

Одесский центр Linux

Администрирование Linux

Выпуск 1

0. Необходимые пояснения

0.1. Несколько слов о профессиональном сленге

У каждой профессии есть свой язык. И служит он не для того, чтобы скрывать свои мысли от «непосвященных» (как думают многие «непосвященные»). Совсем наоборот: он служит для того, чтобы выражаться кратко и избегать неточностей. Профессиональные области знаний содержат множество понятий, для обозначения которых в обычном русском (английском, китайском...) языке не хватает слов. Поэтому недостающие слова заимствуются из других языков или придумываются (зачастую, самым неожиданным образом).

Профессиональный жаргон не разрабатывают филологи, он не приходит из университетов и академий. Жаргон появляется сам по себе, когда профессионалы разговаривают на общие для них темы, читают одни и те же книги.

Большая часть программистского сленга пришла из английского языка. Этому не стоит удивляться, потому что вся теория и практика computer science and information technology развивалась в англоязычных странах. Этот язык (в урезанном, «техническом» варианте) является языком межнационального общения программистов всего мира. Программист должен знать английский язык, чтобы читать документацию в оригинале и должен писать документацию по-английски, чтобы ее могли прочесть другие.

Поэтому всякий раз, когда мы будем вводить новый термин, мы будем указывать в скобках его английский эквивалент. Знание английского слова пригодится вам при поиске.

Часто бывает, что авторы русскоязычных книг (в особенности, книг для начинающих) боятся, что жаргон отпугнет их читателей. Поэтому, сражаясь за чистоту русского языка, они изобретают свой жаргон, который приводит к еще большей путанице и недоразумениям. В результате, более всех страдают именно новички, которые оказываются не в состоянии задать профессионалам вопрос так, чтобы они их поняли.

Мы не будем бояться жаргона.

Другие пытаются бороться за чистоту английского языка и учат своих читателей правильному произношению, чем опять же только увеличивают путаницу. Программисты произносят английские слова так, как их удобно произносить и с той лишь целью, чтобы их поняли (привыкшее к русскому языку ухо не всегда в состоянии уловить детали английского произношения).

Мы будем пользоваться общепринятым произношением.

Но, в то же самое время, мы призываем вас не злоупотреблять жаргоном. Так поступают те, кто хочет казаться глубоким специалистом, не являясь им на самом деле. Пользуйтесь простым правилом: *говорить нужно так, чтобы вас поняли*.

0.2. Хакеры

Происхождение термина *хакер* (hacker) выяснить довольно трудно. Но появился он примерно в 1960-е гг. и означал человека, которому нравится изучать внутреннее устройство какой-либо сложной программной системы, находить ее сильные и слабые стороны, использовать ее возможности в новом, неожиданном ключе. Он легко учится, в совершенстве изучил программирование и получает удовольствие не от результата, а от самого процесса программирования.

До середины 1980-х гг. этот термин был неизвестен широкой публике. Но, когда журналисты подхватили это слово, оно приобрело в их устах совершенно другой смысл и стало обозначать асоциальную личность, распространяющую компьютерные вирусы, похищающую деньги из банков и взламывающую сверхсекретные компьютерные системы.

Мы будем использовать слово «хакер» в его первоначальном смысле, как это принято повсеместно в мире Unix. Это толкование термина «хакер» закреплено, например, RFC 1983 — нормативным документом сети Internet. Там, где речь будет идти о несанкционированном вторжении в компьютерную систему, мы будем использовать слово «злоумышленник». Еще одним подходящим термином является *cracker*, но он сравнительно мало распространен.

У хакеров нет (и не может быть) какой-либо организационной структуры. Нельзя однозначно определить, является ли человек хакером или нет — это, скорее, вопрос собственного мироощущения и, отчасти, признания в профессиональной среде.

Как и во многих других случаях, изменить закрепившуюся общественную установку уже вряд ли возможно. Правильное толкование термина «хакер», вполне вероятно, так и останется уделом профессионалов.

Что почитать:

1. Стивен Леви, «Хакеры: герои компьютерной революции» — <http://cooler.irk.ru/hackers/>

Эта книга рассказывает о тех, кто стоял у истоков компьютерной революции, занимаясь программированием и электроникой «просто ради удовольствия».

2. Free On-Line Dictionary Of Computing — <http://www.foldoc.org/>

См. термины hacker, cracker.

3. RFC 1983 — Internet Users' Glossary — <http://www.faqs.org/rfcs/rfc1983.html>

Глоссарий терминов, имеющих отношение к сети Internet.

4. Hacker-HOWTO — <http://www.catb.org/esr/faqs/hacker-howto.html>

Это полущуточное-полусерьезное HOWTO Эрика Раймонда формулирует основные принципы хакерской этики и хакерского образа жизни и дает один из рецептов, который, по мнению автора, сможет привести вас на путь хакерства.

5. Eric S. Raymond, *The Art of Unix Programming* — <http://www.faqs.org/docs/artu/>

Эта книга ставит своей целью собрать и передать не столько знания о работе отдельных Unix-приложений, сколько общую культуру, сложившуюся в Unix-мире и лежащую в основе всего, что называют *Unix-way*.

6. The Glider: Proposal for a Hacker Emblem — <http://www.catb.org/hacker-emblem/>

Глайдер — известная фигура из игры «Жизнь» была предложена Эриком Раймондом в качестве эмблемы хакерского сообщества.

0.3. Дополнительная информация

Материалы, которые вы держите в руках (или читаете в электронном виде) — это краткий конспект лекции, те сведения, которые необходимо знать. Мы также рекомендуем ими пользоваться при заочном обучении. В конце каждого раздела есть секция «Что почитать» — это список источников, где можно найти дополнительную информацию, которая может понадобиться в реальной жизни.

Мы отдаем предпочтение первоисточникам, потому что в них всегда находится самая полная и самая свежая информация. Мы также упоминаем руководства для углубленного изучения, многие из которых считаются классическими. Руководства «для начинающих» вам вряд ли понадобятся, потому что они не содержат больше, чем рассказывается здесь. Но поскольку мы пользовались некоторыми из них при составлении этих конспектов, было бы несправедливо вообще о них не упомянуть, поэтому мы собрали их в конце этого раздела.

Предлагаемые нами дополнительные материалы находятся, в основном, в сети Internet или включены в состав дистрибутивов Linux. Есть и полезные печатные руководства, но их не всегда можно найти в свободной продаже.

Мы упоминаем русские переводы англоязычных руководств, если нам о них известно. Но нужно быть готовым к тому, что русский перевод не всегда содержит самые последние изменения, внесенные в оригинальный текст, зато очень часто содержит внесенные переводчиком ошибки.

Помимо документации к каждой конкретной программе, чрезвычайно полезным источником знаний по Linux являются HOWTO и mini-HOWTO. В них авторы делятся опытом по решению какой-либо конкретной задачи, описывают имеющиеся средства, а зачастую и дают подробное теоретическое введение в проблему. Если вы не знаете, какую программу использовать и, вообще, с чего начать, попробуйте найти соответствующий HOWTO (не забывайте обратить внимание на дату его последнего обновления). Большое количество различных HOWTO собрано в рамках Linux Documentation Project (LDP). Кроме этого, в LDP можно найти поистине бесценные руководства по системному и сетевому администрированию, безопасности и др. Подборки материалов из LDP можно встретить на дисках и в составе дистрибутивов Linux. Много HOWTO существует и за пределами LDP.

Но есть один справочник, которым можно и нужно пользоваться всегда, когда ничто другое не помогает. Это Internet. Linux появился и развивался в сети Internet. Благодаря открытости Linux, в Internet можно быстро и легко найти документацию, статьи, пошаговые руководства, а также замечания тех, у кого возникали какие-то проблемы и кому удавалось с ними справиться. Если же ничего не нашлось — приходите на великое множество форумов и списков рассылки, посвященных Linux (в целом и по частям), задавайте свой вопрос и можете рассчитывать на профессиональный ответ. Но будьте готовы и к тому, что вас отошлют к списку часто задаваемых вопросов (FAQ), какому-то разделу документации или статье в Internet.

Что почитать:

1. The Linux Documentation Project — <http://www.tldp.org/>

Этот проект ставит своей целью создание качественной свободной документации для Linux.

2. The Linux Cookbook — <http://www.dsl.org/cookbook/>

Сборник полезных советов на каждый день для пользователей Linux. В эту книгу можно заглядывать от случая к случаю, чтобы пополнять свой арсенал трюков.

0.4. Обозначения

Мы будем пользоваться следующими условными обозначениями, пришедшими из языка C и получившими широкое распространение в литературе по Unix:

`foo()`

Круглые скобки после имени говорят о том, что речь идет о *функции*, т.е., выражаясь простыми словами, фрагменте программы, выполняющем какую-то полезную работу. Функции ядра Linux мы будем называть *системными вызовами* (system call).

`0157`

Ноль, идущий перед числом, показывает, что число записано в *восьмеричной* (octal) системе счисления.

`0x19AF`

0x перед числом показывает, что число записано в *шестнадцатиричной* (hexadecimal) системе счисления.

1. Введение в Unix

1.1. История Unix

Идея создания операционной системы, способной обеспечивать совместную работу нескольких пользователей на одной физической машине (*системы с разделением времени*, time-sharing system), относится к началу 1960-х гг. Первой относительно удачной попыткой создания такой системы была CTSS ([C]ompatible [T]ime[S]haring [S]ystem). За ней последовала MULTICS ([M]ULTiplexed [I]nformation and [C]omputing [S]ervice) — мощная для своего времени система, но крайне перегруженная функциональностью и сложная для понимания из-за отсутствия единой архитектуры. Во времена расцвета MULTICS количество ее инсталляций во всем мире не превышало 100 штук.

Операционная система Unix (названная так в насмешку над MULTICS) увидела свет в 1969 году. Ее автором считается Кен Томпсон (Ken Thompson) из Bell Labs — подразделения телефонной компании AT&T (American Telephone & Telegraph). Главными особенностями Unix являются:

1. *Простота.* Unix обладает единой стройной архитектурой, основанной на четких правилах, которые легко проверить. Благодаря этому Unix является эталоном надежности, защищенности и эффективности работы. Уже в начале своей истории Unix могла с успехом использоваться на машинах, которые считались слишком слабыми и устаревшими. Первый вариант Unix был написан за два дня, что разительно контрастировало с MULTICS и ее многотомными спецификациями.
2. *Расширяемость.* Unix предоставил программисту самую удобную из существовавших на тот момент сред программирования. Его открытая архитектура и простые спецификации способствовали появлению большого количества дополнительных программ, которые расширяли функциональность Unix, органично вписываясь в его программное окружение. Unix строится из «кирпичиков», которые можно комбинировать в зависимости от задачи. Поэтому в Unix довольно сложно определить (разве что административными методами), что является частью операционной системы, а что — прикладными программами. Комплект из операционной системы и дополнительных программ, собранный для какой-то определенной цели (например, для использования на домашнем компьютере, на сервере, на карманном компьютере и т.п.) называется *дистрибутивом* (distribution).
3. *Переносимость.* Большая часть кода Unix была написана не на ассемблере, а на языке C — языке высокого уровня, придуманном Дэнисом Ритчи (Dennis Ritchie) специально для Unix. Принципы работы Unix были достаточно общими и позволяли легко переносить ее на практически любую аппаратную платформу. Этот процесс называется *портированием* (porting).

AT&T, в то время официально признанная монополией, не имела права, согласно решению суда, заниматься компьютерным бизнесом. Благодаря этому Unix распространялась совершенно свободно, ее исходные тексты были доступны для всех, они публиковались в учебниках. Система дополнялась и улучшалась силами тех, кто использовал ее у себя в университете или учреждении.

С 1978-79 года начинается повсеместное использование Unix в коммерческих целях. Тогда же, благодаря разработкам Berkeley's Computer Science Research Group (подразделения Калифорнийского университета), появилась реализация протоколов TCP/IP для Unix и Unix стала основной платформой в сети ARPANet (предшественницы Internet). Unix стала быстро вытеснять и вытеснила все остальные операционные системы. Все существующие на сегодняшний день операционные системы так или иначе произошли от Unix.

Unix не стал операционной системой для микрокомпьютеров, поскольку на тот момент их возможности были слишком ограничены и не позволяли обеспечить одновременную работу нескольких программ с жестким разделением аппаратных ресурсов между ними. Но операционная система DOS, тем не менее, многое позаимствовала от Unix.

В 1983 году, согласно новому решению суда, Bell Labs была расформирована и AT&T получила возможность заниматься компьютерным бизнесом. С этого момента AT&T перестает бесплатно распространять исходные тексты Unix (на тот момент Unix System V). Коммерческие компании, занимающиеся Unix-бизнесом (Sun, SCO), в целях борьбы с конкурентами, начинают создавать несовместимые между собой «клоны» Unix. Unix распадается на множество «Unix-подобных»¹ операционных систем. Борьба между ними затрагивает, в основном, рынок «больших ЭВМ» и появление на горизонте сравнительно мощных персональных компьютеров остается незамеченным.

Разрабатываемый в Berkeley клон Unix под названием BSD Unix продолжает распространяться свободно.

Предпринимаются различные попытки стандартизации Unix. Стандарт POSIX (P[ortable] O[perating] S[ystem] I[n]terface) упорядочивает многие интерфейсы System V и BSD и становится основным стандартом Unix-мира. Но различия между POSIX-системами остаются еще очень значительными.

В 1985 году Ричард Столлмен (Richard Stallman) начинает проект по созданию полностью свободной (в плане изучения, модификации и распространения) операционной системы. Для того, чтобы защитить свободную ОС от коммерциализации, Столлмен придумал специальное лицензионное соглашение, названное General Public License (GPL). Согласно GPL, если кто-то модифицирует свободную программу, то новая программа должна быть свободной, даже если у свободной программы был позаимствован только небольшой фрагмент. Такое правило получило шуточное название «авторское лево» (copyleft), по аналогии с copyright — авторским правом.

В качестве основы он, разумеется, взял Unix. Но свободная ОС не была обязана быть похожей на Unix, она должна была развиваться свободно. Поэтому Столлмен назвал ее GNU — G[N]U is N[ot] U[n]ix. Для юридической и финансовой поддержки проекта GNU был создан Фонд свободного программного обеспечения (Free Software Foundation — FSF)².

В 1991 году BSD Unix портируют на платформу Intel 80386 (эта ОС получила название 386BSD). В том же году, несколькими месяцами ранее, финский студент Линус Торвальдс (Linus Torvalds) создает первую версию ядра для операционной системы GNU. Ядро (названное в его честь Linux) работает на том же 386-м процессоре и распространяется под GPL.

¹Необходимо отметить, что под Unix всегда понимается все семейство Unix-подобных ОС, тогда как UNIX является зарегистрированной торговой маркой The Open Group.

²Если в названии программы присутствует слово GNU (например, GNU libc или GNU wget), то это означает, что программа находится под юридической защитой Free Software Foundation. В этом случае FSF принадлежат все авторские права на программу.

В 1992-94 годы AT&T подает в суд на BSD, обвиняя ее в нарушении авторских прав и краже кода из System V. В результате почти двухлетнего разбирательства оказывается, что System V содержит код, заимствованный из BSD и дело заканчивается мировым соглашением. BSD становится официально лицензионно чистой. Но, в силу других обстоятельств, Computer Science Research Group в Berkeley прекращает свое существование, а проект 386BSD разделяется на три ветви: FreeBSD, OpenBSD и NetBSD. Все это ослабило изначально более сильный проект 386BSD, многие ключевые разработчики перешли в GNU/Linux и он постепенно стал доминирующей Unix-подобной системой на рынке персональных компьютеров.

Развитие Internet позволило привлечь к GNU/Linux значительно большее количество разработчиков. Преимущество открытой модели разработки стало очевидным.

К 1993 году (поднявшись с нуля всего за два года) GNU/Linux становится полноценной операционной системой. С 1998 года его имя начинает появляться в средствах массовой информации (именно журналисты сократили «GNU/Linux» до просто «Linux», но, если возможна путаница, важно понимать, что Linux — это название только лишь ядра). К 2000 году многие коммерческие Unix-системы под давлением Linux перестают существовать и остается лишь несколько самых развитых, работающих на специфических аппаратных платформах. С этого времени Linux начинает постепенно наступать на операционные системы семейства Windows на всех фронтах — Internet-серверов (где Linux уже доминирует), корпоративных серверов и рабочих станций, офисных и домашних ПК, карманных компьютеров и множестве других устройств. А новые устройства, появляющиеся в последние годы, зачастую оснащаются Linux изначально.

