 -a dir host:dest_dir
```

12.2.8. Передача файлов на ваш компьютер

Команда `rsync` предназначена не только для копирования файлов с вашего локального компьютера на удаленный хост. Она способна и передавать файлы с удаленного компьютера на локальный хост. Для этого нужно указать путь к удаленному хосту и удаленному источнику в качестве первого аргумента в командной строке. Например, чтобы перенести каталог *`src_dir`* удаленной системы *`host`* в каталог *`dest_dir`* локального хоста, выполните следующую команду:

```
$ rsync -a host:src_dir dest_dir
```

ПРИМЕЧАНИЕ

Как упоминалось ранее, можете использовать rsync для дублирования каталогов на своем локальном компьютере: просто опустите параметр host: в обоих аргументах.

12.2.9. Больше информации о rsync

Для копирования множества файлов одновременно чаще всего используется именно утилита `rsync`. Особенно полезно запускать ее в режиме пакетной обработки для копирования одного и того же набора файлов на несколько хостов, поскольку этот режим ускоряет процесс длительной передачи и позволяет возобновить ее после прерывания.

Утилита `rsync` используется также для создания резервных копий. Например, вы можете подключить интернет-хранилище, такое как Amazon S3, к своей системе Linux, а затем применить команду `rsync --delete` для периодической синхронизации файловой системы с сетевым хранилищем. Таким образом вы создадите у себя очень эффективную систему резервного копирования.

Для утилиты `rsync` существует гораздо больше параметров командной строки, чем описано в этом разделе. Чтобы вывести краткий список параметров, запустите команду `rsync --help`. Более подробная информация об утилите имеется на странице руководства `rsync(1)`, а также на домашней странице `rsync` (rsync.samba.org).

12.3. Совместное использование файлов

В вашей сети, скорее всего, имеется не один компьютер с системой Linux — вероятно, их несколько, а это значит, что совместное применение файлов в конце

концов понадобится. В оставшейся части главы мы сначала рассмотрим совместное использование файлов компьютерами с Windows и macOS, и вы узнаете больше о том, как Linux адаптируется к взаимодействию с совершенно чужими средами. А в конце главы поговорим о SSHFS и применении протокола сетевого доступа к файловым системам (Network File System, NFS) в качестве клиента для обмена файлами между компьютерами Linux или доступа к файлам с устройства сетевого хранилища (Network Area Storage, NAS).

12.3.1. Производительность совместного использования файлов

Первое, о чем нужно задуматься в ходе работы с любой системой совместного применения файлов: почему вы выбрали именно ее? В традиционных сетях на базе Unix были две основные причины: удобство и отсутствие локального хранилища. Один пользователь мог войти на одну из нескольких машин в сети, каждая из которых имела доступ к его домашнему каталогу. Это позволяло гораздо экономичнее сконцентрировать хранилище на небольшом количестве централизованных серверов и не покупать и не обслуживать большое количество локальных хранилищ для каждой машины в сети.

Преимущества подобной модели действий нивелируются одним серьезным недостатком, который остается неизменным на протяжении многих лет: производительность сетевого хранилища ниже, чем локального. Проблемы производительности не характерны для некоторых типов данных для совместного использования: например, современное оборудование и сети не имеют проблем с потоковой передачей видео- и аудиоданных с сервера на медиаплеер отчасти потому, что схема доступа к данным очень предсказуема. Сервер, отправляющий данные из большого файла или потока, может предварительно загрузить и эффективно буферизировать данные, поскольку знает, что клиент, скорее всего, будет обращаться к ним последовательно.

Однако если вы выполняете более сложные манипуляции или получаете доступ к множеству разных файлов одновременно, процессор чаще всего будет находиться в процессе ожидания в сети. *Задержка* (latency) — одна из основных причин этого. Это время, необходимое для получения данных из любого случайного (произвольного) доступа к сетевым файлам. Перед отправкой каких-либо данных клиенту сервер должен принять и расшифровать запрос, а затем найти и загрузить данные. Первичные действия — самые медленные и выполняются почти для каждого нового доступа к файлу.

Суть этих рассуждений заключается в следующем: перед использованием файлообменной сети спросите себя, для чего вам это нужно. Если речь идет о больших объемах данных, не требующих частого произвольного доступа, проблем, скорее всего, не возникнет. Но если, например, вы редактируете видео или разрабатываете программную систему любого значительного размера, эффективнее сохранять свои файлы в локальном хранилище.

12.3.2. Безопасность совместного доступа к файлам

Ранее безопасность в протоколах совместного доступа к файлам обычно не была обязательной. И в результате пришлось задумываться о том, как и где вы хотите реализовать совместный доступ к файлам. Если у вас есть какие-либо основания сомневаться в безопасности сети (сетей) между компьютерами, совместно использующими файлы, применяйте как авторизацию/аутентификацию, так и шифрование. Эффективные авторизация и аутентификация означают, что доступ к файлам имеют только стороны с верными учетными данными (и сервер действительно тот, за кого себя выдает), а шифрование гарантирует, что никто не сможет украсть данные файла при его передаче.

Параметры совместного доступа к файлам, которые проще всего настроить, обычно наименее безопасны, и, к сожалению, не существует стандартизированных способов защиты подобных типов доступа. Однако можно приложить усилия для подключения правильных элементов и задействовать такие инструменты, как `stunnel`, IPSec и VPN, которые могут защитить уровни ниже основных протоколов обмена файлами.

12.4. Совместное использование файлов с помощью Samba

Если вы применяете компьютеры с системой Windows, разрешите доступ к файлам и принтерам для системы Linux с помощью стандартного сетевого протокола Windows (Server Message Block, SMB). Система macOS также поддерживает общий доступ к файлам SMB, может использовать и протокол SSHFS, описанный в разделе 12.5.

Стандартный набор программного обеспечения совместного доступа к файлам для Unix называется Samba. Samba не только позволяет компьютерам Windows вашей сети подключаться к системе Linux, но и наоборот: вы можете печатать и получать доступ к файлам на серверах Windows со своего компьютера Linux с помощью клиентского программного обеспечения.

Чтобы настроить сервер Samba, выполните следующие действия:

1. Создайте файл `smb.conf`.
2. Добавьте разделы совместного доступа к файлам в `smb.conf`.
3. Добавьте разделы совместного доступа к принтерам в `smb.conf`.
4. Запустите демоны `Samba nmbd` и `smbd`.

После установки Samba из пакета дистрибутива система должна выполнить эти действия, используя значения по умолчанию для сервера. Однако она не сможет определить, какие конкретные *ресурсы* нужно сделать общедоступными.

ПРИМЕЧАНИЕ

В этой главе мы не будем досконально изучать работу сервера Samba — рассмотрим только то, что позволяет компьютерам Windows в одной подсети видеть автономную машину Linux через сетевое окружение Windows. Существует бесчисленное множество способов настройки сервера Samba из-за большого количества возможностей управления доступом и топологии сети. Чтобы узнать о том, как настроить крупномасштабный сервер, изучите книгу Using Samba, 3-е издание (O'Reilly, 2007), Джеральда Картера, Джея Т. С. и Роберта Экштейна (Gerald Carter, Jay Ts, and Robert Eckstein). Можете также посетить сайт Samba (samba.org).

12.4.1. Настройка сервера

Центральным файлом конфигурации Samba является `smb.conf`, который большинство дистрибутивов размещают в каталоге `etc` (`/etc/samba`). Он может находиться и в каталоге `lib`, например в `/usr/local/samba/lib`, так что, возможно, его придется поискать.

Формат файла `smb.conf` аналогичен формату XDG, который встречается в других конфигурациях (например, формату конфигурации `systemd`), и разбивается на несколько секций, обозначенных квадратными скобками — `[global]` и `[printers]`. Раздел `[global]` в `smb.conf` содержит общие параметры, которые применяются ко всему серверу и ко всем совместно используемым ресурсам. Эти параметры в первую очередь относятся к конфигурации сети и управлению доступом. В примере раздела `[global]` показано, как задать имя сервера, описание и рабочую группу:

```
[global]
# server name
netbios name = name
# server description
server string = My server via Samba
# workgroup
workgroup = MYNETWORK
```

Параметры и их действия:

- `netbios name` — имя сервера. Если опустить его, Samba использует имя хоста Unix. NetBIOS — это интерфейс API, с помощью которого хосты SMB общаются друг с другом;
- `server string` — краткое описание сервера. По умолчанию задействуется номер версии Samba;
- `workgroup` — имя рабочей группы Windows. Если вы используете домен Windows, задайте для параметра его имя.

12.4.2. Контроль доступа к серверу

Добавьте параметры в файл `smb.conf`, чтобы указать, какие машины и пользователи могут получить доступ к серверу Samba. Перечислим несколько из множества

параметров, которые вы можете задать в секции `[global]` и в секциях, управляющих отдельными совместно используемыми ресурсами (рассмотрим далее в этой главе):

- `interfaces` — позволяет серверу Samba прослушивать (принимать подключения) в заданных сетях или интерфейсах, например:

  ```
  interfaces = 10.23.2.0/255.255.255.0
  interfaces = enp0s31f6
  ```

- `bind interfaces only` — в значении `yes` (при помощи параметра `interfaces`) позволяет ограничить доступ только теми машинами, к которым вы можете напрямую подключиться;
- `valid users` — разрешает доступ определенным пользователям, например:

  ```
  valid users = jruser, bill
  ```

- `guest ok` — в значении `true` дает доступ к совместно применяемому ресурсу анонимным пользователям в сети. Задействуйте этот параметр, только если уверены, что сеть частная;
- `browseable` — открывает доступ к совместно используемому ресурсу для просмотра сетевыми браузерами. Если установить для этого параметра значение `no` для каких-либо ресурсов, вы все равно сможете получить доступ к ним на сервере Samba, но для этого необходимо знать их точные имена.

12.4.3. Пароли

Как правило, необходимо разрешать доступ к вашему серверу Samba только с помощью аутентификации по паролю. К сожалению, базовая система паролей в Unix отличается от системы паролей в Windows, поэтому, если не указать сетевые пароли открытым текстом или не проверить их подлинность с помощью сервера домена Windows, придется настроить альтернативную систему паролей. В этом разделе мы рассмотрим, как сделать это с помощью серверной *базы данных Trivial* (Trivial Database, TDB) Samba, которая подходит для небольших сетей.

Следующие строки в секции `[global]` вашего файла `smb.conf` помогут 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 управлять базой данных паролей Samba с помощью команды `smbpasswd`. Параметр `obey pam restrictions` гарантирует, что любой пользователь, меняющий свой пароль с помощью команды `smbpasswd`, должен соблюдать любые правила, которые PAM (подключаемые модули аутентификации, описанные в главе 7) применяет для обычных изменений пароля. Дополнительно к параметру `passdb`

backend можно после двоеточия указать путь к файлу TDB, например tdbsam:/etc/samba/private/passwd.tdb.

ПРИМЕЧАНИЕ

Если у вас есть доступ к домену Windows, можете установить параметр security = domain, чтобы сервер Samba использовал систему аутентификации домена и не применял базу данных паролей. Но чтобы пользователи домена могли получить доступ к машине, на которой запущен Samba, у каждого пользователя домена должна быть локальная учетная запись с тем же именем пользователя на этой машине.

Добавление и удаление пользователей

Чтобы дать пользователю Windows доступ к вашему серверу Samba, необходимо добавить его в базу данных паролей с помощью команды smbpasswd -a:

```
# smbpasswd -a username
```

Параметр *username* команды smbpasswd должен содержать точное имя пользователя в вашей системе Linux.

Как и системная утилита passwd, smbpasswd попросит вас дважды ввести пароль нового пользователя SMB. После того как пароль пройдет все необходимые проверки надежности, smbpasswd подтвердит создание нового пользователя.

Чтобы удалить пользователя, примените для команды smbpasswd параметр -x:

```
# smbpasswd -x username
```

Чтобы временно деактивировать пользователя, задействуйте параметр -d. Параметр -e активирует пользователя:

```
# smbpasswd -d username
# smbpasswd -e username
```

Изменение паролей

Вы можете изменить пароль Samba от имени суперпользователя с помощью команды 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, самостоятельно введя команду smbpasswd в командной строке.

Существует одно слепое пятно в конфигурации сервера, которого следует остерегаться. Будьте внимательны, если встретили в своем файле smb.conf такую строку:

```
unix password sync = yes
```

Она приводит к тому, что команда `smbpasswd` изменяет обычный пароль пользователя вместе с паролем к серверу Samba. В результате пользователь может изменить свой пароль Samba на пароль, отличный от пароля Linux, и из-за этого не сможет больше войти в систему Linux. Более того, некоторые дистрибутивы устанавливают этот параметр по умолчанию в своих пакетах серверов Samba!

12.4.4. Запуск сервера вручную

Беспокоиться о запуске сервера вручную не нужно, если вы установили Samba из пакета дистрибутива. С помощью команды `systemctl --type=service` это можно проверить. Однако если Samba установлен из исходного кода, запустите `nmbd` и `smbd` со следующими аргументами, где *`smb_config_file`* — это полный путь к вашему файлу `smb.conf`:

```
# nmbd -D -s smb_config_file
# smbd -D -s smb_config_file
```

Демон `nmbd` является сервером имен NetBIOS, а `smbd` выполняет обработку запросов на общий доступ. Параметр `-D` указывает режим демона. Если изменить файл `smb.conf` во время работы `smbd`, уведомить демон об изменениях можно с помощью сигнала HUP. Однако лучше позволить `systemd` контролировать сервер, и в этом случае команда `systemctl` выполнит работу самостоятельно.

12.4.5. Диагностика и файлы журналов

Если при запуске сервера Samba что-то пойдет не так, в командной строке появится сообщение об ошибке. Однако диагностические сообщения во время выполнения отправляются в файлы журналов `log.nmbd` и `log.smbd`, которые обычно находятся в каталоге `/var/log`, например в `/var/log/samba`. В нем можно найти и другие файлы журналов, например индивидуальные журналы для каждого отдельного клиента.

12.4.6. Настройка совместного использования файлов

Чтобы предоставить SMB-клиенту совместный доступ к каталогу, добавьте в файл `smb.conf` секцию, как показано далее, где *`label`* — это совместно применяемый ресурс (например, `mydocuments`), а *`path`* — полный путь к каталогу:

```
[label]
path = path
comment = share description
guest ok = no
writable = yes
printable = no
```

Параметры, применяемые при совместном использовании каталогов:

- `guest ok` — значение `yes` разрешает гостевой доступ к ресурсу. Параметр `public` действует так же;
- `writable` — значение `yes` или `true` открывает доступ к ресурсу для чтения и записи. Не открывайте гостевой доступ на чтение и запись;
- `printable` — разрешает вывод на печать. Конечно же, для совместного каталога должен быть установлен в значение `no` или `false`;
- `veto files` — запрещает экспорт любых файлов, соответствующих заданным шаблонам. Необходимо поставить каждый шаблон между косыми чертами (чтобы он выглядел так — `/шаблон/`). В примере далее отображаются объектные файлы, а также любые файлы или каталоги с именем `bin`:

  ```
  veto files = /*.o/bin/
  ```

12.4.7. Домашние каталоги

Добавьте секцию `[homes]` в файл `smb.conf`, если хотите экспортировать домашние каталоги для других пользователей. Она должна выглядеть следующим образом:

```
[homes]
comment = home directories
browseable = no
writable = yes
```

По умолчанию Samba считывает строку `/etc/passwd` вошедшего в систему пользователя, чтобы определить его домашний каталог для секции `[homes]`. Однако если вы не хотите, чтобы Samba действовал подобным образом, то есть хотите сохранить домашние каталоги Windows в месте, отличающемся от домашних каталогов Linux, то замените параметр `path` на `%S`. Пример того, как можно переключить каталог пользователя `[homes]`в `/u/user`:

```
path = /u/%S
```

Samba заменяет текущее имя пользователя на `%S`.

12.4.8. Совместное применение принтеров

Вы можете открыть доступ к принтерам для клиентов Windows, добавив секцию `[printers]` в файл `smb.conf`. Пример того, как выглядит секция при использовании стандартной системы печати Unix CUPS:

```
[printers]
comment = Printers
browseable = yes
printing = CUPS
```

```
path = cups
printable = yes
writable = no
```

Чтобы использовать параметр `printing = CUPS`, настройте и свяжите сервер Samba с библиотекой CUPS.

ПРИМЕЧАНИЕ

В зависимости от конфигурации вы можете разрешить гостевой доступ к своим принтерам с помощью параметра guest ok = yes и не предоставлять пароль или учетную запись Samba всем, кому необходим доступ к принтерам. Например, можно легко ограничить доступ к принтеру одной подсетью с помощью правил брандмауэра.

12.4.9. Клиенты сервера Samba

Клиентская программа Samba `smbclient` может выполнять вывод в удаленные совместные ресурсы Windows и получать к ним доступ. Программа пригодится в среде, в которой необходимо взаимодействовать с серверами Windows, не используя дружественные Unix средства коммуникации.

Для начала работы с `smbclient` с помощью параметра `-L` получите список совместно используемых ресурсов на удаленном сервере с именем *SERVER*:

```
$ smbclient -L -U username SERVER
```

Параметр `-U` *username* не нужен, если ваше имя пользователя Linux совпадает с вашим именем пользователя на *SERVER*.

После выполнения команда `smbclient` запросит пароль. Чтобы получить доступ к совместному ресурсу в качестве гостя, нажмите клавишу Enter, в противном случае введите свой пароль на *SERVER*. После успешного выполнения команды появится список доступных ресурсов, как в примере далее:

```
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` отображает тип совместного ресурса. Обращайте внимание только на типы `Disk` и `Printer` (ресурсы IPC используются для удаленного управления). В приведенном ранее списке есть два совместно применяемых диска и два общих принтера. Название в столбце `Sharename` открывает доступ к каждому ресурсу.

Доступ к файлам в качестве клиента

Если вам нужен единичный доступ к файлам на совместно используемом дисковом ресурсе, примените следующую команду (опять же можете опустить параметр `-U username`, если ваше имя пользователя Linux совпадает с именем пользователя на сервере):

```
$ smbclient -U username '\\SERVER\sharename'
```

После успешного выполнения команды в командной строке появится приглашение, которое показывает, что теперь вы можете передавать файлы:

```
smb: \>
```

В этом режиме передачи файлов работа команды `smbclient` похожа на работу `ftp` Unix и позволяет выполнять следующие действия:

- `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` — изменяет каталог на удаленном сервере на *dir*;
- `lcd localdir` — изменяет текущий локальный каталог на *localdir*;
- `pwd` — выводит текущий каталог на удаленном сервере, включая имена сервера и ресурсов;
- `!command` — выполняет команду *command* на локальном хосте. Используйте команды `!pwd` и `!ls` для определения статуса каталога и файла на локальной стороне;
- `help` — выводит полный список команд.

Файловая система CIFS

Чтобы сделать доступ к файлам на сервере Windows более удобным, можете подключить совместный ресурс непосредственно к своей системе с помощью команды `mount`. Ее синтаксис таков (обратите внимание, используется формат *SERVER:sharename*, а не *\\SERVER\sharename*):

```
# mount -t cifs SERVER:sharename mountpoint -o user=username,pass=password
```

Чтобы вы могли воспользоваться командой `mount`, в вашей системе должны быть установлены утилиты общего протокола файловой системы интернета (Common Internet File System, CIFS). Большинство дистрибутивов поставляют их в виде отдельного пакета утилит.

12.5. Клиентская программа SSHFS

Поскольку системы общего доступа к файлам Windows остаются в стороне, в этом разделе обсудим обмен файлами между системами Linux. Для несложных сценариев удобно задействовать программу SSHFS. Это не что иное, как файловая система пользовательского пространства, которая открывает SSH-соединение и представляет файлы, имеющиеся на другой стороне, в точке подключения на вашем компьютере. Большинство дистрибутивов не устанавливают ее по умолчанию, поэтому нужно вручную установить пакет SSHFS.

Синтаксис SSHFS в командной строке внешне похож на команды SSH, которые мы встречали ранее. Для начала, конечно, необходимо указать совместно используемый каталог *dir* на удаленном хосте *host* и желаемую точку монтирования *mountpoint*:

```
$ sshfs username@host:dir mountpoint
```

Как и в SSH, вы можете опустить параметр *username@*, если имена пользователя те же, что и на удаленном хосте, и не применять параметр *:dir*, если хотите подключить домашний каталог. При необходимости эта команда запрашивает пароль на второй стороне.

Поскольку это файловая система пользовательского пространства, то если вы задействуете ее от имени обычного пользователя, необходимо отключить ее с помощью команды `fusermount`:

```
$ fusermount -u mountpoint
```

Суперпользователь может монтировать эти файловые системы также с помощью команды `umount`. Этот тип файловой системы обычно лучше всего монтировать от имени обычного пользователя для поддержки согласованности прав доступа и безопасности.

Преимущества SSHFS:

- Минимальная настройка. Единственное требование к удаленному хосту — включить протокол `SFTP`, большинство SSH-серверов делают это автоматически по умолчанию.
- Нет зависимости от какой-либо конкретной конфигурации сети. Если вы можете открыть SSH-соединение, SSHFS будет работать независимо от того, находится ли оно в защищенной локальной сети или в небезопасной удаленной сети.

Недостатки SSHFS:

- Снижение производительности. На шифрование, трансляцию и транспортировку уходит много ресурсов (но все не так плохо, как можно подумать).
- Многопользовательские настройки ограничены.

Если для вас преимущества важнее недостатков, определенно стоит попробовать программу SSHFS в действии.

12.6. NFS

Одной из наиболее часто применяемых систем для совместного использования файлов в системах Unix является NFS, и для разных сценариев существует множество разных ее версий. NFS можно обслуживать через TCP и UDP с большим количеством параметров аутентификации и шифрования (к сожалению, очень немногие из них включены по умолчанию). NFS — это довольно обширная тема, особенно из-за такого количества вариантов версий и действий, поэтому мы ограничимся базовой информацией.

Для подключения удаленного каталога *directory* на сервере *server* с NFS используйте тот же базовый синтаксис, что и для подключения каталога CIFS:

```
# mount -t nfs server:directory mountpoint
```

Технически параметр `-t nfs` не нужен, потому что команда `mount` сработает самостоятельно. Список всех параметров можно изучить на странице руководства `nfs(5)`. Параметр `sec` предоставляет несколько различных вариантов обеспечения безопасности. Многие администраторы используют в небольших закрытых сетях управление доступом на основе хоста. Более сложные методы, такие как проверка подлинности на основе Kerberos, требуют дополнительной настройки в других частях системы.

Если вы все чаще работаете с файловыми системами по сети, настройте автомонтирование так, чтобы система монтировала файловые системы только тогда, когда вы хотите их использовать, что предотвратит проблемы с зависимостями при загрузке. Ранее для этого применялся инструмент `automount`, у которого даже появилась более новая версия под названием `amd`. Однако большая часть их функционала теперь заменена юнитом типа `automount` в `systemd`.

СЕРВЕРЫ NFS

Настройка сервера NFS для совместного с другими машинами Linux использования файлов сложнее, чем настройка клиентской стороны. Вам необходимо запустить демоны сервера (`mountd` и `nfsd`) и настроить файл `/etc/exports` так, чтобы он отображал каталоги, к которым вы предоставляете совместный доступ. Однако мы не будем рассматривать серверы NFS в первую очередь потому, что гораздо удобнее совместно использовать хранилище по сети, просто купив устройство NAS для его обработки. Многие из этих устройств основаны на Linux, поэтому они, естественно, будут поддерживать сервер NFS. Поставщики повышают ценность своих устройств NAS, предлагая собственные инструменты администрирования, позволяющие облегчить выполнение утомительных задач, таких как настройка конфигураций RAID и резервное копирование в облако.

12.7. Облачное хранилище

Что касается облачных резервных копий, еще одним вариантом сетевого хранилища файлов являются облачные хранилища AWS S3 или Google. Эти системы не настолько производительны, как хранилища в локальной сети, но они имеют два существенных преимущества: их не нужно обслуживать, и они сами позаботятся о резервном копировании.

Помимо веб- и программных интерфейсов, которые предлагают все поставщики облачных хранилищ, существуют способы подключения большинства видов облачных хранилищ в системе Linux. В отличие от большинства файловых систем, которые мы встречали ранее, почти все они реализованы как интерфейсы FUSE (File System in User Space — файловая система в пользовательском пространстве). У популярных поставщиков облачных хранилищ, например у S3, существует даже несколько вариантов подключений. В этом есть смысл: программа обработки FUSE — это не что иное, как демон пользовательского пространства, который действует как посредник между источником данных и ядром.

В этой книге мы не будем рассматривать особенности настройки клиента облачного хранилища, потому что все они различаются между собой.

12.8. Состояние совместного доступа к файлам в сети

На этом этапе может показаться, что мы недостаточно полно обсудили NFS и совместный доступ к файлам в целом. Так и есть, но лишь настолько, насколько сами по себе системы совместного доступа к файлам неполноценны.

Мы обсудили проблемы производительности и безопасности в подразделах 12.3.1 и 12.3.2. В частности, у NFS базовый уровень безопасности довольно низок, а это требует дополнительных настроек. Системы CIFS в этом отношении несколько лучше, поскольку необходимые уровни шифрования встроены в современное программное обеспечение. Однако проблему ограничения производительности сложно преодолеть, не говоря уже о том, насколько плохо может работать система, если она временно не может получить доступ к своему сетевому хранилищу.

Разработчики несколько раз пытались решить эту проблему. Возможно, наиболее успешной попыткой является файловая система Эндрю (Andrew File System, AFS), разработанная в 1980-х годах. Так почему же не все используют AFS?

На этот вопрос нет однозначного ответа, но многие причины сводятся к отсутствию гибкости в дизайне системы. Например, для механизма безопасности требуется система аутентификации Kerberos. Хотя она доступна повсеместно, но все же не является стандартом в системах Unix и требует серьезной работы по настройке и обслуживанию (для этого необходимо настроить сервер).

Для крупной организации настройка требований по типу Kerberos — не проблема. Именно в таких организациях, например в университетах и финансовых учреждениях, чаще всего применяется система AFS. Но для обычного пользователя нет необходимости настраивать Kerberos, ведь существуют варианты проще, например NFS или CIFS. Такого рода ограничения распространяются даже на Windows: начиная с Windows 2000 Microsoft переключилась на Kerberos для проверки подлинности по умолчанию на своих серверах, но небольшие сети, как правило, не являются доменами Windows с подобными серверами.

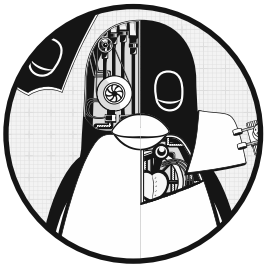
Помимо необходимости аутентификации существует проблема технического характера. Многие клиенты сетевых файловых систем являются кодом ядра, в частности NFS. К сожалению, требования сетевых файловых систем довольно сложны, и это вызывает проблемы. Одна только аутентификация не может происходить в ядре. Реализация клиента ядра также серьезно ограничивает потенциальную базу разработчиков для сетевой файловой системы, затрудняя работу системы в целом. В некоторых случаях клиентский код находится в пространстве пользователя, но под ним всегда лежит определенная настройка ядра.

