Использование операционной системы Linux при программировании

Учебный курс D-Link

Автор: Шибанов В.А.

Версия 1.0

Рязань 2015

Оглавление

Предисловие

Уважаемый читатель, вы приступаете к изучению учебного курса, посвященного использованию операционной системы Linux. Этот курс несколько отличается от большинства учебных курсов и книг по Linux. Большинство книг по Linux ориентируются на одну из двух категорий читателей.

Первая категория – это обычные пользователи, использующие компьютер для решения офисных задач. В них содержатся элементарные знания по командам Linux, и обычно приводится описание конкретных программ с указаниями, когда и куда надо щелкнуть мышью в той или иной ситуации.

Вторая категория – это системные администраторы. В книгах второй категории содержатся описания работы с программами для настройки сетевых и других устройств, сетевых сервисов и т.п.

Книги, предназначенные для программистов, описывающие как решать с помощью Linux задачи, стоящие перед программистом присутствуют, к сожалению, не в таком количестве, как книги двух озвученных групп.

Данный учебный курс задумывался как первый курс по Linux для читателей являющихся IT-профессионалами – программистами и тестировщиками, или студентами старших курсов IT-специальностей, дающий целостный и, по-возможности, непредвзятый взгляд на Linux как на мощный инструмент программиста.

Опыт преподавательской деятельности автора свидетельствует о том, что даже на IT- специальностях систематическое обучение с использованием Linux не проводится. В большинстве случаев используются Windows-программы, иногда на не вполне законных основаниях. Правда в последнее время ситуация несколько улучшается, как в плане внедрения Linux в учебный процесс, так и в плане перехода на лицензионные программы. Именно в области программирования Linux может быть высокоэффективным инструментом, покрывающим большую часть профессиональных потребностей программиста. Тем не менее, сегодня изучение Linux – удел в основном студентов-энтузиастов самоучек. Настоящий учебный курс является попыткой внести свой вклад в изменение сложившейся ситуации.

Все материалы курса можно условно разделить на три части.

Первая часть, состоящая из единственной первой главы, дает общую информацию об операционной системе Linux.

Вторая часть, включающая вторую, третью, четвертую, пятую, шестую и седьмую главы, посвящена работе с программным окружением Linux.

Третья часть, включающая восьмую, девятую, десятую и одиннадцатую главы, посвящена инструментальным средствам Linux, используемым программистами.

Автор выражает благодарность сотрудникам компании D-Link Александру Булычеву, Евгению Юдину, Елене Богдановой, Алесе Дунаевой, Ольге Кузьминой и доцентам Рязанского государственного радиотехнического университета Александру Бакулеву и Алексею Сапрыкину за помощь в создании данного учебного курса.

Глава 1. Общие сведения об ОС Linux

Общие сведения о Linux

Операционная система представляет собой, пожалуй, самую сложную компьютерную программу. Она является мостом между аппаратными устройствами компьютера и выполняющимися на нем прикладными программами. Именно операционная система управляет всей работой компьютера, организует все взаимодействия устройств и программ, а так же программ между собой. Как и любая достаточно сложная система, операционная система вообще, и Linux в частности, характеризуется множеством свойств и характеристик. Рассмотрим эти свойства.

Linux (как и ее предок Unix) изначально создавалась как **многопользовательская** система, т.е. на одном и том же компьютере может работать несколько пользователей (в том числе и одновременно). В многопользовательских системах для каждого пользователя хранятся его личные настройки. Кроме этого, каждый пользователь может определить правила доступа к своим данным других пользователей, а так же, для каждого пользователя задаются полномочия, в пределах которых он может заниматься настройкой компьютера. Если в этом аспекте сравнить Linux с другими популярными операционными системами, то Mac OS X, тоже является многопользовательской. Что касается Windows, то ее первые версии были разработаны на основе однопользовательской операционной системы DOS. Современные версии Windows построены на базе Windows NT, и являются многопользовательскими, однако при реализации данной функции по факту пришлось искать компромисс с обратной совместимостью прикладных программ.

Кроме этого, Linux является **многозадачной** операционной системой, т.е. позволяет выполнять несколько программ одновременно. Правда для современной операционной системы это является само собой разумеющимся, а немногозадачные операционные системы сейчас практически не используются.

Работа с файловыми системами – одна из сильных сторон Linux. Unix изначально была построена вокруг концепции файлов, один из ее принципов гласит: “Все является файлом”. Для большей гибкости при работе с файлами в Linux существует специальный модуль – Виртуальная файловая

система (Virtual File System, VFS). VFS используется как промежуточный уровень, и позволяет Linux поддерживать очень большое количество файловых систем. В частности, Linux может использовать диски как с собственными форматами данных (ext, ext2, ext3, ext4), так и диски Windows (ntfs), Mac OS (hfs), старых версий Unix (sysv), DOS (msdos), файловые системы CD-ROM, DVD- и flash-устройств (iso9660 и др.), а так же сетевые и специальные файловые системы (NFS, ProcFS и др.).