Постепенно растет и общественная значимость Linux. Linux становится для широкого круга людей не просто ядром или операционной системой, а технологией, идеологией, общественным институтом и способом самовыражения.

Что почитать:

1. Eric S. Raymond, *The Art of Unix Programming* — <http://www.faqs.org/docs/artu/>

В разделе *History* сделан достаточно подробный экскурс в историю Unix и историю развивавшейся параллельно Unix-культуры.

2. Free On-Line Dictionary Of Computing — <http://www.foldoc.org/>

См. термины CTSS, MULTICS, Unix, POSIX.

3. Стивен Леви, «Хакеры: герои компьютерной революции» — <http://cooler.irk.ru/hackers/>

Эта книга рассказывает о тех, кто стоял у истоков компьютерной революции, занимаясь программированием и электроникой «просто ради удовольствия».

4. UNIX history — <http://www.levenez.com/unix/>

На этой странице собрана информация и ссылки по истории Unix, перечислены известные автору Unix-подобные ОС и нарисована громадная схема их родственных связей, которую можно распечатать и повесить на стену.

1.2. Файл. Имя файла

Мы не будем давать определение файла, потому что это определение способно скорее запутать, чем разъяснить. Ограничимся интуитивным пониманием файла как некоторой порции информации, которая хранится в каком-то определенном месте и имеет имя.

Имя файла в Linux может содержать любые символы, кроме слэша (/) и символа NUL (символ с кодом 0). Длина имени файла ограничена 255 символами¹. Точка (.) в имени файла — такой же символ, как и все остальные. Она может служить для указания расширения файла, но это совсем не обязательно. Например, в имени `wget_1.7-3.diff.gz` можно увидеть два расширения (`.diff` и `.gz`), и, в то же самое время, точка присутствует и в номере версии (1.7).

Заглавные и строчные буквы в имени файла считаются *разными буквами*. `FuBar` и `fubAR` — это *разные* имена файлов.

Файлы могут объединяться в *каталоги* (directory). В каталоге могут находиться не только файлы, но и подкаталоги и т.д. Запись `foo/bar` означает: «файл `bar` в каталоге `foo`», а `x/y/z` — «файл `z` в каталоге `y`, который находится в каталоге `x`». Обратите внимание, что имена каталогов в Unix разделяются символом «/» — *слэш* (slash), в отличие от Windows-подобных систем, где разделителем служит «\» — *обратный слэш* (backslash). Количество слэшей не имеет значения: `foo//bar///baz` означает то же самое, что и просто `foo/bar/baz`.

Каталоги с подкаталогами образуют иерархическую файловую систему или *дерево каталогов* (directory tree). Корнем этого дерева является *корневой каталог* (root directory). Корневой каталог называется просто / (слэш).

С точки зрения Unix, каталог — это файл, такой же как все остальные файлы, в котором хранится список файлов, находящихся в каталоге и информация о их местонахождении. Если мы возьмем, например, такое имя файла:

```
/usr/X11R6/lib/X11/fonts/encodings/large/big5.eten-0.enc.gz
```

то Unix будет искать файл следующим образом: откроет корневой каталог, найдет там элемент `usr`, откроет каталог `/usr`, найдет там элемент `X11R6`, откроет каталог `/usr/X11R6`, ... и так далее, пока не дойдет до последнего элемента.

Если, как в предыдущем примере, имя файла начинается со слэша, то есть корневого каталога, то такое имя называется *абсолютным именем файла* (absolute filename или fully-qualified filename). Это имя еще иногда называют *путем* (path или pathname), потому что оно действительно задает путь, следуя по которому можно найти файл.

Имя вроде `foo/bar/biz` не начинается с корневого каталога, поэтому, для того, чтобы найти файл по такому имени, нужно знать, с какого каталога начинать поиск. Другими словами, нужно знать имя *базового каталога* (base directory). Имя `foo/bar/biz` задает имя *относительно* базового каталога, поэтому оно называется *относительным именем файла* (relative filename).

¹Таковы ограничения, которые накладывает сам Linux. Но, поскольку Linux может работать с различными файловыми системами, у каждой из них могут быть свои, более жесткие ограничения.

Превратить относительное имя в абсолютное очень просто: если базовый каталог называется `/home/user`, то абсолютное имя будет `/home/user/foo/bar/biz` — мы просто записали имена одно за другим и поставили между ними слэш.

Имя каталога можно заканчивать символом `/`. Так обычно и делают, чтобы подчеркнуть, что речь идет о каталоге, например `/etc/init.d/`.

Если имя файла начинается с точки, например, `.muttrc`, то такой файл будет *скрытым*. Но это не совсем соответствует тому, что обычно подразумевают под скрытым файлом в Windows-подобных системах.

В любом каталоге есть два элемента: `.` (точка) и `..` (две точки). Элемент `.` указывает на сам каталог, а элемент `..` — на вышестоящий, *родительский каталог* (parent directory). В корневом каталоге элемент `..` указывает на сам корневой каталог.

Таким образом, `/home/user/../../vasya/` означает то же, что и `/home/vasya/`, а `/home/ftp/./pub/` — это просто `/home/ftp/pub/`.

Имена `.` и `..` особенно удобны при указании относительных имен файлов. Пусть, например, базовым каталогом будет `/etc/rc3.d/`, а относительное имя будет `../init.d/sysklogd`. Тогда абсолютное имя получим приписыванием:

```
/etc/rc3.d/../init.d/sysklogd
```

а это означает то же самое, что и

```
/etc/init.d/sysklogd
```

Последнее имя, получившееся после удаления всех `.` и `..` называется *каноническим именем файла* (canonical filename).

Имя файла само по себе ничего не говорит о том, где файл находится. Файл, например, `/usr/share/doc/README` может находиться на винчестере, дискете, компакт-диске, на другой машине в сети или вообще в оперативной памяти. В отличие от Windows-подобных систем, где нужно указывать имя диска (`A:`, `C:`), чтобы система могла найти файл, в Unix это не требуется. Система сама знает, где файл находится, и пользователю или прикладной программе об этом беспокоиться не нужно.

1.3. Специальные файлы

Файл в Unix — это универсальный способ доступа к какой-либо информации. То, что мы обычно называем файлом — порция информации, хранящаяся на диске — в Unix называется *обычным файлом* (regular file). Каталог с точки зрения Unix — это тоже файл, в котором хранится список файлов, находящихся в каталоге и информация о их местонахождении. Для того, чтобы отличить обычные файлы от каталогов, в Unix существует понятие *тип файла* (file type). Тип файла хранится в каталоге вместе с именем файла.

Кроме обычных файлов и каталогов в Unix есть еще много других типов файлов. Остановимся на двух из них, которые называют *специальными файлами* (special files) или *файлами устройств* (device files).

Файл устройства, как правило, связан с каким-нибудь аппаратным устройством. Читая этот файл, программа может читать информацию с устройства (например, узнать информацию о местоположении мыши). Записывая в этот файл, программа может отправить информацию на устройство (например, отправить текст на принтер).

Различают два типа специальных файлов: *символьный специальный файл* (character special file) и *блочный специальный файл* (block special file).

- **Символьный специальный файл** связан с устройством, которое можно читать лишь последовательно, символ за символом, и писать на него тоже можно только последовательно. Очевидно, например, что информацию о положении мыши можно читать только лишь последовательно — нельзя прочесть этот файл с середины и узнать, где будет находиться мышь в будущем. Аналогично, звук на звуковое устройство тоже можно выводить только последовательно.
- **Блочный специальный файл** связан с устройством, которое можно читать в любом порядке и с любого места. Обычно это какой-нибудь носитель информации: магнитный диск, компакт-диск, flash-носитель и т.п. Если прочесть этот файл от начала до конца, мы получим *образ диска* (disk image), который потом можно сохранить как резервную копию или записать на другое устройство. Таким простым способом в Unix можно копировать, например, дискеты. Но самый привычный способ использования блочного устройства — это создать на нем файловую систему.

Как правило, файлы устройств находятся в каталоге `/dev/`. Например, `/dev/lp0` — это принтер, а `/dev/psaux` — это порт PS/2, куда обычно подключают мышь. Но файл устройства может находиться и в любом другом месте файловой системы и называться как угодно. О том, что это не обычный, а специальный файл, говорит тип файла.

Узнать, с каким именно устройством связан этот специальный файл, можно благодаря паре чисел, которые есть у любого специального файла — *major and minor device number*. Major number указывает на тип устройства (принтер, мышь, IDE устройство, SCSI устройство и т.п.), а minor number — на конкретное устройство этого типа. Соответствие номеров устройствам устанавливает организация LANANA (Linux Assigned Names And Numbers Authority). Все новые типы устройств должны быть там зарегистрированы

для выделения им постоянного номера. Есть специальный диапазон номеров, предназначенный для экспериментов.

Символьные и блочные устройства нумеруются независимо друг от друга. Например, символьное устройство 13,0 — это джойстик, а блочное устройство 13,0 — это MFM-диск.

К сожалению, номеров устройств очень мало. Диапазон `major & minor number` — от 0 до 255 и он уже практически весь исчерпан. В новых версиях ядра Linux, начиная с серии 2.6, этот диапазон расширен. Он составляет 0–4095 для `major number` и 0–1048576 для `minor number`.

Некоторые из специальных файлов соответствуют не реальным, а виртуальным устройствам. Перечислим те из них, которые должен знать каждый:

`/dev/null`

Вся информация, записанная в это устройство, отправляется в никуда. При попытке чтения сразу выдается конец файла.

`/dev/zero`

При чтении из этого файла выдается бесконечный ряд нулевых байтов.

`/dev/random`

При чтении из этого файла выдается ряд случайных байтов. Генератор случайных чисел в Linux основан на сборе «природных шумов» от драйверов различных устройств и годится к применению, например, для генерации криптографических ключей. `/dev/random` выдает ровно столько «шума», сколько он накопил за время работы системы, а если нужно больше — придется подождать.

`/dev/urandom`

При чтении из этого файла выдается бесконечный ряд случайных байтов. В отличие от `/dev/random`, это устройство выдает столько «шума», сколько потребуется. Если по-настоящему случайного «шума» не хватает, устройство генерирует псевдослучайные числа.

Кроме чтения и записи с файлами устройств можно выполнять и другие операции, специфичные для каждого конкретного устройства (например, выдвинуть лоток устройства чтения компакт-дисков). Эти операции называются операциями *управления вводом/выводом* (IOCTL — **I**nterface **O**utput **C**ontrol).

Что почитать:

1. LANANA Linux Device List — <http://www.lanana.org/docs/device-list/>

Список устройств, зарегистрированных в Linux с присвоенными им номерами и краткими пояснениями. Копия этого документа распространяется вместе с исходными текстами ядра в файле `/usr/src/linux/Documentation/devices.txt`.

2. `random(4)`

Здесь описано назначение устройств `/dev/random` и `/dev/urandom`.

3. The Wonderful World of Linux 2.6 — <http://kniggit.net/wwol26.txt>

В этой статье приведен обширный список усовершенствований, появившихся в ядрах Linux серии 2.6, по сравнению с ядрами серии 2.4.

1.4. Inode. Ссылки

Дисковое пространство в файловой системе Unix выделяется блоками. Если размер блока, например, 1К, то файл длиной 1765 байт будет занимать 2 блока, то есть 2К дискового пространства.

Кроме блоков, где хранится *тело файла*, каждому файлу выделяется небольшой блок, в котором хранится всяческая служебная информация (когда файл создан, в каких блоках находится тело файла и т.д.). Этот блок называется *заголовком файла, управляющим блоком* (file control block) или *inode*. У каждого inode есть номер (его иногда называют *индексом файла* (file index)), который однозначно идентифицирует файл в рамках файловой системы (то есть в рамках одного диска или одного раздела диска, если диск поделен на разделы).

Итак, в каталоге, где хранится имя файла, нет никакой информации о его физическом размещении на диске, дате создания и т.п. В каталоге присутствует только номер inode, в котором вся эта информация и находится. Поэтому обычно не говорят, что «каталог содержит файл», а говорят, что «каталог ссылается на файл». А элемент каталога называется *ссылкой* (link) на файл.

Один и тот же номер inode может быть указан в двух разных элементах каталога (или каталогов). Таким образом у одного файла может быть несколько имен. Все эти имена равноправны. Если удалить любую из ссылок, файл будет продолжать существовать, пока на него есть другие ссылки. Для того, чтобы это обеспечить, в inode есть счетчик ссылок. Когда этот счетчик становится равным 0, файл удаляется.

Если программа открыла файл, а ссылку в этот момент удалили, то файл будет удален только тогда, когда он будет закрыт.

Таким образом, в Unix нет операции удаления файла, а есть только операция удаления ссылки — `unlink()`. За удаление самого файла отвечает операционная система.

Элемент каталога `.` (точка) представляет собой ссылку на сам каталог, а элемент `..` (две точки) — это ссылка на родительский каталог. Таким образом, у любого каталога счетчик inode равен как минимум 2.

В файловой системе не может быть двух каталогов с одним и тем же номером inode. Это требование операционной системы введено для того, чтобы в файловая система представляла с собой иерархию и ее можно было от начала до конца просмотреть, пользуясь простым алгоритмом. Если в файловой системе будут петли, это создаст дополнительные проблемы, усложнит логику программ и увеличит расход памяти.

Количество файлов на диске ограничено не только свободным пространством, но и наличием свободных inode. Если все inode исчерпаны, создать новый файл не удастся. В некоторых файловых системах таблица inode имеет фиксированный размер, а в некоторых эта таблица автоматически расширяется, если возникает нехватка inode.

1.5. Процесс

Когда Unix запускает программу, он запускает *процесс* (process). Процессу выделяется оперативная память, процессорное время и другие системные ресурсы.

Каждый процесс в системе имеет номер, который называют *идентификатором процесса* (process identifier — PID). PID'ы выделяются процессам последовательно, в порядке возрастания, начиная с 0.

Одновременно в системе может существовать до 32 тысяч процессов (в Linux 2.6 — до миллиарда), если, конечно, хватит оперативной памяти. Выделить каждому процессу по отдельному процессору, разумеется, нереально. Поэтому каждый процесс получает на свою работу небольшой промежуток времени, который называют *квантом времени* (timeslice). По истечении кванта времени процессор передается следующему процессу и так далее по очереди. Процесс может прервать свою работу и раньше, до истечения своего кванта времени, если ему нужно ждать какого-то внешнего события (например, нажатия клавиши на клавиатуре) или окончания длительной операции (например, чтения/записи на диск). Такая стратегия распределения процессорного времени между процессами называется *вытесняющей многозадачностью* (preemptive multitasking), потому что система принудительно вытесняет процессы, время работы которых истекло.

Процессы в Unix полностью изолированы друг от друга, от операционной системы и от аппаратуры. Благодаря этому процессы не могут повлиять на работу друг друга и на функционирование системы в целом.

Одни процессы могут порождать другие процессы. При этом порождающий процесс называется *родительским* (parent) или *процессом-предком*, а порожденный — *дочерним* (child) или *процессом-потомком*.

Что почитать:

1. The Wonderful World of Linux 2.6 — <http://kniggit.net/wwol26.txt>

В этой статье приведен обширный список усовершенствований, появившихся в ядрах Linux серии 2.6, по сравнению с ядрами серии 2.4.

1.6. Пользователи. Группы пользователей

Unix — многопользовательская система. Он обеспечивает одновременную работу на одной машине множества пользователей (порядка 65 тысяч, а в Linux 2.6 — более 4 миллиардов). Для этого, в первую очередь, каждый пользователь должен быть известен системе.

У каждого пользователя в системе есть свой уникальный номер, который называют *идентификатором пользователя* (user identifier — UID). Кроме этого у пользователя есть уникальное *имя* (user name) или *логин* (login) и *пароль* (password). Всю эту информацию вместе называют *аккаунтом пользователя* (user account).

Логин должен быть коротким и состоять только из строчных латинских букв. Подчеркивания (`_`), точки, цифры и некоторые другие символы разрешены в логинах, но их следует, по возможности, избегать. Заглавные и строчные буквы считаются *различными* буквами, но, традиционно, заглавными буквами в логинах не пользуются. Рекомендуем придерживаться этих соглашений, поскольку с логинами, не отвечающими этим соглашениям, могут возникать проблемы в совершенно неожиданных местах и лучше сразу избегать проблем, чем потом их преодолевать.

Поскольку логины вроде `maddog` вряд ли имеют какую-то связь с реальным именем пользователя, мы будем преимущественно пользоваться термином *логин*, а не *имя*, хотя для операционной системы логин — это имя пользователя.