На данный момент в мире Linux/Unix нет стандартных средств сетевого обмена файлами (по крайней мере, если у вас не крупный сайт или вы не инвестировали большой объем работ в систему). Возможно, вскоре эта ситуация изменится.

Когда провайдеры начали предлагать облачные хранилища, стало ясно, что традиционные формы сетевого обмена файлами не подходят. В облаке методы доступа основаны на механизмах безопасности, таких как TLS, которые позволяют получать доступ к хранилищу без настройки большой системы, такой как Kerberos. Как упоминалось в предыдущем разделе, FUSE предлагает множество параметров доступа к облачному хранилищу. Мы больше не зависим от ядра для какой-либо части клиента — любой вид аутентификации, шифрования или обработки легко выполнить в пространстве пользователя. Все это означает, что в будущем вполне могут появиться проекты совместного применения файлов, обеспечивающие большую гибкость в области безопасности и др., например перевод имен файлов.

13

Пользовательское окружение



Основное внимание в этой книге уделяется тем частям системы Linux, которые лежат в основе серверных процессов и интерактивных пользовательских сеансов. И в конце концов система и пользователь должны где-то встретиться, верно? Файлы запуска системы на этом этапе играют важную роль, поскольку устанавливают значения по умолчанию для оболочки и других интерактивных программ. Они определяют, как ведет себя система, когда пользователь заходит в нее.

Большинство пользователей не обращают внимания на свои файлы запуска, задействуя их в случае необходимости добавить что-то для удобства работы, например псевдоним. Со временем файлы загромождаются ненужными переменными среды и проверками, которые могут привести к неприятным или довольно серьезным проблемам.

Если вы ранее работали с Linux, то могли заметить, что в домашнем каталоге со временем накапливается невероятно большой массив файлов запуска. Их иногда называют *файлами с точкой* (дотфайлами), потому что они практически всегда начинаются с точки (`.`). Благодаря такому формату их можно исключать из отображения по умолчанию в `ls` и большинстве файловых менеджеров. Многие из них создаются автоматически при первом запуске программы, и их не нужно менять. В этой главе в первую очередь рассмотрим файлы запуска оболочки, которые, скорее всего, придется изменить или переписать с нуля. Сначала давайте узнаем, насколько внимательно необходимо относиться к работе с этими файлами.

13.1. Создание файлов запуска

Создавая файл запуска, не забывайте о пользователях системы. Если вы единственный работаете на компьютере, беспокоиться не о чем, потому что все ошибки затронут только вас и вы легко сможете их исправить. Однако если вы создаете файлы запуска, которые должны применяться по умолчанию для всех новых пользователей на компьютере/в сети или которые можно скопировать и задействовать на другой машине, к процессу создания нужно отнестись гораздо внимательнее. Если вы допустили ошибку в файле запуска, который распространили среди 10 пользователей, то ее придется исправлять также 10 раз.

При создании файлов запуска для других пользователей придерживайтесь двух основных принципов:

- **Простота.** Файлов запуска должно быть немного, а сами они — как можно более короткие и простые, чтобы их было легко изменять, но трудно взломать. Каждый элемент в файле запуска — это одна потенциальная проблема.
- **Читаемость.** Делайте в файлах подробные комментарии, чтобы пользователи понимали, что делает каждая часть файла.

13.2. Изменение файлов запуска

Прежде чем вносить изменения в файл запуска, подумайте, действительно ли нужно это делать. Перечислим веские причины для изменения файлов запуска:

- Необходимо изменить приглашение по умолчанию.
- Необходимо изменить критически важное программное обеспечение. (Для этой задачи сначала стоит использовать сценарии-обертки.)
- Существующие файлы запуска повреждены.

Если все перечисленное к вам не относится, то лучше не изменять файлы запуска: иногда они по умолчанию взаимодействуют с другими файлами в `/etc`.

Тем не менее вы не читали бы эту главу, если бы не были заинтересованы в изменении файлов запуска, поэтому рассмотрим важные детали этого процесса.

13.3. Элементы файла запуска оболочки

Что входит в файл запуска оболочки? Некоторые элементы, например путь к команде и настройка приглашения, сразу приходят на ум. Но что именно *должен* содержать путь, как выглядит приглашение и как много данных может содержать файл запуска?

В этом разделе мы рассмотрим основные элементы файла запуска оболочки, от пути к команде, приглашения и псевдонимов до маски прав доступа.

13.3.1. Путь к команде

Наиболее важной частью любого файла запуска оболочки является *путь к команде* (command path). Путь должен включать в себя каталоги, содержащие все приложения и представляющие интерес для обычного пользователя. По крайней мере, путь должен содержать компоненты в следующем порядке:

```
/usr/local/bin
/usr/bin
/bin
```

Этот порядок гарантирует, что пользователь может переопределить стандартные программы по умолчанию на локальные программы, расположенные в `/usr/local`.

Большинство дистрибутивов Linux устанавливают исполняемые файлы почти для всех пакетов пользовательских программ в каталог `/usr/bin`. Иногда могут встречаться исключения, но это случайные различия, которые развились с годами. Например, сейчас игры (`/usr/games`) и графические приложения размещаются в другом месте, поэтому сначала проверьте настройки вашей системы по умолчанию. Также убедитесь, что все программы общего пользования в системе доступны в одном из только что перечисленных каталогов. Если это не так, то система, возможно, вышла из-под контроля. Не изменяйте путь по умолчанию в своей пользовательской среде, чтобы разместить каждый новый каталог установки программного обеспечения. Самый простой способ для этого — применять символические ссылки в каталоге `/usr/local/bin`.

ДОБАВЛЕНИЕ ТОЧКИ К ПУТИ

Существует один небольшой, но спорный элемент пути к командам — точка. Добавление точки (`.`) в путь позволяет запускать программы в текущем каталоге без использования символов `./` перед именем программы.

Может показаться, что это довольно удобно при написании сценариев или компиляции программ, но этого делать не стоит, и вот почему:

- Может возникнуть проблема с безопасностью — нельзя ставить точку в начале пути. Например, злоумышленник может поместить троянский вирус с именем `ls` в архив, который доступен всем в интернете. Даже если точка поставлена в конце пути, система все равно станет уязвимой, ведь злоумышленник будет отслеживать опечатки по типу `sl` или `ks`.
- Это может вызвать путаницу из-за своей непоследовательности. Точка в пути может означать, что поведение команды изменится в соответствии с текущим каталогом.

Многие пользователи создают собственный каталог `bin` для хранения сценариев и программ оболочки, поэтому можете добавить его в начало пути:

```
$HOME/bin
```

ПРИМЕЧАНИЕ

В настоящее время двоичные файлы размещаются в каталоге в $HOME/.local/bin.

Если вы работаете с системными утилитами, например `sysctl`, `fdisk` и `lsmod`, добавьте в свой путь каталоги `sbin`:

```
/usr/local/sbin
/usr/sbin
/sbin
```

13.3.2. Путь к странице руководства

Ранее путь к странице руководства определялся переменной среды `MANPATH`, но сейчас ее не нужно устанавливать, чтобы не переопределить системные значения по умолчанию в файле `/etc/manpath.config`.

13.3.3. Приглашения prompt

Опытные пользователи, как правило, избегают длинных, сложных, бесполезных приглашений. Для сравнения: многие администраторы и дистрибутивы перетаскивают все данные в приглашение по умолчанию. Часто даже приглашения оболочки по умолчанию загромождены бесполезными данными. Например, приглашение `bash` по умолчанию содержит имя оболочки и версию. Приглашение вашей системы должно соответствовать потребностям пользователей: укажите в нем текущий рабочий каталог, имя хоста и имя пользователя, если это необходимо.

Кроме того, не указывайте важные для оболочки символы, например:

```
{ } = & < >
```

ПРИМЕЧАНИЕ

Особенно избегайте символа >, который может вызвать появление ошибочных пустых файлов в текущем каталоге, если случайно скопировать и вставить часть окна оболочки (символ > перенаправляет вывод в файл).

Пример простой настройки приглашения для `bash`, в котором оно заканчивается обычным символом `$` (приглашение `csh` заканчивается символом %):

```
PS1='\u\$ '
```

Выражение `\u` — это выражение, которое оболочка вычисляет для текущего имени пользователя (см. раздел `PROMPTING` на странице руководства `bash(1)`). Список других выражений:

- `\h` — имя хоста (краткая форма, без доменных имен);
- `\!` — номер в истории;
- `\w` — текущий каталог. Вывод полного пути каталога может оказаться длинным, поэтому можно ограничить отображение только последним компонентом, используя выражение `\W`;
- `\$` — символ `$` появляется при запуске от имени учетной записи обычного пользователя, а `#` — при запуске от имени суперпользователя.

13.3.4. Псевдонимы

Одним из наиболее сложных элементов в современных средах пользователя является роль *псевдонимов* — функции оболочки, которая заменяет одну строку на другую перед выполнением команды. Псевдонимы можно эффективно применять в качестве ярлыков, которые позволяют сэкономить время на вводе текста. Однако у них есть несколько недостатков:

- С их помощью может быть непросто передавать аргументы.
- Они не очевидны и могут приводить к путанице — встроенная в оболочку команда `which` может определить псевдоним, но не скажет, откуда он взялся и где был создан.
- Они не используются в подоболочках и неинтерактивных оболочках, не передаются в дочерние оболочки.

Одна из классических ошибок при определении псевдонима — добавление дополнительных аргументов в существующую команду, например создание псевдонима `ls` для `ls -F`. В лучшем случае станет затруднительно удалить аргумент `-F`. В худшем — пользователь не поймет, что применяет аргументы не по умолчанию, что может серьезно повлиять на его работу.

Учитывая эти недостатки, следует избегать псевдонимов — проще написать функцию оболочки или совершенно новый сценарий оболочки. Система может запускать и выполнять оболочки так быстро, что разница между псевдонимом и совершенно новой командой будет незаметна.

Тем не менее псевдонимы используют, если нужно изменить часть среды оболочки. Вы не можете изменить переменную среды с помощью сценария оболочки, потому что сценарии выполняются как подоболочки. Но вместо этого можете определить функции оболочки для выполнения этой задачи.

13.3.5. Маска прав доступа

Как описано в главе 2, встроенная функция `umask` (маска прав доступа) оболочки устанавливает права доступа по умолчанию. Включите команду `umask` в один из файлов запуска, чтобы убедиться, что любая запущенная вами программа создает

файлы с требуемыми правами доступа. Два подходящих для этого варианта таковы:

- `077` — самая строгая маска прав доступа, она не дает другим пользователям доступа к новым файлам и каталогам. Подходит для многопользовательской системы, где нужно запретить другим пользователям просматривать какие-либо ваши файлы. Но когда эта маска установлена по умолчанию, при определенных обстоятельствах могут возникнуть проблемы, например ваши пользователи хотят обмениваться файлами, но не понимают, как правильно задать права доступа (неопытные пользователи могут переводить файлы в режим доступа world-writable — доступен для записи всем);
- `022` — эта маска предоставляет другим пользователям доступ для чтения к новым файлам и каталогам. Полезна в однопользовательской системе, потому что многие демоны, работающие как псевдопользователи, не смогут видеть файлы и каталоги, созданные с помощью более строгой маски `umask 077`.

ПРИМЕЧАНИЕ

Некоторые приложения, особенно почтовые программы, переопределяют umask, изменяя ее значение на 077, потому что запрограммированы считать, что их файлы должен видеть только их владелец и больше никто.

13.4. Порядок и примеры файлов запуска

Мы уже говорили о том, что́ можно помещать в файлы запуска оболочки, теперь пришло время рассмотреть конкретные примеры. Удивительно, но одним из самых сложных и запутанных процессов при создании файлов запуска является определение того, какой из возможных файлов запуска следует использовать. В этом разделе рассмотрим две наиболее популярные оболочки Unix: `bash` и `tcsh`.

13.4.1. Оболочка bash

В оболочке `bash` можно выбирать из файлов запуска `.bash_profile`, `.profile`, `.bash_login` и `.bashrc`. Но какие из них подходят для пути к команде, пути к странице руководства, приглашения, псевдонимов и маски прав доступа? У вас должен быть файл `.bashrc`, сопровождаемый символической ссылкой `.bash_profile`, которая указывает на `.bashrc`. Это необходимо потому, что в системе существует несколько различных типов экземпляров оболочки `bash`, два основных из них — это интерактивные и неинтерактивные.

Нас интересуют только интерактивные оболочки, потому что неинтерактивные (например, запускающие сценарии оболочки) не читают файлы запуска. Интерактивные оболочки используются для выполнения команд с терминала, таких как те, которые мы встречали ранее в книге. Их можно классифицировать как оболочки входа в систему (`login`) и оболочки без входа в систему (`non-login`).

Оболочки входа в систему

Чаще всего оболочка входа в систему — это то, что появляется в окне терминала при первом входе в систему с помощью такой программы, как `/bin/login`. Удаленный вход в систему с помощью SSH также может использовать оболочку входа в систему. Основная суть заключается в том, что оболочка входа в систему является начальной оболочкой. Вы можете определить, является ли оболочка оболочкой входа в систему, запустив команду `echo $0`: если первый символ `-`, то это оболочка входа.

Когда `bash` применяется как оболочка входа в систему, она запускает `/etc/profile`. Затем ищет файлы пользователя — `.bash_profile`, `.bash_login` и `.profile` — и запускает только первый попавшийся из них.

Как бы странно это ни звучало, Linux позволяет запустить неинтерактивную оболочку в качестве оболочки входа в систему, чтобы она активировала файлы запуска. Для этого необходимо запустить оболочку с параметром `-l` или `--login`.

Оболочки без входа в систему

Оболочка без входа в систему — это дополнительная оболочка, которую можно запустить после основного входа в систему. По сути, это просто любая интерактивная оболочка, которая не является оболочкой входа в систему. Оконные системные терминальные программы (`xterm`, терминал GNOME и т. д.) запускают оболочки без входа в систему по умолчанию.

При запуске в качестве оболочки без входа в систему `bash` запускает файл `/etc/bash.bashrc`, а затем — файл пользователя `.bashrc`.

Преимущества двух типов оболочки

Два разных типа файлов запуска появились потому, что раньше пользователи входили в систему через традиционный терминал с помощью оболочки входа, а затем запускали дочерние оболочки без входа в систему с помощью оконных подсистем или программы `screen`. Было решено, что это расточительно — повторно настраивать пользовательскую среду и запускать множество уже запущенных программ для подоболочек без входа в систему. С помощью оболочек входа в систему вы можете запускать необычные команды запуска в таком файле, как `.bash_profile`, оставляя для `.bashrc` только псевдонимы и другие легковесные вещи.

Сейчас большинство пользователей настольных компьютеров входят в систему через графический менеджер (подробнее об этом вы узнаете в следующей главе). Большинство из них начинаются с одной неинтерактивной оболочки входа в систему, что сохраняет выстроенную логику входа. Если оболочки запускаются подругому, необходимо настроить все окружение (путь, путь к страницам руководства

и т. д.) в файле `.bashrc`, иначе ничего из окружения не появится в оболочке окна терминала. Если вы захотите войти в систему через консоль или удаленно, *понадобится* также файл `.bash_profile`, потому что неинтерактивные оболочки входа не используют файл `.bashrc`.

Пример файла .bashrc

Каким нужно создать файл `.bashrc` (который можно использовать также в качестве файла `.bash_profile`), чтобы задействовать его для обоих типов оболочек — со входом и без входа? Приведем один очень простой, но емкий 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
```

В этом файле запуска путь `$HOME/bin` отображается в начале, поэтому исполняемые файлы из этого каталога приоритетнее, чем системные версии. Если вы хотите использовать системные исполняемые файлы, добавьте `/sbin` и `/usr/sbin`.

Как сказано ранее, можно применять файл `.bashrc` как файл `.bash_profile`, используя символическую ссылку, или, что еще понятнее, создать файл `.bash_profile` со следующей командой:

```
. $HOME/.bashrc
```

Проверка того, является ли оболочка входа интерактивной

Если файл `.bashrc` соответствует файлу `.bash_profile`, нет необходимости выполнять дополнительные команды для оболочек входа в систему. Но если вы хотите назначить различные действия для оболочек со входом и без входа, добавьте следующее условие в свой файл `.bashrc`, который проверяет переменную `$-` оболочки на наличие символа `i`:

```
case $- in
 *i*) # далее указываются интерактивные команды
    command
    --пропуск--
    ;;
*)    # далее указываются неинтерактивные команды
    command
    --пропуск--
    ;;
esac
```

13.4.2. Оболочка tcsh

Стандартной оболочкой `csh` практически во всех системах Linux является оболочка `tcsh` (улучшенная оболочка C), которая популяризировала такие функции, как редактирование командной строки и многорежимное заполнение имени файла и команд. Даже если вы не задействуете `tcsh` в качестве новой пользовательской оболочки по умолчанию (по умолчанию применяется `bash`), все равно необходимо предоставить файлы запуска оболочке `tcsh` на случай, если пользователи вашей системы захотят задействовать ее.

Для оболочки `tcsh` нет разницы между оболочками входа в систему и оболочками без входа. При запуске `tcsh` ищет файл `.tcshrc`, а если не найдет, будет искать файл запуска оболочки `csh` — `.cshrc`. Такой порядок обусловлен тем, что файл `.tcshrc` можно использовать для расширений `tcsh`, которые не работают в `csh`. Лучше использовать вместо `.tcshrc` традиционный файл `.cshrc` — маловероятно, что кто-либо когда-нибудь станет применять ваши файлы запуска с `csh`. И если пользователь вашей сети столкнется с оболочкой `csh` в какой-то другой системе, файл `.cshrc` все равно будет работать.

Пример файла .cshrc

Вот пример файла `.cshrc`:

```
# Command path.
setenv PATH $HOME/bin:/usr/local/bin:/usr/bin:/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 написания файлов запуска и выбора значений по умолчанию для новых пользователей — поэкспериментировать. Создайте тестового пользователя с пустым домашним каталогом и не копируйте в него собственные файлы запуска. Напишите новые файлы запуска с нуля.

Когда вы решите, что настройки готовы, войдите в систему как новый тестовый пользователь всеми возможными способами — через консоль, удаленно и др. Протестируйте как можно больше функций, включая работу оконной системы и страницы руководства. После тестирования создайте второго тестового пользователя, скопировав файлы запуска первого. Если все по-прежнему работает, значит, вы создали подходящий набор файлов запуска, который можно распространять среди новых пользователей.

В этом разделе мы опишем разумные значения по умолчанию для новых пользователей.

13.5.1. Параметры оболочки по умолчанию

Для любого нового пользователя в системе Linux по умолчанию должна задействоваться оболочка `bash`, потому что:

- пользователи взаимодействуют с той же оболочкой, которую применяют для написания сценариев оболочки (по многим причинам, о которых мы не будем говорить, оболочка `csh` не подходит для написания сценариев, об этом даже задумываться не стоит);
- оболочка `bash` используется по умолчанию в дистрибутивах Linux;
- оболочка `bash` применяет библиотеку GNU readline для приема вводных данных, поэтому ее интерфейс идентичен интерфейсу многих других инструментов;

- оболочка bash позволяет легко понимать и контролировать перенаправление ввода-вывода и программы обработки файлов.

Однако многие опытные пользователи Unix работают с оболочкой csh или tcsh просто потому, что она им знакома и им сложно с нее переключиться. Конечно, вы можете выбрать любую оболочку, однако эффективнее всего именно оболочка bash. Также установите bash в качестве оболочки по умолчанию для любого нового пользователя в системе. (Пользователи могут изменить свою оболочку с помощью команды chsh в соответствии с собственными предпочтениями.)

ПРИМЕЧАНИЕ

Существует множество других оболочек (rc, ksh, zsh, es и т. д.). Некоторые из них не подходят в качестве оболочек для начинающих, но zsh и fish иногда применяют новые пользователи, которые ищут альтернативную оболочку.

13.5.2. Текстовый редактор

В традиционных системах редактором по умолчанию является vi или emacs. Это единственные редакторы, которые используются (или, по крайней мере, доступны) практически в любой системе Unix. И это означает, что в долгосрочной перспективе они вызывают наименьшее количество проблем у нового пользователя. Однако дистрибутивы Linux часто настраивают в качестве редактора по умолчанию nano, потому что он наиболее прост в применении и подходит для новичков.

Как и в случае с файлами запуска оболочки, избегайте больших файлов запуска редактора по умолчанию. Добавление команды set showmatch в файл запуска .exrc еще никому не повредило: в этом случае vi выделяет соответствующие скобки. Однако избегайте настроек, которые существенно меняют поведение или внешний вид редактора, например, функции showmode, автоматической расстановки отступов и переноса на следующую строку.

13.5.3. Пейджер

Пейджер — это программа, например less, которая отображает текст по одной странице за раз. Для переменной среды PAGER полезно установить по умолчанию значение less.

13.6. Подводные камни в файлах запуска

Избегайте следующих ошибок в файлах запуска:

- Не помещайте графические команды в файл запуска оболочки — не все оболочки работают в графических средах.

- Не устанавливайте переменную среды `DISPLAY` в файле запуска оболочки — это может привести к неправильному поведению вашей графической сессии. (Мы рассмотрим графические среды позднее.)
- Не устанавливайте тип терминала в файле запуска оболочки.
- Не забывайте оставлять подробные комментарии в файлах запуска по умолчанию.
- Не выполняйте в файле запуска команды, которые выводят данные в `stdout`.
- Никогда не устанавливайте переменную `LD_LIBRARY_PATH` в файле запуска оболочки (см. подраздел 15.1.3).

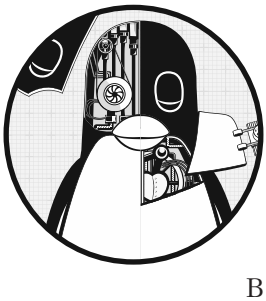
13.7. Далее о файлах запуска

Поскольку в этой книге исследуется только базовая система Linux, мы не будем рассматривать файлы запуска оконной среды. Это обширная тема, потому что диспетчер отображения, который регистрирует пользователя в системе Linux, имеет собственный набор файлов запуска, таких как `.xsession`, `.xinitrc`, и бесконечные комбинации элементов, связанных с GNOME и KDE.

Варианты работы с окнами могут показаться странными, и в Linux нет единого способа запуска оконной среды. В следующей главе мы рассмотрим лишь некоторые из множества способов. Изучать и дополнять файлы, относящиеся к графической среде, довольно интересно, однако не усложняйте жизнь остальным пользователям вашей системы. Принцип простоты файлов запуска оболочки творит чудеса и для файлов запуска графического интерфейса. Скорее всего, у вас вообще не будет необходимости менять файлы запуска графического интерфейса.

14

Краткий обзор рабочего стола Linux. Вывод на печать



В этой главе кратко разберем начальные компоненты, которые встречаются в типичной настольной системе Linux. Настольная среда — один из самых необычных и интересных видов программного обеспечения в системах Linux. А все потому, что в Linux существует так много сред и приложений, которые можно тестировать и использовать.

В отличие от других частей системы Linux, таких как хранилище и сеть, для создания структуры рабочего стола не нужна обширная иерархия уровней. Все дело в том, что каждый компонент выполняет определенную задачу, взаимодействуя с другими компонентами по мере необходимости. Некоторые составляющие, в частности библиотеки для графических наборов инструментов, имеют общие строительные блоки. Их можно представить как простые абстрактные слои системы, но на самом деле их функции так же полезны, как и работа всех остальных компонентов.

В этой главе мы немного обсудим составляющие настольных компьютеров в целом, а две из них рассмотрим подробнее: основную инфраструктуру большинства настольных компьютеров и D-Bus — службу межпроцессной связи, используемую во многих частях системы. В практической части изучим примеры работы нескольких диагностических утилит, которые хотя и не нужны для выполнения ежедневных задач (большинство графических интерфейсов не требуют ввода команд оболочки для взаимодействия с ними), но могут помочь понять основную механику системы и, возможно, даже попутно развлекут. Мы также кратко рассмотрим процесс печати, поскольку настольные рабочие компьютеры часто используют общий принтер.

14.1. Компоненты рабочего стола

Благодаря всевозможным конфигурациям настольные компьютеры Linux обеспечивают для пользователей выполнение множества функций. Большая часть того, что видит пользователь Linux (оформление рабочего стола), создается приложениями или стандартными блоками приложений. Если вам не нравится какое-то конкретное приложение, ему легко найти замену. А если такой альтернативы нет, можете написать приложение сами. У разработчиков Linux, как правило, широкий спектр предпочтений в отношении того, какие функции должен выполнять рабочий стол, что создает множество вариантов выбора.

Чтобы приложения работали сообща, у них должно быть что-то общее.

На момент написания этой главы ядро рабочего стола Linux находится в переходном состоянии. С самого начала и до недавнего времени настольные компьютеры Linux использовали X (X Window System, также известная как Xorg). Сейчас это положение дел меняется — многие дистрибутивы создают оконные системы с помощью набора программного обеспечения, основанного на протоколе Wayland.

Чтобы понять, почему происходят подобные изменения именно в базовых элементах системы, сделаем шаг назад и рассмотрим основы работы графики.

14.1.1. Фреймбуфер

В основе любого механизма графического отображения находится *фреймбуфер (кадровый буфер)* — фрагмент памяти, который графическое оборудование считывает и передает на экран. Каждый пиксел дисплея представлен несколькими отдельными байтами в фреймбуфере, поэтому, если вы хотите изменить внешний вид какого-либо приложения, необходимо записать новые значения в память фреймбуфера.

Одной из задач, которую решает оконная система, является управление записью в фреймбуфер. В любой современной системе окна или наборы окон принадлежат отдельным процессам, выполняющим все обновления графики независимо друг от друга. Итак, если пользователю разрешено передвигать окна и размещать одни поверх других, как приложение узнает, где и какую графику рисовать, и как убедиться, что одному приложению не разрешено перезаписывать графику других окон?

14.1.2. Система X Window

У системы X Window есть сервер (X-сервер), который действует как своего рода ядро рабочего стола для управления всем, от визуализации окон до настройки дисплеев и обработки ввода с устройств, таких как клавиатуры и мыши. X-сервер не контролирует действия и вид элементов. Вместо этого клиентские программы X обрабатывают пользовательский интерфейс. Базовые клиентские приложения X (например, окна терминалов и браузеры) подключаются к X-серверу и запрашивают отрисовку окна. В ответ сервер определяет, где разместить окна и отображать

клиентскую графику, и применяет визуализацию графики в фреймбуфере. Также при необходимости он направляет входные данные клиенту.

Из-за того что X-сервер действует как посредник для всех действий системы, может значительно страдать ее производительность. Кроме того, он выполняет множество функций, которые больше не используются, и сам по себе изобретен довольно давно, еще в 1980-х годах. Тем не менее сервер остался достаточно гибким для того, чтобы вместить множество новых функций, и поэтому применяется до сих пор. Мы рассмотрим основы взаимодействия с системой X Window позже в этой главе.