Работа с сетью – еще одно сильное место Linux. Unix была первой операционной системой, в которой был реализован сетевой стек TCP/IP. Linux, как ее потомок, так же поддерживает как большое количество сетевых протоколов (практически все распространенные протоколы), так и огромный перечень сетевой аппаратуры. Кроме того, в комплекте с дистрибутивами Linux изначально поставляется множество программ, предназначенных для работы в сети, например, web-серверы, ftp-серверы, средства сетевой диагностики и настройки и т.п. Не удивительно, что Linux очень широко применяется в качестве операционной системы для сетевых серверов, а так же для различных сетевых устройств, например, роутера D-Link DIR-320 и других устройств.

Одной из удобных особенностей Linux является наличие **систем управления пакетами**. Для данных систем нет полноценного аналога в мире Windows. Системы управления пакетами позволяют автоматизировать процессы установки, удаления и обновления программ, загрузки программ из локальных или удаленных репозиториев. При установке программ автоматически обрабатываются ситуации, связанные с конфликтами версий различных пакетов. В мире Linux существуют два основных формата программных пакетов и систем управления пакетами, построенных на их основе, - это формат RPM, характерный для Red Hat Linux, и формат DEB, характерный для Debian Linux и его производных, в частности Ubuntu и Mint.

При создании любой новой операционной системы очень остро встает вопрос наличия для нее прикладных программ. Операционная система сама по себе, без соответствующего прикладного программного обеспечения для пользователя практически бесполезна. В то же время для существующих операционных систем уже существует большое количество программ, переписывать которые с нуля будет достаточно накладно. Поэтому одной из важных задач при создании ОС является наличие в ней **средств эмуляции других операционных систем**, которые позволят выполнять предназначенные для других ОС программы. В настоящее время наиболее

распространенной операционной системой является Windows. Для Windows разработано множество программ, предназначенных для использования в различных областях человеческой деятельности. Поэтому эмуляция Windows-окружения – первоочередная задача для любого разработчика новой ОС. В Linux таким средством является программа WINE, представляющая собой независимую реализацию программных интерфейсов Windows (WinAPI). WINE позволяет запускать в Linux многие Windows-программы, однако есть и те, которые не выполняются с помощью WINE. Так же для Linux разработано несколько виртуальных машин, позволяющих эмулировать выполнение на Linux-машине других операционных систем. Примерами виртуальных машин являются QEMU, VirtualBox и VmWare.

Наконец, последней особенностью Linux, которую мы рассмотрим, является то, что операционная система Linux, изначально разработанная для процессора Intel 386, сейчас портирована на очень большое количество **аппаратных платформ**. Если мы рассмотрим наиболее распространенные ОС для настольных компьютеров, то они достаточно жестко связаны с одной или несколькими платформами, например работа Mac OS X официально гарантируется только на компьютерах Macintosh[1], а Windows работает на Intel x86, AMD в 32-х и 64-х разрядном варианте (плюс разработан вариант Windows для ARM). Что же касается Linux, то благодаря тому, что эта операционная система является открытой и свободной, она была портирована на очень большое количество аппаратных платформ. Это и платформы для персональных компьютеров – Intel x86, IA-64, и платформы для встроенных систем – ARM, MIPS, AVR32, и платформы для мейнфреймов, на которых работали старые версии Unix, например, IBM System/390, и платформы, характерные для рабочих станций, Power PC, Spark и др.

История создания ОС Linux и Unix

1) Возникновение Unix. Bell Labs. Язык C.

Linux ведет свою родословную от операционной системы Unix, первая версия которой разработана еще в 1969 году. Знать историю Linux важно хотя бы потому, что многие технические и организационные особенности Linux имеют исторические корни. История создания Linux и других Unix-

[1] Существуют модификации Mac OS X, позволяющие запускать ее на компьютерах других производителей с архитектурой x86 (т.н. хакинтош), однако они нарушают лицензионное соглашение Apple.

систем замечательно описана в книге Эрика Реймонда “Искусство программирования в Unix” [1]. Далее мы попытаемся кратко рассмотреть некоторые важные вехи в истории Unix.

Первая версия Unix была разработана сотрудниками исследовательской лаборатории Bell Labs (подразделения компании AT&T) Кеном Томпсоном, Деннисом Ритчи и Дугласом Макилроем в 1969 году, а выпущена в свет в 1971 году. Первоначально Unix была не официальным проектом компании, а являлась инициативной разработкой сотрудников – они создавали удобное средство для работы на ЭВМ для самих себя, для выполнения своих собственных задач. То есть Unix с самого начала создавался программистами как **инструмент для программистов**. Разработка оказалась весьма удачной и стала распространяться сначала внутри самой компании, для поддержки ее собственного парка компьютеров, используемых для решения внутренних задач.