Пароли следует выбирать длинные (не меньше 8 символов), не состоящие из словарных слов и не имеющие никакой связи логином или реальным именем пользователя. Лучше всего, если это будет смесь из заглавных и строчных латинских букв и цифр, сгенерированная случайным образом какой-нибудь программой-генератором паролей. Но не следует использовать в пароле какие-либо другие символы кроме вышеперечисленных, чтобы не создавать себе проблем со вводом таких паролей. Свой пароль нужно помнить наизусть и не доверять его никому. Если злоумышленнику станет известен логин и пароль хотя бы одного пользователя системы, это подвергнет значительно большей опасности всю систему в целом. Поэтому на важных системах нужно постараться уменьшить количество пользовательских аккаунтов.

У каждого пользователя в системе есть *домашний каталог* (home directory). Он предназначен для файлов, относящихся только к этому пользователю. Как правило, домашний каталог пользователя называется `/home/логин/` (например, `/home/maddog/`).

UID'ы в диапазоне от 0 до 99 зарезервированы системой в качестве системных пользователей. UID'ы от 100 до 999 могут использоваться различными прикладными программами, если им понадобятся пользовательские аккаунты для работы. UID'ы реальных пользователей начинаются, как правило, с 1000.

Пользователь с UID'ом 0 — это особый системный пользователь, которого называют *суперпользователем* (superuser). Такой пользователь традиционно имеет логин `root` («корневой»), поэтому его часто называют просто *рутом*. Root имеет неограниченные права в системе, особый домашний каталог (`/root`), а в экстренных ситуациях он — единственный, кто может войти в систему. Пароль рута должен быть надежно защищен (как

указано выше), а пользоваться рутовым аккаунтом нужно как можно реже, чтобы не подвергать систему лишнему риску. От имени рута следует запускать только проверенные программы, в работе которых вы уверены.

Пользователи могут объединяться в группы. У каждой группы есть числовой *идентификатор группы* (group identifier — GID) и *имя группы* (group name). Имя группы должно отвечать тем же требованиям, что и имя пользователя. Имена пользователей и имена групп могут совпадать.

Каждый пользователь должен входить по крайней мере в одну группу, которую называют *группой по умолчанию* (default group). В последнее время стало принято создавать для каждого пользователя отдельную группу, имя которой совпадает с его логином (то есть для пользователя `alan` создается и группа `alan`). Кроме этого, пользователь может входить и в другие группы, которые называют *дополнительными группами* (supplementary groups). Таких групп может быть до 32.

GID'ы распределяются по тому же принципу, что и UID'ы (см. выше). Группа с GID'ом 0 обычно в Linux называется группой `root`¹ и ее можно наделить дополнительными привилегиями.

В Unix нет и не может быть анонимных пользователей. У любого файла и у любого процесса есть пользователь-владелец и группа-владелец.

Любой пользователь, который каким-либо образом подключается к системе, должен подтвердить, что он — именно тот, за кого себя выдает. Этот процесс называется *аутентификацией* (authentication). Самая простая аутентификация — это ввод логина и пароля. Если в системе зарегистрирован пользователь с таким логином и паролем, доступ разрешается. С другой стороны, система прав Unix не исключает, что пользователя можно аутентифицировать и другим способом: с помощью ключа шифрования, смарт-карты или обратившись по защищенному каналу к авторитетному серверу.

Что почитать:

1. Debian Policy Manual — <http://www.debian.org/doc/debian-policy/>

В разделе «9.2.2. UID and GID classes» идет речь о распределении UID и GID в системе.

2. Users and Groups in the Debian System —
[/usr/share/doc/base-passwd/users-and-groups.txt.gz](http://usr/share/doc/base-passwd/users-and-groups.txt.gz)

В этом документе собраны имена всех системных пользователей и групп и объяснено их предназначение. Этот список является стандартом только для дистрибутива Debian GNU/Linux, в других дистрибутивах могут быть отличия.

3. LSB User & Group Names —
<http://www.linuxbase.org/spec/booksets/LSB-Core/LSB-Core/usernames.html>

Список системных пользователей и групп, которые должны присутствовать во всех Linux-системах, соответствующих стандарту Linux Standard Base (LSB).

4. The Wonderful World of Linux 2.6 — <http://kniggit.net/wwol26.txt>

В этой статье приведен обширный список усовершенствований, появившихся в ядрах Linux серии 2.6, по сравнению с ядрами серии 2.4.

¹ в других Unix-системах она называется `wheel`.

1.7. Права доступа

Любая пользовательская или системная операция выполняется при помощи процесса. Любой системный ресурс, любой источник информации можно представить в виде файла. Таким образом, чтобы разграничить права пользователей и групп, достаточно одного-единственного механизма проверки, позволяющего разрешить или запретить доступ определенных процессов к определенным файлам. В Unix для этого служат UID и GID, которые есть у каждого процесса и у каждого файла, и *биты прав доступа* (access permission bits).

rw	x		rw	x		rw	x	
user			group			other		

Основными являются 9 битов, которые разделены на три группы по 3 бита, как показано на схеме (поэтому очень удобно права указывать в восьмеричном виде — 3 бита как раз обозначаются одной восьмеричной цифрой). Далее проверка идет следующим образом:

- Если UID процесса совпадает с UID владельца файла, то при проверке используются первые 3 бита (user).
- Если GID процесса (основной или любой из дополнительных) совпадает с GID группы-владельца файла, то при проверке используются вторые 3 бита (group).
- В противном случае используются последние 3 бита (other).

И, наконец, проверяются сами биты — чтение файла разрешается, если установлен read bit (r), запись в файл — если установлен write bit (w), запуск файла на выполнение — если установлен execute bit (x).

Обратите внимание, что проверяется только одна тройка битов. Если, например, процесс является владельцем файла, а права установлены так (буквы указывают на установленные биты, а прочерки (-) — на сброшенные):

r	-	-	rw	-	rw	-
4			6		6	

то процесс не получит права на запись в файл, несмотря на то, что все остальные это право имеют.

Для каталогов право на чтение каталога означает право обращаться к файлам, находящимся в этом каталоге и подкаталогах. Таким образом, чтобы записать в файл `/foo/bar/baz`, недостаточно иметь право на запись самого файла `baz` — нужно еще иметь право читать каталоги `/foo/` и `/foo/bar/`.

Право на запись каталога означает право создавать и удалять файлы в этом каталоге. Право на выполнение каталога означает право *искать* файлы в этом каталоге.

Пользователь `root` пренебрегает правами на доступ к файлам. Единственное исключение — рут может запустить файл только тогда, когда execute bit установлен хотя бы в одной тройке битов. Это необходимо в целях безопасности, чтобы рут случайно не запустил незапускаемый файл.

Биты прав однозначно указывают, какой файл является запускаемым, а какой нет. Поэтому в Unix традиционно у исполняемых файлов нет никакого специфического расширения (по аналогии с `.com` или `.exe` в DOS).

Есть особый случай, когда каталог сделан общедоступным — с правами

```
rwX  rwX  rwX
7    7    7
```

В этом случае любой пользователь может создавать файлы в каталоге, но и любой может их удалять, независимо от того, кто является владельцем файла. Для предотвращения этого служит специальный sticky bit (`t`). Если он установлен, то удалить файл сможет только владелец этого файла (при этом ему, конечно, нужны будут и права на запись в каталог).

Sticky bit присутствует не во всех Unix-системах.

Существует в современных реализациях Unix и более сложный механизм распределения полномочий — access control list (ACL). С его помощью можно, например, дать доступ к файлу двум пользователям, не объединенным в группу. Но необходимость в ACL возникает только для очень сложных систем. В большинстве же случаев лучше хорошо продумать схему доступа на уровне пользователей и групп, так вы с большей вероятностью избежите ошибок.

Что почитать:

1. info libc, File System Interface :: File Attributes

Описание различных атрибутов файла, их назначения, битов прав доступа и правил проверки доступа.

2. capabilities(7)

Здесь описаны привилегии, которыми обладает пользователь `root`.

2. Введение в shell

2.1. Терминал. Виртуальные терминалы

В начале компьютерной эры конструкторы задумались о создании более удобных для человека способов вывода информации с компьютера, чем просто перемигивание лампочек на пульте. Для этого к компьютеру подключили телетайп — уже широко использовавшееся к тому времени устройство, состоящее из печатающего устройства и клавиатуры. Все символы, введенные с клавиатуры, телетайп преобразовывал в цифровую форму (телетайпный код) и передавал по проводу, а все символы, полученные по проводу, печатал на бумажной ленте. Телетайпный код также включал специальные символы, обозначающие конец строки, конец страницы, конец или приостановку передачи и т.п.

Телетайп стал одним из первых *терминалов* (terminal) ЭВМ, т.е. «оконечных устройств». Терминал не обязан быть устройством, взаимодействующим с человеком — в качестве терминала может служить модем, производственная линия или телефонная станция. Но чаще всего терминалом называют принтер с клавиатурой или, в современном варианте, дисплей с клавиатурой, которые точнее будет назвать *консолью* (console). Терминал всегда совмещает в себе ввод и вывод информации. Обозначают терминал той же аббревиатурой, что и телетайп — TTY (**T**ele**TY**pe).

Unix-система приспособлена для работы на компьютере, к которому подключено множество терминалов. Даже на персональном компьютере, у которого есть только один дисплей и клавиатура, в Unix-системе они представлены в виде терминала. Это устройство называется `/dev/console`. (Кстати, встроенный динамик (PC speaker) тоже считается частью терминала.)

В современных Unix-системах для удобства пользователя, у которого есть только одна консоль, придуманы так называемые *виртуальные терминалы* (virtual terminals — VT). В системе существует множество виртуальных терминалов, один из которых является *текущим* (current VT) и виден на экране, а на другие можно переключаться нажатием клавиш `Alt-Fn` или `Ctrl-Alt-Fn`, где *n* — номер терминала (от 1 до 63). Это позволяет одновременно работать с несколькими программами, работающими в полноэкранном режиме.

Виртуальные терминалы представлены в системе устройствами `/dev/ttyn`. `/dev/tty0` всегда соответствует текущему терминалу.

Как правило, на клавиатуре нет 63 функциональных клавиш, поэтому, если нужно работать с большим количеством виртуальных консолей, можно запрограммировать для переключения на них другие клавиши. Клавишами `Alt-→` и `Alt-←` можно переключиться на следующий и предыдущий терминал соответственно, а комбинация `Alt-PrintScreen` переключает на предыдущий активный терминал.

Содержимое экрана текущего терминала находится в видеопамяти. Как правило, ее хватает для отображения более чем одного экрана текста. Поэтому, если программа вывела настолько длинный текст, что он не поместился в экран, можно воспользоваться клавишами `Shift-PageUp` и `Shift-PageDown` и увидеть, что находится в видеопамяти за пределами экрана.

При переключении на другой виртуальный терминал содержимое экрана копируется в *буфер терминала* (terminal buffer) — при этом все, что находилось за пределами экрана,

пропадает.

Виртуальный терминал появляется в системе в тот момент, когда какая-нибудь программа что-нибудь на этот терминал выведет. Это называется **распределением терминала** (VT allocation). Пока терминал не распределен, на него нельзя переключиться — его не существует.

2.2. Основы управления терминалом

Как правило, все символы, введенные с терминала, попадают прямо в программу, которая с этого терминала читает, а все символы, выводимые программой на терминал, попадают прямо на экран. Однако, некоторые символы, называемые *управляющими* (control characters) могут обрабатываться драйвером терминала (или самой аппаратурой, если терминал аппаратный) специальным образом, например, как команды перевода строки, очистки экрана, удаления символа и т.д. К управляющим символам относятся символы с кодами 0-31 и символ с кодом 127 (DEL).

У всех управляющих символов есть двух- или трехбуквенное обозначение и каждому из них соответствует комбинация клавиш **Ctrl-буква**. Хотя для некоторых из них есть специальные клавиши (Tab, Enter, Backspace), но комбинации **Ctrl-буква** гарантированно работают всегда и на всех терминалах.

Управляющие коды различаются для различных терминалов, потому что не у всех терминалов одинаковые возможности (если терминал, например, выводит на принтер, он не может стереть символ). Терминал в Linux соответствует стандарту VT102 с Linux-специфичными расширениями. Вот некоторые из символов, которые при вводе с клавиатуры обрабатываются специальным образом:

DEL (0x7F, ^?, Backspace)

DElete. Удаляет последний введенный символ. Обратите внимание, что клавиша **Backspace** должна выводить именно этот код, а не **BS** (0x08, ^H), чтобы Linux-терминал ее воспринял.

HT (0x09, ^I, Tab)

Horisontal Tab. Переходит на следующую позицию табуляции (на экране IBM PC — каждый восьмой символ).

LF (0x0A, ^J, Enter)

Line Feed. Завершает строку и переводит курсор в начало следующей строки.

NAK (0x15, ^U)

Negative AcKnowledge. Удаляет всю введенную строку.

DC2 (0x12, ^R)

Выводит всю введенную строку на экран заново.

Клавиши, для которых нет специальных символов, выдают Esc-последовательности, т.е. целые цепочки символов, начинающиеся с символа **ESC** (0x1B, ^[). Например, клавиша **Delete** выдает **ESC [3~**.

На терминалах VT100 есть клавиша-переключатель **Meta**. Комбинация **Meta-буква** выдает последовательность **ESC буква**. На клавиатуре IBM PC этой клавиши нет и можно просто нажать последовательно **Esc** и букву. Часто, для удобства, в раскладках клавиатуры клавишу **Meta** заменяют клавишей **Alt**.

Эти коды работают всегда, т.е. программе не нужно заботиться о том, чтобы обеспечить пользователя минимальными средствами для редактирования введенной строки. Естественно, что в этом случае программа может получать текст с клавиатуры только лишь строка за строкой. Такой режим работы терминала называется *cooked*.

Программа может переключить терминал в режим *raw* и получать с клавиатуры отдельные символы и даже просто скан-коды. В этом случае вся обработка этих символов целиком ложится на программу. Это удобно для всяческих полноэкранных программ и игр.

Если терминал находится в режиме *echo*, то все введенные символы, не обработанные самим терминалом, выводятся на экран. Если вы вводите, например, пароль, то *echo*-режим, как правило, выключают и символы на экране не отображаются. Но *cooked*-режим при этом включен и все клавиши редактирования работают. Программы могут выключать *echo* и в других случаях, когда им мешает автоматическое отображение символов на экране.

Для управления экраном тоже существует большое количество управляющих кодов и Esc-последовательностей. Но они *никак не связаны* с кодами, вводимыми с клавиатуры. Комбинация **Meta-c** в *echo*-режиме не выведет **ESC** с на экран терминала, а просто отобразится как `^c`. Для управления терминалом нужно выводить спецсимволы из программы.

Что почитать:

1. `console_codes(4)`

Здесь описаны специальные коды Linux-терминала.

2. Keyboard-and-Console-HOWTO —

<http://www.tldp.org/HOWTO/Keyboard-and-Console-HOWTO.html>

Информация об использовании в Linux клавиатуры и консоли, а также об использовании не-ASCII символов.

3. `info libc, Low-Level Terminal Interface :: Terminal Modes :: Special Characters`

В этом разделе и подразделах подробно описаны символы, воспринимаемые терминалом специальным образом.

2.3. Shell

Простейший интерфейс Unix — это *интерфейс командной строки* (command-line interface, CLI). Вы вводите с клавиатуры команду, получаете ответ на экране, затем вводите следующую команду и т.д. Все, что можно сделать в Unix, можно сделать с командной строки, и это обеспечивает управляемость Unix даже в самых сложных ситуациях.

Этот интерфейс обеспечивает семейство программ, называемых *оболочками* (shell). Шеллов довольно много, но, как правило, пользуются только одним.

Самым первым шеллом в Unix был **sh** или Bourne shell. От него произошли **ksh** (Korn shell), **bash** (Bourne Again shell), **zsh** и другие. В BSD появилось другое семейство шеллов, с C-подобным синтаксисом — **csch** (C shell), от которого произошел **tcsh** (Turbo C shell).

В Linux самым популярным шеллом является **bash**, хотя многие предпочитают **zsh** — **bash**-подобный шелл с расширенными возможностями редактирования.

В Unix для каждого пользователя можно задать свой шелл, который будет запускаться при входе в систему с консоли (локально или удаленно). Этот шелл называется *login shell*.

Наличие в любой системе **sh** и **csch** необходимо для совместимости. Но, для того, чтобы сэкономить усилия и место на диске (особенно, если диск маленький — дискета, например), в Linux **sh** является хардлинком на **bash**, а **csch** — на **tsch**. При запуске любой процесс получает «0-й аргумент» — имя, под которым он был запущен. Поэтому **bash**, запущенный как **sh**, отключает свои дополнительные возможности и старается быть похожим на **sh**.