14.1.3. Протокол Wayland

В отличие от системы X, протокол Wayland спроектирован значительно децентрализованным. У него нет сервера отображения, управляющего фреймбуфером для ряда графических приложений, и нет централизованных прав для визуализации графики. Вместо этого каждое приложение получает собственный буфер памяти (своего рода подфреймбуфер) для своего окна, а часть программного обеспечения, называемая *композитным менеджером окон*, объединяет все буферы клиентов в форму, необходимую для копирования во фреймбуфер экрана. Поскольку эти задачи поддерживаются аппаратным обеспечением, композитный менеджер окон довольно эффективен.

В каком-то смысле графическая модель в Wayland не слишком отличается от того, каким образом уже много лет используется система X. Вместо того чтобы запрашивать поддержку X-сервера, большинство приложений отображают все свои данные в виде растрового изображения, а затем отправляют его на X-сервер. В связи с этим система X обзавелась композитным расширением, которое используется уже несколько лет.

Для передачи ввода в нужное приложение большинство настроек Wayland и многие X-серверы задействуют библиотеку `libinput` (стандартизация событий для клиентов). Протокол Wayland для своей работы не требует подключения библиотеки, однако в настольных системах она почти всегда используется. Мы обсудим библиотеку `libinput` в подразделе 14.3.2.

14.1.4. Менеджеры окон

Основное различие между системами X и Wayland заключается в *менеджере окон* — программе, которая определяет порядок расположения окон на экране и играет центральную роль в работе пользователя. В системе X оконный менеджер — это клиент, который действует как помощник сервера: он добавляет «украшения» окон (например, строки заголовка и кнопки закрытия), обрабатывает события ввода для этих украшений и сообщает серверу, куда перемещать окна.

В Wayland менеджер окон — это нечто, похожее на сервер. Он отвечает за объединение всех буферов клиентского окна в фреймбуфер дисплея и обрабатывает передачу событий устройства ввода. В результате этот менеджер выполняет больше работы,

чем менеджер в X, но большая часть кода менеджера окон может быть одинаковой для различных реализаций.

В обеих системах существует множество реализаций менеджера окон, но в системе X их гораздо больше. Однако большинство популярных менеджеров, таких как Mutter (в GNOME) и Kwin (от KDE), теперь включают в себя также композитную поддержку Wayland. Независимо от базовой технологии маловероятно, что когда-либо появится стандартный оконный менеджер Linux: вкусы и требования пользователей разнообразны и постоянно меняются, поэтому регулярно появляются новые менеджеры окон.

14.1.5. Наборы инструментов

Настольные приложения включают в себя общие элементы, например кнопки и меню, которые называются *виджетами*. Чтобы ускорить разработку и обеспечить общий внешний вид, программисты используют графические наборы инструментов для создания подобных элементов. В операционных системах Windows или macOS поставщик предоставляет общий набор инструментов, и большинство программистов работают именно с ним. В Linux набор инструментов GTK+ является одним из наиболее распространенных, но часто можно встретить и виджеты, основанные на платформе Qt и др.

Наборы инструментов обычно состоят из общих библиотек и вспомогательных файлов, таких как изображения и информация о теме.

14.1.6. Окружение рабочего стола

Наборы инструментов обеспечивают пользователю единый внешний вид рабочего стола, однако некоторые детали требуют определенной степени взаимодействия между различными приложениями. Например, одно приложение может поделиться данными с другим или обновить общую панель уведомлений на рабочем столе. Чтобы это осуществить, наборы инструментов и другие библиотеки объединяются в более крупные пакеты, называемые *окружениями рабочего стола*. GNOME, KDE и Xfce — это распространенные окружения рабочего стола Linux.

В основе большинства окружений рабочего стола лежат наборы инструментов, но для создания единого рабочего стола окружения должны включать также множество вспомогательных файлов, таких как значки и конфигурации, которые составляют общие темы. Все это связано с документами, описывающими правила проектирования, например, как должны отображаться меню и заголовки приложений и как последние должны реагировать на определенные системные события.

14.1.7. Приложения

В верхней части рабочего стола находятся приложения, такие как веб-браузеры и окно терминала. Приложения системы X могут варьироваться от примитивных (например, древняя программа xclock) до сложных (например, веб-браузер Chrome

и пакет LibreOffice). Эти приложения обычно автономны, но при этом часто используют межпроцессную связь, чтобы быть в курсе соответствующих событий. Например, приложение может активироваться при подключении нового устройства хранения данных либо получении нового электронного письма или сообщения. Эта связь осуществляется через шину D-Bus, которая описана в разделе 14.5.

14.2. Wayland или X?

Прежде чем обсуждать практическую часть, нужно определить, какая графическая система у вас используется. Для этого откройте оболочку и проверьте значение переменной окружения `$WAYLAND_DISPLAY`. Если значение выглядит как `wayland-0`, система применяет протокол Wayland. Если значений нет, вы работаете с системой X (могут встречаться исключения, но вы вряд ли столкнетесь с ними при подобной проверке).

Обе системы не исключают друг друга. Если ваша система использует Wayland, она также, скорее всего, задействует сервер совместимости X. К тому же Linux позволяет запустить менеджер окон Wayland внутри системы X, хотя это и выглядит странно (подробнее об этом позже).

14.3. Обзор протокола Wayland

Мы начнем с протокола Wayland, потому что он новее и уже стал стандартом по умолчанию во многих дистрибутивах. Однако отчасти из-за его дизайна и новизны для него существует не так много инструментов, как для системы X. Но разработчики активно стараются это исправить.

Для начала поговорим о том, что такое Wayland. Название Wayland относится к протоколу связи между композитным менеджером окон и графической клиентской программой. Большой пакет ядра Wayland в сети не отыскать, но можно найти библиотеку Wayland, которую большинство клиентов используют для связи с протоколом (по крайней мере, на данный момент).

Существует также *эталонная* реализация композитного менеджера Wayland под названием Weston и несколько связанных с ней клиентов и утилит. *Эталонная* в этом случае значит то, что Weston содержит необходимые функции композитного менеджера, но он не предназначен для широкой публики из-за простого интерфейса. Суть в том, что разработчики композитных менеджеров окон могут взглянуть на исходный код Weston и понять, как правильно реализовать важнейшие функции.

14.3.1. Композитный менеджер окон

Как бы странно это ни звучало, пользователь может не знать, какой композитный менеджер окон Wayland он применяет. Можно попробовать найти название в информации о версии интерфейса, но одного определенного места для ее поиска нет. Однако можно найти запущенный процесс менеджера, отследив доменный сокет

Unix, который он использует для связи с клиентами. Сокет — это отображаемое имя в переменной окружения `WAYLAND_DISPLAY`, которая обычно равна `wayland-0` и находится в файле `/run/user/<uid>`, где `<uid>` — идентификатор пользователя (если это не так, проверьте переменную окружения `$XDG_RUNTIME_DIR`). От имени суперпользователя вы можете найти процесс, прослушивающий этот сокет с помощью команды `ss`, но вывод команды будет выглядеть странно:

```
# ss -xlp | grep wayland-
u_str              LISTEN              0                    128
/run/user/1000/wayland-0 755881
* 0              users:(("gnome-shell",pid=1522,fd=30),("gnome-
shell",pid=1522,fd=28))
```

Однако стоит только присмотреться, как становится понятно, что процесс менеджера окон — это `gnome-shell, PID 1522`. К сожалению, это еще одно косвенное обращение: оболочка GNOME — это подключаемый модуль Mutter, который является композитным менеджером окон, используемым в окружении рабочего стола GNOME. (Называть оболочку GNOME подключаемым модулем — то же самое, что называть Mutter библиотекой.)

ПРИМЕЧАНИЕ

Одним из наиболее необычных аспектов систем Wayland является механизм отрисовки оконных «украшений», например заголовков. В системе X все это делает менеджер окон, а в первых реализациях Wayland отрисовкой занимались клиентские приложения, из-за чего иногда окна на одном экране выглядели по-разному. В настоящее время существует часть протокола под названием XDG-Decoration, через которую менеджер окон и приложение договариваются о том, кто будет отрисовывать «украшения».

В контексте композитного менеджера окон Wayland вы можете рассматривать дисплей как видимое пространство, представленное фреймбуфером. Дисплей может охватывать сразу несколько подключенных мониторов.

Также вы можете запускать более одного композитного менеджера одновременно, хотя требуется это редко. Для этого необходимо запустить компиляторы на отдельных виртуальных терминалах. Для первого менеджера система установит отображаемое имя `wayland-0`, для второго — `wayland-1` и т. д.

Чтобы изучить свой композитный менеджер, используйте команду `weston-info`, которая отображает характеристики интерфейсов, доступных менеджеру. Однако эта команда не отобразит много данных вдобавок к информации на вашем дисплее и некоторых устройствах ввода.

14.3.2. Библиотека libinput

Чтобы получить входные данные от устройств (например, клавиатуры), а также передать от ядра к клиентам, менеджеру окон Wayland необходимо собрать эти входные данные и направить их соответствующему клиенту в стандартизированной

форме. Библиотека `libinput` осуществляет поддержку, необходимую для сбора входных данных с различных устройств ядра `/dev/input` и их обработки. Менеджер Wayland не просто передает событие ввода как есть — перед отправкой клиенту он преобразует его в протокол Wayland.

Сама по себе библиотека `libinput` не особо интересна, однако она поставляется с небольшой утилитой, также называемой `libinput`, которая позволяет проверять устройства ввода и события в том виде, в каком они представлены ядром. Попробуйте выполнить следующую команду, чтобы просмотреть доступные устройства ввода (скорее всего, вывод будет длинный):

```
# libinput list-devices
--пропуск--
Device:           Cypress USB Keyboard
Kernel:           /dev/input/event3
Group:            6
Seat:             seat0, default
Capabilities:     keyboard
Tap-to-click:     n/a
Tap-and-drag:     n/a
Tap drag lock:    n/a
Left-handed:      n/a
--пропуск--
```

В этой части вывода отображаются тип устройства (`Device`) — клавиатура (`Keyboard`) и расположение устройства `evdev` ядра Kernel (`/dev/input/event3`). Устройство появляется, когда вы прослушиваете подобные события:

```
# libinput debug-events --show-keycodes
-event2   DEVICE_ADDED     Power Button
                       seat0 default group1 cap:k
--пропуск--
-event5   DEVICE_ADDED     Logitech T400
                       seat0 default group5 cap:kp left scroll-nat scroll-button
-event3   DEVICE_ADDED     Cypress USB  Keyboard
                       seat0 default group6 cap:k
--пропуск--
 event3   KEYBOARD_KEY      +1.04s      KEY_H (35) pressed
 event3   KEYBOARD_KEY      +1.10s      KEY_H (35) released
 event3   KEYBOARD_KEY      +3.06s      KEY_I (23) pressed
 event3   KEYBOARD_KEY      +3.16s      KEY_I (23) released
```

При выполнении этой команды переместите указатель мыши и нажмите несколько клавиш. Появятся новые выходные данные, описывающие эти события. Помните, что библиотека `libinput` — это всего лишь система для записи событий ядра. Она используется не только в Wayland, но и в системе X Window.

14.3.3. Совместимость X и Wayland

Перед тем как обсуждать систему X Window в целом, изучим ее совместимость с Wayland. Множество полезных приложений поддерживают только систему X,

что затрудняет переход на протокол Wayland. Существует два варианта, как обойти это ограничение.

В первом варианте можно создать собственное приложение Wayland. Большинство графических приложений, работающих на X, уже используют наборы инструментов, похожие на наборы в GNOME и KDE. Благодаря тому что поддержка Wayland в эти наборы инструментов уже добавлена, легче переделать приложение X в приложение Wayland. Разработчики должны поддерживать оформление окон и конфигурации устройств ввода, а также разбираться с редкими случайными зависимостями библиотек X в приложениях. Для многих основных приложений все это уже сделано.

Второй вариант — запустить приложение X через уровень совместимости в Wayland с помощью всего X-сервера, работающего в качестве клиента Wayland. Такой сервер называется Xwayland и является просто еще одним слоем под X-клиентами, запускаемым по умолчанию большинством последовательностей запуска композитного менеджера. Серверу Xwayland необходимо переводить входящие события и поддерживать свои буферы окон отдельно. Дополнительный посредник, подобный Xwayland, всегда немного (не существенно) замедляет работу системы.

В обратную сторону этот способ не сработает. Нельзя запускать клиенты Wayland на X таким же образом (теоретически, можно написать подобную систему, но в ней нет особого смысла). Однако вы можете запустить композитный менеджер внутри окна X. Например, если используете X, запустите команду `weston` в командной строке, чтобы вызвать композитный менеджер. Откройте окно терминала и любое другое приложение Wayland в менеджере, и сможете запускать X-клиенты внутри композитного менеджера, если сервер Xwayland запущен правильно.

Однако если запустить композитный менеджер и вернуться к обычному сеансу X, некоторые утилиты начнут работать по-другому: например, они могут отобразиться в окне менеджера, а не в окне X. Причина в том, что многие приложения в таких системах, как GNOME и KDE, теперь создаются с поддержкой как X, так и Wayland. Сначала приложение будет искать менеджер Wayland, и по умолчанию код в `libwayland` будет искать установки по умолчанию `wayland-0`, если переменная окружения `WAYLAND_DISPLAY` не задана. Если приложение найдет работающий композитный менеджер, то начнет использовать его. Чтобы избежать путаницы, не запускайте композитный менеджер внутри системы X или одновременно с X-сервером.

14.4. Обзор системы X Window

В отличие от систем на базе Wayland, система X Window (www.x.org) всегда была крупной — с базовым дистрибутивом, включающим сервер X, клиентов и библиотеки их поддержки. В связи с появлением настольных сред, таких как GNOME и KDE, роль X со временем изменилась, и теперь внимание уделяется основному серверу, который управляет устройствами визуализации и ввода, а также упрощенной клиентской библиотекой.

X-сервер довольно легко найти в системе — он называется X или Xorg. Проверьте его наличие в списке процессов — обычно он используется с рядом параметров, как в примере:

```
Xorg -core :0 -seat seat0 -auth /var/run/lightdm/root/:0 -nolisten tcp vt7
-novtswitch
```

Значение `:0` называется *X display* (дисплейным менеджером X) — это идентификатор, представляющий один или несколько мониторов, к которым можно получить доступ с помощью обычной клавиатуры и/или мыши. Обычно дисплей соответствует только одному монитору, подключенному к вашему компьютеру, но вы можете разместить несколько мониторов под одним дисплеем. Для процессов, запущенных в сеансе X, переменной среды `DISPLAY` присвоен идентификатор дисплея.

ПРИМЕЧАНИЕ

Дисплеи можно дополнительно разделить на экраны, например со значениями :0.0 и :0.1, но это довольно редкое явление, потому что расширения для системы X, такие как RandR, могут объединять несколько мониторов в один виртуальный экран большего размера.

В Linux X-сервер работает на виртуальном терминале. В приведенном ранее примере аргумент `vt7` запущен в `/dev/tty7` (обычно сервер запускается на первом доступном виртуальном терминале). Вы можете запускать несколько X-серверов одновременно в Linux на отдельных виртуальных терминалах, причем каждый сервер будет иметь уникальный идентификатор дисплея. Переключаться между серверами можно с помощью комбинации клавиш Ctrl+Alt+Fn или команды `chvt`.

14.4.1. Дисплейные менеджеры

Не нужно запускать X-сервер через командную строку, потому что запуск сервера не определяет клиентов, которые должны на нем работать. Если вы запустите только сервер, появится пустой экран. Самый распространенный способ запустить X-сервер — это использовать *дисплейный менеджер*, программу, которая запускает сервер и выводит на экран окно входа в систему. При входе в систему дисплейный менеджер запускает набор клиентов, таких как оконный менеджер и файловый менеджер.

Существует множество различных дисплейных менеджеров, к примеру `gdm` (для GNOME) и `kdm` (для KDE). Менеджер `lightdm` в списке аргументов вызова X-сервера из приведенного ранее примера — это кроссплатформенный дисплейный менеджер, предназначенный для запуска сеансов GNOME или KDE.

Если вы все равно хотите запустить сеанс X с виртуальной консоли, а не через дисплейный менеджер, запустите команду `startx` или `xinit`. Однако этот сеанс будет очень простым и совершенно не похожим на сеанс дисплейного менеджера, потому что их механика и файлы запуска различаются.

14.4.2. Сетевая прозрачность

Одной из особенностей X является сетевая прозрачность. Поскольку клиенты общаются с сервером по протоколу, можно запускать клиенты на сервер, работающий на другой машине, непосредственно по сети, при этом X-сервер прослушивает TCP-соединения на порте 6000. Клиенты, подключающиеся к нему, могут проходить проверку подлинности, а затем отправлять окна на сервер.

К сожалению, этот метод обычно не обеспечивает никакого шифрования, из-за чего оказывается небезопасным. Чтобы избежать этого, большинство дистрибутивов теперь отключают сетевой прослушиватель X-сервера (параметр `-nolisten tcp` для сервера). Однако вы все равно можете запускать X-клиенты с удаленной машины с помощью туннелирования SSH, как описано в главе 10, подключив сокет домена Unix X-сервера к сокету на удаленной машине.

ПРИМЕЧАНИЕ

Не существует простого способа удаленно запустить Wayland, потому что у клиентов есть собственная память экрана, к которой композитный менеджер должен получить прямой доступ. Однако многие новые системы, такие как RDP (Remote Desktop Protocol — протокол удаленного рабочего стола), могут работать совместно с композитным менеджером, чтобы подключаться удаленно.

14.4.3. Способы исследования X-клиентов

Обычно командная строка для работы с графическим пользовательским интерфейсом не применяется, однако существует несколько утилит, которые позволяют изучить части системы X Window. В частности, можно инспектировать клиенты по мере их запуска.

Одни из самых простых инструментов — `xwininfo`. При запуске без каких-либо аргументов он просит вас щелкнуть на окне:

```
$ xwininfo
xwininfo: Please select the window about which you
          would like information by clicking the
          mouse in that window.
[Перевод: Пожалуйста, выберите окно, о котором вы хотите получить информацию,
          щелкнув на нем мышью]
```

Далее он выводит список сведений об окне — его местоположение и размер:

```
xwininfo: Window id: 0x5400024 "xterm"

  Absolute upper-left X: 1075
  Absolute upper-left Y: 594
--пропуск—
```

Обратите внимание на идентификатор окна `window id` в примере. X-сервер и менеджеры окон используют его для отслеживания окон. Чтобы вывести список всех идентификаторов окон и клиентов, задействуйте команду `xlsclients -l`.

14.4.4. События на сервере (X events)

X-клиенты получают входящие данные и другую информацию о состоянии сервера через систему событий. События X работают так же, как и другие асинхронные события межпроцессного взаимодействия, например udev и D-Bus. Сервер X получает информацию из источника, к примеру от устройства ввода, а затем перераспределяет этот ввод в виде события любому заинтересованному клиенту X.

Вы можете экспериментировать с событиями с помощью команды `xev`. При ее запуске появится новое окно, можно щелкнуть на нем мышью и ввести команду. При этом `xev` генерирует выходные данные, описывающие события X, которые он получает от сервера. Пример вывода для события движения мыши:

```
$ xev
--пропуск--
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 значения в скобках. Первая пара представляет координаты `x` и `y` указателя мыши внутри окна, а вторая (`root:`) — расположение указателя на всем дисплее.

Другие события низкого уровня — это нажатия клавиш и кнопок, а события уровнем выше отображают, появился ли указатель мыши в окне или переместился него, использует ли менеджер окон текущее окно или переключился на другое и т. д. Например, вот события выхода `LeaveNotify event` и потери фокусировки `FocusOut event`:

```
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
```

Чаще всего команда `xev` используется для извлечения кодов клавиш `keycodes` и символов клавиш `key symbols` для разных клавиатур при настройке раскладки клавиатуры. Результат нажатия клавиши `L` — `KeyPress event`, код ключа `keycode` здесь `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
```

Вы также можете прикрепить команду `xev` к существующему идентификатору окна с помощью параметра `-id` *`id`*. Замените *`id`* подходящим идентификатором из `xwininfo` (это будет шестнадцатеричное число, начинающееся с `0x`).

14.4.5. X-ввод и настройки предпочтений

Одна из наиболее непростых характеристик системы X заключается в том, что существует несколько способов настройки ее под себя, и некоторые могут не сработать. Например, одной из распространенных настроек клавиатуры в системах Linux является переназначение клавиши Caps Lock на клавишу Ctrl. Существуют разные варианты — от внесения небольших изменений с помощью старой команды `xmodmap` до использования совершенно новой раскладки клавиатуры с помощью утилиты `setxkbmap`. Как узнать, какой из способов (если они есть) применить? Для этого необходимо знать и понимать, какие части системы должны участвовать в процессе, но определить это довольно сложно. Имейте в виду, что окружение рабочего стола может предоставлять собственные настройки и переопределения. С учетом сказанного рассмотрим способы управления настройками системы.

Устройства ввода (общие)

X-сервер использует расширение X Input Extention для управления вводом с множества различных устройств. Существует два основных типа устройств ввода: клавиатура и указатель (мышь), и вы можете подключить столько устройств, сколько захотите. Для одновременной обработки нескольких устройств одного и того же типа расширение X-input создает виртуальное основное устройство, которое перенаправляет ввод устройства на сервер X.

Чтобы узнать настройки устройства на своем компьютере, выполните 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)]
```

С каждым устройством связан идентификатор, который можно использовать с `xinput` и другими командами. В приведенном выводе идентификаторы `id 2` и `id 3` являются основными устройствами, а `id 8` и `id 9` — реальными устройствами. Обратите внимание на то, что кнопки питания `Power Button` на устройстве также рассматриваются как устройства ввода `X`.

Большинство клиентов X прослушивают ввод от основных устройств, потому что им не нужно знать, какое конкретное устройство инициирует событие. На самом деле большинство клиентов ничего не знают о расширении X Input Extention. Однако клиент может использовать расширение для выделения конкретного устройства.

Каждое устройство имеет набор связанных свойств. Чтобы вывести их, введите команду `xinput` с номером устройства:

```
$ 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
--пропуск—
```

Некоторые свойства можно менять с помощью параметра `--set-prop`. Дополнительную информацию найдете на странице руководства `xinput(1)`.

Мышь

Настройками, связанными с устройством, можно управлять с помощью команды `xinput`. Многие из наиболее полезных параметров относятся к настройке мыши (указателя pointer). Вы можете изменить многие настройки непосредственно в качестве свойств, но проще использовать специализированные параметры `--set-ptr-feedback` и `--set-button-map` для команды `xinput`. Например, чтобы изменить порядок кнопок (удобно для левшей) на трехкнопочной мыши, подключенной к устройству *`dev`*, выполните следующую команду:

```
$ xinput --set-button-map dev 3 2 1
```

Клавиатура

Довольно сложно интегрировать множество различных раскладок клавиатуры в любую оконную систему. У системы X всегда была внутренняя возможность отображения клавиатуры в своем основном протоколе, которым можно управлять с помощью команды `xmodmap`. Однако любая новая система использует XKB (расширение клавиатуры X).

Разобраться в работе XKB непросто, и многие пользователи до сих пор применяют `xmodmap`, чтобы быстро вносить изменения. Основная идея XKB заключается в том, что вы можете определить раскладку клавиатуры, скомпилировать ее с помощью команды `xkbcomp`, а затем загрузить и активировать эту карту на сервере X с помощью команды `setxkbmap`. Эта система имеет две особенно интересные особенности:

- может добавить частичные раскладки в дополнение к существующим раскладкам. Это особенно удобно для таких задач, как замена клавиши Caps Lock на клавишу Ctrl, и используется многими графическими настройками клавиатуры в окружениях рабочего стола;
- может настраивать отдельные раскладки для каждой подключенной клавиатуры.

Фон рабочего стола

Корневое окно (root window) X-сервера является фоном дисплея. Старая команда X `xsetroot` позволяет задавать цвет фона и другие характеристики корневого окна, но не действует на большинстве машин, потому что окно никогда не отображается. Вместо этого большинство окружений рабочего стола размещают большое окно позади всех других окон, чтобы включить функции по типу «активных обоев» и просмотра файлов на рабочем столе. Также существуют способы изменить фон из командной строки (например, с помощью команды `gsettings` в некоторых установках GNOME), но если вы действительно хотите это сделать, значит, у вас очень много свободного времени.

Команда xset

Самой старой командой настройки, вероятно, является `xset`. Она больше не используется, но ее можно быстро запустить с параметром `q`, чтобы вывести статус нескольких функций. Возможно, наиболее полезными являются настройки заставки и DPMS (Display Power Manager Signaling — сигнализации управления энергопотребления дисплея).

14.5. Шина D-Bus

Одной из наиболее важных разработок для рабочего стола Linux является *механизм рабочего стола D-Bus* (Desktop Bus) — система передачи сообщений. D-Bus служит важным механизмом межпроцессной связи, который позволяет настольным приложениям взаимодействовать друг с другом и который большинство систем Linux используют для уведомления процессов о системных событиях, например вставке USB-накопителя.

Механизм D-Bus содержит библиотеку, которая стандартизирует межпроцессное взаимодействие с протоколом и поддерживает функции для любых двух процессов, которые могут взаимодействовать друг с другом. Сама по себе эта библиотека

предлагает только необычную версию средств IPC, например доменные сокеты Unix. Максимальную пользу приносит центральный узел D-Bus — `dbus-daemon`.

Процессы, которым необходимо реагировать на события, могут подключаться к `dbus-daemon` и регистрироваться для получения определенных видов событий. Само подключение процессов также создает события. Например, процесс `udisks-daemon` отслеживает udev на предмет событий на диске и отправляет их в `dbus-daemon`, который затем повторно передает события всем заинтересованным приложениям.

14.5.1. Системные и сеансовые экземпляры

Механизм D-Bus стал неотъемлемой частью системы Linux и теперь выходит за рамки рабочего стола. Например, как `systemd`, так и Upstart имеют каналы связи D-Bus. Однако добавление зависимостей к инструментам рабочего стола внутри базовой системы противоречит правилам разработки Linux.

Для решения этой проблемы существует два вида экземпляров (процессов) `dbus-daemon`. Первый — это *системный экземпляр*, который запускается системой init во время загрузки с параметром `--system`. Системный экземпляр обычно запускается как пользователь D-Bus, а его файл конфигурации — это `/etc/dbus-1/system.conf` (эту конфигурацию не следует изменять). Процессы могут подключаться к системному экземпляру через доменный сокет Unix `/var/run/dbus/system_bus_socket`.

Независимый от системы экземпляр D-Bus — это необязательный *сеансовый экземпляр*, который запускается только при запуске сеанса рабочего стола. К нему подключаются настольные приложения.

14.5.2. Мониторинг сообщений D-Bus

Один из лучших способов увидеть разницу между системными и сессионными экземплярами `dbus-daemon` — это отслеживание событий, проходящих по шине. Используйте 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"
```

В примере появилось сообщение о запуске — это значит, что монитор подключен и получил имя. Большой активности при таком запуске не будет, потому что системный экземпляр чаще всего выполняет немного задач.

Чтобы увидеть какие-либо изменения, попробуйте подключить USB-накопитель.