Сам по себе шелл мало что умеет делать, хотя в современных шеллах и есть некоторый набор встроенных команд. Все операции выполняются при помощи дополнительных утилит, а шелл умеет «склеивать» эти утилиты для выполнения сложных или рутинных операций. Таким образом соблюдается один из основных принципов Unix: предоставлять набор «строительных блоков» из которых можно составить более сложную систему и решить любую задачу. Благодаря этому Unix легко приспосабливается к потребностям пользователя.

Большая часть утилит, пригодных «на все случаи жизни», находится в пакете GNU coreutils.

2.4. Интерфейс командной строки

Интерфейс командной строки работает по очень простому принципу: шелл выводит приглашение на экран; вы вводите команду; команда выполняется, выводя на экран, если необходимо, сообщения о ходе и результатах своей работы; затем шелл опять выводит приглашение и можно вводить следующую команду.

Как правило, если команда была выполнена успешно, она не выводит *никаких* сообщений. Отсутствие сообщений — нормальная ситуация.

Если программа выводит сообщение об ошибке, то оно обычно имеет такой вид:

```
mount: can't find /t in /etc/fstab or /etc/mtab
```

В начале сообщения указывается имя программы, которая вывела это сообщение. Это очень удобно, когда несколько программ работает одновременно — можно сразу узнать, к какой программе какое сообщение относится.

Как правило, последний символ приглашения шелла — это \$ или #. Символ # означает, что команда будет выполнена от имени рута и поэтому нужно быть особенно внимательным. Если же команда выполняется от имени обычного пользователя, символ будет \$. Этому же соглашению придерживаются при написании руководств и примеров в книгах по Unix.

Простейшая команда в Unix выглядит так:

```
имя_программы параметр1 параметр2 ...
```

Заметьте, что первое слово («0-й параметр») — это всегда имя программы. Шелл запускает эту программу и передает ей список параметров, который идет в командной строке вслед за именем. Что делать с этими параметрами — решает программа.

В современных шеллах, в том числе и в `bash`, есть некоторое количество встроенных команд, но подавляющее количество команд — это все же внешние программы. Unix shell вполне может обходиться и без встроенных команд.

Регистр букв в имени программы *имеет значение*. Это вполне понятно, поскольку имя программы — это имя файла. Принято, что подавляющее большинство программ в Unix имеют имена, состоящие только из строчных букв.

Выйти из шелла можно командой `exit`. Кроме этого есть еще команда `logout`, которой можно выйти из login shell, но не из шелла, запущенного каким-либо другим способом.

Ключи

Параметры, начинающиеся с символа - (минус), называются *ключами* (keys), *переключателями* (switches) или *опциями* (options). Они служат для управления поведением программы. Опции бывают короткие, однобуквенные:

```
gcc -E
```

Такие опции, как правило, можно объединять, например:

```
ls -al
```

то же самое, что и

```
ls -a -l
```

Бывают и длинные опции:

```
ps --forest
```

Длинные опции начинаются с двух минусов (`--`) — таков стандарт GNU. Но некоторые программы не придерживаются этого стандарта:

```
cdrecord -v -blank=fast -speed=12
```

В этом случае однобуквенные опции в такой программе нельзя объединять, иначе возникнет путаница.

Стандарт GNU требует, чтобы любая программа понимала, как минимум, такие длинные опции:

`--help`

Вывести краткую справку об использовании программы. В Unix не принято, чтобы программа выводила справку об использовании, если ее запустить безо всяких ключей.

`--version`

Вывести имя программы и номер ее версии. Здесь должно выводиться настоящее имя программы, что бывает полезно, когда несколько хардлинков указывают на одну и ту же программу.

`--`

Этот ключ указывает на то, что все ключи в командной строке закончились и все дальнейшие параметры уже не являются ключами, даже если они начинаются с символа `-`.

Что делать, если программе нужно передать имя файла, начинающееся с минуса, например, `-f`? Можно воспользоваться параметром `--`:

```
rm -- -f
```

Или использовать такой трюк:

```
rm ./-f
```

Редактирование командной строки

Bash предоставляет обширные возможности для редактирования командной строки. Здесь мы опишем только основные из них, но крайне полезно прочесть документацию на эту тему.

Кроме стандартных **Backspace** и **Ctrl-U** bash позволяет передвигать курсор по всей строке при помощи клавиш **←** и **→**, **Home** и **End** и удалять символы клавишей **Delete**. Комбинация **Ctrl-K** удаляет все символы от текущей позиции курсора до конца строки.

Клавишами **↑** и **↓** можно перебирать ранее набранные команды.

Если вы набираете в командной строке имя файла, можно набрать только начало имени и нажать клавишу **Tab** — bash сам дополнит имя файла. Если нельзя однозначно определить, о каком файле идет речь (например, набрано **fil**, а в каталоге есть **file1** и **file2**), то bash дополнит имя настолько, насколько возможно (в нашем случае, до **file**) и даст звуковой сигнал. Если нажать **Tab** еще раз, bash выведет возможные варианты дополнения. Если вариантов будет слишком много, он выведет предупреждение.

Bash понимает, что первое слово в командной строке — это имя программы, поэтому при нажатии **Tab** в этом месте он будет перебирать имена программ, а не файлов. Есть еще много различных ситуаций, где bash сам догадывается, что нужно дополнять и, кроме того, автодополнение можно настроить по своему усмотрению.

Что почитать:

1. **bash(1)**

Обширное и исчерпывающее руководство по bash.

2. **info standards, Program Behavior :: Command-Line Interfaces**

Стандарты GNU, касающиеся обработки параметров командной строки.

2.5. Работа с файлами и каталогами

Текущий каталог

У любого процесса в Unix есть *текущий рабочий каталог* (current working directory). Как правило, если вы указываете не абсолютное имя файла, начинающееся с /, а относительное, то оно отсчитывается от текущего каталога. Узнать имя текущего каталога можно с помощью команды `pwd`.

```
$ pwd
/home/user
```

Заметим, что процесс хранит не имя текущего каталога, а номер его inode. Поэтому, если текущий каталог переместить в другое место файловой системы, то его имя изменится автоматически.

Сменить текущий каталог в шелле можно командой `cd`. Она задается так:

```
cd каталог
```

Указанный каталог становится текущим. Помните, что можно использовать относительные пути и имена `.` и `...`. Можно не указывать каталог:

```
cd
```

Тогда мы перейдем в свой домашний каталог.

Команда `cd` является встроенной по очень простой причине: если бы она запускалась отдельным процессом, она не могла бы сменить текущий каталог шелла — нарушилась бы изоляция процессов друг от друга — фундаментальное свойство Unix.

Команда

```
pushd каталог
```

действует аналогично `cd` и при этом запоминает в специальном стеке каталогов имя предыдущего каталога. Вернуться назад можно командой

```
popd
```

`Pushd` и `popd` выводят на экран содержимое стека каталогов. Кроме этого, содержимое стека можно узнать в любой момент командой

```
dirs
```

ls

Всем описанным ниже командам можно передавать в качестве параметров как один файл или каталог, так и список.

Команда `ls` выводит список файлов в каталоге.

`ls каталог`

Если запустить `ls` без параметров, то будет выведено содержимое текущего каталога. Можно узнать и информацию об отдельном файле:

`ls имя_файла`

У `ls` есть огромное количество ключей, задающих, что и в каком формате выводит. Приведем только самые необходимые:

`-a`

Обычно `ls` не выводит скрытые файлы, то есть файлы, имя которых начинается с точки (.). `ls -a` выведет все файлы, в том числе и скрытые.

`-d`

`ls -d имя_каталога` выводит не содержимое каталога, а информацию о самом каталоге.

`-R`

Выводит информацию подкаталогах, файлах в подкаталогах и т.д. рекурсивно.

`-p`

Указывает тип файла, ставя после имен каталогов символ / и т.п.

`-F`

Аналогично `-p`, но еще отмечает исполняемые файлы, ставя после их имени * (звездочку).

`-l`

Выводит подробную информацию о файлах.

Строка, которую выводит `ls -l` имеет такой вид:

```
drwxr-xr-x    2 root    root      4096 Feb 29 00:03 /bin/
```

Здесь указаны, справа налево: имя файла, дата и время последней записи в файл, размер файла, имя владельца и группы-владельца, количество ссылок на inode и права доступа. Первый символ в маске прав указывает на тип файла:

-

Обычный файл.

d

Каталог.

c

Символьный специальный файл.

b

Блочный специальный файл.

Для специальных файлов вместо размера указывается major и minor number:

```
crw-----    1 root    tty        5,    1 Apr 15 10:49 /dev/console
```

Время указывается в сокращенном формате — если в файл последний раз записывали не в этом году, то указывается год записи, в противном случае — время записи.

Создание, удаление, перемещение

Команда `touch` обновляет время последнего чтения из файла. Если файл не существует, она его создает, поэтому ее можно использовать просто для создания пустых файлов:

```
touch имя_файла
```

Каталоги создаются командой `mkdir`:

```
mkdir имя_каталога
```

Команда `mkdir /foo/bar` создаст только каталог `bar`, `/foo/` уже должен существовать. `mkdir -p /foo/bar` создаст каталог и все его родительские каталоги, если они не существуют.

```
rmdir имя_каталога
```

Удаляет пустой каталог. `rmdir -p` удалит и родительские каталоги, если они окажутся пустыми.

```
rm имя_файла
```

Удаляет файлы. Если указан не файл, а каталог, он игнорируется. Командой `rm -r` можно удалить каталог вместе со всеми файлами и подкаталогами.

Ни `rm`, ни `rmdir` не задают никаких вопросов пользователю. Они уверены, что пользователь знает, что делает. Есть лишь одно исключение: `rm` попросит подтверждения, если у вас нет прав на запись в файл, который вы хотите удалить. Это не значит, что вы не можете удалить файл — для удаления достаточно иметь право на запись **в каталог**, в котором этот файл находится. Так что вопрос `rm` — это просто мера предосторожности. `rm -f` позволяет избавиться от любых вопросов. `rm -i`, наоборот, задает вопрос перед каждым удалением.

Копировать файлы можно командой `cp`. По одному:

```
cp файл_источник файл_получатель
```

Или целый список файлов в каталог:

```
cp файл1 файл2 ... каталог
```

Если в файл-получатель уже существует, `cp` его перезапишет безо всяких вопросов. `cp -f` попытается удалить файл-получатель, если его не удастся перезаписать. `cp -i` будет спрашивать, прежде чем перезаписывать.

Если в списке файлов встретятся каталоги, `cp` их копировать не будет. `cp -r` скопирует каталог со всем содержимым, подкаталогами и т.д.

Команда `mv` перемещает файлы. Ее синтаксис и ключи аналогичны `cp`. Если перемещение происходит в рамках одной файловой системы, то перемещаются только ссылки на файл из одного каталога в другой, `inode` остается на месте. Между файловыми системами происходит копирование аналогично `cp`.

`Mv` перемещает каталоги наравне с обычными файлами, поэтому ключ `-r` ей не нужен.

Что почитать:

1. `bash(1)`

Здесь находится информация по встроенным командам `pwd`, `cd`, `pushd`, `popd` и `dirs`.

2. `ls(1)`, info `ls`

Документация к утилите `ls`.

3. `touch(1)`, info `touch`

Документация к утилите `touch`.

4. `mkdir(1)`, info `mkdir`

Документация к утилите `mkdir`.

5. `rmdir(1)`, info `rmdir`

Документация к утилите `rmdir`.

6. `rm(1)`, info `rm`

Документация к утилите `rm`.

7. `cp(1)`, info `cp`

Документация к утилите `cp`.

8. `mv(1)`, info `mv`

Документация к утилите `mv`.

2.6. Просмотр файлов

Утилита `cat` выводит содержимое файла:

```
cat файл1 файл2 ...
```

Если ей передать список из нескольких файлов, она соединит их один за другим (поэтому она и называется `cat` — от слова *catenation*).

Файл может быть очень большим — гораздо больше экрана. Клавиша **Shift-PageUp** может помочь, но не всегда. Лучше воспользоваться соответствующими утилитами.

Утилита `more` выводит файл постранично:

```
more файл1 файл2 ...
```

После каждой страницы она будет ждать от пользователя нажатия клавиши. По клавише пробел будет показана следующая страница, клавиша **q** означает выход. Нажав **h**, можно увидеть справку по остальным командам.

`More` читает файл один раз и не позволяет вернуться обратно. Это позволяет ей быть очень простой и не занимать много места в памяти. Но существует и более удобная утилита `less`.

```
less файл1 файл2 ...
```

`Less` позволяет использовать для прокрутки клавиши управления курсором, **PageUp** и **PageDown**, прокручивать как вниз, так и вверх. Естественно, как и везде в `Unix`, она позволяет обойтись только алфавитноцифровыми клавишами, на случай несовместимости терминалов, когда другие клавиши на терминале не работают. `Less` поддерживает все команды `more` и добавляет новые.

Кроме однобуквенных клавиш, в `less` есть и своя командная строка. Ее можно вызвать, нажав **:** (двоеточие).

Клавиша **!** позволяет выполнять команды шелла, не выходя из `less`. Если нажать **/** и набрать какое-нибудь слово, `less` будет искать его во входном файле, начиная с текущей позиции и далее вперед. Клавиша **?** тоже выполняет поиск, но назад. **/** и **?** без параметров повторяют предыдущий поиск. Это же делает клавиша **n**.

`Less` корректно работает с нетекстовыми файлами.

Описанные выше команды характерны для большинства полноэкранных программ, работающих в консоли `Unix`. Технология их написания хорошо отработана и позволяет им успешно работать на любых терминалах.

Что почитать:

1. `cat(1)`, `info cat`

Документация к утилите `cat`.

2. `more(1)`

Документация к утилите `more`.

3. `less(1)`

Документация к утилите `less`.

2.7. Введение в man

`man имя_программы`

Выводит подробное руководство по работе с программой. Это руководство называется *man-страницей* (manpage). Man является на сегодняшний день наиболее распространенной и универсальной системой ведения документации во всех разновидностях Unix. Поставлять manpage с каждой программой — хороший тон в Unix. В Debian это является частью требований к программному обеспечению.

Вообще говоря, man-страницы существуют не только для программ, но и для некоторых файлов, всех системных функций, устройств, и просто отвлеченных тем, например, лицензий. Для упорядочения man разделен на 9 секций:

- 1
Программы.
- 2
Системные функции.
- 3
Библиотечные функции.
- 4
Специальные файлы.
- 5
Форматы файлов.
- 6
Игры.
- 7
Прочее.
- 8
Административные команды.
- 9
Ядро.

Секцию можно указать непосредственно в командной строке:

`man секция имя_страницы`

Если же секция не указана, man откроет первую найденную страницу с указанным именем¹. В литературе по Unix принято ссылаться на man, указывая в скобках номер

¹Порядок поиска настраивается в `/etc/manpath.config` (директива SECTION).

секции, например — `find(1)` — любой, кто имел дело с Unix, сразу поймет, что речь идет о `man`.

`Man` выводит страницу при помощи программы `pager`, которая является просто ссылкой на `more` или `less` — в зависимости от предпочтений пользователя.

`Man`-страницы традиционно делятся на разделы: `NAME` — краткая информация о программе, `SYNOPSIS` — синтаксис командной строки, `DESCRIPTION` — подробное описание назначения программы, `OPTIONS` — список ключей с описанием, `SEE ALSO` — перекрестные ссылки и т.д.

Можно просто узнать краткую информацию о команде, не читая `man`-страницу целиком. Для этого служит команда `whatis` или `man -f`.

Если вы не знаете точное имя команды, можно воспользоваться поиском — программа `apropos` или `man -k` ищет указанный текст в именах и описаниях `man`-страниц.

Что почитать:

1. `man(1)`

Документация к утилите `man`, описание секций и разделов `man`.

2. `whatis(1)`

Документация к утилите `whatis`.

3. `apropos(1)`

Документация к утилите `apropos`.

2.8. Дескрипторы файлов. Перенаправление

Каждому файлу, открытому процессом, присваивается номер. Этот номер уникален в рамках процесса и называется *дескриптором файла* (file descriptor). Дескрипторы выделяются последовательно, начиная с 0. Более того, в момент открытия файла ему всегда выделяется наименьший из свободных дескрипторов. Т.е., если процесс открывает 4 файла, то им присваиваются дескрипторы 0, 1, 2, 3. Если теперь процесс закроет файл с дескриптором 2, а затем откроет еще один файл, то этот файл получит дескриптор 2.

У любого процесса, запускаемого с командной строки, открыты три дескриптора — 0, 1 и 2. Эти дескрипторы имеют специальное название и назначение.

0, `stdin`

Стандартный ввод. Отсюда процесс читает все, что должно быть введено с клавиатуры.

1, `stdout`

Стандартный вывод. Сюда процесс записывает все, что должно быть выведено на экран.

2, `stderr`

Стандартный вывод для ошибок. Сюда процесс выводит сообщения об ошибках.

Изначально все эти три дескриптора указывают на терминал и процесс действительно читает с клавиатуры и выводит на экран. Но дескрипторы можно перенаправить, чтобы процесс читал из файла и/или выводил в файл.

команда $n >$ *имя_файла*

Указанный файл открывается на запись и указанный дескриптор перенаправляется в этот файл. Здесь n — это номер дескриптора. Если номер дескриптора не указывать

команда $>$ *имя_файла*

то перенаправляется `stdout`.

Если файл уже существует, он будет усечен и все его содержимое будет уничтожено. Важно понимать, что при этом будут сохранены все права на файл, тогда как, если бы мы удалили файл и затем создали его заново, он был бы создан уже с другим владельцем и другими правами.

При таком перенаправлении файл не усекается, и все, что выводит процесс, будет *дописано* в конец файла:

команда $n >>$ *имя_файла*

Опять же, дескриптор можно не указывать.

команда $n <$ *имя_файла*

Указанный файл открывается на чтение и указанный дескриптор перенаправляется в этот файл. Если номер дескриптора не указывать, то перенаправляется stdin.

Заметьте, что разделение stdout и stderr играет очень важную роль. Если stdout перенаправлен в файл (например, процесс выводит полезную информацию и нам хотелось бы ее сохранить), то stderr можно оставить направленным на экран и мы увидим все сообщения об ошибках, которые могут произойти в ходе работы.

Что почитать:

1. bash(1)

В разделе REDIRECTION описаны различные виды перенаправлений.

2.9. Цепочки процессов

Конструкция вида

команда1 | команда2

означает, что обе команды должны быть запущены одновременно, причем *stdout команды1* будет перенаправлен на *stdin команды2*.

Если говорить более точно, то между двумя этими процессами устанавливается связь при помощи *канала* или «*трубы*» (pipe). Pipe — это безымянный файл, существующий только в памяти. Первый процесс кладет информацию в один конец трубы, а второй забирает ее из противоположного конца. Труба имеет конечный размер, и если первый процесс выводит информацию слишком быстро, так что второй процесс не успевает ее забирать, труба заполняется и первый процесс приостанавливается, пока в трубе не освободится место. Аналогично, если второй процесс вынимает информацию очень быстро, труба может опустеть, и тогда второй процесс приостанавливается до поступления новых данных.

Такая комбинация называется *цепочкой процессов* (pipeline), а сам символ | часто называют «pipe».

Цепочки процессов демонстрируют мощь Unix, позволяя из небольших утилит, каждая из которых выполняет одну простую задачу, составлять сколь угодно сложные комбинации. Например, ls не умеет выводить длинный список файлов постранично, но этого можно легко достичь комбинацией ls и less:

```
ls | less
```

Заметим, что less здесь запущена без указания имени файла — в этом случае она будет читать информацию с stdin. Таким образом, две следующие конструкции эквивалентны¹:

```
less имя файла
```

и

```
less < имя файла
```

Так ведут себя more, cat и еще очень многие утилиты в Unix, обрабатывающие текстовые файлы. Это позволяет легко встраивать их в подобные цепочки, и поэтому эти утилиты часто называют *фильтрами*.

Что почитать:

1. bash(1)

Цепочки процессов описаны в подразделе Pipelines раздела SHELL GRAMMAR.

¹С той лишь разницей, что в первом случае файл будет открыт шеллом, а во втором — самой утилитой less. Иногда эта разница существенна.

2.10. Списки и скобки

Команды, перечисленные через точку с запятой (;), просто выполняются одна за другой:

команда1; команда2; команда3; ...

Команда, заключенная в скобки, выполняется в отдельном экземпляре шелла. Например:

```
(ls foo;ls bar)|less
```

Здесь будет запущен отдельный экземпляр шелла, в котором будут последовательно выполнены команды `ls foo` и `ls bar`, а весь вывод из этих двух команд будет перенаправлен в `less`.

Люди, знакомые с математикой могут прочесть эту командную строку по-другому: это ведь просто выражение, где простейшие команды связаны между собой операциями `|` и `;`; а скобки поставлены потому, что операция `|` имеет больший приоритет.

Что почитать:

1. `bash(1)`

См. в разделе SHELL GRAMMAR подразделы Lists и Compound Commands.

2.11. Тильда

Символ ~ в командной строке заменяет имя домашнего каталога текущего пользователя. Таким образом,

```
cd ~/foo/
```

это то же самое, что и

```
cd /home/user/foo/
```

Можно указать и на домашний каталог какого-нибудь другого пользователя при помощи конструкции

```
~имя_пользователя
```

Важно понимать, что *эту замену осуществляет шелл*. Программа получит параметр уже без тильды, ей не нужно будет об этом заботиться. Поэтому, какую бы вы программу не запускали, в командной строке можно будет всегда воспользоваться тильдой и еще многими другими конструкциями, которые предоставляет шелл.

Конструкция с тильдой характерна для bash и отсутствует в оригинальном sh.

Что почитать:

1. bash(1)

См. в разделе EXPANSION подраздел Tilde Expansion.

2.12. Шаблоны

В командной строке можно указывать не только конкретное имя файла, но и *шаблон* или *маску* (pattern или wildcard):

```
ls -l *.dvi
```

Увидев маску, шелл заменяет ее списком файлов, которые под эту маску подходят. Т.е., если в текущем каталоге есть файлы `a.dvi`, `b.dvi` и `c.dvi`, то `ls` в нашем примере получит такой список параметров:

```
ls -l a.dvi b.dvi c.dvi
```

В маске можно использовать следующие символы:

*

Может быть заменена на любую последовательность символов, в том числе и пустую.

?

Может быть заменен на любой одиночный символ.

[...]

Такая конструкция может быть заменена на любой символ из тех, которые перечислены в квадратных скобках. Т.е. `[afz]` может быть заменена на `a`, `f` или `z`, а `[c-h]` может быть заменена на любую букву от `c` до `h` включительно.

Ни один из перечисленных выше символов не может быть заменен на `/`, т.е. под маску `a*b` подойдет файл `abizb`, но не подойдет файл `a/b`. А вот точка (`.`) — это самый обычный символ. Таким образом,

```
cp foo/* .
```

скопирует все файлы из каталога `foo/` в текущий каталог. А команда

```
cp foo/*.* .
```

скопирует только те файлы, в имени которых есть точка.

Есть одно исключение: если имя файла начинается с точки, т.е. файл является скрытым, он не подходит под обычную маску — описанная выше команда не скопирует скрытые файлы. Первую точку в маске нужно обязательно указывать:

```
rm .*
```

Такая команда удалит все скрытые файлы.

Если не найден ни один файл, соответствующий шаблону, то шаблон не будет заменен — он останется в списке параметров без изменений.

Процесс замены шаблона на список файлов называется *globbing*, а специальные символы шаблона называются *globbing characters*.

Что почитать:

1. `glob(7)`

Здесь подробно описаны маски файлов.

2. `bash(1)`

См. в разделе EXPANSION подраздел Pathname Expansion.

2.13. Кавычки и escape-символы

Утилита `echo` выводит на экран свои параметры. Ее можно использовать для вывода произвольного текста:

```
$ echo This is me
This is me
```

`Echo` всегда выводит возврат каретки в конце строки. Если указать ключ `-n`, возврата каретки не будет:

```
$ echo -n one; echo two
onetwo
```

Шелл обрабатывает параметры `echo` точно так же, как и параметры любой другой команды, и на примере `echo` очень легко проследить этот процесс. Например:

```
$ echo ma ~ ma
ma /home/user ma
```

Для того, чтобы шелл проигнорировал специальный символ (например, тильду), достаточно поставить перед этим символом обратный слэш (`\`):

```
$ echo ma \~ ma
ma ~ ma
```

Для того, чтобы вывести `\`, нужно поставить два обратных слэша подряд (`\\`):

```
$ echo \ma
ma
$ echo \\ma
\ma
```

Такой способ превращения специальных символов в обычные называется *escaping*, а символ `\` в нашем случае — это *escape-символ*. Обратный слэш очень часто служит escape-символом в самых разных приложениях.

Можно нейтрализовать специальные символы, заключив строку в одинарные кавычки (`'`). Строка в одинарных кавычках всегда воспринимается буквально:

```
$ echo * hi, all\! *
1.jpg 2.jpg 3.jpg 5.jpg hi, all! 1.jpg 2.jpg 3.jpg 5.jpg
$ echo '* hi, all\! *'
* hi, all\! *
```

Внутри кавычек даже пробелы теряют свое специальное значение, и строка, заключенная в кавычки, всегда считается *одним* параметром.

Что почитать:

1. `bash(1)`

Кавычки и escape-символы описаны в разделе QUOTING.

2.14. Управление заданиями. Группы процессов и сессии. Демоны

Когда программа запускается с командной строки обычным способом, она получает терминал, на котором она запущена, в свое полное распоряжение. Шелл при этом приостанавливается и ждет, пока программа завершится.

Если поставить после команды амперсанд (&),

команда &

то команда будет запущена в *фоновом режиме* (in background). В этом случае шелл не будет ждать ее завершения и выдаст приглашение на ввод следующей команды, а запущенный фоновый процесс продолжит выполняться, сколько ему потребуется. Это удобно для тех процессов, которые занимают длительное время (например, копирование большого файла) и ничего не запрашивают с клавиатуры.

Строго говоря, команда может состоять не из одного, а из нескольких процессов. Например:

```
ls | wc -l &
```

Даже если в командной строке фигурирует только одна программа, она может запускать несколько процессов в ходе своей работы. Поэтому говорят, что с командной строки запущена *группа процессов* (process group) или *задание* (job). Можно запустить одновременно несколько групп процессов, одна из которых может выполняться «на переднем плане» (foreground job), а все остальные — в фоне (background jobs).

Все группы процессов, и фореграундные и бэкграундные, вместе образуют *сеанс* или *сессию* (session). Общим у всех процессов одной и той же сессии является терминал, с которого они все были запущены. Этот терминал называется *управляющим терминалом* (controlling terminal).

Процесс по своей инициативе может перейти из одной группы в другую или создать свою группу, при условии, что все это происходит в рамках одного сеанса. Процесс также может покинуть сеанс и стать *лидером* (leader) нового сеанса — при этом он, конечно, распрощается со своим управляющим терминалом. Такие процессы, у которых нет управляющего терминала, называются *демонами* (daemon).

Специальный файл `/dev/tty` всегда соответствует управляющему терминалу того процесса, который к этому устройству обращается.

У каждого задания есть номер. Он выводится, например, когда вы отправляете задание в бэкграунд:

```
$ cp /multimedia/iso/weapon.iso /tmp/ &  
[1] 5381
```

Здесь в квадратных скобках — номер задания (1), а рядом — PID процесса (5381). Можно получить и список заданий, которые сейчас выполняются, при помощи команды `jobs`:

```
$ jobs
[1]+  Running                  cp /multimedia/iso/weapon.iso /tmp/ &
```

Зная номер задания, легко отправить его в фореграунд:

```
fg %1
```

Или в бэкграунд:

```
bg %1
```

В командах `fg` и `bg`, поскольку они работают только с заданиями, можно не указывать знак `%` и писать просто `fg 1` или `bg 1`. В других командах этот знак нужен, чтобы подчеркнуть, что речь идет о задании, а не о PID процесса.

Обратите внимание на знаки `+` и `-`, которыми помечены два задания из списка `jobs`. `+` — это «текущее» задание, а `-` — «предыдущее». Текущим заданием можно управлять, не указывая его номер:

```
bg
```

Если номер текущего задания нужно передать какой-либо другой команде, то вместо него можно писать `%%` или `%+`.

Номер предыдущего задания заменяет `%-` или, в случае `fg` или `bg` — просто знак `-`:

```
bg -
```

Задание, которым мы манипулировали последний раз, становится текущим, а то, которое до этого было текущим, становится предыдущим.

Как отправить в бэкграунд задание, которое сейчас выполняется в фореграунде? Для этого нужно нажать `Ctrl-Z` и фореграундная группа процессов приостановится:

```
$ cp /multimedia/iso/weapon.iso /tmp/
^Z
[1]+  Stopped                  cp /multimedia/iso/weapon.iso /tmp/
```

После этого мы получим доступ к командной строке и сможем снова запустить приостановленное задание командой `fg` или `bg`.

Бэкграундный процесс получит сигнал о приостановке, если попытается прочесть что-то с терминала. Даже если он проигнорирует этот сигнал, он не сможет ничего прочесть. Он также получит сигнал при попытке вывести что-либо на терминал, но в этом случае, если процесс не приостановится, он сможет писать на терминал.

Лидером сеанса в наших примерах является шелл. Только лидер сеанса может управлять заданиями, отправлять их в фореграунд или бэкграунд. Поэтому команды `fg`, `bg`, `jobs` и другие, которые принимают в качестве аргумента номер задания, являются встроенными в шелл, а не отдельными утилитами.

Что почитать:

1. `bash(1)`

Здесь описаны встроенные в `bash` команды `fg`, `bg` и `jobs`. Раздел `JOB CONTROL` подробно описывает управление заданиями в `bash`.

2. `info libc`, `Job Control`

В этом разделе и подразделах подробно описано внутреннее устройство системы управления заданиями и поведение групп процессов в различных ситуациях.

2.15. Список процессов

Список процессов, запущенных в системе, можно получить при помощи команды **ps**:

```
$ ps
  PID TTY          TIME CMD
 4117 pts/5    00:00:00 bash
 5625 pts/5    00:00:00 ps
```

Здесь указаны PID процесса, имя его управляющего терминала, чистое время выполнения и команда, которой он был запущен. Чистое время выполнения — это время, в течение которого процесс занимал процессор. Сюда не входит время ожидания ввода с клавиатуры, чтения диска и т.п.

У утилиты **ps** есть множество ключей — они указывают, что и в каком виде нужно выводить. Среди них есть однобуквенные ключи с одним знаком минус (POSIX-style), многобуквенные с двумя знаками минус (GNU-style) и однобуквенные ключи без знаков (BSD-style). Это не создает путаницы, поскольку кроме ключей, у **ps** нет других параметров.

Команда **ps** без параметров выводит только процессы, принадлежащие текущему пользователю и запущенные на текущем терминале. Ключ **x** указывает **ps**, что не нужно обращать внимания на терминал, на котором запущен процесс. Ключ **a** заставляет **ps** игнорировать пользователя, являющегося владельцем процесса. Таким образом, команда **ps ax** выведет список всех процессов в системе.

ps u выведет более подробную информацию о процессах, включая статистику использования памяти, состояние процесса (выполняется, приостановлен и т.п.), логин владельца и пр.

Ps выводит командную строку в сокращенном виде, урезанную по ширине экрана. Но **ps -w** выведет ее полностью. Поскольку такую информацию может получить любой пользователь, очень не рекомендуется указывать в командной строке пароли. Впрочем, любой процесс может отредактировать свою командную строку и выводить в ней только ту информацию, которую считает необходимым сообщить пользователю.

ps --forest выводит список процессов в виде дерева. То же самое, но немного в другом виде, выводит утилита **pstree**. Аккуратное дерево не получится, поскольку процессы рождаются и умирают в произвольном порядке.

Что почитать:

1. **ps(1)**

Документация к утилите **ps**, информация о ключах и обозначения, используемые при выводе.

2. **pstree(1)**

Документация к утилите **pstree**.

2.16. Сигналы

Сигнал (signal) — это механизм, позволяющий сообщить процессу о том, что произошло какое-либо внешнее событие. Сигнал — это очень простой механизм, он лишь сообщает: «такое-то событие произошло» и не несет никакой дополнительной информации.

Для посылки сигналов существует утилита `kill`. Существует строго лимитированный набор сигналов, за большей частью из которых закреплено предопределенное значение. Полный список сигналов можно получить командой `kill -l`, а их значение указано в соответствующих руководствах (см. в конце этого раздела). Часть сигналов предназначена для того, чтобы сообщить о каких-то системных событиях, часть посылается в результате срабатывания ловушек защиты процессора (выполнение недопустимой операции или обращение к защищенной памяти), часть предназначена для событий, определяемых самой программой.

У каждого сигнала, кроме номера, есть буквенное обозначение — его обычно пишут с префиксом `SIG`. Таким образом, сигнал 9, сигнал `KILL` и `SIGKILL` — это три равноправных способа обозначения одного и того же сигнала.