Для сравнения: экземпляры сеансов совершают гораздо больше действий. Предположим, что вы вошли в сеанс рабочего стола, введите следующую команду:

```
$ dbus-monitor --session
```

Далее попробуйте использовать приложение рабочего стола, например файловый менеджер: если ваш рабочий стол поддерживает D-Bus, появится огромное количество сообщений о различных изменениях. Имейте в виду, что не все приложения будут выдавать сообщения.

14.6. Печать

Печать документа в Linux — это многоступенчатый процесс.

1. Программа печати преобразует документ в форму PostScript. Это необязательный шаг.
2. Программа отправляет документ на сервер печати.
3. Сервер печати получает документ и помещает его в очередь на печать.
4. Когда приходит черед документа в очереди, сервер печати отправляет его в фильтр печати.
5. Если документ не в форме PostScript, фильтр печати может выполнить преобразование.
6. Если принтер назначения не понимает форму PostScript, драйвер принтера преобразует документ в формат, совместимый с принтером.
7. Драйвер принтера добавляет в документ дополнительные инструкции, такие как параметры лотка для бумаги и двусторонней печати.
8. Сервер печати использует серверную часть для отправки документа на принтер.

Самая запутанная часть этого процесса заключается в форме PostScript. На самом деле PostScript — это язык программирования, поэтому при печати файла с его помощью вы, по сути, отправляете на принтер программу. PostScript служит стандартом для печати в Unix-подобных системах во многом так же, как формат `.tar` служит стандартом архивирования. (Некоторые приложения используют вывод в формате PDF, но его довольно легко конвертировать.)

Мы подробнее поговорим о формате печати, но сначала рассмотрим систему очереди на печать.

14.6.1. Сервер печати CUPS

Стандартная система печати в Linux — это сервер CUPS (www.cups.org), который является той же системой, которая используется в macOS. Демон сервера CUPS называется `cupsd`, и в качестве простого клиента для отправки ему файлов можно применять команду `lpr`.

Одной из важных особенностей CUPS является то, что он реализует *протокол межсетевой печати* (Internet Print Protocol, IPP) — систему, которая позволяет передавать

файлы, подобные HTTP, между клиентами и серверами на TCP-порте 631. На самом деле, если в системе запущен сервер CUPS, вы сможете подключиться к `http://localhost:631/`, чтобы просмотреть текущую конфигурацию и проверить все задания принтера. Большинство сетевых принтеров и серверов печати поддерживают как IPP, так и Windows, что значительно упрощает настройку удаленных принтеров.

Управлять системой из веб-интерфейса не очень безопасно из-за ее настройки по умолчанию. Для добавления и изменения принтеров в дистрибутиве есть графический интерфейс настроек. Эти инструменты управляют файлами конфигурации, которые обычно находятся в файле `/etc/cups`. Конфигурация может быть довольно сложной, поэтому лучше позволить этим инструментам выполнять всю работу самостоятельно. Даже если вы столкнетесь с проблемой и вам потребуется выполнить настройку вручную, лучше всего создать принтер с помощью графических инструментов.

14.6.2. Фильтры преобразования форматов и печати

Многие принтеры, включая почти все бюджетные модели, не понимают форматы PostScript и PDF. Для поддержки таких принтеров система печати Linux должна преобразовывать документы в формат, специфичный для данного принтера. CUPS отправляет документ в обработчик растровых изображений (Raster Image Processor, RIP) для создания растрового изображения. RIP почти всегда задействует программу Ghostscript (`gs`) для выполнения большей части работы, но это довольно сложно, потому что растровое изображение должно соответствовать формату принтера. Поэтому драйверы принтера, используемые CUPS, обращаются к файлу определения принтера PostScript (PPD) конкретного устройства, чтобы определить настройки, например разрешение печати и размеры бумаги.

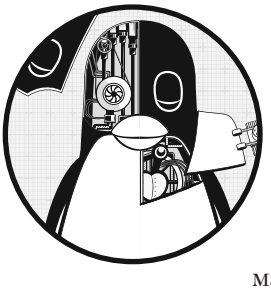
14.7. Дополнительно об окружениях рабочего стола

У окружений рабочего стола Linux есть одна интересная особенность — возможность выбрать, какие части вы хотите использовать, а какие нет. Изучить множество различных проектов со списками рассылки и ссылками на проекты можно на сайте www.freedesktop.org.

Еще одной важной разработкой для рабочего стола Linux является проект с открытым исходным кодом Chromium OS и его аналог Google Chrome OS для компьютеров Chromebook. Это система Linux, которая использует большую часть технологии настольных компьютеров, описанной в этой главе, но сосредоточена на веб-браузерах Chromium/Chrome. Многое из того, что можно найти на традиционном рабочем столе, в Chrome OS не применяется. Экспериментировать с окружениями рабочего стола может быть довольно интересно, однако наше обсуждение на этом закончено. Если эта глава вызвала у вас интерес и вы хотите узнать больше, для начала необходимо понять, как работают инструменты разработчика. Именно этот вопрос мы изучим далее.

15

Инструменты разработки



Система Linux очень популярна у программистов не только из-за огромного количества доступных инструментов и окружений, но и потому что она отлично описана и прозрачна. Чтобы пользоваться инструментами разработки на компьютере Linux, не нужно быть программистом, а ведь эти инструменты играют бóльшую роль в управлении системами Linux, чем в других операционных системах. Достаточно уметь определять инструменты и понимать, как они запускаются.

В этой небольшой главе я собрал много информации по теме, но не обязательно все изучать досконально. Я привожу очень простые примеры, и чтобы повторить их, не нужно знать, как писать код. Вы можете быстро просмотреть эту главу и вернуться к ней позже. Разделяемые библиотеки — вот самая важная тема, которую необходимо изучить. Однако чтобы понять, откуда берутся разделяемые библиотеки, сначала нужно узнать, как создавать программы.

15.1. Компилятор C

Чтобы понять, как создавать программы в системе Linux, необходимо уметь использовать компилятор языка программирования C. Большинство утилит Linux и многие приложения в системах Linux написаны на языках C или C++. В этой главе в основном будут приведены примеры на языке C, но вы сможете перенести их и на C++.

Программы на языке C разрабатываются традиционно: вы пишете и компилируете их, и они запускаются. То есть когда вы пишете программный код на языке C

и хотите его запустить, вы должны *скомпилировать* читаемый человеком код в низкоуровневую двоичную форму, понятную процессору компьютера. Код, который вы создаете, называется *исходным кодом* и может включать в себя множество файлов. Можете сравнить это с языками сценариев, когда не нужно ничего компилировать (их мы обсудим позже).

ПРИМЕЧАНИЕ

В большинстве дистрибутивов нет инструментов, необходимых для компиляции кода на языке C. Если вы не нашли некоторые из описанных здесь инструментов, установите необходимый для сборки пакет build-essential для Debian/Ubuntu или yum groupinstall для Fedora/CentOS. В противном случае поищите пакеты gcc или C compiler.

Компилятор GNU C, `gcc` (часто называемый `cc`), является компилятором C, исполняемым в большинстве систем Unix, однако сейчас набирает популярность более новый компилятор `clang` проекта LLVM. Файлы исходного кода C заканчиваются на `.c`. Взгляните на автономный файл исходного кода C `hello.c`, взятый из книги Брайана Кернигана и Денниса М. Ритчи «Язык программирования Си» (Вильямс, 2019):

```
#include <stdio.h>

int main() {
    printf("Hello, World.\n");
}
```

Сохраните исходный код в файле `hello.c`, а затем запустите компилятор с помощью команды

```
$ cc hello.c
```

В результате вы получите исполняемый файл с именем `a.out`, который можно запускать, как и любой другой исполняемый файл в системе. Однако следует дать исполняемому файлу другое имя, например `hello`. Для этого используйте параметр компилятора `-o`:

```
$ cc -o hello hello.c
```

Для подобных небольших программ компиляция состоит из всего нескольких действий. Может потребоваться добавить дополнительную библиотеку или включить каталог. Однако прежде чем переходить к этим темам, рассмотрим более крупные программы.

15.1.1. Компиляция нескольких исходных файлов

Большинство программ на языке C слишком велики, чтобы уместиться в одном файле исходного кода. Становится сложно организовывать такие огромные файлы

и управлять ими, и у компиляторов могут появиться проблемы с обработкой больших файлов. Поэтому разработчики обычно разделяют исходный код на составные части и помещают каждую в собственный файл.

При компиляции большинства файлов `.c` исполняемый файл не создается сразу. Вместо этого необходимо использовать параметр компилятора `-c` для каждого файла, чтобы создать *объектные файлы*, содержащие двоичный *объектный код*, который в конечном итоге войдет в основной исполняемый файл. Для наглядности предположим, что у нас есть два файла, `main.c` (он запускает программу) и `aux.c` (выполняет фактическую работу), которые выглядят следующим образом:

- Файл `main.c`:

```
void hello_call();

int main() {
    hello_call();
}
```

- Файл `aux.c`:

```
#include <stdio.h>

void hello_call() {
    printf("Hello, World.\n");
}
```

Следующие две команды компилятора выполняют большую часть работы по разработке программы — создают объектные файлы:

```
$ cc -c main.c
$ cc -c aux.c
```

После их выполнения мы получаем два объектных файла: `main.o` и `aux.o`.

Объектный файл — это двоичный файл, который процессор почти понимает, но не до конца. Во-первых, операционная система не знает, как запустить объектный файл, а во-вторых, чтобы создать полноценную программу, необходимо объединить несколько объектных файлов и часть системных библиотек.

Для создания полностью функционирующей исполняемой программы из одного или нескольких объектных файлов необходимо запустить *компоновщик* (linker, команда `ld` в Unix). Однако программисты редко используют `ld` в командной строке, поскольку компилятор C сам запускает программу компоновщика. Чтобы создать исполняемый файл с именем `myprog` из этих двух объектных файлов, для их связывания выполните команду

```
$ cc -o myprog main.o aux.o
```

ПРИМЕЧАНИЕ

Можно скомпилировать каждый исходный файл с помощью отдельной команды, однако предыдущий пример уже намекает на то, что отслеживать их все в процессе компиляции может быть сложно, и эта проблема возрастает по мере увеличения количества исходных файлов. Система make, описанная в разделе 15.2, является традиционным стандартом Unix для управления и автоматизации компиляций. Вы увидите, насколько важна подобная система, когда мы перейдем к управлению файлами, описанными в следующих двух разделах.

Давайте вернемся к файлу `aux.c`. Как говорилось ранее, его код выполняет фактическую работу программы. Может существовать много файлов наподобие объектного файла `aux.o`, которые необходимы для сборки программы. Теперь представьте, что другие приложения хотят применять написанные нами программы. Можем ли мы повторно использовать эти объектные файлы? Вот об этом и поговорим далее.

15.1.2. Связывание с библиотеками

Запуск компилятора с исходным кодом не создает достаточного количества объектного кода, который самостоятельно создал бы полезную исполняемую программу. Для создания полноценных программ необходимы библиотеки. Библиотека C — это набор общих предварительно скомпилированных компонентов, которые можно встроить в программу. На самом деле это ненамного больше, чем набор объектных файлов (наряду с некоторыми файлами заголовков, о которых мы поговорим в подразделе 15.1.4). Например, существует стандартная математическая библиотека, которой пользуются многие исполняемые файлы, поскольку она предоставляет тригонометрические функции и т. п.

Библиотеки задействуются в основном во время компоновки, когда программа компоновщика (`ld`) создает исполняемый файл из объектных файлов. Компоновку с использованием библиотеки часто называют *связыванием с библиотекой*. Именно в этом месте может возникнуть проблема. Например, у вас есть программа, которая задействует библиотеку `curses`, но вы не связали их с помощью компилятора. Из-за этого компоновщик выведет ошибку, как в примере далее:

```
badobject.o(.text+0x28): undefined reference to 'initscr'
```

Наиболее важные части этих сообщений об ошибках выделены жирным шрифтом. Проверив объектный файл `badobject.o`, программа компоновщика не смогла найти функцию, выделенную жирным шрифтом, и, следовательно, не смогла создать исполняемый файл. В данном примере видно, что библиотека не связана, потому что отсутствует функция `initscr()`. Если поискать в интернете имя функции, почти всегда можно найти страницу руководства или другую ссылку на нужную библиотеку.

ПРИМЕЧАНИЕ

Ошибка с ненайденной ссылкой (undefined reference) не всегда означает, что компилятору не хватает библиотеки. В команде компоновки может отсутствовать один из объектных файлов программы. Можно легко отличить библиотечные функции от функций в объектных файлах, потому что вы узнаете те, которые написали сами, или, по крайней мере, сможете их найти.

Чтобы устранить эту проблему, сначала необходимо найти нужную библиотеку, а затем использовать параметр компилятора `-l` для связи с ней. Библиотеки разбросаны по всей системе, хотя большинство находятся в подкаталоге `lib` (`/usr/lib` — системное расположение по умолчанию). В предыдущем примере основным файлом библиотеки является `libcurses.a`, поэтому имя библиотеки — `curses`. Собрав все это вместе, можно связать программу следующим образом:

```
$ cc -o badobject badobject.o -lcurses
```

Необходимо сообщать компоновщику о нестандартном расположении библиотек, параметр для этого — `-L`. Предположим, что программе `badobject` требуется библиотека `libcrud.a` в каталоге `/usr/junk/lib`. Чтобы скомпилировать и создать исполняемый файл, используйте команду, показанную далее:

```
$ cc -o badobject badobject.o -lcurses -L/usr/junk/lib -lcrud
```

ПРИМЕЧАНИЕ

Если вы хотите найти в библиотеке определенную функцию, используйте команду nm с фильтром символов --defined-only. Будьте готовы к большому выводу. Например, попробуйте выполнить команду nm --defined-only libcurses.a. Во многих дистрибутивах вы также можете использовать команду less для просмотра содержимого библиотеки. (Возможно, потребуется команда locate, чтобы найти libcurses.a: многие дистрибутивы сейчас помещают библиотеки в подкаталоги /usr/lib, зависящие от архитектуры, например, в /usr/lib/x86_64-linux-gnu/.)

В системе есть библиотека, называемая *стандартной библиотекой C*. Она содержит фундаментальные компоненты, которые считаются частью языка программирования C. Ее основной файл — `libc.a`. При компиляции программы эта библиотека всегда используется по умолчанию, если специально не исключить ее. Большинство программ в системе работают с разделяемой версией этой библиотеки, поэтому давайте обсудим, что это такое.

15.1.3. Разделяемые библиотеки

Файл библиотеки, заканчивающийся на `.a` (например, `libcurses.a`), называется *статической библиотекой*. Когда вы связываете программу со статической библиотекой, компоновщик копирует необходимый машинный код из файла библиотеки в ваш исполняемый файл. Как только это произойдет, конечному

исполняемому файлу больше не нужен будет файл библиотеки при запуске. Также, поскольку у вашего исполняемого файла есть собственная копия кода библиотеки, поведение исполняемого файла не изменится, если изменить сам файл `.a`.

Однако размеры статических библиотек постоянно увеличиваются, как и их количество, а сами они начинают потреблять больше дискового пространства и памяти. Кроме того, если определенную статическую библиотеку признают неподходящей или небезопасной, станет невозможно изменить исполняемые файлы, которые были связаны с ней, за исключением перекомпиляции каждого исполняемого файла.

Решение всех этих проблем — это *разделяемые библиотеки*. Связывание программы с разделяемой библиотекой не копирует код в конечный исполняемый файл — оно просто добавляет ссылки на имена, содержащиеся в коде файла библиотеки. При запуске программы система загружает код библиотеки в область памяти процесса только при необходимости. Многие процессы могут совместно использовать один и тот же код общей библиотеки в памяти. И если вам нужно немного изменить код библиотеки, это можно сделать без перекомпиляции каких-либо программ. При обновлении программного обеспечения в дистрибутиве Linux обновляемые пакеты могут включать разделяемые библиотеки. Когда менеджер обновлений хочет перезагрузить компьютер, это может значить, что он проверяет, что каждая часть вашей системы использует новые версии разделяемых библиотек.

У разделяемых библиотек есть недостатки: сложное управление и сложный процесс связывания. Однако можно легко научиться управлять такими библиотеками, если знать четыре важные вещи:

- Как составить список разделяемых библиотек, необходимых исполняемому файлу.
- Как исполняемый файл ищет разделяемые библиотеки.
- Как связать программу с разделяемой библиотекой.
- Как избежать распространенных ошибок при использовании разделяемых библиотек.

В следующих разделах мы рассмотрим, как применять и поддерживать разделяемые библиотеки в вашей системе. Если хотите узнать, как работают разделяемые библиотеки или компоновщики в целом, ознакомьтесь со следующими работами: *Linkers and Loaders* (Morgan Kaufmann, 1999) Джона Р. Левина (John R. Levine), *The Inside Story on Shared Libraries and Dynamic Loading* (Computing in Science & Engineering, 2001, сентябрь/октябрь) Дэвида М. Бизли, Брайана Д. Уорда и Яна Р. Кука (David M. Beazley, Brian D. Ward, Ian R. Cooke). Можете также просмотреть онлайн-ресурсы, например библиотеку программ HOWTO (https://bit.ly/3q3MbS6). Страницу руководства `ld.so(8)` также стоит изучить.

Как перечислить зависимости разделяемой библиотеки

Файлы разделяемых библиотек обычно находятся в тех же местах, что и статические библиотеки. Двумя стандартными каталогами библиотек в системе Linux являются `/lib` и `/usr/lib`, а многие другие могут быть разбросаны по всей системе. Каталог `/lib` не должен содержать статических библиотек.

Разделяемая библиотека имеет суффикс, который содержит `.so` (Shared Object — разделяемый объект), например `libc-2.15.so` и `libc.so.6`. Чтобы узнать, какие разделяемые библиотеки использует программа, запустите команду `ldd` *`prog`*, где *`prog`* — это имя исполняемого файла. Вот пример для оболочки:

```
$ ldd /bin/bash
    linux-vdso.so.1 (0x00007ffff31cc000)
    libgtk3-nocsd.so.0 => /usr/lib/x86_64-linux-gnu/libgtk3-nocsd.so.0
    (0x00007f72bf3a4000)
    libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007f72bf17a000)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f72bef76000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f72beb85000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f72be966000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f72bf8c5000)
```

Для достижения более оптимальной производительности и гибкости исполняемые файлы обычно не знают расположения своих разделяемых библиотек — знают только названия и примерное местоположение. Небольшая программа `ld.so` (динамический компоновщик/загрузчик среды выполнения) находит и загружает разделяемые библиотеки для программы. Вывод команды `ldd` в приведенном ранее примере показывает имена библиотек в левой части `=>`. В правой части показано, где `ld.so` находит библиотеку.

В последней строке вывода приведено фактическое местоположение `ld.so` — в `/lib/ld-linux.so.2`.

Как ld.so находит разделяемые библиотеки

Одной из распространенных проблем, связанных с разделяемыми библиотеками, является то, что динамический компоновщик не может найти библиотеку. Первое место, где динамический компоновщик *обязательно* ищет разделяемые библиотеки, — это *предварительно настроенный путь поиска библиотеки времени исполнения* (Runtime Library Search Path) `rpath`, если он существует. Далее мы рассмотрим, как быстро создать этот путь.

Затем динамический компоновщик просматривает системный кэш `/etc/ld.so.cache`, чтобы узнать, находится ли библиотека там, где должна быть. Это быстрый кэш имен библиотечных файлов, найденных в каталогах, перечисленных в файле конфигурации кэша `/etc/ld.so.conf`.

ПРИМЕЧАНИЕ

Файл ld.so.conf может включать в себя ряд файлов из каталога, такого как /etc/ld.so.conf.d, что типично для многих файлов конфигурации Linux, которые мы встречали ранее.

Каждая строка в файле `ld.so.conf` или в файлах, которые он содержит, — это имя каталога, который нужно включить в кэш. Список каталогов обычно короткий и содержит данные, как в примере:

```
/lib/i686-linux-gnu
/usr/lib/i686-linux-gnu
```

Каталоги стандартной библиотеки `/lib` и `/usr/lib` являются неявными, что означает: их не нужно включать в файл `/etc/ld.so.conf`.

Если вы измените файл `ld.so.conf` или один из каталогов разделяемой библиотеки, то должны вручную пересобрать файл `/etc/ld.so.cache` с помощью команды

```
# ldconfig -v
```

Параметр `-v` предоставляет подробную информацию о библиотеках, которые `ldconfig` добавляет в кэш, и обо всех изменениях, которые он обнаруживает.

Существует еще одно место, где `ld.so` ищет разделяемые библиотеки, — переменная среды `LD_LIBRARY_PATH`. Мы вскоре это обсудим.

Не добавляйте что-либо в файл `/etc/ld.so.conf`. Вы должны знать, какие общие библиотеки находятся в системном кэше, и если поместить в кэш каждый маленький каталог разделяемых библиотек, это может вызвать конфликты в системе. При компиляции программного обеспечения, для которого требуется неявный путь к библиотеке, укажите исполняемому файлу встроенный путь поиска библиотеки времени исполнения `rpath`. Рассмотрим, как это сделать.

Как связать программы с разделяемыми библиотеками

Допустим, у вас есть разделяемая библиотека `libweird.so.1` в `/opt/obscure/lab`, с которой вам нужно связать библиотеку `myprog`. Такого пути нет в `/etc/ld.so.conf`, поэтому нужно передать этот путь компоновщику. Свяжите программу следующим образом:

```
$ cc -o myprog myprog.o -Wl,-rpath=/opt/obscure/lib -L/opt/obscure/lib -lweird
```

Параметр `-Wl,-rpath` указывает компоновщику добавить указанный каталог в путь поиска библиотеки времени исполнения. Однако, даже если вы используете параметр `-Wl,-rpath`, вам все равно нужен флаг `-L`.

Если вы хотите изменить `rpath` существующего двоичного файла, задействуйте программу `patchelf`, однако лучше изменять путь во время компиляции. (ELF,

исполняемый и связанный формат, — это стандартный формат, применяемый для исполняемых файлов и библиотек в системах Linux.)

Проблемы разделяемых библиотек

Разделяемые библиотеки очень гибки, не говоря уже о некоторых действительно эффективных возможностях. Однако библиотеками лучше не злоупотреблять, иначе в системе начнется беспорядок. Перечислим три важные проблемы, которые могут возникнуть:

- отсутствующие библиотеки;
- ужасная производительность;
- несовпадение библиотек.

Главной причиной всех проблем разделяемой библиотеки является переменная окружения `LD_LIBRARY_PATH`. Эта переменная, установленная для набора имен каталогов, разделенных двоеточием, заставляет `ld.so` искать данные в указанных каталогах еще до поиска разделяемой библиотеки. Это простейший способ заставить программы работать при перемещении библиотеки, когда у вас нет исходного кода программы, нельзя использовать `patchelf` или не хочется собирать исполняемые файлы. К сожалению, ничем хорошим это не заканчивается.

Никогда не устанавливайте переменную `LD_LIBRARY_PATH` в файлах запуска оболочки или при компиляции программного обеспечения. Когда компоновщик динамической среды выполнения обнаруживает эту переменную, ему приходится просматривать все содержимое каждого указанного каталога неоправданно много раз. Это приводит к значительному снижению производительности, а что более важно, вы можете столкнуться с конфликтами и несоответствующими библиотеками, потому что компоновщик среды выполнения просматривает эти каталоги для *каждой* программы.

Если вам *необходимо* использовать `LD_LIBRARY_PATH` для запуска какой-нибудь незначительной программы, для которой у вас нет исходного кода, либо приложения, которое вы предпочли бы не перекомпилировать, например Firefox или другие, применяйте сценарий-обертку. Допустим, необходимо добавить в исполняемый файл `/opt/crummy/bin/crummy.bin` несколько разделяемых библиотек в `/opt/crummy/lib`. Напишите сценарий-оболочку `crummy`, который выглядит следующим образом:

```
#!/bin/sh
LD_LIBRARY_PATH=/opt/crummy/lib
export LD_LIBRARY_PATH
exec /opt/crummy/bin/crummy.bin $@
```

Не используйте переменную `LD_LIBRARY_PATH`, и это предотвратит большинство проблем с разделяемыми библиотеками. Еще одна существенная сложность, которая иногда возникает у разработчиков, заключается в том, что API библиотеки

может незначительно меняться от одной младшей версии к другой, нарушая работу уже установленного программного обеспечения. Есть два превентивных решения: использовать либо согласованную методологию для установки общих библиотек `c -Wl, -rpath` для создания пути поиска библиотеки времени исполнения, либо статические версии малоизвестных библиотек.

15.1.4. Файлы заголовков (Include) и каталоги

Файлы заголовков C — это дополнительные файлы исходного кода, которые содержат объявления типов и библиотечных функций, часто для библиотек, которые мы встречали ранее. Например, `stdio.h` является файлом заголовка (см. простую программу в разделе 15.1).

Множество проблем с компиляцией связано с файлами заголовков. Большинство таких сбоев возникает, когда компилятор не может найти файлы заголовков и библиотеки. Существуют даже случаи, когда программист забывает добавить директиву `#include` в код, содержащий нужный файл заголовка, что приводит к сбою при компиляции части исходного кода.

Проблемы с файлами заголовков

Отследить правильные файлы заголовков не всегда легко. Иногда можно найти их с помощью команды `locate`, но обычно в разных каталогах имеется несколько файлов с одинаковыми именами и неясно, какой из них правильный. Когда компилятор не может найти файл заголовка, он выводит следующее сообщение об ошибке:

```
badinclude.c:1:22: fatal error: notfound.h: No such file or directory
```

Оно говорит о том, что компилятор не может найти файл заголовка `notfound.h`, на который ссылается файл `badinclude.c`. Если посмотреть на файл `badinclude.c` (в строке 1, как указывает ошибка), мы увидим строку, которая выглядит так:

```
#include <notfound.h>
```

Подобные директивы `Include` не указывают, где именно должен располагаться файл заголовка. Они сообщают только то, что файл должен находиться либо в месте, установленном по умолчанию, либо в месте, указанном в командной строке компилятора. В названиях большинства из этих мест имеется слово `include`. По умолчанию каталог `include` в Unix — это `/usr/include`, компилятор всегда проверяет его, если не запретить ему это делать. Конечно, если файл находится на своем месте по умолчанию, ошибки не будет, поэтому давайте посмотрим, как заставить компилятор искать в других каталогах `include`.

Предположим, вы нашли `notfound.h` в каталоге `/usr/junk/include`. Чтобы компилятор добавил его в свой путь поиска, используйте параметр `-I`:

```
$ cc -c -I/usr/junk/include badinclude.c
```

Теперь компилятор больше не будет останавливаться на строке кода в `badinclude.c`, которая ссылается на файл заголовка.

ПРИМЕЧАНИЕ

Мы обсудим, как найти отсутствующие файлы include, в главе 16.

Будьте внимательны с файлами `include`, которые используют двойные кавычки (`""`) вместо угловых скобок (`<>`), например:

```
#include "myheader.h"
```

Двойные кавычки означают, что файла заголовка нет в системном каталоге `include` и что файл `include` находится в том же каталоге, что и исходный файл. Если возникла проблема с двойными кавычками, это значит, что вы, вероятно, пытаетесь скомпилировать неполный исходный код.