У всех сигналов существует предопределенное действие, которое выполняется при поступлении этого сигнала. Это может быть завершение или приостановка процесса, генерация *посмертного дампа* (core dump) процесса или игнорирование сигнала. Но процесс может перехватить сигнал и проигнорировать его или выполнить любое желаемое действие. Существует лишь два сигнала, которые нельзя перехватить: `SIGKILL` (9) приводит к завершению процесса, а `SIGSTOP` (19) — к его приостановке.

Пользовательский процесс может послать сигнал только процессу, принадлежащему тому же самому пользователю. Привилегированный процесс может послать сигнал любому процессу. Можно послать сигнал и самому себе. Сигнал `SIGCONT` — это особый случай: он может быть послан любому процессу в текущей сессии, независимо от его владельца.

Сигнал не может прервать выполнение ядра (хотя могут быть и исключения). Поэтому, если процесс вызвал, например, чтение с диска, и выполнение этой операции затянулось (из-за сбоя носителя, например), то процесс нельзя убить сигналом.

Формат команды `kill`:

```
kill [ -сигнал ] pid
```

Сигнал можно обозначать как числом, так и буквенным идентификатором. Если номер сигнала не указан, посылается `SIGTERM` (15).

Существует версия команды `kill`, встроенная в `bash`. Ей можно передавать не только PID процесса, но и номер задания в таком виде:

```
kill -STOP %1
```

Команда `killall` позволяет послать сигнал процессам, указывая не PID, а имя программы. Такой командой можно убить все работающие `bash`:

```
killall bash
```

Существует достаточно много аналогичных программ и скриптов, позволяющих отбирать процессы по различным критериям и посылать им сигналы: `skill`, `pkill`, `pgrep`, `pidof` и т.д. Они позаимствованы из различных клонов Unix и могут быть полезны в том или ином случае.

Что почитать:

1. `signal(7)`

Здесь подробно описаны все сигналы, присутствующие в Linux на различных архитектурах.

2. `kill(1)`, `info kill`

Документация к утилите `kill`, а также список сигналов с предопределенными действиями.

3. `killall(1)`

Документация к утилите `killall`.

2.17. Некоторые специальные символы

Перечисленные ниже специальные символы связаны с посылкой сигналов или другими специфическими действиями, не относящимися напрямую к редактированию текста. Обратите внимание, что эти символы обрабатываются самим терминалом и поэтому срабатывают независимо от того, какой процесс сейчас владеет этим терминалом.

^D

Процесс, читающий в данный момент с терминала, получит конец файла — аналогично тому, как он достиг бы конца файла, если бы читал из обычного файла.

^C

Эта комбинация клавиш посылает сигнал SIGINT всей группе процессов, выполняющейся сейчас в фореграунде. По умолчанию, процессы должны завершиться при получении такого сигнала.

^Z

Эта комбинация клавиш посылает сигнал SIGTSTP всей группе процессов, выполняющейся сейчас в фореграунде. По умолчанию, процессы должны приостановиться при получении такого сигнала.

^S

Эта комбинация клавиш приостанавливает вывод на терминал. Процесс, который пишет на терминал, тоже, как правило, приостанавливается.

^Q

Эта комбинация клавиш возобновляет вывод на терминал, приостановленный с помощью ^S.

Что почитать:

1. info libc, Low-Level Terminal Interface :: Terminal Modes :: Special Characters

В этом разделе и подразделах подробно описаны символы, воспринимаемые терминалом специальным образом.

2.18. Hangup

Когда пользователь выходит из шелла (это может случиться и в результате разрыва связи с удаленным терминалом), шелл завершается и сессия остается без лидера. В этом случае все процессы в сессии получают сигнал SIGHUP (1). По умолчанию это должно привести к завершению всех процессов, входящих в сессию.

Утилита `nohup` призвана защитить процессы от сигнала HUP. Ее синтаксис:

```
nohup команда параметр1 параметр2 ...
```

`Nohup` запускает указанную команду с указанными параметрами, а ее вывод перенаправляет в файл `nohup.out` в текущем каталоге (или в домашнем каталоге, если в текущем каталоге файл открыть не удалось). Файл открывается для дополнения.

`Nohup` позволяет запустить из командной строки какой-нибудь длительный процесс (например, выкачку) и выйти из шелла, не опасаясь, что процесс прервется.

Процессы, принадлежащие группе, у которой нет лидера, называются *orphaned*.

Что почитать:

1. `nohup(1)`, `info nohup`

Документация к утилите `nohup`.

2. `info libc`, Job Control :: Orphaned Process Groups

В этом разделе описано поведение системы в случае завершения лидера сессии.

2.19. Обзор текстовых редакторов

Самым первым и простейшим текстовым редактором в Unix можно считать `ed`. Он очень удобен для терминалов, где устройством вывода является принтер. `Ed` не выводит на экран файл целиком, а редактирует его построчно. Для редактирования существуют специальные команды, которые вводятся с командной строки редактора.

Более совершенный редактор — это «визуальный редактор» — `vi`. Он выводит на экран файл целиком и предоставляет более удобные средства редактирования — естественно, в пределах того, что предоставляет терминал. `Vi` особенно удобен на медленных и нестандартных терминалах.

`Vi` использует в работе только алфавитноцифровые клавиши (хотя можно настроить его и на использование функциональных, клавиш управления курсором и т.д.) — это позволяет ему работать на любых терминалах. Но, поскольку один и тот же набор символов служит и для ввода символов, и для управления редактором, в `vi` существует понятие *режима* (mode) — нормальный режим, режим вставки, режим выделения блоков и т.д.

В современных Linux есть более совершенные, чем первоначальная, реализации `vi`: `nvi` (New vi), `Elvis` и `ViM` (`[Vi] I[m]proved`). `ViM` — это полноценная среда программирования с подсветкой синтаксиса, взаимодействием с компилятором, сворачиванием блоков и т.д.

`Ed` и `vi` можно вполне использовать и в неинтерактивном режиме, написав файл с командами и перенаправив `stdin` редактора.

`Emacs` (`[E]diting [Mac]ro[s]`) — это мощный программистский редактор, написанный основателем движения свободного ПО Ричардом Столлменом. `Emacs` — первая программа, распространявшаяся по лицензии GPL. Сейчас существует две реализации `Emacs`: `GNU Emacs` и `XEmacs`. `Emacs` написан на Lisp и позволяет дописывать новые режимы и менять конфигурацию при помощи того же Lisp. Этот редактор фактически является «системой в системе» — в нем есть не только работа с отладчиком, но и веб-браузер, календарь, калькулятор, IRC-клиент и многое другое. В нем очень много удобных интерфейсов для редактирования файлов различных форматов. Недостатком его многие считают чрезмерную громоздкость и интегрированность по принципу «все в одном», а также использование всех клавиш-переключателей: `Meta`, `Alt`, `Ctrl` и `Shift`. Споры между любителями `ViM` и `Emacs` не утихают до сих пор.

Простейший текстовый редактор — это `pico` или чуть более мощный `GNU nano`. Это привычный для не-юниксоидов «безрежимный» (modeless) редактор. Аналогичен ему и популярный в свое время `joe`.

Кроме этого для Linux существует еще несколько сотен редакторов. Перечислим только самые популярные: `jEdit`, `Nedit`, `le`, `ne`, `Cooledit` и др. Есть редакторы, работающие как в текстовом, так и в графическом режиме (в том числе и упомянутые `ViM` и `Emacs`), написанные на Java, редакторы специальных форматов и т.д. Заметьте, что здесь мы не касаемся систем подготовки документов, поддерживающих шрифты, форматирование, картинки и потому относящихся совсем к другой категории.

3. Углубляясь в файловую систему

3.1. СИМЛИНКИ

Символическая ссылка (symbolic link) — для краткости, просто *симлинк* (symlink) — это файл, содержащий в себе имя другого файла. Симлинк — это специальный тип файла, который в выводе `ls -l` обозначается буквой `l`:

```
lrwxrwxrwx    1 root    root          18 Nov  1  2002 S10sysklogd -> ../in
it.d/sysklogd
```

При попытке открыть симлинк и прочитать или записать в него все обращения автоматически перенаправляются и все операции производятся с тем файлом, на который симлинк указывает. Только в случае удаления или переименования удаляется и переименовывается симлинк, а не файл. Для работы непосредственно с симлинками предусмотрены отдельные функции.

Биты прав, установленные для симлинка, не имеют никакого значения. Доступом к файлу управляют его собственные права, а доступ к симлинку определяется правами на каталог, в котором он находится.

Поскольку симлинк, в отличие от хардлинка, указывает на имя файла, а не на inode, он может указывать на файл, находящийся и на другой файловой системе. Симлинк может указывать и на несуществующий файл — в этом случае при попытке открыть файл будет возвращена ошибка. Поэтому симлинки иногда называют *мягкими ссылками* (softlink).

В случае с симлинком очень легко отличить линк от оригинального файла — по типу файла. Поэтому разрешается создавать симлинки, указывающие на каталог. Если программа хочет обойти все дерево каталогов, она может просто игнорировать симлинки и таким образом избежать зацикливания.

Симлинк может указывать на другой симлинк. Но существует ограничение на глубину такой вложенности, чтобы избежать циклической ссылки — симлинка, ссылающегося на самого себя.

Хардлинки и симлинки создаются командой `ln`:

```
ln файл имя_симлинка
```

Обратите внимание на порядок аргументов: имя создаваемого линка указывается вторым аргументом. Это аналогично тому, как аргументы указываются у `cp`, `mv` и подобных — создается всегда файл, указанный последним в списке аргументов.

Можно одной командой создать линки сразу на несколько файлов:

```
ln файл1 файл2 ...[ имя_каталога/ ]
```

В указанном каталоге создаются линки на все указанные файлы. Имена линков будут совпадать с именами файлов. Если имя каталога не указано, линки создаются в текущем каталоге.

`ln` по умолчанию создает хардлинки. Для создания симлинков нужно указать ключ `-s`:

```
ln -s ../RU/README README.txt
```

Обратите внимание, что в случае создания симлинка его содержимым будет имя файла, записанное именно в том виде, в котором оно указано в команде `ln`: если в команде указан относительный путь, то и в симлинк будет записан относительный путь. Таким образом, *относительный путь здесь отсчитывается не от текущего каталога, а от симлинка*. Если симлинк, содержащий относительное имя файла, переместить в другой каталог, то он может указывать уже на другой файл. Иногда это именно то, чего хотят добиться, иногда — нет.

Если файл, который нужно создать, уже существует, то `ln` по умолчанию его не перезаписывает, если не указан ключ `-f`.

Что почитать:

1. `ln(1)`, `info ln`

Документация к утилите `ln` и общая информация о хардлинках и симлинках.

3.2. Создание специальных файлов

Специальные файлы создаются командой `mknod`:

```
mknod имя_файла тип major minor
```

Тип файла указывается одной буквой:

b

блочный специальный файл;

c

символьный специальный файл.

Например:

```
mknod /dev/console c 5 1
```

Как правило, в каталоге `/dev/` создают все файлы-устройства, которые только могут понадобиться. Для этого служит находящийся там же скрипт `/dev/MAKEDEV`. Он создает файлы в текущем каталоге, поэтому перед его использованием нужно перейти в каталог `/dev/`. Как правило, у этого скрипта есть дополнительные параметры, которыми можно задавать, какие файлы нужно создать и с какими правами. Эти параметры варьируются в зависимости от дистрибутива.

Что почитать:

1. `mknod(1)`, `info mknod`

Документация к утилите `mknod`.

3.3. SUID и SGID

Дочерний процесс всегда наследует права — идентификатор владельца и группы-владельца — от родительского процесса. Но что делать, если простому пользователю нужно выполнить привилегированную операцию (например, записать на CD) или выполнить какую-то операцию от имени другого пользователя. Для этого в маске прав служат два бита — SUID и SGID — которые устанавливаются на файле с программой.

Если файлу с программой установлен SUID-бит, то при запуске эта программа будет запущена с UID не родительского процесса, а с UID того, пользователя, который является владельцем файла. Аналогично действует и SGID-бит, управляющий переключением GID. Вот пример атрибутов файла:

```
-rwsr-xr-x    1 root    root          15244 Nov 19  2001 ping
```

Программа `ping` запускается с правами рута, какой бы пользователь ее ни запустил.

Для каталогов SGID-бит имеет другое значение: он говорит о том, что все файлы, создаваемые в этом каталоге, должны принадлежать той же группе, что и сам каталог. Подкаталоги, создаваемые в таком каталоге, тоже унаследуют SGID-бит. Такой механизм позволяет упорядочить права в файловой системе, когда в нее пишут несколько пользователей, и он принят по умолчанию в BSD-системах.

Биты SUID, SGID и sticky выводятся командой `ls -l` на том же месте, где и бит `execute` для владельца, группы-владельца и остальных пользователей соответственно. Поэтому, какая буква стоит на этом месте, можно определить значение сразу двух этих битов. Например, для пары SUID/user `execute`:

```
-
    Оба бита сброшены.

x
    Установлен бит execute, SUID сброшен.

S
    Установлен бит SUID, execute сброшен.

s
    Оба бита установлены.
```

SUID и SGID, вообще говоря, меняют только те UID и GID, которые используются для контроля прав — *effective UID* и *effective GID* (EUID и EGID). Но процесс помнит свои *реальные* (real) UID и GID. Дочерние же процессы наследуют в качестве реального UID и GID EUID и EGID их родителя.

Обратите внимание на следующий нюанс. Если мы запустим следующую команду:

```
suprog > /root/a.txt
```

То если `suprog` даже и будет запущена от рута благодаря SUID, файл будет открыт шеллом, а не программой, а у шелла прав рута, вообще говоря, нет.

Другой нюанс связан со скриптами. Установка SUID-бита на файле со скриптом ни к чему не приведет просто потому, что скрипт сам по себе не запускается — запускается интерпретатор (например, `bash`), а скрипт передается ему на `stdin`¹.

Очень не рекомендуется использовать SUID и SGID слишком часто — они создают опасность для системы распределения прав, если SUID'ные программы написаны не очень аккуратно. Поэтому во многих системах о появлении новых SUID'ных программ тут же докладывается сисадмину.

Что почитать:

1. `info libc`, `Users and Groups`

В этом разделе и подразделах подробно описано, как процесс получает и меняет своего владельца.

¹Строго говоря, SUID'ные скрипты в некоторых клонах Unix все-таки возможны, несмотря на все вышесказанное. Но в Linux и *BSD они запрещены, потому что очень легко повлиять на выполнение скрипта, манипулируя окружением. Поэтому секьюрный SUID'ный скрипт невозможен в принципе и лучше держаться от греха подальше.

3.4. su

Программа `su` позволяет любому пользователю выполнить несколько команд от имени другого пользователя, а потом вернуться в свой шелл. Это особенно удобно, когда вам нужно время от времени выполнять команды от рута, но, в целях безопасности, не хочется работать от рута постоянно или держать на отдельной консоли открытый рутовый шелл.

```
su [имя_пользователя]
```

Запущенная с такими параметрами утилита `su` сначала запросит пароль указанного пользователя, а затем запустит новый экземпляр шелла от его имени. Выйдя из этого шелла, мы вернемся опять в старый.

Если имя пользователя не указано, подразумевается `root`. Если `su` запущена от рута, она не будет спрашивать у него пользовательский пароль¹ (очевидно, что это бесполезно).

Можно не запускать новый шелл, а выполнить от нужного пользователя одну лишь команду:

```
su -с команда [имя_пользователя]
```

Нужно иметь в виду, что команда является *единственным* аргументом ключа `-с`. Если у этой команды есть аргументы, перенаправления и пр., ее нужно целиком заключить в кавычки:

```
su -с 'less /var/log/syslog'
```

По умолчанию `su` запускает для нового пользователя тот шелл, который он предпочитает. Ключ `-р` отменяет эту установку — запускается экземпляр того же шелла, из которого была запущена `su`.

Если указать ключ `-`,

```
su - [имя_пользователя]
```

то новый шелл будет запущен так, как будто пользователь залогинился напрямую с консоли (это называется *login shell*).

Что почитать:

1. `su(1)`, `info su`

Документация к утилите `su`.

¹`Su` является SUID'ной программой, ее `EUID=0`, поэтому выяснить, от какого пользователя она была запущена, она может только по своему `real UID`.

3.5. Монтирование

В Unix путь файла не включает никакого указания на то, где файл находится. Если нужно работать с файлами на нескольких различных носителях, нужно файловые системы, которые на этих носителях находятся *примонтировать* (mount) к каталогам в основной файловой системе. Делается это командой

```
mount блочное_устройство каталог
```

Например:

```
mount /dev/fd0 /floppy
```

После выполнения этой команды в каталоге /floppy вы увидите содержимое корневого каталога устройства /dev/fd0 со всеми подкаталогами — дерево каталогов этого устройства станет поддеревом основной файловой системы. Все предыдущее содержимое каталога /floppy будет скрыто — оно останется на диске, но будет недоступно, пока файловая система не будет *отмонтирована* (unmount) от точки /floppy. Каталог /floppy в нашем случае является *точкой монтирования* (mount point) и должен существовать до того, как будет произведено монтирование.

Тип монтируемой файловой системы можно указать с помощью ключа -t:

```
mount -t iso9660 /dev/hdc /cdrom
```

Если этот ключ не указан или указано -t auto, mount попытается сам определить тип файловой системы. (Блок, в котором находится общая информация о файловой системе, называется *суперблоком* (superblock).) Файловые системы перебираются в том порядке, который указан в файлах /etc/filesystems или /proc/filesystems (второй проверяется, если первый не существует).

По соглашению, принятому в Unix, названия всех файловых систем записываются строчными буквами: fat, ext2, hpfs и т.д.

При монтировании могут указываться дополнительные опции, зависящие, вообще говоря, от типа файловой системы. Это делается параметром -o:

```
mount -t vfat -o noexec,codepage=866,iocharset=koi8-r,umask=022 /dev/hda1 /mu  
stdie
```

Как видите, опции перечисляются через запятую и имеют вид *имя[=значение]*.

Примеры опций:

nodev

Запретить доступ к специальным файлам, размещенным на этой файловой системе.

noexec

Запретить запускать файлы с этой файловой системы (сбросить флаг execute со всех обычных файлов).

nosuid

Запретить распознавание битов SUID и SGID.

ro

Монтировать файловую систему только для чтения.

rw

Монтировать файловую систему для чтения/записи.

remount

Перемонтировать уже смонтированную файловую систему с другими опциями. Эта опция особенно полезна, если файловую систему нельзя просто отмонтировать и примонтировать заново из-за того, что с размещенными на ней файлами активно работают другие процессы.

Отмонтировать файловую систему можно командой **umount**, указав только лишь имя точки монтирования или имя устройства (можно также отобразить файловые системы по типу или по опциям):

```
umount /mustdie
```

Нельзя отмонтировать файловую систему, если на ней открыты какие-либо файлы. Но если указать опцию **-l**, она будет убрана из поля зрения, а подлинное отмонтирование наступит тогда, когда все файлы на ней будут закрыты¹. Опция **-f** позволяет отмонтировать «зависшие» сетевые файловые системы.

Команда **mount** без параметров выводит список всех подмонтированных файловых систем со всеми их опциями. Это список находится в **/etc/mtab** (файл обновляется при каждом монтировании/размонтировании) и в **/proc/mounts** (это виртуальный файл, с помощью которого можно получить информацию непосредственно от ядра).

Для монтирования некоторых файловых систем нужны специальные подготовительные действия. Поэтому для их монтирования предусмотрена отдельная программа — **mount.имя_системы**, например, **mount.smbfs**. Mount запускает эти программы, когда нужно смонтировать данную файловую систему.

Fstab

Для того, чтобы не указывать каждый раз при монтировании весь список параметров, можно записать их в файл **/etc/fstab**. Этот файл имеет простой формат: каждая строка состоит из 6 полей, разделенных пробелами или символами табуляции:

```
/dev/hda1      /mustdie      vfat    defaults,noauto,noexec,codepage=866,  
iocharset=koi8-r,umask=022  0              0
```

Первые 4 поля — это имя монтируемого устройства, имя точки монтирования, тип файловой системы и опции. Записанную в **/etc/fstab** файловую систему можно примонтировать, указывая только лишь имя точки монтирования или имя устройства — все остальное берется из соответствующей строки **/etc/fstab**:

¹Только для ядер Linux выше 2.4.11.

```
mount /mustdie
```

`/etc/fstab` служит и для того, чтобы автоматически примонтировать все нужные файловые системы при старте ОС. Команда `mount -a` монтирует всё, что перечислено в `/etc/fstab`, кроме тех файловых систем, в опциях которых указано `noauto`.

Опция **defaults** включает опции монтирования по умолчанию.

Монтирование — привилегированная операция. Обычный пользователь может примонтировать только те файловые системы, которые указаны в `/etc/fstab`, и в опциях которых указано **user**. Отмонтировать при этом сможет только тот, кто примонтировал. Аналогичная опция **users** позволяет отмонтировать любому. Опция **owner** позволяет монтировать только владельцу файла-устройства.

Вместо имени файла-устройства можно указать метку диска или серийный номер файловой системы (с помощью **LABEL=** или **UUID=**). Это позволяет монтировать указанный диск постоянно в одну и ту же точку, даже если он подключается к разным аппаратным интерфейсам (например, к разным портам USB).

Что почитать:

1. `mount(8)`

Документация к утилите `mount`. Здесь же перечислены опции, которые можно указывать при монтировании различных файловых систем.

2. `umount(8)`

Документация к утилите `umount`.

3. `fstab(5)`

Формат файла `/etc/fstab` .

3.6. Файловые системы Linux

Linux умеет работать практически со всеми существующими файловыми системами. Некоторые из них недокументированы или частично документированы и поэтому их реализация в Linux неполна и не рекомендуется модифицировать их из-под Linux во избежание краха.

Для повышения устойчивости некоторые файловые системы предусматривают наличие журнала, куда записываются все производимые операции со структурой файловой системы. Это позволяет быстро и корректно ее восстановить в случае, например, отключения питания. Такие файловые системы называются *журналируемыми* (journalled).

Следующие файловые системы считаются в Linux «родными» — они поддерживают все необходимые Linux возможности и на них можно устанавливать ОС без проблем:

ext2

Первая и наиболее распространенная «родная» файловая система Linux. Поддерживает все, даже наиболее новые возможности, наиболее простая и не журналируемая.

ext3

Тоже самое, что и ext2, но со включенным журналом. Ее можно монтировать как ext2, если она была корректно отмонтирована, а ext2 легко превращается в ext3 с помощью `tune2fs -j`.

reiserfs

Создана Хансом Рейзером как первая журналируемая файловая система для Linux. Поддерживает «упаковку файловых хвостов», позволяющую экономить место на диске, пропадающее из-за того, что длина файла чаще всего не кратна размеру блока. Активно разрабатывается, поэтому может содержать ошибки.

xfs

Разработана фирмой SGI для ОС Irix. Включена в поставку ядра Linux в последних ядрах из серии 2.4. Журналируемая и оптимизированная.

jfs

Разработана фирмой IBM для ОС AIX. Включена в поставку ядра Linux в последних ядрах из серии 2.4. Журналируемая и оптимизированная. Содержит собственную систему управления томами.

Если сравнивать их по производительности, то reiserfs (без упаковки хвостов), xfs и jfs будут идти примерно вровень, обгоняя друг друга в зависимости от использованных тестов. Медленнее будет ext2 и ext3 в режиме по умолчанию (опция `data=ordered`) и reiserfs с упаковкой хвостов. Ext3 со включенным журналированием как структуры, так и данных (опция `data=journal`) будет значительно медленнее.

Ниже представлены некоторые из «сторонних» файловых систем, поддерживаемых Linux. Они не предназначены для работы в качестве основной файловой системы Unix ОС.

iso9660

Файловая система, в которой хранятся файлы на CD-ROM. Не рассчитана на модификацию файлов.

msdos

Более известная как FAT (FAT12, FAT16, FAT32) файловая система для MS-DOS. Не поддерживает длинные имена, права доступа, симлинки и т.п.

vfat

Расширение FAT для поддержки длинных имен файлов. Используется в MS Windows 9x.

ntfs

Файловая система MS Windows NT. Недокументирована, поэтому реализация записи на нее из-под Linux находится пока в стадии тестирования. Не поддерживает Unix-подобную систему прав.

Что почитать:

1. fs(5)

Краткое описание наиболее распространенных файловых систем.

3.7. Часто возникающие проблемы с vfat

Из-за ограниченности файловой системы vfat при работе с ней в Linux возникают типичные проблемы, которые мы собрали в этом разделе.

Права доступа

Права доступа для файлов на vfat устанавливаются согласно опции `umask=` при монтировании или в соответствии с `umask` процесса, который произвел монтирование. Ядра ветки 2.6 поддерживают установку прав отдельно для файлов и каталогов (`fmask=` и `dmask=`). Обратите внимание, что маска указывается в восьмеричном виде и является *инверсной*, то есть в ней установлены те биты прав, которые нужно сбросить. Очень рекомендуется также указать опции `nodev`, `nosuid` и `noexec`.

Владелец

Владелец и группа-владелец для файлов на vfat устанавливаются согласно опциям `uid=` и `gid=` при монтировании или в соответствии с UID и GID процесса, который произвел монтирование.

Кодировка

Опция `codepage=` указывает кодировку для коротких имен файлов, которые хранятся на диске в 8-битной кодировке (`codepage=866` для русского языка). Опция `iocharset=` указывает в какой кодировке должны читаться длинные имена файлов, которые хранятся на диске в Unicode, т.е. в `iocharset=` должна быть указана та кодировка, которой мы пользуемся в Linux (например, `iocharset=koi8-r`).

Что почитать:

1. `mount(8)`

См. опции в разделах `fat`, `msdos` и `vfat`.

3.8. Проверка файловых систем

В случае внезапного краха операционной системы (например, при отключении питания) в файловой системе могут оказаться частично измененные данные и структура, из-за чего она может не работать совсем или работать с ошибками, которые могут проявиться только через некоторое время. Поэтому в таких случаях нужно файловую систему проверить. Делается это командой **fsck**:

```
fsck имя_устройства
```

Обычно при каждом старте ОС автоматически выполняется **fsck -a**. В этом случае проверяются все файловые системы, перечисленные в `/etc/fstab`. Число в 6-м столбце этого файла указывает очередность, в которой файловые системы должны проверяться. Если там стоит 0, то проверки не будет.

Каждая файловая система проверяется по своим правилам, поэтому **fsck** — это семейство программ — **fsck.ext2**, **fsck.minix**, **fsck.xfs** и т.д. **Fsck** запускает нужную, распознав тип файловой системы.

Специальные параметры файловой системы **ext2** позволяют форсировать проверку «на всякий случай» по прошествии определенного промежутка времени или после определенного количества монтирований. Можно также указать, что делать в случае обнаружения ошибок. Они настраиваются через **tune2fs** и параметром **errors** в опциях монтирования.

Команда **sync** приказывает записать все данные из дискового кэша на диски, позволяя минимизировать последствия краха файловой системы.

Запуск **fsck** на примонтированной файловой системе может привести к повреждению ее структуры и потере данных (содержимое кэша не согласуется с теми изменениями, которые **fsck** производит на диске). Несмотря на то, что **fsck** позволяет форсировать проверку на примонтированной файловой системе, это следует делать только тогда, когда файловая система ошибочно считается примонтированной, а на самом деле отмонтирована. В крайнем случае, можно запускать **fsck** на файловой системе, примонтированной **read-only**.

Что почитать:

1. **fsck(8)**

Документация к утилите **fsck**.

2. **e2fsck(8)**

Документация к утилите **fsck**, специфичной для файловой системы **ext2**.

3.9. Badblocks

Fsck проверяет только целостность логической структуры файловой системы — согласованность каталогов, inode и таблиц свободных блоков. Для проверки на физические повреждения предназначена программа **badblocks**

```
badblocks устройство [конечный_блок] [начальный_блок]
```

Badblocks по умолчанию тестирует диски путем чтения каждого блока. Если указан ключ **-n**, производится чтение-запись без уничтожения данных на диске. Ключ **-w** включает самый разрушительный тест.

Badblocks выводит список номеров сбойных блоков. Этот список можно записать в файл (перенаправив вывод или указав ключ **-o**) и передать этот файл e2fsck (ключ **-l имя_файла**). Все указанные блоки будут привязаны к специальному inode, предназначенному для сбойных блоков. При этом крайне важно указать badblocks правильный размер блока, иначе будут выведены неправильные номера. Для упрощения этой операции у e2fsck (fsck.ext2) есть специальный ключ **-с**, позволяющий запустить badblocks непосредственно из e2fsck. Если этот ключ указан дважды (**-сс**), будет запущен неразрушительный тест на чтение-запись.

Что почитать:

1. badblocks(8)

Документация к утилите badblocks.

2. e2fsck(8)

Документация к утилите fsck, специфичной для файловой системы ext2.

3.10. Loopback и binding

Binding

Начиная с ядер Linux версии 2.4 можно примонтировать любую ветку файловой системы в любое другое место. Получается два «отражения» одних и тех же файлов в двух разных местах. Делается это так:

```
mount --bind olddir newdir
```

Каталог `newdir` становится полным «отражением» `olddir`. Если к какому нибудь подкаталогу `olddir` были примонтированы другие файловые системы, они не переносятся автоматически в `newdir`. Можно это сделать отдельными командами `mount --bind` или сразу:

```
mount --rbind olddir newdir
```

Loopback

Можно примонтировать образ файловой системы, находящийся в файле (например образ дискеты или CD). Делается это опцией монтирования `loop`:

```
mount -o loop mydisk.iso /cdrom
```

В этом случае файл `mydisk.iso` связывается со специальным виртуальным loopback-устройством `/dev/loop0`, `/dev/loop1` и т.д., а это устройство монтируется в нужную точку. Размер файла раз и навсегда ограничивает размер файловой системы — она не может увеличиваться. Можно привязать loopback-устройство к файлу и отдельной командой:

```
losetup /dev/loop0 mydisk.iso
```

И отвязать:

```
losetup -d /dev/loop0
```

Образы файловых систем, хранящиеся в файлах, полезны и в том случае, когда файловую систему нужно зашифровать. Для этого у `mount` и `losetup` есть соответствующие опции (а поддержка шифрования должна присутствовать в ядре). Образ файловой системы также не обязан начинаться с начала файла.

Что почитать:

1. `mount(8)`
См. опцию `loop` и параметры `--bind` и `--rbind`.
2. `losetup(8)`
Документация к утилите `losetup`.

3.11. Копирование дисков

Утилита `dd` позволяет копировать диски поблочно и создавать образы дисков. Опции у нее нестандартные, выглядят они как *имя=значение*:

`if=`

Имя входного файла (по умолчанию используется `stdin`). Это может быть устройство.

`of=`

Имя выходного файла (по умолчанию используется `stdout`). Это может быть устройство.

`bs=, ibs=, obs=`

Размер блока, используемый при копировании. Можно указать отдельно размер входного и выходного блоков.

`count=`

Количество блоков, которые нужно скопировать.

`skip=`

Количество блоков, которые нужно пропустить во входном файле.

`seek=`

Количество блоков, которые нужно пропустить в выходном файле.

Можно пользоваться суффиксами `c`, `w`, `b`, `kB`, `K`, `MB`, `M`, `GB`, `G` и т.д., чтобы указать размер в словах, тысячах байт, килобайтах, миллионах байт, мегабайтах и т.д.

Так, например, можно создать файл в несколько мегабайт, заполненный нулями:

```
dd if=/dev/zero of=dummy.bin bs=1M count=7
```

Утилита `dd` умеет делать простейшие преобразования — переставлять пары байт местами, дополнять блоки нулевыми символами или пробелами и пр.

Что почитать:

1. `dd(1)`, `info dd`

Документация к утилите `dd`.

3.12. Создание файловых систем

Файловые системы создаются командой `mkfs`:

```
mkfs [-t тип] устройство
```

Если тип файловой системы не указан, подразумевается `ext2`.

Mkfs позволяет создать файловую систему только на части диска, а также проверить диск на наличие сбойных блоков, прежде чем создавать файловую систему.

Каждая файловая система создается по своим правилам, поэтому `mkfs` — это семейство программ — `mkfs.ext2`, `mkfs.minix`, `mkfs.xfs` и т.д. Mkfs запускает нужную, зная тип файловой системы.

Что почитать:

1. `mkfs(8)`

Документация к утилите `mkfs`.

2. `mke2fs(8)`

Документация к утилите `mkfs`, специфичной для файловой системы `ext2`.

3.13. Стандартная файловая иерархия

Размещение большей части файлов в файловой системе Linux стандартизовано. Файловая иерархия подобрана так, чтобы различные ее части удобно было размещать на разных дисках с разными файловыми системами и монтировать с различными опциями. В этом разделе дан обзор основных каталогов, назначение которых закреплено в Filesystem Hierarchy Standard (FHS).