Препроцессор C

На самом деле компилятор C не ищет файлы `include`. Эта задача ложится на препроцессор C — программу, которую компилятор запускает на исходном коде перед анализом фактической программы. Препроцессор переписывает исходный код в форму, понятную компилятору, что облегчает чтение исходного кода и позволяет предоставить ярлыки.

Команды препроцессора в исходном коде называются *директивами*, они начинаются с символа `#`. Существует три основных типа директив:

- **Файлы Include.** Директива `#include` предписывает препроцессору добавить весь файл целиком. Обратите внимание на то, что флаг компилятора `-I` на самом деле является параметром, который заставляет препроцессор искать в указанном каталоге файлы `include`, как показано в предыдущем разделе.
- **Определение макроса.** Строка, например `#define BLAH` *`something`*, указывает препроцессору заменить *`something`* для всех вхождений `BLAH` в исходном коде. Соглашение требует, чтобы названия макросов состояли только из прописных букв. Однако неудивительно то, что программисты используют макросы, имена которых выглядят как функции и переменные. (Время от времени это вызывает массу проблем. Многие программисты активно работают с препроцессором только из спортивного интереса.)

ПРИМЕЧАНИЕ

Вместо определения макросов в исходном коде их можно определить, передав параметры компилятору: -DBLAH=something сработает так же, как и предыдущая директива #define.

- **Условные операторы.** Выделить определенные фрагменты кода можно с помощью директив `#ifdef`, `#if` и `#endif`. Директива `#ifdef` *`MACRO`* проверяет,

определен ли макрос препроцессора, и *условие* `#if condition` проверяет, является ли *условие* `condition` ненулевым. Для обеих директив, если условие, следующее за оператором `if`, является ложным, препроцессор не передает компилятору ни одной строчки программы между `#if` и следующим `#endif`. К этому следует привыкнуть, если вы планируете просматривать какой-либо код на языке C.

Рассмотрим пример условной директивы. Когда препроцессор видит следующий код, он проверяет, определен ли макрос `DEBUG`, и если да, передает строку, содержащую `fprintf()`, компилятору. В противном случае препроцессор пропускает эту строку и продолжает обрабатывать файл после `#endif`:

```
#ifdef DEBUG
  fprintf(stderr, "This is a debugging message.\n");
#endif
```

ПРИМЕЧАНИЕ

Препроцессор C ничего не знает о синтаксисе C, переменных, функциях и других элементах. Он понимает только собственные макросы и директивы.

В Unix препроцессор C называется `cpp`, его можно запустить также с помощью команды `gcc -E`. Однако редко возникает необходимость запускать препроцессор самостоятельно.

15.2. Утилита make

Программа, имеющая более одного файла исходного кода или требующая странных параметров компилятора, слишком громоздка для компиляции вручную. Эта проблема существует уже много лет, и для ее решения традиционно применяется утилита управления компиляцией Unix `make`. Если вы используете систему Unix, следует немного изучить утилиту `make`, потому что системные утилиты задействуют ее для некоторых задач. Однако все, что мы обсудим в этой главе, — всего лишь верхушка айсберга. Существуют полноценные книги по этой теме, например *Projects with GNU Make*, 3-е издание (O'Reilly, 2005), Роберта Мекленбурга (Robert Mecklenburg). Кроме того, большинство пакетов Linux создаются с помощью дополнительного слоя вокруг `make` или аналогичного инструмента. Существует множество систем сборки, мы рассмотрим одну из них под названием `autotools` в главе 16.

Утилита `make` — это большая система, при этом довольно легко понять, как она работает. Когда вы видите файл с именем Makefile или `makefile`, это значит, что используется утилита `make`. (Попробуйте запустить команду `make`, чтобы посмотреть, сможете ли вы что-нибудь настроить.)

Основная идея утилиты `make` — это *цель*, которой вы хотите достичь. Ею может быть файл (файл `.o`, исполняемый файл и т. д.) или метка. Кроме того, некоторые

целевые объекты зависят от других целевых объектов. Так, вам потребуется полный набор файлов `.o`, прежде чем вы сможете связать свой исполняемый файл. Эти требования называются *зависимостями*.

Чтобы создать цель, `make` следует *правилу*, например, определяющему, как перейти от исходного файла `.c` к объектному файлу `.o`. Утилита `make` по умолчанию уже знает несколько правил, но вы можете создать собственные.

15.2.1. Makefile

Возьмем за основу примеры файлов `aux.c` и `main.c` из подраздела 15.1.1 и с помощью Makefile создадим программу `myprog`:

```
❶ # object files
❷ OBJS=aux.o main.o

❸ all: ❹ myprog

  myprog: ❺ $(OBJS)
          ❻ $(CC) -o myprog $(OBJS)
```

Символ `#` в первой строке этого файла Makefile ❶ обозначает комментарий.

Следующая строка — это определение макроса, которое устанавливает переменную `OBJS` в два имени файлов объектов ❷. Почему это важно, узнаем позже. На данный момент обратите внимание на то, как вы определяете макрос и как ссылаетесь на него в дальнейшем (`$(OBJS)`).

Следующий элемент в файле Makefile содержит свою первую цель `all` ❸. Первая цель всегда используется по умолчанию, это цель, которую `make` хочет создать, когда вы запускаете ее в командной строке.

Правило сборки цели следует за двоеточием. Для цели `all` в этом файле Makefile нужно выполнить что-то под названием `myprog` ❹. Это первая зависимость в файле, все зависит от `myprog`. Обратите внимание на то, что `myprog` может быть реальным файлом или целью другого правила. В данном случае это и то и другое (правило для всех и цель `OBJS`).

Для сборки `myprog` файл Makefile использует макрос `$(OBJS)` в зависимостях ❺. Макрос расширяется до `aux.o` и `main.o`, указывая, что `myprog` зависит от этих двух файлов (они должны быть реальными, потому что в файле Makefile нет целей с такими именами).

ПРИМЕЧАНИЕ

Отступ перед $(CC) ❻ — это табуляция. Утилита make очень строго относится к табуляциям.

Файл Makefile предполагает, что у вас есть два исходных файла C с именами `aux.c` и `main.c` в одном каталоге. Запустите команду `make` для файла Makefile, чтобы отобразить команды, которые выполняет утилита `make`:

```
$ make
Cc    -c -o aux.o aux.c
cc    -c -o main.o main.c
cc -o myprog aux.o main.o
```

Диаграмма зависимостей показана на рис. 15.1.

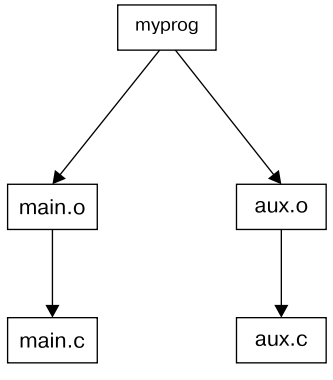


Рис. 15.1. Зависимости Makefile

15.2.2. Встроенные правила

Если файл `aux.c` отсутствует в файле Makefile, то как `make` переходит с файла `aux.c` на файл `aux.o`? Это происходит потому, что у `make` есть встроенные правила, которым нужно следовать. Благодаря им утилита знает, как найти файл `.c` и как запустить в нем `cc -c`, чтобы создать файл `.o`.

15.2.3. Финальная сборка программы

Последний шаг создания программы `myprog` сложнее всех остальных, но суть его довольно проста. После того как вы создали два объектных файла в `$(OBJ)`, запустите компилятор C, как показано далее, где `$(CC)` расширяется до имени компилятора:

```
    $(CC) -o myprog $(OBJS)
```

Как уже говорилось, отступ перед `$(CC)` означает табуляцию. *Необходимо* вставлять табуляцию перед любой системной командой в отдельной строке.

Остерегайтесь следующего вывода:

```
Makefile:7: *** missing separator. Stop.
```

Ошибка из приведенного ранее примера означает, что файл Makefile поврежден. Табуляция является разделителем, и, если его нет или используются лишние символы, возникает ошибка.

15.2.4. Обновление зависимостей

Последнее и важнейшее, о чем необходимо знать относительно утилиты `make`, заключается в том, что ее задача — привести цели в соответствие с их зависимостями. Для этого она использует минимальное количество шагов, что значительно экономит время. Если в предыдущем примере ввести команду `make` два раза подряд, первая команда создаст `myprog`, а вторая выдаст следующий результат:

```
make: Nothing to be done for 'all'.
```

Во второй раз команда `make` просматривает свои правила и видит, что программа `myprog` уже существует, поэтому ее не нужно создавать снова, так как ни одна из зависимостей не изменилась с момента последнего создания. Экспериментируя с этим, можете выполнить следующие действия:

1. Запустите команду `touch aux.c`.
2. Запустите команду `make` еще раз. На этот раз она определяет, что зависимость `aux.c` новее, чем `aux.o`, уже находящаяся в каталоге, поэтому утилита снова компилирует `aux.o`.
3. Программа `myprog` зависит от `aux.o`, и так как последняя обновилась, утилита `make` должна снова создать программу `myprog`.

Это очень типичная цепная реакция.

15.2.5. Аргументы и параметры командной строки

Знание аргументов и параметров командной строки для утилиты `make` значительно увеличивает эффективность ее использования. Наиболее действенный вариант — указать единичную цель в командной строке. Для предыдущего файла Makefile запустите `make aux.o`, чтобы задействовать только файл `aux.o`.

Кроме того, `make` позволяет определить макрос в командной строке. Например, чтобы использовать компилятор `clang`, введите

```
$ make CC=clang
```

В примере `make` применяет определение `CC` вместо своего компилятора по умолчанию `cc`. С помощью макросов командной строки тестируются определения и библиотеки препроцессоров, особенно это касается макросов `CFLAGS` и `LDFLAGS`, которые мы вскоре обсудим.

На самом деле для запуска утилиты `make` даже не нужен файл. Если встроенные правила `make` соответствуют цели, команда `make` сама создаст цель. Например, если

у вас есть исходный код очень простой программы под названием `blah.c`, введите команду `make blah`. Вывод команды `make` будет выглядеть следующим образом:

```
$ make blah
Cc   blah.o   -o blah
```

Однако такой способ сработает только для самых элементарных программ на языке C, а если программе нужны библиотека или специальный каталог `include`, то следует написать файл Makefile. Запуск утилиты `make` без файла Makefile больше подходит для работы с языками типа Fortran, Lex или Yacc. И если вы не знаете, как работают компилятор или утилита, можете использовать именно этот способ.

Утилите `make` способна самостоятельно выяснить необходимую информацию: даже если `make` не сможет создать цель, она все равно выведет данные о том, как использовать тот или иной инструмент.

Два самых распространенных параметра:

- `-n` — выводит команды, необходимые для сборки, но не позволяет `make` действительно выполнять какие-либо команды;
- `-f`*`file`* — указывает `make` на чтение из файла *`file`*, а не из Makefile или `makefile`.

15.2.6. Стандартные макросы и переменные

Для утилиты `make` существует много специальных макросов и переменных. Трудно определить разницу между макросом и переменной, но здесь термин «макрос» используется для обозначения элемента, который не меняется в процессе сборки цели с помощью `make`.

Как мы видели ранее, можно установить макросы в начале файла Makefile. Перечислим наиболее распространенные макросы:

- `CFLAGS`. Параметры компилятора языка C. При создании объектного кода из файла `.c` утилита `make` передает макрос в качестве аргумента компилятору.
- `LDFLAGS`. Похож на `CFLAGS`, но предназначен для компоновщика при создании исполняемого файла из объектного кода.
- `LDLIBS`. Если вы используете `LDFLAGS`, но не хотите объединять путь поиска с параметрами имени библиотеки, поместите последние в этот файл.
- `CC`. Компилятор языка C. Значение по умолчанию — `cc`.
- `CPPFLAGS`. Параметры препроцессора C. Когда `make` каким-либо образом запускает препроцессор C, она передает этот макрос в качестве аргумента.
- `CXXFLAGS`. GNU `make` использует этот макрос для флагов компилятора C++.

Значение переменных `make` меняется по мере сборки целей. Переменные начинаются со знака доллара (`$`). Существует несколько способов добавления переменных, но

часть наиболее распространенных переменных автоматически устанавливается в целевых правилах:

- `$@` — внутри правила переменная расширяется до текущей цели;
- `$<` — внутри правила переменная расширяется до первой зависимости целевого объекта;
- `$*` — переменная расширяется до базового имени текущего целевого объекта. Например, если вы создаете `blah.o`, переменная расширяется до `blah`.

Приведем пример, иллюстрирующий общий шаблон правила, использующего `myprog` для создания файла `.out` из файла `.in`:

```
.SUFFIXES: .in
.in.out: $<
        myprog $< -o $*.out
```

Правило `.c.o` встречается во многих файлах Makefile, определяющих индивидуальный способ запуска компилятора C для создания объектного файла.

Полный список переменных `make` в Linux можно найти на странице руководства `make info`.

ПРИМЕЧАНИЕ

Имейте в виду, что для GNU make существует множество расширений, встроенных правил и функций, которых нет в других вариантах. При использовании Linux проблем нет, но если вы перейдете на системы Solaris или BSD, вас ждет сюрприз, так как параметры в них работать не будут. Для решения этой проблемы была создана мультиплатформенная система сборки GNU autotools.

15.2.7. Стандартные цели

Большинство разработчиков включают в свои файлы Makefiles несколько дополнительных стандартных целей, которые выполняют вспомогательные задачи, связанные с компиляцией:

- `clean`. Цель `clean` используется повсеместно, команда `make clean` удаляет все объектные и исполняемые файлы, чтобы можно было начать все заново или упаковать программное обеспечение. Пример правила для файла Makefile `myprog`:

  ```
  clean:
          rm -f $(OBJS) myprog
  ```

- `distclean`. Файл Makefile, созданный с помощью системы GNU autotools, всегда включает в себя цель `distclean`, которая удаляет все, что не стало частью исходного дистрибутива, в том числе файл Makefile. Подробнее об этом вы узнаете в главе 16. В очень редких случаях разработчик решает не

удалять исполняемый файл с целью `distclean`, используя вместо этого цель `realclean`.

- `install`. Эта цель копирует файлы и скомпилированные программы в подходящее по мнению файла Makefile место в системе. Это может быть опасно, поэтому всегда запускайте команду `make -n install`, чтобы посмотреть, что произойдет, прежде чем запускать другие команды.
- `test` или `check`. Некоторые разработчики добавляют цель `test` или `check`, чтобы убедиться, что после окончательной сборки все работает правильно.
- `depend`. Эта цель создает зависимости, вызывая компилятор с параметром `-M` для проверки исходного кода. Это необычная цель, поскольку она часто изменяет сам файл Makefile. Сейчас эта цель используется нечасто, но если вы найдете инструкцию с указанием добавить цель `depend`, обязательно сделайте это.
- `all`. Как упоминалось ранее, обычно это первая цель в файле Makefile. Вы часто будете встречать отсылки на нее вместо реального исполняемого файла.

15.2.8. Организация файла Makefile

Несмотря на то что существует множество различных стилей файлов Makefile, большинство программистов придерживаются некоторых общих правил. Во-первых, в первой части файла Makefile (внутри определений макросов) должны находиться библиотеки и файлы `include`, сгруппированные в соответствии с пакетом:

```
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 флага компилятора и компоновщика получает макрос, как в примере далее:

```
CFLAGS=$(CFLAGS) $(MYPACKAGE_INCLUDES) $(PNG_INCLUDES)
LDFLAGS=$(LDFLAGS) $(MYPACKAGE_LIB) $(PNG_LIB)
```

Обычно объектные файлы группируются в соответствии с исполняемыми файлами. Предположим, у вас есть пакет, который создает исполняемые файлы `boring` и `trite`. Каждый из них имеет собственный исходный файл `.c` и требует кода в `util.c`. Вот как может выглядеть код:

```
UTIL_OBJS=util.o

BORING_OBJS=$(UTIL_OBJS) boring.o
TRITE_OBJS=$(UTIL_OBJS) trite.o

PROGS=boring trite
```

Остальная часть файла Makefile может выглядеть следующим образом:

```
all: $(PROGS)

boring: $(BORING_OBJS)
        $(CC) -o $@ $(BORING_OBJS) $(LDFLAGS)

trite: $(TRITE_OBJS)
        $(CC) -o $@ $(TRITE_OBJS) $(LDFLAGS)
```

Можно объединить две цели исполняемых файлов в одно правило, но не стоит этого делать, потому что будет нелегко переместить правило в другой файл Makefile, удалить исполняемый файл или сгруппировать исполняемые файлы по-другому. Кроме того, зависимости станут неверными: если бы у вас было только одно правило для файлов `boring` и `trite`, то `trite` зависел бы от `boring.c`, `boring` — от `trite.c`, и утилита `make` пыталась бы пересобрать обе программы при изменении одного из файлов с исходным кодом.

ПРИМЕЧАНИЕ

Если вам нужно определить специальное правило для объектного файла, поместите правило для него чуть выше правила, которое создает исполняемый файл. Если несколько исполняемых файлов используют один и тот же объектный файл, поместите правило объекта выше всех исполняемых правил.

15.3. Программы Lex и Yacc

С программами Lex и Yacc вы можете встретиться в процессе компиляции программ, которые считывают файлы конфигурации или команды. Эти инструменты являются строительными блоками для языков программирования.

- Lex — это разметчик, который преобразует текст в пронумерованные теги с метками. Версия для GNU/Linux называется flex. Для работы с Lex может понадобиться флаг компоновщика `-ll` или `-lfl`.
- Yacc — это синтаксический анализатор, который пытается считывать токены в соответствии с грамматикой. Синтаксический анализатор GNU — это `bison`, чтобы начать работу с Yacc, запустите `bison -y`. Возможно, понадобится также флаг компоновщика `-ly`.

15.4. Языки сценариев

Давным-давно среднему системному администратору Unix не нужно было переживать о языках сценариев кроме оболочки Bourne shell и awk. Сценарии оболочки (см. главу 11) по-прежнему являются важной частью Unix, а сценарии awk теперь используются редко. В настоящее время появилось множество более мощных и эффективных преемников, и многие системные программы фактически перешли

с языка C на языки сценариев (к примеру, программа `whois`). Рассмотрим основы написания сценариев.

Первое, что нужно знать о любом языке сценариев, — то, что первая строка сценария выглядит как фрагмент сценария Bourne shell. Например, скрипт на Python начинается так:

```
#!/usr/bin/python
```

Возможен и такой вариант — он запускает первую версию Python в командном пути, вместо того чтобы всегда переходить в `/usr/bin`:

```
#!/usr/bin/env python
```

Как мы обсуждали в главе 11, исполняемый текстовый файл, начинающийся с символов `#!` (шебанг), является сценарием. Путь, указанный после данного префикса, — это исполняемый файл интерпретатора языка сценариев. Когда Unix пытается запустить исполняемый файл, который начинается с `#!`, он запускает программу с остальной частью файла в качестве стандартного ввода. Следовательно, сценарий может выглядеть даже так:

```
#!/usr/bin/tail -2
This program won't print this line,
but it will print this line...
and this line, too.
```

Часто в первой строке сценария оболочки содержится одна из наиболее распространенных основных проблем сценария — неверный путь к интерпретатору языка сценариев. Предположим, вы назвали скрипт из приведенного ранее примера `myscript`. Что если команда `tail` на самом деле находится в `/bin`, а не в каталоге `/usr/bin` вашей системы? В этом случае запуск скрипта приведет к такой ошибке:

```
bash: ./myscript: /usr/bin/tail: bad interpreter: No such file or directory
```

Помните, что в первой строке сценария не сработает более одного аргумента. То есть значение `-2` в предыдущем примере может сработать, но если добавить еще один аргумент, система может рассмотреть значение `-2` и новый аргумент как один большой аргумент с пробелами и всем прочим. Такая реакция может варьироваться от системы к системе, и не стоит испытывать терпение, тратя время на такие мелочи.

Рассмотрим несколько существующих языков сценариев.

15.4.1. Язык Python

Python — это очень популярный язык сценариев со множеством функций, таких как обработка текста, доступ к базе данных, создание сетей и многопоточность. У него есть мощный интерактивный режим и очень организованная объектная модель.

Исполняемый файл Python — это `python`, обычно он находится в каталоге `/usr/bin`. Однако Python используется не только из командной строки для сценариев. Он встречается повсюду — от анализа данных до создания веб-приложений. Книга Дэвида М. Бизли (David M. Beazley) *Python Distilled* (Addison-Wesley, 2021) подойдет для глубокого знакомства с Python.

15.4.2. Язык Perl

Одним из старых сторонних языков сценариев Unix является Perl. Он обладает невероятным множеством полезных функций и инструментов программирования. Хотя в последние годы Perl используется все реже из-за появления Python, он превосходит его в работе с текстом, преобразовании и обработке файлов. Существует множество инструментов, созданных с помощью языка Perl. Чтобы познакомиться с этим языком, обратитесь к книгам «Perl. Изучаем глубже» (Символ-Плюс, 2013) и *Modern Perl* (Onyx Neon Press, 2016) Рэндала Л. Шварца, Брайана Д. Фоя и Тома Феникса (Randal L. Schwartz, Brian D Foy, Tom Phoenix).

15.4.3. Другие языки сценариев

Вы можете также встретить следующие языки сценариев:

- **PHP.** Язык обработки гипертекста, часто встречающийся в динамических веб-сценариях. Некоторые пользователи применяют его для автономных сценариев. Веб-сайт PHP: www.php.net.
- **Ruby.** Фанаты объектно-ориентированного подхода и многие веб-разработчики любят программировать на нем (www.ruby-lang.org).
- **JavaScript.** Этот язык используется в веб-браузерах в основном для управления динамическим контентом. Большинство опытных программистов не применяют его как отдельный язык сценариев из-за многочисленных недостатков. Однако без JavaScript нельзя обойтись, если вы занимаетесь веб-программированием. В последние годы его реализация Node.js распространяется в серверном программировании и написании сценариев, ее исполняемое имя — `node`.
- **Emacs Lisp.** Разновидность языка программирования Lisp, используемого текстовым редактором Emacs.
- **MATLAB, Octave.** MATLAB — это коммерческий матричный и математический язык программирования и библиотека. Octave — очень похожий на MATLAB бесплатный проект программного обеспечения.
- **R.** Популярный бесплатный язык статистического анализа. Больше информации о нем можно найти на странице www.r-project.org, а также в книге Нормана Мэтлоффа «Искусство программирования на R. Погружение в большие данные» (Питер, 2019).

- **Mathematica.** Еще один коммерческий язык математического программирования с библиотеками.
- **m4.** Это язык обработки макросов, обычно встречающийся только в GNU autotools.
- **Tcl** (tool command language — инструментальный командный язык). Это простой язык сценариев, обычно связанный с инструментарием графического интерфейса пользователя Tk и утилитой автоматизации Expect. Хотя сейчас Tcl не так широко распространен, как раньше, не стоит сбрасывать со счетов его эффективность. Многие опытные разработчики предпочитают Tk именно из-за его встроенных возможностей. Больше информации найдете на сайте www.tcl.tk.

15.5. Язык Java

Java — это компилируемый язык, похожий на C, с более простым синтаксисом и мощной поддержкой объектно-ориентированного программирования. Он встречается в нескольких областях систем Unix, например, часто используется в качестве среды веб-приложений и популярен при создании специализированных приложений. Приложения для Android обычно пишутся на языке Java. Несмотря на то что он нечасто встречается на типичном рабочем столе Linux, необходимо знать, как работает Java как минимум в том, что касается автономных приложений.

Существует два вида компиляторов Java: собственные компиляторы для создания машинного кода системы (например, компилятор C) и компиляторы байт-кода для интерпретатора (иногда он называется виртуальной машиной, которая тем не менее отличается от виртуальной машины гипервизора, рассматриваемой в главе 17). В системе Linux байт-код встречается практически всегда.

Файлы байт-кода Java заканчиваются на `.class`. Среда выполнения Java (Java Runtime Environment, JRE) содержит все программы, необходимые для запуска байт-кода Java. Чтобы запустить файл байт-кода *`file`*, используйте команду

```
$ java file.class
```

Существуют файлы байт-кода, которые заканчиваются на `.jar` и представляют собой коллекции архивированных файлов `.class`. Чтобы запустить файл `.jar`, примените следующую команду:

```
$ java -jar file.jar
```

Иногда может быть необходимо установить переменную окружения `JAVA_HOME` в префикс пути места установки Java, а также задействовать переменную `CLASSPATH`, чтобы включить любые каталоги, содержащие необходимые программе классы, то есть набор каталогов, разделенных двоеточием, похожий на обычную переменную `PATH` для исполняемых файлов.

Если необходимо скомпилировать файл `.java` в байт-код, понадобится набор для разработки — Java Development Kit (JDK). Можете запустить компилятор `javac` из JDK для создания файлов `.class`:

```
$ javac file.java
```

JDK поставляется и с `jar` — программой, которая может собирать и разбирать файлы `.jar`. Работает так же, как `tar`.

15.6. А что дальше? Компиляция пакетов

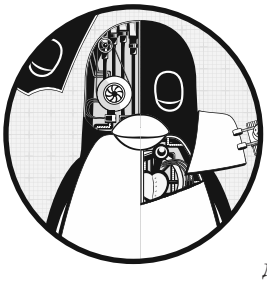
Мир компиляторов и языков сценариев огромен и постоянно расширяется. На момент написания этой главы новые компилируемые языки, такие как Go (`golang`) и Rust, набирают популярность в прикладном и системном программировании.

Инфраструктуры компиляции LLVM (llvm.org) значительно упростили разработку компилятора. Если вам интересно, как разработать и реализовать компилятор, изучите две подходящие книги: «Компиляторы. Принципы, технологии и инструментарий» (Вильямс, 2001) Альфреда В. Ахо и др., *Modern Compiler Design*, 2-е издание (Springer, 2012), Дика Груна и др. (Dick Grune et al.). Для разработки языка сценариев лучше всего пользоваться онлайн-ресурсами, так как реализации сильно различаются.

Теперь, изучив основы инструментов программирования в системе, мы можем увидеть, на что они способны. Следующая глава посвящена созданию пакетов в Linux из исходного кода.

16

Компиляция программ из исходного кода на языке C



Большинство общедоступных пакетов программного обеспечения Unix сторонних производителей поставляются в виде исходного кода, который пользователь может собрать и установить. Одна из причин этого заключается в том, что Unix (и сам Linux) имеет так много различных вариантов и архитектур, что было бы трудно распространять двоичные пакеты для всех возможных комбинаций платформ. Другая причина, не менее важная, заключается в том, что широкое распространение исходного кода в сообществе Unix побуждает пользователей вносить исправления ошибок и новые функции в программное обеспечение, придавая смысл *открытому* исходному коду (open source). Практически все, что находится в системе Linux, от ядра и библиотеки C до веб-браузеров, можно представить в виде исходного кода. Можно даже обновить и дополнить всю систему, повторно установив ее фрагменты из исходного кода. Однако *не следует* обновлять систему, устанавливая вообще *все* из исходного кода, разве что вам нравится процесс или у вас есть на это причины.

Дистрибутивы Linux позволяют обновлять основные части системы более простыми способами, например, с помощью программ из каталога `/bin`. Важным свойством дистрибутивов является возможность быстро решать проблемы безопасности. Однако не ожидайте, что дистрибутив предоставит все необходимое без вашего участия.

Причины, по которым стоит установить определенные пакеты самостоятельно:

- Управление параметрами конфигурации.
- Установка программного обеспечения туда, куда вам необходимо. Вы даже можете установить несколько разных версий одного и того же пакета.
- Управление версией, которую вы устанавливаете. Дистрибутивы не всегда содержат последние версии всех пакетов, особенно дополнений к программным пакетам, таким как библиотеки Python.
- Изучение того, как функционируют пакеты.

16.1. Системы сборки программ

В Linux существует множество сред программирования, от традиционного C до интерпретируемых языков сценариев, таких как Python. Как правило, они имеют по крайней мере одну отдельную систему для создания и установки пакетов в дополнение к инструментам, предоставляемым дистрибутивом Linux.

В этой главе мы рассмотрим компиляцию и установку исходного кода C с помощью лишь одной из этих систем сборки — сценариев конфигурации, созданных из пакета GNU autotools. Эта система считается стабильной, и многие базовые утилиты Linux ее используют. Поскольку она основана на существующих инструментах, похожих на `make`, после изучения ее работы вы сможете применять свои знания в других системах сборки.

Устанавливая пакет из исходного кода C, сделайте следующие шаги:

1. Распакуйте архив исходного кода.
2. Сконфигурируйте пакет.
3. Запустите команду `make` или другую команду сборки программ.
4. Выполните команду `make install` или команду установки конкретного дистрибутива, чтобы установить пакет.

ПРИМЕЧАНИЕ

Прежде чем приступить к этой главе, необходимо изучить основы — главу 15.

16.2. Распаковка пакетов с исходным кодом C

Исходный код пакета поставляется в формате `.tar.gz` (файл `.tar.bz2` или `.tar.xz`), и его необходимо распаковать, как описано в разделе 2.18. Однако перед распаковкой проверьте содержимое архива с помощью команды `tar tvf` или `tar ztvf`, так как некоторые пакеты не создают собственные подкаталоги в каталоге, куда извлекается архив.

Вывод, как в следующем примере, означает, что пакет можно распаковать:

```
package-1.23/Makefile.in
package-1.23/README
package-1.23/main.c
package-1.23/bar.c
--пропуск--
```

Однако можно заметить, что не все файлы находятся в общем каталоге (например, `package-1.23` из предыдущего примера):

```
Makefile
README
main.c
--пропуск--
```

Если извлечь подобный архив, то в текущем каталоге может начаться хаос. Чтобы избежать этого, создайте новый каталог и перейдите в него, прежде чем извлекать содержимое архива.

Наконец, будьте внимательны с пакетами, содержащими файлы с абсолютными путями, как в примере далее:

```
/etc/passwd
/etc/inetd.conf
```

Скорее всего, они вам не встретятся, но если что, просто удалите архив из своей системы, потому что он может содержать троян или какой-то другой вредоносный код.

После распаковки содержимого исходного архива перед вами предстанет множество различных файлов. Давайте разберемся в них. Найдите файлы с именем типа `README` и `INSTALL`. Всегда сначала просматривайте любые файлы `README`, потому что они содержат описание пакета, краткое руководство, советы по установке и другую полезную информацию. Многие пакеты поставляются также с файлами `INSTALL`, содержащими инструкции по компиляции и установке пакета. Обратите особое внимание на специальные параметры и определения компилятора.

Помимо файлов `README` и `INSTALL` вы увидите другие файлы пакетов, которые делятся примерно на три категории:

- Файлы, относящиеся к системе `make`, такие как `Makefile`, `Makefile.in`, `configure` и `CMakeLists.txt`. Некоторые очень старые пакеты поставляются с файлом Makefile (его, скорее всего, нужно будет изменить), однако большинство используют утилиту настройки, например GNU autoconf или Cmake. Они поставляются со сценарием или файлом конфигурации (например, `configure` или `CMakeLists.txt`), чтобы помочь создать файл Makefile из файла `Makefile.in` в соответствии с настройками системы и параметров конфигурации.

- Файлы исходного кода, заканчивающиеся на `.c`, `.h` или `.cc`. Файлы исходного кода C могут появляться практически в любом месте каталога пакета. Файлы исходного кода C++ обычно имеют окончания `.cc`, `.C` или `.cxx`.
- Объектные файлы, заканчивающиеся на `.o`, или двоичные файлы. Обычно в дистрибутивах исходного кода нет объектных файлов, но вы можете найти их в тех редких случаях, когда создателю пакета нельзя выпускать определенный исходный код, и вам нужно выполнить определенные действия, чтобы использовать объектные файлы. В большинстве случаев объектные или двоичные исполняемые файлы в исходном дистрибутиве означают, что пакет собран неправильно, поэтому запустите команду `make clean`, чтобы убедиться, что компиляция актуальна.

16.3. Утилита GNU Autoconf

Несмотря на то что исходный код C обычно легко портируется, отличия, существующие на каждой платформе, делают невозможной компиляцию большинства пакетов с помощью одного файла Makefile. Раньше проблема решалась за счет предоставления отдельных файлов Makefile для каждой операционной системы либо легкоизменяемого файла Makefile. Этот подход перешел в сценарии, которые генерируют файлы Makefile на основе анализа системы, применяемой для создания пакета.

GNU autoconf — популярная система для автоматической генерации файлов Makefile. Пакеты, использующие эту систему, поставляются с файлами `configure`, `Makefile.in` и `config.h.in`. Файлы `.in` являются шаблонами, смысл заключается в том, чтобы запустить сценарий настройки, узнать характеристики системы, а затем произвести замены в файлах `.in` для создания реальных файлов сборки. Для конечного пользователя все просто — чтобы создать файл Makefile из `Makefile.in`, следует запустить команду `configure`:

```
$ ./configure
```

Команда выведет множество диагностических сообщений, поскольку сценарий проверяет систему на соответствие необходимым условиям. Если все хорошо, `configure` создает один или несколько файлов Makefile и файл `config.h`, а также файл кэша (`config.cache`), чтобы не нужно было повторять проверку.

Теперь можно запустить `make` для компиляции пакета. Однако успешное выполнение команды `configure` не обязательно означает, что компиляция `make` сработает. (В разделе 16.6 даны советы по устранению неполадок при неудачной настройке и компиляции.)

Познакомимся с этим процессом поближе.

ПРИМЕЧАНИЕ

На этом этапе в вашей системе должны быть доступны все необходимые инструменты сборки. Чтобы убедиться в этом в дистрибутивах Debian и Ubuntu, необходимо установить пакет для сборки build-essential, в системах, подобных Fedora, воспользуйтесь groupinstall в разделе с инструментами разработки (Development Tools).

16.3.1. Пример использования Autoconf

Прежде чем обсудить, как изменять поведение утилиты `autoconf`, рассмотрим простой пример ее работы. Установите пакет GNU `coreutils` в свой домашний каталог, чтобы убедиться, что вы не повредите систему. Загрузите с сайта ftp.gnu.org/gnu/coreutils пакет (лучше всего последнюю версию), распакуйте его, перейдите в его каталог и настройте следующим образом:

```
$ ./configure --prefix=$HOME/mycoreutils
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
--пропуск--
config.status: executing po-directories commands
config.status: creating po/POTFILES
config.status: creating po/Makefile
```

Теперь запустите утилиту `make`:

```
$ make
  GEN      lib/alloca.h
  GEN      lib/c++defs.h
--пропуск--
make[2]: Leaving directory '/home/juser/coreutils-8.32/gnulib-tests'
make[1]: Leaving directory '/home/juser/coreutils-8.32'
```

Затем попробуйте запустить один из только что созданных исполняемых файлов, например `./src/ls`, и запустите команду `make check`, чтобы выполнить серию проверок пакета. (Этот процесс может занять некоторое время, но за ним интересно наблюдать.)

Наконец, можно установить пакет. Сделайте пробный запуск (без реальной установки) с помощью команды `make -n`, чтобы увидеть, что сделает `make install`:

```
$ make -n install
```

Просмотрите вывод, и если все в порядке (например, не появилась странная установка пакета в любом другом месте, кроме каталога `mycoreutils`), выполните реальную установку:

```
$ make install
```

Теперь в вашем домашнем каталоге появится подкаталог `mycoreutils`, который содержит подкаталоги `bin`, `share` и др. Ознакомьтесь с программами

в подкаталоге `bin` (только что были созданы многие из основных инструментов, описанных в главе 2). Наконец, поскольку вы настроили каталог `mycoreutils` так, чтобы он был независим от остальной части системы, его можно полностью удалить, не нанеся ей вреда.

16.3.2. Установка с помощью инструментов создания пакетов

В большинстве дистрибутивов можно установить новое программное обеспечение в виде пакета, который вы можете поддерживать с помощью инструментов для работы с пакетами дистрибутива. Дистрибутивы на основе Debian, такие как Ubuntu, пожалуй, самые простые в этом плане — вместо простой установки `make` нужно установить пакет с помощью утилиты `checkinstall` следующим образом:

```
# checkinstall make install
```

Эта команда выводит настройки, относящиеся к создаваемому пакету, и дает вам возможность их изменить. Во время установки `checkinstall` отслеживает все файлы, которые должны быть установлены в системе, и помещает их в файл `.deb`. Затем можно использовать команду `dpkg` для установки и удаления нового пакета.

Создание пакета RPM немного сложнее, потому что сначала необходимо создать дерево каталогов для пакета(ов). Это можно сделать с помощью команды `rpmdev-setuptree`. По завершении с помощью утилиты `rpmbuild` выполните остальные шаги. В онлайн-руководстве даны более точные инструкции.

16.3.3. Параметры сценария configure

Ранее мы рассмотрели один из наиболее полезных параметров сценария `configure` — использование `--prefix` для указания каталога установки. По умолчанию цель `install` из файла Makefile, созданного с помощью `autoconf`, использует префикс `/usr/local`. Это значит, что двоичные программы помещаются в `/usr/local/bin`, библиотеки — в `/usr/local/lib` и т. д. Изменить этот префикс можно следующим образом:

```
$ ./configure --prefix=new_prefix
```

В большинстве версий сценария `configure` есть параметр `--help`, который перечисляет другие параметры конфигурации. К сожалению, список настолько длинный, что трудно понять, какие параметры важнее, поэтому перечислим несколько основных:

- `--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` — запрещает пакету создавать разделяемые библиотеки. Экономит время обработки в зависимости от библиотеки (см. подраздел 15.1.3);
- `--with-`*`package`*`=`*`directory`* — сообщает `configure`, что пакет *`package`* находится в каталоге *`directory`*. Удобно, когда необходимая библиотека располагается в нестандартном месте. К сожалению, не все сценарии `configure` распознают этот параметр, поэтому может быть трудно определить точный синтаксис.

ОТДЕЛЬНЫЕ КАТАЛОГИ СБОРКИ

Вы можете создать отдельные каталоги сборки, чтобы поэкспериментировать с некоторыми из перечисленных ранее параметров. Для этого создайте новый каталог в любом месте системы и из него запустите сценарий `configure` в каталоге исходного кода пакета. Сценарий `configure` создаст ферму символических ссылок в новом каталоге сборки, где все они будут указывать на исходное дерево в исходном каталоге пакета. (Некоторые разработчики предпочитают, чтобы пакеты создавались именно таким образом, потому что исходное дерево никогда не изменяется. Также этот способ полезен, если вы хотите выполнить сборку для более чем одной платформы или набора параметров конфигурации с использованием одного и того же пакета исходного кода.)

16.3.4. Переменные окружения

Вы можете влиять на `configure` с помощью переменных окружения, которые сценарий `configure` помещает в переменные `make`. Наиболее важными из них являются `CPPFLAGS`, `CFLAGS` и `LDFLAGS`. Однако имейте в виду, что сценарий `configure` очень требователен к переменным окружения. Например, следует использовать `CPPFLAGS` вместо `CFLAGS` для каталогов файлов заголовков, поскольку `configure` часто запускает препроцессор независимо от компилятора.

Для оболочки `bash` самый простой способ отправить переменную окружения `configure` — это поместить определение переменной перед `./configure` в командной строке. Например, чтобы определить макрос `DEBUG` для препроцессора, используйте следующую команду:

```
$ CPPFLAGS=-DDEBUG ./configure
```

Также можно передать переменную в качестве параметра `configure`, например:

```
$ ./configure CPPFLAGS=-DDEBUG
```

Переменные окружения особенно полезны, когда сценарий `configure` не знает, где искать сторонние включаемые файлы и библиотеки. Например, чтобы выполнить поиск препроцессора в *`include_dir`*, выполните следующую команду:

```
$ CPPFLAGS=-Iinclude_dir ./configure
```

Чтобы компоновщик искал файлы в *lib_dir* (см. подраздел 15.2.6), используйте команду

```
$ LDFLAGS=-Llib_dir ./configure
```

Если в каталоге *lib_dir* есть разделяемые библиотеки (см. подраздел 15.1.3), команда из приведенного ранее примера, скорее всего, не установит путь динамического компоновщика времени исполнения. В этом случае используйте параметр компоновщика `-rpath` вместе с параметром `-L`:

```
$ LDFLAGS="-Llib_dir -Wl,-rpath=lib_dir" ./configure
```

Будьте внимательны при настройке переменных. Маленькая ошибка может привести к сбою компилятора и сценария `configure`. К примеру, вы забыли добавить символ `-` в `-I`, как показано далее:

```
$ CPPFLAGS=Iinclude_dir ./configure
```

Это приводит к ошибке:

```
configure: error: C compiler cannot create executables
See 'config.log' for more details
```

Изучение файла `config.log`, сгенерированного в результате этой неудачной попытки, приводит к следующему:

```
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 Autoconf

Как только сценарий `configure` заработает, вы увидите, что сгенерированный им файл Makefile обладает рядом полезных целей в дополнение к стандартным `all` и `install`:

- `make clean` — удаляет все объектные файлы, исполняемые файлы и библиотеки (см. главу 15);
- `make distclean` — эта цель похожа на `make clean`, за исключением того, что она удаляет все автоматически сгенерированные файлы, включая файлы Makefile, `config.h`, `config.log` и т. д. Идея заключается в том, что после запуска `make distclean` исходное дерево должно выглядеть, как недавно распакованный дистрибутив;
- `make check` — некоторые пакеты поставляются с набором тестов для проверки правильной работы скомпилированных программ, команда `make check` запускает эти тесты;

- `make install-strip` — похожа на `make install`, но при установке удаляет таблицу символов и другую отладочную информацию из исполняемых файлов и библиотек. Урезанные двоичные файлы занимают гораздо меньше места.

16.3.6. Файлы журналов Autoconf

Если во время настройки что-то пойдет не так, а причина неочевидна, стоит обратиться к файлу `config.log`. Однако `config.log` — это гигантский файл, что может затруднить поиск точного источника проблемы.

В этой ситуации необходимо перейти в самый конец файла `config.log` (например, введя прописную букву `G` для команды `less`), а затем прокручивать страницы журнала назад, пока не увидите проблему. Однако в конце файла все еще находится множество данных, потому что `configure` помещает туда всю свою среду, включая выходные переменные, переменные кэша и другие определения. Поэтому перейдите к концу вывода и выполните поиск в обратном направлении для строки, например, `for more details` или какого-либо другого фрагмента текста неудачного вывода `configure`. (Можно инициировать обратный поиск в меньшем количестве данных с помощью команд `less` и `?`.) Скорее всего, ошибка окажется прямо над строкой, найденной при поиске.

16.3.7. Утилита pkg-config

Хранить множество сторонних библиотек в системе может быть непросто. При этом установка каждой из них с отдельным префиксом может привести к проблемам со сборкой пакетов, для которых требуются эти сторонние библиотеки. Например, если вы хотите скомпилировать OpenSSH, вам понадобится библиотека OpenSSL. Какие именно библиотеки требуются и как сообщить процессу настройки OpenSSH местоположение библиотек OpenSSL?

Большинство библиотек теперь используют утилиту `pkg-config` не только для указания местоположения своих файлов и библиотек, но и для задания точных флагов, необходимых для компиляции и компоновки программы. Синтаксис выглядит следующим образом:

```
$ pkg-config options package1 package2 ...
```

Например, чтобы найти библиотеки, необходимые для популярной библиотеки сжатия zlib, выполните следующую команду:

```
$ pkg-config --libs zlib
```

Вывод будет выглядеть следующим образом:

```
-lz
```

Чтобы просмотреть все библиотеки, известные утилите `pkg-config`, включая краткое описание каждой из них, выполните команду

```
$ pkg-config --list-all
```

Как работает утилита pkg-config

Утилита `pkg-config` находит информацию о пакете, читая файлы конфигурации, которые заканчиваются на `.pc`. Например, вот так выглядит пакет `openssl.pc` для библиотеки сокетов OpenSSL, как показано в системе Ubuntu (находится в каталоге `/usr/lib/x86_64-linux-gnu/pkgconfig`):

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib/x86_64-linux-gnu
includedir=${prefix}/include

Name: OpenSSL
Description: Secure Sockets Layer and cryptography libraries and tools
Version: 1.1.1f
Requires:
Libs: -L${libdir} -lssl -lcrypto
Libs.private: -ldl -lz
Cflags: -I${includedir} exec_prefix=${prefix}
```

Вы можете изменить этот файл, например добавив `-Wl,-rpath=${libdir}` к флагам библиотеки, чтобы задать путь поиска библиотеки времени исполнения. Однако более важный вопрос заключается в том, как `pkg-config` находит файлы `.pc`. По умолчанию `pkg-config` просматривает каталог `lib/pkgconfig` своего установочного префикса.

Например, `pkg-config`, установленная с префиксом `/usr/local`, просматривает каталог `/usr/local/lib/pkgconfig`.

ПРИМЕЧАНИЕ

Если не установить пакет разработки, то вы не сможете найти файлы .pc для многих пакетов. Например, чтобы добавить openssl.pc в системе Ubuntu, необходимо установить пакет libssl-dev.

Как установить файлы pkg-config в нестандартных каталогах

К сожалению, по умолчанию `pkg-config` не считывает файлы `.pc` за пределами своего установочного префикса. Это означает, что файл `.pc`, находящийся в нестандартном месте, например в `/opt/openssl/lib/pkgconfig/openssl.pc`, будет недоступен для любой стандартной установки `pkg-config`. Существует два основных способа сделать файлы `.pc` доступными за пределами установочного префикса `pkg-config`:

- создать символические ссылки (или копии) из фактических файлов `.pc` в основной каталог `pkgconfig`;
- установить переменную окружения `PKG_CONFIG_PATH` так, чтобы она включала любые дополнительные каталоги `pkgconfig`. Этот вариант неэффективен с точки зрения системы в целом.

16.4. Процесс установки

Умение собирать и устанавливать программное обеспечение, — это хорошо, но еще более полезно знание того, когда и где устанавливать собственные пакеты. Дистрибутивы Linux стараются добавить как можно больше программного обеспечения при установке, поэтому всегда нужно проверять, не лучше ли установить пакет вручную. Перечислим преимущества ручной установки:

- Возможность настроить параметры пакета по умолчанию.
- При ручной установке пакета зачастую возникает более ясное представление о том, как его использовать.
- Управление версией пакета.
- Проще создать резервную копию пользовательского пакета.
- Проще распространять самостоятельно установленные пакеты по сети (при условии, что архитектура согласована, а место установки относительно изолировано).

Недостатки ручной установки:

- Если пакет уже установлен в вашей системе, то при повторной установке он может перезаписать важные файлы, а это способно вызвать проблемы. Чтобы избежать этого, используйте префикс установки `/usr/local`. Даже если пакет не установлен в вашей системе, следует проверить, имеется ли он для данного дистрибутива. Если есть, помните об этом, чтобы случайно не установить пакет дистрибутива.
- Занимает много времени.
- Пользовательские пакеты не обновляются автоматически. Дистрибутивы поддерживают большинство пакетов в актуальном состоянии, не требуя от вас ручных обновлений. Это особенно важно для пакетов, которые взаимодействуют с сетью, поскольку обновления для системы безопасности всегда должны быть актуальными.
- Если вы установите пакет и не будете его использовать, то напрасно потратите время.
- Возможна неправильная настройка пакетов.

Нет особого смысла устанавливать пакеты, подобные тем, что содержатся в пакете `coreutils`, который мы создали ранее в этой главе (`ls`, `cat` и т. д.), конечно, если

вы не создаете максимально настраиваемую под себя систему. Но если вы хотите использовать сетевые серверы, например Apache, и полностью их контролировать, стоит установить серверы вручную.

16.4.1. Места установки

Префикс `/usr/local` — это префикс по умолчанию в GNU autoconf и многих других пакетах, традиционный каталог для локально установленного программного обеспечения. Обновления операционной системы игнорируют префикс `/usr/local`, поэтому ничего не удаляют и не перезаписывают в нем. Для небольших локальных пакетов программного обеспечения каталог `/usr/local` отлично подойдет. Единственная проблема заключается в том, что если у вас установлено много пользовательских приложений, в каталоге начнется хаос. Он заполнится тысячами странных, маленьких, непонятно откуда взявшихся файлов.

Если каталог заполнился данными и им трудно управлять, создайте собственные пакеты, как описано в подразделе 16.3.2.

16.5. Применение исправлений

Большинство изменений в исходном коде программного обеспечения доступны в виде ветвей (branches) от исходного кода разработчика, например репозитория Git. Однако время от времени вам может потребоваться применить исправление (патч) к исходному коду, чтобы поправить ошибки или добавить функции. Использование `diff` в данном случае равноценно добавлению патча, так как команда предназначена для определения требуемых исправлений.

Начальная часть патча выглядит примерно так:

```
--- 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 более чем одного файла. Три дефиса подряд (`---`) показывают файлы, в которых есть изменения. Всегда обращайте внимание на начальную часть исправления, чтобы определить необходимый рабочий каталог. Предыдущий пример относится к `src/file.c`, поэтому перед применением исправления следует перейти в каталог, содержащий `src`, а *не* в сам каталог `src`.

Чтобы применить патч-исправление, введите команду *`patch`*:

```
$ patch -p0 < patch_file
```

Если все в порядке, команда *`patch`* тихо выполнит обновление определенного набора файлов. Однако она может задать следующий вопрос:

```
File to patch:
```

Это означает, что вы не в нужном каталоге, а также может указывать на то, что ваш исходный код не соответствует исходному коду патча. В этом случае вам не повезло: даже если бы вы могли найти нужную часть файлов, другие файлы не обновятся. В результате останется исходный код, который нельзя скомпилировать.

В некоторых случаях патч может ссылаться на версию пакета, как в примере далее:

```
--- 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 у вас другой номер версии или вы переименовали каталог, дайте задание команде *patch* удалить начальные компоненты пути. Предположим, что вы находитесь в каталоге, содержащем `src` (как в примере). Чтобы команда *patch* проигнорировала часть пути `package-3.42/` (то есть удалила один начальный компонент пути), используйте параметр `-p1`:

```
$ patch -p1 < patch_file
```

16.6. Устранение ошибок компиляции и установки

Если вы понимаете разницу между ошибками и предупреждениями компилятора, ошибками компоновщика и проблемами с разделяемой библиотекой, которые описаны в главе 15, у вас не возникнет особых проблем с исправлением многих ошибок при создании программного обеспечения. В этом разделе рассмотрим самые распространенные проблемы и неполадки. При создании программ с помощью autoconf вероятность столкнуться с подобными ошибками значительно ниже, однако в любом случае стоит понимать, как распознавать такие ошибки.

Прежде чем мы перейдем к рассмотрению проблем, убедитесь, что вы можете читать и понимаете определенные выходные данные команды `make`. Важно знать, в чем заключается разница между ошибкой и игнорируемой ошибкой. Далее приведен пример реальной ошибки:

```
make: *** [target] Error 1
```

Часть файлов Makefiles подозревает, что может возникнуть ошибка, но знает, что эти ошибки безвредны. Любые подобные сообщения можно игнорировать:

```
make: *** [target] Error 1 (ignored)
```

Кроме того, в больших пакетах GNU `make` часто вызывает сама себя, причем каждый экземпляр `make` в сообщении об ошибке помечен `[N]`, где `N` — это номер. Сбой можно быстро найти, посмотрев на конкретную ошибку `make`, которая появляется непосредственно после сообщения об ошибке 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 в примере отображают всю необходимую информацию. Проблема с файлом `file.c`, расположенным в каталоге `/home/src/package-5.0/src`. К сожалению, дополнительный вывод так велик, что может быть трудно определить, что важно. Чтобы быстрее находить и определять ошибки, необходимо научиться их фильтровать.

16.6.1. Особые ошибки

Перечислим распространенные ошибки сборки программного обеспечения.

- **Проблема.** Сообщение об ошибке компилятора:

  ```
  src.c:22: conflicting types for 'item'
  /usr/include/file.h:47: previous declaration of 'item'
  ```

  Причина и исправление. Разработчик сделал ошибку, повторно добавив *item* в строке 22 `src.c`. Чтобы исправить это, удалите ошибочную строку с помощью комментария, `#ifdef` или др.

- **Проблема.** Сообщение об ошибке компилятора:

  ```
  src.c:37: 'time_t' undeclared (first use this function)
  --пропуск--
  src.c:37: parse error before '...'
  ```

  Причина и исправление. Разработчик не указал важный файл заголовка. Чтобы найти отсутствующий файл заголовка, перейдите к страницам руководства. Для исправления этой ошибки прежде всего посмотрите на ошибочную строку (в данном случае строка 37 в `src.c`). Скорее всего, это объявление переменной:

  ```
  time_t v1;
  ```

  Далее выполните поиск `v1` в программе, где выполняется какой-либо вызов функции, например:

  ```
  v1 = time(NULL);
  ```

  Теперь запустите `man 2 time` или `man 3 time`, чтобы найти системные и библиотечные вызовы с именем `time()`. В этом случае на странице руководства в разделе 2 есть нужная информация:

```
SYNOPSIS
    #include <time.h>

    time_t time(time_t *t);
```

Это значит, что `time()` требует файл `time.h`. Поместите `#include <time.h>` в начало `src.c` и повторите попытку.

- **Проблема.** Сообщение об ошибке компилятора (препроцессора):

```
src.c:4: pkg.h: No such file or directory
(long list of errors follows)
```

Причина и исправление. Компилятор запустил препроцессор C на `src.c`, но не смог найти include-файл `pkg.h`. Исходный код, скорее всего, зависит от библиотеки, которую необходимо установить. Другой вариант — может потребоваться предоставить компилятору нестандартный путь include. Для этого нужно добавить параметр `-I` пути include к флагам препроцессора C (`CPPFLAGS`). (Имейте в виду, что может потребоваться также флаг компоновщика `-L` для include-файлов.)

Если дело не в библиотеке, возможно, вы пытаетесь выполнить компиляцию для операционной системы, которую этот исходный код не поддерживает. Проверьте файлы Makefile и `README`, чтобы найти подробную информацию о платформах.

Если вы используете дистрибутив на базе Debian, попробуйте применить команду `apt-file` к имени файла заголовка:

```
$ apt-file search pkg.h
```

Это действие поможет найти необходимый пакет разработки. Для дистрибутивов, использующих `yum`, задействуйте следующую команду:

```
$ yum provides */pkg.h
```

- **Проблема.** Сообщение об ошибке `make`

```
make: prog: Command not found
```

Причина и исправление. Чтобы можно было создать пакет, в системе должна быть программа `prog`. Если `prog` — это `cc`, `gcc` или `ld`, в вашей системе не установлены утилиты разработки. Но если вы знаете, что `prog` уже установлена в системе, попробуйте изменить файл Makefile, чтобы указать полный путь к программе.

В редких случаях при использовании неаккуратного исходного кода `make` выполняет сборку `prog`, а затем немедленно ее применяет, предполагая, что текущий каталог (`.`) находится в вашем командном пути. Если ваш `$PATH` не включает текущий каталог, можете отредактировать файл Makefile и изменить `prog` на `./prog`. В качестве альтернативы можно временно добавить символ «точка» (`.`) в путь.

16.7. Что дальше?

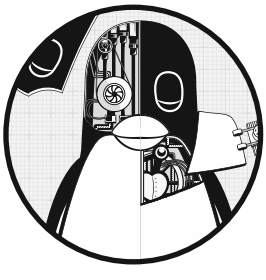
В этой главе мы лишь коснулись основ создания программного обеспечения. После того как вы научитесь собирать программное обеспечение, можете изучить эту тему глубже и узнать:

- как использовать системы сборки, отличные от autoconf, такие как CMake и SCons;
- как настроить сборки для собственного программного обеспечения. Чтобы писать программное обеспечение, необходимо выбрать систему сборки и научиться ее использовать. Для изучения системы сборки GNU autoconf подойдет книга Джона Калькоте (John Calcote) *Autotools*, 2-е издание (No Starch Press, 2019);
- как скомпилировать ядро Linux. Система сборки ядра кардинально отличается от системы сборки других инструментов. Она имеет собственную систему конфигурации, адаптированную для настройки ядра и модулей. Однако вся процедура довольно проста, и если понимать, как работает загрузчик, то никаких проблем с ним возникнуть не должно. Тем не менее нужно внимательно подходить к этому вопросу: убедитесь, что у вас есть старое ядро на случай, если не сможете загрузиться с новым;
- как выглядят исходные пакеты, относящиеся к конкретному дистрибутиву. Дистрибутивы Linux поддерживают собственные версии исходного кода программного обеспечения в виде специальных пакетов исходного кода. Вы можете найти полезные патчи, которые расширяют функциональность или устраняют проблемы в других, неподдерживаемых пакетах. Системы управления исходными пакетами включают инструменты для автоматической сборки, такие как относящийся к Debian `debuild`, а также `mock` на базе RPM.

Создание программного обеспечения является ступенькой к изучению программирования и разработки программного обеспечения. Инструменты, которые мы встречали в этой и предыдущей главах, раскрывают тайну происхождения системного программного обеспечения. И уже не так трудно перейти к изучению исходного кода, внесению изменений и созданию собственного программного обеспечения.

17

Виртуализация



Термин *«виртуальный»* в вычислительных системах довольно расплывчатый. Он применяется в основном для обозначения посредника, который переводит сложный или фрагментированный базовый слой в упрощенный интерфейс для одного или нескольких пользователей. Вернемся к описанию, которое мы видели ранее: виртуальная память позволяет нескольким процессам обращаться к большому банку памяти, создавая отдельный банк памяти для каждого из них.

Подобное определение не совсем очевидно. Можно объяснить типичную цель виртуализации так: это создание изолированных сред для запуска нескольких систем одновременно и без конфликтов.

Поскольку работу виртуальных машин относительно легко понять на более высоком уровне, именно с этого начнем изучение виртуализации. Более того, мы останемся на этом уровне, чтобы объяснить некоторые термины, с которыми вы можете столкнуться, когда будете работать с виртуальными машинами, не вдаваясь в разнообразие реализаций.

Затем мы подробнее рассмотрим технические детали контейнеров. Они строятся с помощью технологии, которую мы обсуждали ранее в этой книге, благодаря чему мы сможем увидеть взаимосвязи элементов в системе. Более того, относительно просто исследовать контейнеры интерактивно.

17.1. Виртуальные машины

Концепция виртуальных машин схожа с концепцией виртуальной памяти, за исключением того, что виртуальная машина охватывает *все* аппаратное обеспечение

машины, а не только память. В случае виртуальных машин вы создаете совершенно новую машину — процессор, память, интерфейсы ввода-вывода и т. д. — с помощью программного обеспечения и запускаете в ней целую операционную систему, включая ядро. Этот тип виртуальной машины называется *системной виртуальной машиной*, он существует уже несколько десятилетий. Например, мейнфреймы IBM традиционно применяют системные виртуальные машины для создания многопользовательской среды, в свою очередь, пользователи получают собственную виртуальную машину, работающую под управлением CMS — простой однопользовательской операционной системы.

Вы можете создать виртуальную машину полностью из программы (эмулятора) или по максимуму использовать базовое оборудование, как происходит при создании виртуальной памяти. Мы будем рассматривать второй вариант создания виртуальной машины из-за его отличной производительности. Однако обратите внимание на то, что ряд популярных эмуляторов поддерживает старые компьютерные и игровые системы, такие как Commodore 64 и Atari 2600.

Мир виртуальных машин разнообразен, в нем используется огромное количество терминов, в которых нужно разбираться. Мы сосредоточимся в первую очередь на том, как сочетаются эта терминология и то, что может встретить обычный пользователь Linux. А также обсудим некоторые различия в виртуальном оборудовании.

ПРИМЕЧАНИЕ

К счастью, использование виртуальных машин намного проще, чем их описание. Например, в VirtualBox вы можете применить графический интерфейс для создания и запуска виртуальной машины или даже использовать средство управления VBoxManage командной строки, если нужно автоматизировать этот процесс в сценарии. Веб-интерфейсы облачных служб также облегчают администрирование виртуальных машин. Благодаря такой простоте использования мы можем сосредоточиться скорее на понимании технологии и терминологии виртуальных машин, чем на деталях эксплуатации.

17.1.1. Гипервизоры

Гипервизор, или *монитор виртуальных машин* (hypervisor, или virtual machine monitor, VMM), — это часть программного обеспечения, которая работает аналогично тому, как операционная система управляет процессами. Существуют два типа гипервизоров, и от них зависит способ применения виртуальной машины. Большинству пользователей лучше знаком *гипервизор типа 2*, поскольку он работает в обычной операционной системе, такой как Linux. Например, VirtualBox — это гипервизор типа 2, и вы можете запустить его в своей системе без значительных изменений. Возможно, вы уже использовали его во время чтения этой книги для тестирования и изучения различных типов систем Linux.

Гипервизор типа 1 больше похож на операционную систему (особенно ядро), созданную специально для быстрого и эффективного запуска виртуальных машин. Этот

тип гипервизора может задействовать обычную сопутствующую систему, такую как Linux, для управления виртуальными машинами. Даже если вы, возможно, никогда не запускали гипервизор типа 1 на своем компьютере, вы все время с ним взаимодействуете. Все службы облачных вычислений работают как виртуальные машины под гипервизорами типа 1 (например, Xen). Чаще всего веб-сайты используют программное обеспечение, работающее на такой виртуальной машине. Создание экземпляра операционной системы в облачной службе, например в AWS, — это создание виртуальной машины на гипервизоре типа 1.

В целом, виртуальная машина со своей операционной системой называется *гостевой*. Хост — это то, что управляет гипервизором. Для гипервизоров типа 2 хост — это просто ваша собственная система. Для гипервизоров типа 1 хостом является сам гипервизор, возможно, в сочетании со специализированной сопутствующей системой.

17.1.2. Оборудование виртуальной машины

Теоретически, гипервизор должен легко предоставлять аппаратные интерфейсы для гостевой системы. Например, чтобы предоставить устройство виртуального диска, вы можете создать большой файл где-нибудь на хосте и дать ему доступ, как к диску со стандартной эмуляцией ввода-вывода устройства. Этот подход эмулирует конкретное аппаратное обеспечение в виде виртуальной машины, однако он неэффективен. Чтобы виртуальные машины стали более гибкими и эффективными для решения различных задач, требуются некоторые изменения.

Большинство различий между реальным и виртуальным оборудованием являются результатом прокидывания (briging), которое позволяет гостям получать более прямой доступ к ресурсам хоста. Обход виртуального оборудования между хостом и гостем известен как *паравиртуализация*. Сетевые интерфейсы и блочные устройства относятся к числу наиболее вероятных объектов паравиртуализации. К примеру, устройство `/dev/xvd` на инстансе облачных вычислений представляет собой виртуальный диск Xen, использующий драйвер ядра Linux для прямого подключения к гипервизору. Иногда паравиртуализация применяется только для удобства: например, в настольной системе, такой как VirtualBox, доступны драйверы для координации движения мыши между окном виртуальной машины и средой хоста.

Какой бы механизм ни использовался, цель виртуализации всегда состоит в том, чтобы гостевая операционная система могла обращаться с виртуальным оборудованием так же, как с любым другим устройством. Это гарантирует, что все слои поверх устройства функционируют должным образом. Например, в гостевой системе Linux это позволяет ядру получать доступ к виртуальным дискам, как к блочным устройствам, чтобы можно было разбивать их на разделы и создавать файловые системы с помощью обычных инструментов.

Режимы процессора виртуальной машины

В рамках этой книги мы не будем обсуждать большую часть деталей работы виртуальных машин, однако процессор заслуживает особого внимания, так как мы уже разбирали разницу между режимом ядра и режимом пользователя. Названия этих режимов различаются в зависимости от процессора (например, процессоры x86 применяют систему, называемую *кольцами защиты*), но суть их не меняется. В режиме ядра процессор может выполнять практически все, в пользовательском режиме некоторые действия запрещены, а доступ к памяти ограничен.

Первые виртуальные машины для архитектуры x86 работали в пользовательском режиме. Это вызывало проблемы, потому что ядро, работающее внутри виртуальной машины, стремится находиться в режиме ядра. Чтобы противостоять этому, гипервизор может обнаруживать любые служебные действия (инструкции), поступающие из виртуальной машины, и реагировать на них (перехватывать). Дальше гипервизор эмулирует служебные инструкции, позволяя виртуальным машинам работать в режиме ядра на архитектуре, не предназначенной для этого. Поскольку большинство инструкций, выполняемых ядром, ничем не ограничены, они выполняются нормально и влияние на производительность минимально.

После внедрения такого рода гипервизора производители процессоров поняли, что существует спрос на процессоры, которые могли бы помочь гипервизору, устранив необходимость в перехвате команд и эмуляции. Intel и AMD выпустили наборы функций VT-x и AMD-V соответственно, и большинство гипервизоров теперь поддерживают их. В некоторых случаях они просто необходимы.

Если вы хотите узнать больше о виртуальных машинах, начните с книги Джима Смита и Рави Наира (Jim Smith, Ravi Nair) *Virtual Machines: Versatile Platforms for Systems and Processes* (Elsevier, 2005). В ней речь идет также о процессных виртуальных машинах, таких как Java virtual machine, которую мы не будем рассматривать.

17.1.3. Использование виртуальных машин

В системах Linux применение виртуальных машин часто попадает в одну из нескольких категорий:

- **Тестирование.** Существует множество вариантов использования виртуальных машин, позволяющих протестировать что-либо за пределами обычного продакшена (production) операционной среды. Например, когда вы разрабатываете программное обеспечение для продакшена, важно тестировать его на отдельной машине. Другой вариант — экспериментировать с новым программным обеспечением (к примеру, новым дистрибутивом) в безопасной одноразовой среде. Виртуальные машины позволяют это делать, не приобретая новое оборудование.

- **Совместимость приложений.** Когда нужно запустить что-либо в операционной системе, отличной от основной, используются виртуальные машины.
- **Серверы и облачные службы.** Как упоминалось ранее, все облачные службы построены на технологии виртуальных машин. Если вам нужно запустить интернет-сервер (например, веб-сервер), самый быстрый способ сделать это — купить экземпляр виртуальной машины у поставщика облачных служб. Облачные провайдеры предлагают также специализированные серверы, такие как базы данных, чье программное обеспечение настроено на работу на виртуальных машинах.

17.1.4. Недостатки виртуальных машин

В течение многих лет виртуальные машины были основным методом изоляции и масштабирования служб. Поскольку есть возможность создавать виртуальные машины несколькими щелчками мыши или с помощью API, становится очень удобно создавать серверы, для которых не требуется устанавливать и обслуживать оборудование. Тем не менее некоторые аспекты виртуальных машин имеют недостатки.

- Установка и/или настройка системы и приложения может быть громоздкой и трудоемкой. Такие инструменты, как Ansible, способны автоматизировать этот процесс, но все равно требуется значительное время, чтобы создать систему с нуля. Если вы используете виртуальные машины для тестирования программного обеспечения, время, затрачиваемое на настройку, будет расти.
- Даже при правильной настройке виртуальные машины запускаются и перезагружаются довольно медленно. Есть несколько способов это обойти, но все равно придется загружать полную систему Linux.
- Необходимо поддерживать полную систему Linux, постоянно обновляя ее и обеспечивая безопасность на каждой виртуальной машине. В таких системах есть `systemd` и `sshd`, а также любые инструменты, от которых зависит приложение.
- В приложении могут возникнуть конфликты со стандартным программным обеспечением, установленным на виртуальной машине. Некоторые приложения имеют странные зависимости и не всегда хорошо ладят с программным обеспечением, найденным на продакшен-машине. Кроме того, зависимости, например библиотеки, могут изменяться при обновлении машины, нарушая работу уже настроенных зависимостей.
- Изоляция служб на отдельных виртуальных машинах стоит довольно дорого. Нужна изоляция для того, чтобы запускать в системе не более одной надежной и простой службы приложения. Кроме того, некоторые службы могут дополнительно сегментироваться, так что если вы запускаете несколько веб-сайтов, лучше всего хранить их на разных серверах. Однако это только

увеличивает затраты, особенно при использовании облачных служб, которые взимают плату за каждый экземпляр виртуальной машины.

Эти проблемы на самом деле ничем не отличаются от тех, с которыми вы столкнулись бы при запуске служб на реальном оборудовании, и не являются препятствиями для работы с небольшими службами. Но как только вы начнете запускать больше служб, недостатки станут заметнее, а это потребует времени и денег. Именно в таком случае стоит рассмотреть контейнеризацию служб.

17.2. Контейнеры

Виртуальные машины отлично подходят для изоляции всей операционной системы и ее набора запущенных приложений, но случается так, что для этого нужен другой, более легкий способ. В настоящее время лучший вариант — контейнеры. Прежде чем перейти к деталям, давайте сделаем шаг назад, чтобы увидеть, как появилась технология контейнеризации.

Традиционный способ работы компьютерных сетей заключался в запуске множества служб на одной и той же физической машине: например, сервер имен мог также выступать в качестве сервера электронной почты и выполнять другие задачи. Тем не менее не стоит доверять любому программному обеспечению, включая серверы, и считать его полностью безопасным или стабильным. Для повышения безопасности системы и предотвращения взаимодействия служб друг с другом существует несколько основных способов создания барьеров вокруг серверных демонов.

Одним из методов изоляции служб является использование системного вызова `chroot()` для изменения корневого каталога на что-либо другое, отличное от фактического системного корня. Программа может изменить свой корневой каталог на, к примеру, каталог `/var/spool/my_service` и больше не сможет получить доступ к чему-либо за его пределами. На самом деле существует программа `chroot`, которая позволяет запускать программу с новым корневым каталогом. Этот тип изоляции иногда называют *тюрьмой chroot* (chroot jail), потому что процессы обычно не могут избежать ее.

Другим типом изоляции является функция ограничения ресурсов (`rlimit`) ядра, которая ограничивает количество процессорного времени, которое может потреблять процесс, или ограничивает размер его файлов.

Идеи, на базе которых выстраиваются контейнеры, основываются на том, что пользователь сменяет окружение и ограничивает ресурсы, с помощью которых выполняются процессы. Хотя нет единой определяющей функции, *контейнер* можно свободно определить как ограниченное окружение выполнения для набора процессов. Подразумевается, что эти процессы не могут влиять на что-либо в системе за пределами этого окружения. В целом это называется *виртуализацией на уровне операционной системы*.

Важно иметь в виду, что машина, на которой работает один или несколько контейнеров, все еще имеет только одно базовое ядро Linux. Однако процессы внутри контейнера могут работать в среде пользовательского пространства из дистрибутива Linux, отличного от базовой системы.

Ограничения в контейнерах заданы с помощью ряда функций ядра. Некоторые из важных аспектов процессов, выполняемых в контейнере, заключаются в следующем:

- У них есть свои собственные группы управления.
- У них есть собственные устройства и файловая система.
- Они не могут видеть любые другие процессы в системе или взаимодействовать с ними.
- У них есть собственные сетевые интерфейсы.

Собрать все эти детали воедино — сложная задача. Все можно изменить вручную, но и это может быть непросто — даже связь с группами управления уже затрудняет процесс. Существует множество инструментов, которые могут выполнять подзадачи, необходимые для создания эффективных контейнеров и управления ими. Два из самых популярных — Docker и LXC. В этой главе основное внимание мы уделим инструменту Docker, но коснемся и LXC, чтобы изучить их различия.

17.2.1. Docker, Podman и привилегии

Для выполнения упражнений, данных в этой книге, вам понадобится инструмент управления контейнерами. Приведенные в этом разделе примеры построены с помощью инструмента Docker, который можно без проблем установить с помощью дистрибутива.

Помимо Docker существует инструмент под названием Podman. Основное различие между ними заключается в том, что Docker требует, чтобы сервер работал при использовании контейнеров, в то время как Podman это не нужно. От этого зависит, как системы настраивают контейнеры. Большинство конфигураций Docker требуют привилегий суперпользователя для доступа к функциям ядра, применяемым его контейнерами, и демон `docker` выполняет соответствующую работу. В отличие от Docker, запустить Podman вы можете от имени обычного пользователя, что называется *rootless-операцией*. При таком запуске он применяет различные методы изоляции.

Можно запустить Podman и от имени суперпользователя, заставив его переключиться на методы изоляции, которые использует Docker. И наоборот, более новые версии Docker поддерживают режим `rootless`.

К счастью, команды Podman соответствуют командам Docker командной строки. Это означает, что вы можете заменить `podman` на `docker` в приведенных далее

примерах и они все равно будут работать. Однако существуют различия в реализациях, особенно при запуске Podman в `rootless`-режиме (в примерах отмечено, когда можно применять `podman`).

17.2.2. Пример использования инструмента Docker

Простейший способ изучить работу контейнеров — приступить к практике. Приведенный далее пример иллюстрирует основные функции Docker, благодаря которым контейнеры работают. Однако мы не будем подробно изучать здесь этот инструмент. Для ознакомления используйте онлайн-руководство, а также книгу Найджела Пультона (Nigel Poulton) *Docker Deep Dive* (2016).

Для запуска контейнера нужно создать *образ*, который содержит файловую систему и несколько других определяющих функций. Эти образы почти всегда будут основаны на уже готовых образах, загруженных из интернет-репозитория.

ПРИМЕЧАНИЕ

Легко перепутать образы и контейнеры. Представляйте образ как файловую систему контейнера: процессы не выполняются в образе, но выполняются в контейнерах. Этот пример не совсем точен (в частности, меняя файлы в контейнере Docker, вы не вносите изменений в образ), но на данный момент его достаточно.

Установите Docker в своей системе (дополнительный пакет вашего дистрибутива), создайте где-нибудь новый каталог, перейдите в него и создайте файл `Dockerfile`, содержащий строки

```
FROM alpine:latest
RUN apk add bash
CMD ["/bin/bash"]
```

В этой конфигурации используется облегченный дистрибутив Alpine. Единственное изменение, которое мы вносим, — это добавление оболочки `bash`. Она нужна не только для дополнительного удобства применения в интерактивном режиме, но и для создания уникального образа и просмотра того, как работает эта процедура. Можно использовать общедоступные образы и не вносить в них никаких изменений (часто так и происходит). В этом случае вам не требуется файл `Dockerfile`.

Соберите образ с помощью следующей команды, которая считывает файл `Dockerfile` в текущем каталоге и применяет идентификатор `hlw_test` к образу:

```
$ docker build -t hlw_test .
```

ПРИМЕЧАНИЕ

Возможно, вам потребуется добавить свою учетную запись в группу docker в системе, чтобы иметь возможность запускать команды Docker как обычному пользователю.

Появится большой вывод. Прочитайте его внимательно, чтобы понять, как работает Docker. Разделим вывод на части, соответствующие строкам файла `Dockerfile`. Первая задача — получить последнюю версию дистрибутивного контейнера Alpine из реестра Docker registry:

```
Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM alpine:latest
latest: Pulling from library/alpine
cbdbe7a5bc2a: Pull complete
Digest: sha256:9a839e63dad54c3a6d1834e29692c8492d93f90c59c978c1ed79109ea4b9a54
Status: Downloaded newer image for alpine:latest
 ---> f70734b6a266
```

Обратите внимание на то, что часто используются дайджесты SHA256 и более короткие идентификаторы. Это нормально — инструменту Docker нужно отслеживать множество мелких частей. На этом этапе Docker создает новый образ с идентификатором `f70734b6a266` для базового образа дистрибутива Alpine. Вы можете обратиться к этому конкретному образу позже, но вряд ли это придется делать, потому что это не окончательный образ. Docker добавит слои поверх него. Образ, который не станет конечным, называется *промежуточным*.

ПРИМЕЧАНИЕ

При использовании инструмента Podman вывод будет отличаться, но последовательность шагов та же.

Следующая часть нашей конфигурации — это установка пакета оболочки `bash` в Alpine. Скорее всего, вы узнали вывод, полученный в результате выполнения команды `apk add bash` (выделено жирным шрифтом):

```
Step 2/3 : RUN apk add bash
 ---> Running in 4f0fb4632b31
fetch http://dl-cdn.alpinelinux.org/alpine/v3.11/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.11/community/x86_64/APKINDEX. tar.gz
(1/4) Installing ncurses-terminfo-base (6.1_p20200118-r4)
(2/4) Installing ncurses-libs (6.1_p20200118-r4)
(3/4) Installing readline (8.0.1-r0)
(4/4) Installing bash (5.0.11-r1)
Executing bash-5.0.11-r1.post-install
Executing busybox-1.31.1-r9.trigger
OK: 8 MiB in 18 packages
Removing intermediate container 4f0fb4632b31
 ---> 12ef4043c80a
```

Неочевидно, *как* именно это действие произошло. Скорее всего, на вашей машине не используется Alpine. Как же вы можете запустить принадлежащую дистрибутиву Alpine команду `apk`, которой, вероятнее всего, нет на машине?

Ключом к пониманию является строка `Running in 4f0fb4632b31` (Выполняется в 4f0fb4632b31). Вы еще не запрашивали контейнер, но Docker настроил новый

контейнер с промежуточным образом Alpine из предыдущего шага. Контейнеры также имеют идентификаторы, которые, к сожалению, ничем не отличаются от идентификаторов образов. Еще бóльшую путаницу вносит то, что Docker называет временный контейнер *промежуточным контейнером (intermediate container)*, который отличается от промежуточного образа. Промежуточные образы остаются после сборки, промежуточные контейнеры — нет.

После настройки временного контейнера с идентификатором `4f0fb4632b31` Docker запускает внутри него команду `apk` для установки `bash`, а затем сохраняет полученные изменения в файловой системе в новый промежуточный образ с идентификатором `12ef4043c80a`. Обратите внимание на то, что Docker также удаляет контейнер после завершения.

Наконец, Docker вносит окончательные изменения, необходимые для запуска оболочки `bash` при запуске контейнера из нового образа:

```
Step 3/3 : CMD ["/bin/bash"]
 ---> Running in fb082e6a0728
Removing intermediate container fb082e6a0728
 ---> 1b64f94e5a54
Successfully built 1b64f94e5a54
Successfully tagged hlw_test:latest
```

ПРИМЕЧАНИЕ

Все, что делается с помощью команды RUN в файле настройки, происходит во время сборки образа, а не после, когда вы запускаете контейнер с образом. Команда CMD предназначена для времени исполнения контейнера, вот почему она выполняется в конце.

Теперь у вас есть окончательный образ с идентификатором `1b64f94e5a54`, но поскольку вы его пометили (двумя отдельными шагами), то можете ссылаться на него, также как на `hlw_test` или `hlw_test:latest`. Запустите `docker images`, чтобы убедиться, что ваш образ Alpine присутствует:

```
$ docker images
REPOSITORY       TAG         IMAGE ID         CREATED          SIZE
hlw_test         latest      1b64f94e5a54     1 minute ago     9.19MB
alpine           latest      f70734b6a266     3 weeks ago      5.61MB
```

Запуск контейнеров Docker

Теперь можно запустить контейнер. Есть два основных способа: можно либо создать контейнер, а затем запустить что-то внутри него (в два отдельных шага), либо просто создать и запустить его за один шаг. Давайте сразу перейдем ко второму варианту и начнем с образа, который собрали ранее:

```
$ docker run -it hlw_test
```

Далее появится приглашение оболочки `bash`, в котором вы сможете запускать команды в контейнере. Эта оболочка работает от имени суперпользователя.

ПРИМЕЧАНИЕ

Если не добавить параметры -it (Interactive — интерактивный, connect a terminal — подключается к терминалу), приглашение не появится и работа контейнера завершится почти сразу. Эти параметры, особенно -t, нечасто используются для повседневных задач.

Чтобы осмотреть контейнер, выполните команды, такие как `mount` и `ps`, и изучите файловую систему в целом. Хотя большинство данных выглядят так же, как данные в типичной системе Linux, некоторые части системы выглядят по-другому. Например, если вывести полный список процессов, появятся только две строки:

```
# ps aux
PID   USER     TIME COMMAND
    1 root      0:00 /bin/bash
    6 root      0:00 ps aux
```

Из примера видно, что в контейнере оболочка имеет идентификатор процесса 1 (в обычной системе используется `init`), и кроме вывода списка процессов больше ничего не выполняется.

На этом этапе важно помнить, что это те же процессы, которые можно встретить в обычной (хостовой) системе. Если вы откроете другое окно оболочки в своей хост-системе, то сможете найти процесс контейнера в списке. Вот как это будет выглядеть:

```
root     20189 0.2 0.0     2408 2104 pts/0     Ss+ 08:36     0:00 /bin/bash
```

Это наше первое знакомство с одной из функций ядра для контейнеров — пространства имен ядра Linux специально для идентификаторов процессов. Процесс может создать совершенно новый набор идентификаторов процессов для себя и своих дочерних элементов, начиная с PID 1, и лишь эти идентификаторы будут видимы для дерева процессов.

Объединенные файловые системы Overlay

Теперь изучим файловую систему в контейнере. Она сама по себе минимальна, потому что основана на дистрибутиве Alpine. Мы используем Alpine не только потому, что он небольшой, но и потому, что он отличается от привычных дистрибутивов. Однако, если посмотреть, как монтируется корневая файловая система, можно увидеть, что она сильно отличается от обычного монтирования устройства:

```
overlay on / type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay2/l/
C3D66CQYRP4SCXWFFY6HHF6X5Z:/var/lib/docker/overlay2/l/K4BLIOMNRROX3SS5GFPB
7SFISL:/var/lib/docker/overlay2/l/2MKIOXW5SUB2YDOUBNH4G4Y7KF❶,upperdir=/
var/lib/docker/overlay2/d064be6692c0c6ff4a45ba9a7a02f70e2cf5810a15bcb2b728b00
dc5b7d0888c/diff,workdir=/var/lib/docker/overlay2/d064be6692c0c6ff4a45ba9a7a02
f70e2cf5810a15bcb2b728b00dc5b7d0888c/work)
```

Это объединенная *файловая система (overlay)* — функция ядра, которая позволяет создавать файловую систему путем объединения существующих каталогов в виде слоев с изменениями, хранящимися в одном месте. Если вы посмотрите на свою хост-систему, то увидите эту систему и получите доступ к каталогам компонентов, а также найдете место, где утилита Docker запустила исходное монтирование.

ПРИМЕЧАНИЕ

В режиме rootless утилита Podman использует версию FUSE файловой системы overlay. В этом случае вы не увидите эту подробную информацию при монтировании файловой системы, но сможете получить аналогичную информацию, изучив процессы fuse-overlayfs в хост-системе.

В выводе команды `mount` есть параметры каталогов `lowerdir`, `upperdir` и `workdir`. Нижний каталог на самом деле представляет собой серию каталогов, разделенных двоеточием, и если вы посмотрите на них в своей хост-системе, то обнаружите, что последний каталог ❶ — это базовый дистрибутив Alpine, который был настроен на первом этапе сборки образа (внутри виден корневой каталог дистрибутива). При просматривании двух предыдущих каталогов видно, что они соответствуют двум другим шагам сборки. Поэтому эти каталоги накладываются друг на друга справа налево.

Верхний каталог находится поверх всех остальных, и в нем также отображаются любые изменения в смонтированной файловой системе. Он не обязательно будет пустым, когда вы его монтируете, но для работы с контейнерами нет смысла что-либо в него добавлять. Рабочий каталог — это место, где драйвер файловой системы выполняет свои задачи перед записью изменений в верхний каталог, и он должен быть пустым при монтировании.

В целом, образы контейнеров со многими этапами сборки довольно многослойны, и иногда это может стать проблемой. Для минимизации количества слоев существуют различные стратегии, такие как объединение команд `RUN` и многоуровневых построений. Но мы не будем вдаваться в подробности.

Сеть

Можно выбрать запуск контейнера в той же сети, к которой относится и хост-машина, однако обычно для обеспечения безопасности требуется определенная изоляция в сетевом стеке. Docker помогает добиться изоляции несколькими способами, однако по умолчанию (и чаще всего) используется *сетевой мост* (имеет другой вид пространства имен — сетевое пространство имен `netns`). Перед запуском Docker создает в хост-системе новый сетевой интерфейс (обычно `docker0`), который назначается частной сети, например 172.17.0.0/16, поэтому интерфейс в этом случае получит адрес 172.17.0.1. Эта сеть предназначена для связи между хост-машиной и ее контейнерами.

Затем при создании контейнера Docker создает новое, практически пустое сетевое пространство имен. Сначала новое пространство имен, которое будет находиться в контейнере, содержит только новый частный интерфейс возвратной петли (`lo`). Чтобы подготовить пространство имен для работы, Docker создает на хосте виртуальный интерфейс, который имитирует связь между двумя реальными сетевыми интерфейсами, каждый со своим собственным устройством, и помещает одно из этих устройств в новое пространство имен. При конфигурации сети с использованием адреса в сети Docker (в нашем случае 172.17.0.0/16) на устройстве в новом пространстве имен процессы могут отправлять пакеты в этой сети и получать их на хосте. Это может привести к путанице, поскольку разные интерфейсы в разных пространствах имен могут иметь одно и то же имя (например, контейнер может быть eth0, а также хост-машиной).

Поскольку при этом задействуется частная сеть (а сетевой администратор не будет вслепую направлять что-либо в эти контейнеры и из них), если оставить этот путь, процессы контейнера, использующие данное пространство имен, не смогут выйти за пределы системы. Чтобы обеспечить возможность доступа к внешним хостам, сеть Docker на хосте настраивает преобразование сетевых адресов NAT.

На рис. 17.1 показана стандартная настройка сетевого моста Docker. Она включает в себя физический уровень с интерфейсами, а также межсетевой уровень подсети Docker и NAT, связывающий эту подсеть с остальной частью хост-машины и ее внешними подключениями.

ПРИМЕЧАНИЕ

Может потребоваться изучить подсеть сети интерфейса Docker, так как между ней и сетью на основе NAT, назначенной оборудованием маршрутизатора от телекоммуникационных компаний, могут возникать конфликты.

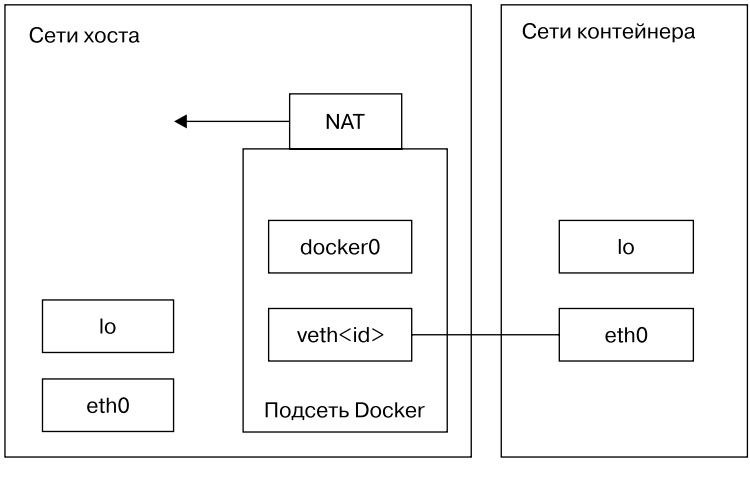


Рис. 17.1. Сетевой мост в Docker. Толстая линия — это связь двух виртуальных интерфейсов

Сеть `rootless` в Podman отличается тем, что для настройки виртуальных интерфейсов требуется доступ суперпользователя. Podman также задействует новое сетевое пространство имен, но ему нужен интерфейс, который можно настроить для работы в пользовательском пространстве. Это интерфейс TAP (обычно в tap0), и в сочетании с демоном пересылки slirp4netns контейнерные процессы могут выходить в сеть. Это не так эффективно — например, контейнеры не могут соединяться друг с другом.

В сети есть гораздо больше возможностей, в том числе как предоставлять порты в сетевом стеке контейнера для использования внешними службами. Однако важнее всего уметь понимать топологию сети.

Работа утилиты Docker

На этом этапе мы могли бы продолжить обсуждение других видов изоляции и ограничений, которые реализует Docker, но это займет много времени, однако суть работы утилиты ясна. Контейнеры создаются не из одной конкретной функции, а скорее из нескольких. Поэтому утилита Docker должна отслеживать все, что происходит при создании контейнера, а также иметь возможность их очищать.

Docker определяет контейнер как запущенный, если в нем запущен процесс. Вы можете вывести текущие запущенные контейнеры с помощью команды `docker ps`:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND      CREATED      STATUS     PORTS   NAMES
bda6204cecf7   hlw_test   "/bin/bash"  8 hours ago  Up 8 hours
        boring_lovelace
8a48d6e85efe   hlw_test   "/bin/bash"  20 hours ago Up 20 hours
       awesome_elion
```

Как только все процессы завершаются, Docker переводит их в состояние выхода, но по-прежнему сохраняет контейнеры (если команда не началась с параметра `--rm`). Это включает в себя изменения, внесенные в файловую систему. Чтобы получить доступ к файловой системе, введите команду `docker export`.

Команда `docker ps` по умолчанию не отображает закончившие работу контейнеры, поэтому, чтобы увидеть все данные, примените параметр `-a`. Такие контейнеры быстро накапливаются, и если приложение, запущенное в контейнере, создает много данных, на диске может не хватить места. Используйте команду `docker rm`, чтобы удалить закончивший работу контейнер.

Это относится и к старым образам. Разработка образа часто повторяется, и если помечать новый образ тем же тегом, что и существующий, Docker не будет удалять исходный. Старый образ просто останется без тега. Запустите команду `docker images`, чтобы отобразить все образы в вашей системе. Вот пример старого образа без тега:

```
$ docker images
REPOSITORY          TAG         IMAGE ID         CREATED          SIZE
hlw_test            latest      1b64f94e5a54     43 hours ago     9.19MB
<none>              <none>      d0461f65b379     46 hours ago     9.19MB
alpine              latest      f70734b6a266     4 weeks ago      5.61MB
```

Используйте команду `docker rmi`, чтобы удалить образ. Команда удаляет и все ненужные промежуточные образы, на которых строится основной. Устаревшие образы также имеют тенденцию накапливаться, если их не удалять. В зависимости от того, что содержится в образах и как они построены, они могут занимать значительное количество места в системе.

В целом, Docker выполняет множество тщательных проверок версий и контрольных точек. Этот уровень управления отражает особую философию по сравнению с такими инструментами, как LXC, с которым мы познакомимся далее.

Модели процессов службы Docker

Одним из потенциально запутанных аспектов контейнеров Docker является жизненный цикл процессов внутри них. Прежде чем процесс сможет полностью завершиться, его процесс-родитель должен собрать («добыть») свой код возврата с помощью системного вызова `wait()`. Однако в контейнере случается, что бездействующие процессы сохраняются, потому что родительские процессы не смогли правильно на них отреагировать. Наряду с тем, как настроены многие образы, напрашивается такой вывод: не нужно запускать несколько процессов или служб внутри контейнера Docker. Однако это неверно.

В контейнере может выполняться множество процессов. Оболочка из приведенного ранее примера запускает новый дочерний процесс при выполнении команды. Здесь имеет значение лишь то, что родительский процесс удаляет все дочерние, после того как они выполнят свои задачи. Так происходит чаще всего, однако при определенных обстоятельствах может возникнуть ситуация, когда родительский процесс не удаляет дочерний, особенно если не знает о его существовании. Это может произойти, когда существует несколько уровней порождения процесса, и PID 1 внутри контейнера оказывается родителем дочернего элемента, о котором не знает.

Чтобы исправить проблему, когда служба просто порождает некоторые процессы и позволяет им «висеть», даже когда контейнер должен закончить свою работу, добавьте параметр `--init` для команды `docker run`. Такая команда создаст очень простой процесс инициализации `init`, который запускается как PID 1 в контейнере и действует как процесс-родитель, удаляющий завершенные дочерние процессы.

Однако если вы запускаете несколько служб или задач внутри контейнера (например, несколько исполнителей для сервера заданий), вместо того чтобы запускать их с помощью сценария, то для запуска и мониторинга можно использовать демон управления процессами, например Supervisor (`supervisord`). Он обеспечивает не

только необходимую функциональность системы, но и больший контроль над процессами служб.

Теперь рассмотрим модель контейнера, которая не связана с инструментом Docker.

17.2.3. Система LXC

Мы так долго обсуждали Docker не только потому, что это самая популярная система для создания образов контейнеров, но и потому, что она позволяет очень легко начать работу и перейти к уровням изоляции, которые обеспечивают контейнеры. Однако существуют и другие пакеты для создания контейнеров, у них разные подходы. Старейшим из них является система LXC. Фактически первые версии Docker были построены на LXC. Если вы поняли, как Docker выполняет свои задачи, у вас не должно возникнуть проблем с техническими концепциями LXC, поэтому обойдемся без примеров. Вместо этого просто рассмотрим некоторые практические различия между инструментами.

Термин «LXC» иногда используется для обозначения набора функций ядра, которые делают возможными контейнеры, но большинство пользователей применяют его специально для обозначения библиотеки и пакета, содержащих ряд утилит для создания контейнеров Linux и управления ими. В отличие от Docker, LXC требует довольно много ручных настроек. Например, вам необходимо создать собственный сетевой интерфейс контейнера и предоставить сопоставления идентификаторов пользователей.

Первоначально система LXC должна была стать как можно большей частью всей системы Linux внутри контейнера, с инициализацией и всем остальным. После установки специальной версии дистрибутива вы можете установить все необходимое для того, что запускали внутри контейнера. Эта часть незначительно отличается от того, что делает Docker, но впереди еще выполнение множества настроек, а с Docker вы просто загружаете кучу файлов — и все готово к работе.

Таким образом, LXC — система более гибкая в том, что касается адаптации к различным потребностям. Например, по умолчанию LXC не применяет файловую систему overlay, которая имеется в Docker, хотя ее можно добавить. Поскольку система LXC построена на C API, по желанию можете использовать эту гранулярность в собственном программном приложении.

Сопутствующий пакет управления под названием LXD помогает разобраться с более тонкими ручными функциями LXC, такими как создание сети и управление образами, и предлагает REST API, что можно использовать для доступа к LXC вместо C API.

17.2.4. Платформа Kubernetes

Контейнеры часто используются для работы со многими видами веб-серверов, потому что вы можете запускать несколько контейнеров из одного образа на нескольких

компьютерах, обеспечивая отличную эффективность. К сожалению, эффективную работу контейнеров нелегко обеспечить. Необходимо выполнять следующие задачи:

- отслеживать, какие машины могут запускать контейнеры;
- осуществлять запуск, мониторинг и перезапуск контейнеров на этих машинах;
- настраивать запуск контейнера;
- настраивать сеть контейнеров по мере необходимости;
- загружать новые версии образов контейнеров и внимательно обновлять все запущенные контейнеры.

И это не полный список, он не отражает всей сложности каждой задачи. Для выполнения этих действий необходимо было новое программное обеспечение, и им стала платформа Google Kubernetes. Возможно, одним из важнейших факторов, обеспечивших ее популярность, является ее способность запускать образы контейнеров Docker.

У Kubernetes есть две основные стороны, как у любого клиент-серверного приложения. Сервер включает в себя машины, доступные для запуска контейнеров, а клиент — это в первую очередь набор утилит командной строки, которые запускают наборы контейнеров и управляют ими. Файлы конфигурации для контейнеров и групп, которые они образуют, могут быть обширными, и большая часть работы, выполняемой на стороне клиента, заключается в создании соответствующей конфигурации.

Вы можете изучить конфигурацию самостоятельно. Если не хотите заниматься настройкой серверов вручную, применяйте инструмент Minikube для установки виртуальной машины с кластером Kubernetes на своей машине.

17.2.5. Ошибки работы контейнеров

Если задуматься, как работает такой сервис, как Kubernetes, можно понять, что создание системы, использующей контейнеры, требует затрат. Как минимум необходима одна или несколько машин для запуска контейнеров, и это должна быть полноценная машина Linux, хоть на реальном оборудовании, хоть на виртуальной машине. Затраты на обслуживание тоже никуда не делись, хотя, возможно, поддерживать эту основную инфраструктуру будет проще, чем конфигурацию, требующую множества пользовательских установок программного обеспечения.

Затраты могут принимать несколько форм. Если вы решите управлять собственной инфраструктурой, потребуется значительное время, а также затраты на оборудование, хостинг и техническое обслуживание. Если же решите использовать службу для контейнеризации, например кластер Kubernetes, то будете оплачивать работу других людей.

Особенности контейнеров:

- **С точки зрения памяти в хранилище контейнерам может потребоваться слишком много места.** Для того чтобы любое приложение могло функционировать внутри контейнера, последний должен иметь всю необходимую поддержку операционной системы Linux, например разделяемые библиотеки. Они могут занимать много места, особенно если не обращать внимания на подбираемый для контейнеров базовый дистрибутив. А насколько велико само приложение? Ситуация улучшается, если использовать файловую систему overlay с несколькими копиями одного и того же контейнера, поскольку они применяют одни и те же базовые файлы. Однако, если ваше приложение создает много данных во время выполнения, верхние слои всех этих наложений могут увеличиться.
- **Работая с контейнерами, невозможно обойти стороной другие системные ресурсы, например процессорное время.** Вы можете настроить ограничения на объем потребления ресурсов контейнерами, но при этом будете по-прежнему ограничены тем, сколько может обработать основная система. И ядро, и блочные устройства также продолжают работу. Если ресурсы системы закончатся, то пострадают контейнеры, система под ними или и то и другое сразу.
- **Возможно, придется изменить место хранения своих данных.** В контейнерных системах по типу Docker, использующих файловые системы overlay, изменения, внесенные в файловую систему во время выполнения, удаляются после завершения процессов. Во многих приложениях все пользовательские данные попадают в базу данных, которую также необходимо администрировать. А что насчет журналов? Для хорошо функционирующего серверного приложения они необходимы, и их нужно где-то хранить. Для масштабных production-систем необходимо иметь отдельную службу журналирования.
- **Большинство инструментов и модели для работы с контейнерами ориентированы на веб-серверы.** Если вы используете обычный веб-сервер, то найдете широкую поддержку и информацию о запуске веб-серверов в контейнерах. В частности, Kubernetes обладает множеством функций безопасности для предотвращения выполнения «runaway» (потерявшего контроль) кода сервера. Это плюс, потому что такая возможность компенсирует то, насколько, откровенно говоря, плохо написано большинство веб-приложений. Однако если попытаться запустить другой тип службы, иногда может показаться, что вы пытаетесь протолкнуть куб в круглое отверстие.
- **Неправильная сборка контейнеров может вызвать проблемы с конфигурацией и сбои в работе.** То, что вы создаете изолированную среду, не защищает вас от ошибок в ней. Возможно, вам не придется сильно беспокоиться о сложностях системы, но многое может пойти не так. Если в системе возникают проблемы, неопытные пользователи, как правило, хаотически что-то добавляют, чтобы их устранить. Система, возможно, и начнет функционировать,

но появится множество дополнительных проблем. Важно понимать, какие изменения вы вносите.

- **Управлять версиями контейнеров может быть довольно проблематично.** Для примеров в книге мы использовали тег latest. Предполагается, что это последняя стабильная версия контейнера. Однако также это значит, что при создании контейнера на основе последнего выпуска дистрибутива или пакета новые элементы могут изменить или нарушить работу вашего приложения. Лучше всего применять определенный тег версии базового контейнера.
- **Проблемы с безопасностью.** Особенно это относится к образам, созданным с помощью Docker. Когда вы основываете свои контейнеры на тех, что находятся в репозитории образов Docker, вы доверяетесь репозиторию, надеясь, что не возникнет еще больших проблем с безопасностью. В отличие от Docker, в LXC рекомендуется создавать собственные контейнеры.

Можно подумать, что контейнеры, у которых столько проблем, имеют больше недостатков по сравнению с другими способами управления системными окружениями. Однако это не так. Независимо от того, какой подход вы выберете, эти проблемы в той или иной степени и форме будут — и некоторыми из них легче управлять именно в контейнерах. Просто помните, что контейнеры не решают все проблемы. Например, если вашему приложению требуется много времени для запуска в обычной системе (после загрузки), оно также будет медленно запускаться в контейнере.

17.3. Виртуализация времени исполнения

Последний вид виртуализации, о котором следует упомянуть, основан на типе окружения, используемого для разработки приложения. Он отличается от системных виртуальных машин и контейнеров, которые мы изучали ранее, потому что в нем приложения не размещаются на разных машинах. Вместо этого выполняется разделение, которое применяется только к определенному приложению.

Причина существования такого рода окружений заключается в том, что несколько приложений в одной и той же системе могут использовать один и тот же язык программирования, что чревато конфликтами. Например, Python применяется в нескольких местах в типичном дистрибутиве и может включать множество дополнительных пакетов. Если задействовать системную версию Python в своем пакете, могут возникнуть проблемы, если нужна другая версия одного из дополнений.

Рассмотрим, как функция *виртуального окружения Python* создает версию Python только с нужными пакетами. Начать можно с создания нового каталога для среды, подобной этой:

```
$ python3 -m venv test-venv
```

ПРИМЕЧАНИЕ

К тому времени, когда вы прочтете этот раздел, вероятно, можно будет ввести python вместо python3.

Теперь посмотрите на новый каталог `test-venv`. Вы увидите ряд системных каталогов, таких как `bin`, `include` и `lib`. Чтобы активировать виртуальное окружение, необходимо не выполнять сценарий как обычный сценарий, а «активировать» его: `test-venv/bin/activate`:

```
$ . test-env/bin/activate
```

Необходимость выбора источника выполнения заключается в том, что активация, по сути, устанавливает переменную окружения, чего вы не можете сделать, запустив исполняемый файл. На этом этапе при запуске Python вы получаете версию в каталоге `test-venv/bin`, который сам по себе является лишь символической ссылкой, а переменная окружения `VIRTUAL_ENV` устанавливается в базовый каталог окружения. Запустите команду `deactivate`, чтобы выйти из виртуального окружения.

С таким набором переменных окружения вы получаете новую, пустую библиотеку пакетов в `test-venv/lib`. Все новое, что вы устанавливаете в окружении, отправляется туда, а не в библиотеку основной системы.

Не все языки программирования допускают использование виртуальных окружений так, как это делает Python, и об этом стоит знать хотя бы для того, чтобы прояснить некоторую путаницу, вызываемую словом *«виртуальный»*.

Библиография

Abrahams P. W., Larson B. UNIX for the Impatient, 2nd ed. Boston: Addison-Wesley Professional, 1995.

Aho A. V., Kernighan B. W., Weinberger P. J. The AWK Programming Language. Boston: Addison-Wesley, 1988.

Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: Principles, Techniques, and Tools, 2nd ed. Boston: Addison-Wesley, 2006.

Aumasson J.-P. Serious Cryptography: A Practical Introduction to Modern Encryption. San Francisco: No Starch Press, 2017.

Barrett D. J., Silverman R. E., Byrnes R. G. SSH, The Secure Shell: The Definitive Guide, 2nd ed. Sebastopol, CA: O'Reilly, 2005.

Beazley D. M. Python Distilled. Addison-Wesley, 2021.

Beazley D. M., Ward B. D., Cooke I. R. The Inside Story on Shared Libraries and Dynamic Loading // Computing in Science & Engineering 3, no. 5 (September/October 2001): 90–97.

Calcote J. Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool, 2nd ed. San Francisco: No Starch Press, 2019.

Carter G., Ts J., Eckstein R. Using Samba: A File and Print Server for Linux, Unix, and Mac OS X, 3rd ed. Sebastopol, CA: O'Reilly, 2007.

Christiansen T., Foy B. D., Wall L., Orwant J. Programming Perl: Unmatched Power for Processing and Scripting, 4th ed. Sebastopol, CA: O'Reilly, 2012.

chromatic. Modern Perl, 4th ed. Hillsboro, OR: Onyx Neon Press, 2016.

Davies J. Implementing SSL/TLS Using Cryptography and PKI. Hoboken, NJ: Wiley, 2011.

Friedl J. E. F. Mastering Regular Expressions, 3rd ed. Sebastopol, CA: O'Reilly, 2006.

Gregg B. Systems Performance: Enterprise and the Cloud, 2nd ed. Boston: Addison-Wesley, 2020.

Grune D., Reeuwijk van K., Bal H. E., Jacobs C. J. H., Langendoen K. Modern Compiler Design, 2nd ed. New York: Springer, 2012.

Hopcroft J. E., Motwani R., Ullman J. D. Introduction to Automata Theory, Languages, and Computation, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006.

Kernighan B. W., Pike R. The UNIX Programming Environment. Upper Saddle River, NJ: Prentice Hall, 1984.

Kernighan B. W., Ritchie D. M. The C Programming Language, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1988.

Kochan S. G., Wood P. Unix Shell Programming, 3rd ed. Indianapolis: SAMS Publishing, 2003.

Levine J. R. Linkers and Loaders. San Francisco: Morgan Kaufmann, 1999.

Lucas M. W. SSH Mastery: OpenSSH, PuTTY, Tunnels, and Keys, 2nd ed. Detroit: Tilted Windmill Press, 2018.

Matloff N. The Art of R Programming: A Tour of Statistical Software Design. San Francisco: No Starch Press, 2011.

Mecklenburg R. Managing Projects with GNU Make, 3rd ed. Sebastopol, CA: O'Reilly, 2005.

Peek J., Todino-Gonguet G., Strang J. Learning the UNIX Operating System: A Concise Guide for the New User, 5th ed. Sebastopol, CA: O'Reilly, 2001.

Pike R., Presotto D., Dorward S., Flandrena B., Thompson K., Trickey H., Winterbottom P. Plan 9 from Bell Labs. Accessed February 1, 2020, https://9p.io/sys/doc/.

Poulton N. Docker Deep Dive. Author, 2016.

Quinlan D., Russell R., Yeoh C., eds. Filesystem Hierarchy Standard, Version 3.0 // Linux Foundation, 2015, https://refspecs.linuxfoundation.org/fhs.shtml.

Raymond E. S., ed. The New Hacker's Dictionary. 3rd ed. Cambridge, MA: MIT Press, 1996.

Robbins A. sed & awk Pocket Reference, 2nd ed. Sebastopol, CA: O'Reilly, 2002.

Robbins A., Hannah E., Lamb L. Learning the vi and Vim Editors: Unix Text Processing, 7th ed. Sebastopol, CA: O'Reilly, 2008.

Salus P. H. The Daemon, the Gnu, and the Penguin. Tacoma, WA: Reed Media Services, 2008.

Samar V., Schemers R. J. III. Unified Login with Pluggable Authentication Modules (PAM) // Open Software Foundation (RFC 86.0). October 1995. http://www.opengroup.org/rfc/rfc86.0.html.

Schwartz R. L., Foy B. D., Phoenix T. Learning Perl: Making Easy Things Easy and Hard Things Possible, 7th ed. Sebastopol, CA: O'Reilly, 2016.

Shotts W. The Linux Command Line, 2nd ed. San Francisco: No Starch Press, 2019.

Silberschatz A., Galvin P. B., Gagne G. Operating System Concepts, 10th ed. Hoboken, NJ: Wiley, 2018.

Smith J., Nair R. Virtual Machines: Versatile Platforms for Systems and Processes. Cambridge, MA: Elsevier, 2005.

Stallman R. M. GNU Emacs Manual, 18th ed. Boston: Free Software Foundation, 2018.

Stevens W. R., Fenner Bill, Rudoff A. M. Unix Network Programming, Volume 1: The Sockets Networking API, 3rd ed. Boston: Addison-Wesley Professional, 2003.

Tanenbaum A. S., Bos H. Modern Operating Systems, 4th ed. Upper Saddle River, NJ: Prentice Hall, 2014.

Tanenbaum A. S., Wetherall D. J. Computer Networks, 5th ed. Upper Saddle River, NJ: Prentice Hall, 2010.

Б. Уорд

Внутреннее устройство Linux

3-е издание

Перевел с английского *С. Черников*

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*Н. Ланцунцевич*
Литературный редактор	*Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, М. Молчанова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 05.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 03.03.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 38,700. Тираж 1000. Заказ 0000.