В каталоге `/boot` находятся файлы, необходимые для загрузки ядра (до монтирования корневой файловой системы. Обычно `/boot` размещают на отдельном маленьком разделе, который после загрузки находится в отмонтированном состоянии (кроме случаев, когда нужно сменить ядро). Если BIOS не умеет читать большие или нестандартные диски, этот раздел стараются разместить поближе к началу диска или на отдельном диске.

В `/bin`, `/sbin`, `/usr/bin` и `/usr/sbin` находятся исполняемые файлы (как бинарные, так и скрипты). В `/bin` и `/usr/bin` — те, которые могут понадобиться любому пользователю, а в `/sbin` и `/usr/sbin` — те, которые могут понадобиться только системному администратору.

Поскольку программ может быть установлено очень много, каталог `/usr` обычно разрастается и под него можно выделить отдельный раздел. Но в случае, если с этим разделом что-то случится, должна существовать возможность запустить систему с минимальными функциями. Поэтому минимальный необходимый набор программ помещают в `/bin` и `/sbin`, а все остальное — в `/usr/bin` и `/usr/sbin`.

Библиотеки, необходимые для работы программ, помещают в `/usr/lib`, а минимальный набор библиотек, необходимых в экстренном случае, — в `/lib`.

Каталог `/usr` содержит только программы и относящиеся к этим программам файлы, которые не меняются в процессе работы. `/usr` может быть подмонтирован `read-only` и помещен, например, на CD-ROM. Все изменяющиеся файлы должны находиться в каталоге `/var`. `/var`, как правило, размещают на быстрой файловой системе.

Каталог `/usr/share` можно разместить на сервере и сделать общим для нескольких машин — в нем находятся файлы, не зависящие от архитектуры и конфигурации конкретной машины (иконки, шрифты и т.п.). Если нужно, программа при установке может создать себе в `/usr/share` отдельный подкаталог.

Документация находится в `/usr/share/doc/имя_программы`. На старых системах она была в `/usr/doc`.

`/usr/local` предназначен для программ, специфичных для конкретной машины. Внутренняя структура этого каталога полностью повторяет структуру `/usr`. В этот каталог предпочитают ставиться программы, собираемые вручную из исходных текстов.

Файлы конфигурации собраны в каталоге `/etc`. Если нужно, программа при установке может создать себе в `/etc` отдельный подкаталог.

Для временных файлов предназначен каталог `/tmp`. Этот каталог очищается при загрузке системы и, кроме того, закрытые и давно не используемые файлы периодически удаляются — программы не должны об этом беспокоиться. Этот каталог открыт на запись для всех пользователей и снабжен `sticky-битом`, чтобы пользователи не могли удалять файлы друг друга. Создавая файлы в `/tmp`, программа (особенно, если она запущена из

под рута), должна проверять, не существует ли уже такой файл, иначе она вполне может записать не туда, куда рассчитывала (в случае симлинков, например).

Если оперативной памяти достаточно, для ускорения доступа к `/tmp` его могут размещать в RAM. Для этого служит, в частности, виртуальная файловая система `tmpfs`.

Кэш, который желательно сохранить между перезагрузками системы, но, при необходимости, можно удалить без последствий, помещают в каталог `/var/cache`.

Другая меняющаяся информация, которую нежелательно удалять, помещается в каталог `/var/lib`.

Информация о работающих программах, как правило, находится в `/var/run`.

В `/var/spool` находятся очереди — например, очередь заданий на печать или очередь почтовых сообщений на передачу.

Почта для пользователей системы хранится в `/var/mail` или, в более старых системах, в `/var/spool/mail`.

Каталог `/var/local` используется программами из `/usr/local`.

В редко используемый в настоящее время каталог `/opt` устанавливают программы, которым нужна своя, специфическая структура каталогов. Все, относящееся к этой программе, собирается в подкаталоге `/opt/имя_программы`.

Домашние каталоги пользователей собраны в каталоге `/home`. Этот каталог тоже может быть помещен на отдельный диск. Но на всякий случай, если файловую систему `/home` подмонтировать не удастся, домашний каталог рута выделен отдельно, в каталог `/root`.

В домашнем каталоге пользователя собрано все, что касается этого пользователя в системе. Если пользователю дозволено иметь свою личную конфигурацию какой-либо из используемых им программ, то файл конфигурации размещается в скрытом файле в домашнем каталоге пользователя. Если глобальный файл конфигурации называется `/etc/programrc`, то локальный, скорее всего, — `~/programrc`. В каком порядке эти файлы будут прочитаны и каким опциям будет отдан приоритет — глобальным или локальным — зависит от программы.

В каталоге `/dev` собраны все файлы-устройства. Если одно и то же устройство используется для записи несколькими программами, то эти программы могут создавать *файлы блокировки* (lock files), чтобы предотвратить одновременную запись из нескольких программ. Эти файлы называются `/var/lock/LCK.имя_устройства`. Внутри такого файла находится PID процесса, захватившего это устройство. Если процесса с таким PID'ом не существует, блокировку можно спокойно удалить и работать с устройством дальше. Каталог `/var/lock` очищается при каждой загрузке системы, чтобы предотвратить случайное появление процессов с тем же PID'ом.

К каталогу `/proc` примонтирована виртуальная файловая система `procfs`, с помощью которой ядро передает прикладным программам различную системную информацию, например, информацию о типе процессора (`/proc/cpuinfo`) или использовании памяти (`/proc/meminfo`). В `/proc` есть отдельный подкаталог для каждого процесса, где собрана относящаяся к этому процессу информация.

Более того, `procfs` позволяет изменять некоторые настройки ядра во время работы системы. Для этого нужно записать значение параметра в соответствующий файл в каталоге `/proc/sys`. Именно этим занимается утилита `sysctl`. Назначение этих файлов можно найти в документации к ядру.

Каталог `/mnt` предназначен для временного монтирования файловых систем (например, дискет или flash-дисков). В настоящий момент есть два подхода к использованию этого каталога. В одном случае для известных устройств (дисковод, CD-ROM и т.п.)

создаются подкаталоги в `/mnt`: `/mnt/cdrom`, `/mnt/floppy` и т.д, и эти же имена прописываются в `/etc/fstab`. Для неизвестных устройств создается каталог `/mnt/tmp`.

Другой подход — не создавать подкаталогов в `/mnt` и монтировать туда только неизвестные устройства. А для известных создавать подкаталоги прямо в корневом каталоге: `/cdrom`, `/floppy` и т.д. Такой подход, принятый в Debian, более соответствует FHS, но приводит к замусориванию корневого каталога.

Новая редакция стандарта FHS рекомендует создавать для известных устройств подкаталоги в каталоге `/media`, а неизвестные монтировать в `/mnt`.

Что почитать:

1. Filesystem Hierarchy Standard (FHS) — <http://www.pathname.com/fhs/>

Стандарт на файловую иерархию для Linux-систем. Подробно описывает, какие каталоги для чего используются и по какой причине. Копия этого документа в Debian находится в каталоге `/usr/share/doc/debian-policy/fhs/` (FHS является частью Debian Policy).

2. `hier(7)`

Здесь перечислены названия и назначение основных каталогов файловой системы. Этот документ является выжимкой из стандарта FHS.

3. `sysctl(8)`

Документация к утилите `sysctl`.

4. `proc(5)`

Структура файловой системы `procfs` и назначение некоторых файлов и каталогов. Это руководство является вторичным по отношению к документации, идущей в комплекте с ядром.

5. `/usr/src/linux/Documentation/filesystems/procfs.txt` — The `/proc` filesystem

Структура файловой системы `procfs` и назначение файлов и каталогов в ней.

3.14. Devfs, sysfs и udev

Для каждого устройства в каталоге `/dev` должен быть создан специальный файл. Имя этого файла не имеет значения (хотя для большинства устройств уже сложились стандарты именования), имеет значение только `major` и `minor numbers`, которые и указывают, какому устройству соответствует этот файл. Как правило, в каталоге `/dev` заранее создают файлы для всех возможных устройств.

Эта схема породила несколько проблем, общих для всех Unix-систем:

- *Нехватка номеров устройств.* Количество разнообразных устройств, подключаемых к компьютеру, растет большими темпами и диапазон 0-255 для `major` и `minor numbers` был исчерпан уже к выходу ядра Linux версии 2.4. Хотя нет на свете компьютера, к которому все эти устройства были бы подключены одновременно. В некоторых случаях требуется подключить несколько тысяч дисков и в системе просто не предусмотрены номера для такого количества однотипных устройств.
- *Какое из устройств нужно?* Один и тот же USB-принтер, подключаемый к разным портам, получает различные имена. Как найти нужное устройство в этом случае.
- *Какие устройства на самом деле существуют в системе?* Для ответа на этот вопрос нужна система извещения пользовательских программ о подключении и отключении устройств — как в процессе работы системы, так и между загрузками.
- *Какой номер присвоить новому устройству?* Эта проблема решается наличием авторитетного органа (LANANA), занимающегося выдачей номеров, но такой подход неудобен.
- *Каталог `/dev` слишком велик.*

Во многих Unix-системах, в том числе и в Linux, начиная с версии 2.4, эта проблема решается наличием виртуальной файловой системы `devfs`, которая примонтируется к каталогу `/dev`. В `devfs` появляются только устройства, которые есть в системе, и для связи файла с устройством больше не требуются `major` и `minor numbers` — связь происходит по имени.

Но и этот подход не решает всех проблем и, кроме того, порождает новые — теперь имя файла-устройства намертво зашито в ядре. Имена, используемые в `devfs`, не совпадают с общепринятыми (например, `/dev/hda1` называется `/dev/ide/host0/bus0/target0/lun0/part1`). В целях совместимости специальный демон, `devfsd`, получив сигнал от ядра о появлении нового устройства, создает симлинки со старых имен на новые. Точно так же, используя скрипты, приходится поступать в случае изменения прав на файлы-устройства.

Массив имен занимает довольно много места в `kernel space` — а это драгоценная оперативная память.

В ядрах Linux серии 2.6 появилась новая виртуальная файловая система — `sysfs`, которая предоставляет расширенную информацию об устройстве. Это позволяет решить

проблему с USB и подобными устройствами, которые подключаются к разным портам — их теперь легко найти через sysfs.

Файловая система devfs становится более ненужной. Получив сообщение от ядра о подключении/отключении устройства, программа `udev`, сверившись с файлом конфигурации и sysfs, создает и удаляет соответствующие файлы в реальной файловой системе. Major и minor numbers выделяются ядром динамически и передаются udev при подключении.

Что почитать:

1. `/usr/src/linux/Documentation/filesystems/devfs/`

Документация по файловой системе devfs.

2. `devfsd(8)`

Документация по devfsd, его конфигурации и принципам работы.

3. The Wonderful World of Linux 2.6 — <http://kniggit.net/wwol26.txt>

В этой статье приведен обширный список усовершенствований, появившихся в ядрах Linux серии 2.6, по сравнению с ядрами серии 2.4.

4. Frequently Asked Questions about udev —

<http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev-FAQ>

Разъяснение основных вопросов, касающихся udev и ссылки на более подробную документацию.

3.15. Работа с дискетами

Для форматирования дискет на физическом («низком») уровне служит утилита **fdformat**:

```
fdformat имя_устройства
```

В Linux существуют отдельные устройства для каждого из стандартных и нестандартных форматов дискет: `/dev/fd0u720`, `/dev/fd1u1440` и т.д. Устройства `/dev/fd0` и `/dev/fd1` пытаются определить формат дискеты самостоятельно. Таким образом, имя устройства, переданное команде **fdformat** однозначно определяет будущий формат дискеты, и никаких дополнительных параметров не требуется.

Форматирование дискеты на логическом уровне («быстрое форматирование»), т.е. создание на ней файловой системы, выполняется, как обычно, командой **mkfs**. Для чтения/записи дискету нужно примонтировать, а перед тем, как ее вынуть — не забыть отмонтировать, иначе данные, находящиеся в кэше, будут утеряны (само устройство не в состоянии помешать пользователю вынуть дискету, если она не отмонтирована).

Утилиты для более тонкого управления дисководом на физическом уровне находятся в пакете **fdutils**.

Утилиты из пакета **mttools** позволяют работать с дискетами, не монтируя их, «как в MS-DOS». Они аналогичны командам MS-DOS и даже используют то же самое соглашение о именовании дисков и файлов:

```
mformat A:  
mmdir A:/foo  
mdel B:/foo_bar.txt
```

Эти утилиты работают медленнее, чем происходила бы работа с примонтированной дискетой, поскольку не используют дисковый кэш. К тому же, чтобы ими пользоваться, нужен доступ на запись к устройству `/dev/fdn`. С помощью **mttools** можно работать и с разделами винчестера, отформатированными под FAT, но это имеет мало смысла.

Следует заметить, что из-за разницы в драйверах дискеты могут не читаться при переносе их между операционными системами, они ненадежны и слишком привязаны к физическим параметрам дисковода. К счастью, дискеты уже давно потеряли свое значение в качестве носителя информации и постепенно уходят в небытие.

Более совершенные накопители на магнитных дисках (ZIP, Jazz) подключаются к другим интерфейсам и, с точки зрения Linux, к дискетам не относятся.

Что почитать:

1. **fd(4)**

Подробное описание имеющихся в Linux специальных файлов для работы с дисководами и их возможностей.

2. **fdformat(8)**

Документация к утилите **fdformat**.

3. info fdutils

Подробное руководство по пакету утилит fdutils, принципам работы с дисковыми драйверами и возможностям драйвера дискового.

4. info mtools

Подробное руководство по пакету утилит mtools.

3.16. Идентификация типов файлов

Выяснить тип файла по его содержимому помогает утилита `file`:

```
file имя_файла
```

Идентифицируются файлы, чаще всего, по специальным меткам — *magic numbers*, которые располагаются в каком-то определенном месте в теле файла, но могут использоваться и другие признаки. Для каждого типа файла есть своя процедура проверки, которая записывается на специальном языке — их можно увидеть в `/usr/share/misc/file/magic` (`/usr/share/misc/file/magic.mgc` содержит те же самые процедуры в скомпилированном виде — для ускорения работы). Свои процедуры можно добавлять в `/etc/magic`.

Поскольку определять тип файла по содержимому каждый раз слишком хлопотно, часто тип указывают прямо в имени при помощи расширения или пользуются комбинацией этих двух способов — расширением и *magic numbers*.

С текстовым описанием, которое выводит `file`, не слишком удобно работать. Но существует и другой — краткий и однозначный способ записи типа файла — так называемый *MIME-type*. Этот способ был придуман для пересылки файлов по электронной почте, отсюда и слово MIME в названии — **M**ultipurpose **I**nternet **M**ail **E**xtensions.

MIME-тип имеет такой формат:

```
основной_тип/подтип
```

Основной тип — это текст (`text/*`), картинка (`image/*`), видео (`video/*`) и т.п. Это может быть и файл данных, специфичный для конкретного приложения (`application/*`).

Подтип указывает на конкретный формат — `image/png`, `text/html`, `application/pdf` и т.п. Неидентифицированный файл — простой поток байтов — обозначается `application/octet-stream`.

`File` может выводить тип файла в таком формате, если ее запустить с ключом `-i`.

Эти типы стандартизуются авторитетной организацией — IANA. Если какой-то фирме понадобятся обозначения для своих специфических форматов файлов (*vendor-specific*), она может получить свое собственное пространство имен и ее типы файлов будут записываться так:

```
основной_тип/vnd.название_фирмы.подтип
```

Например: `text/vnd.motorola.reflex`, `application/vnd.epson.esf`.

Нестандартные, экспериментальные (*experimental*) MIME-типы имеют префикс `x-`: `application/x-chm`, `x-world/x-vrml` и т.п.

В Linux сейчас, зачастую, у каждой программы есть своя таблица MIME-типов, соответствующих им расширений, приложений для их обработки и т.п. Общий стандарт базы данных MIME-типов уже готов и сейчас находится в процессе адаптации различными приложениями и дистрибутивами.

Что почитать:

1. `file(1)`

Документация к утилите file.

2. magic(5)

Формат файла с тестами, используемыми утилитой file для идентификации типа файла.

3. RFC 2046 — Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types — <http://www.faqs.org/rfcs/rfc2046.html>

Способ обозначения типа файла и некоторые основные типы.

4. RFC 2048 — Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures — <http://www.faqs.org/rfcs/rfc2048.html>

Процедура регистрации MIME-типов, обозначения для внутрифирменных, персональных и нестандартных MIME-типов.

5. Internet media type registry — <ftp://ftp.iana.org/in-notes/iana/assignments/media-types/>

Список всех MIME-типов, зарегистрированных в IANA.

6. Shared MIME Database — <http://www.freedesktop.org/Standards/shared-mime-info-spec>

Описание и реализация общей системы MIME-типов.