В 1973 году произошло важное событие – исходный код Unix был переписан на специально разработанном для этого Деннисом Ритчи языке программирования высокого уровня C. До этого операционные системы писались на языке ассемблера, и при смене платформы требовалось переписать весь код операционной системы с нуля. Теперь же вы могли заставить работать Unix на новой платформе, написав только компилятор языка C, и выполнив компиляцию исходных кодов Unix. Кроме облегчения переноса на другие платформы этот шаг позволил сделать исходный код Unix более читаемым и структурированным, что облегчило изменения системы в будущем.

2) Создание Berkley Unix (BSD)

Изначально компания AT&T не рассматривала Unix как коммерческий продукт – в силу антимонопольных ограничений она не имела права заниматься бизнесом в области компьютеров. Поэтому исходные коды системы распространялись среди владельцев компьютеров относительно свободно. В то время компьютеры были достаточно дороги, и их владельцами, как правило, были крупные организации – корпорации, университеты, ведомства. В университетских исследовательских центрах довольно активно начали модифицировать систему для своих нужд и обмениваться друг с другом результатами доработок. Фактически сформировалось своеобразное сообщество пользователей Unix, принимающих участие, как в доработке самой операционной системы, так и

в разработке для нее прикладных программ. Это сообщество было прообразом современного движения Open Source.

Первым и самым активным участником сообщества стал Калифорнийский университет в Беркли. В 1980 году агентство DARPA поручает исследовательскому центру Беркли разработку реализации стека TCP/IP для Unix. До этого Unix имела слабую поддержку сети. После реализации стека TCP/IP в BSD Unix она стала фактически самой передовой Unix-системой и заняла важнейшее место в развивающейся в то время сети Internet. Еще больше эта тенденция усилилась, когда несколько специалистов из исследовательского центра Беркли, в частности Билл Джой, основали в 1982 году корпорацию Sun Microsystems и стали производить рабочие станции и сетевые серверы, функционирующие под управлением разработанного ими варианта Unix, которая сначала называлась SunOS, а позднее была переименована в Solaris.

3) Free Software Foundation. Проект GNU.

Примерно в то же время компания Bell System, частью которой была Bell Labs, была разделена на части и практически сразу превратила Unix, ценность которой она уже осознала, в коммерческий продукт. Заметим, что до этого момента компания Bell System в соответствии с антимонопольным соглашением с министерством юстиции США была серьезно ограничена в возможностях ведения бизнеса в компьютерной отрасли, и продавать Unix не могла. Распространение исходных текстов было прекращено. Это естественно было воспринято многими программистами весьма негативно. Один из них, автор редактора Emacs, Ричард Столлмен, в 1983 объявил о создании проекта GNU, целью которого являлось создание свободной Unix-подобной операционной системы и набора прикладных программ, покрывающих большинство потребностей пользователей. Их нужно было переписать с нуля, не используя код из проприетарных программ. Чуть позже, в 1985 им же был создан Фонд свободного программного обеспечения (Free Software Foundation) – некоммерческая организация, занимающаяся поддержкой проекта GNU. В ходе проекта GNU случались как успехи, так и неудачи. В частности большая часть прикладных программ, используемых в дистрибутивах Linux, была создана именно в рамках проекта GNU (например, компилятор GCC), однако разработка ядра операционной системы GNU Hurd затянулась, и на настоящее время рабочая версия все еще не выпущена.

4) Коммерческие Unix. System V

Мы уже упомянули о разделе Bell System, позволившем AT&T начать продажу Unix. В 1982 была выпущена коммерческая версия UNIX System III, а годом позднее UNIX System V. Последняя получила достаточно широкое распространение. В мире UNIX произошел своеобразный раскол на две части – System V и BSD. В данных версиях стали появляться средства, выполняющие схожие задачи, но не совместимые друг с другом. Самый красноречивый пример – это две несовместимые реализации сетевого стека (сокеты на основе стека TCP/IP распространенные в настоящие время являются реализацией BSD).

Крупные корпорации купили лицензии на Unix System V и стали производить собственный коммерческий вариант UNIX, как правило, оптимизированный для работы на производимых ими компьютерах. Существуют операционные системы AIX от IBM, Xenix, первоначально выпущенная Microsoft, HP-UX от Hewlett-Packard, UnixWare от Novell.

5) Создание и развитие Linux

В начале 80-х годов произошло еще одно событие, на которое в Unix-мире сначала никак не отреагировали. На рынок вышли, и стали массово продаваться персональные компьютеры IBM PC. Процессоры 86 и 286 были слишком простыми, для того, чтобы на них можно было запустить Unix, однако по объемам продаж было понятно, что за этими компьютерами будущее. Фактически, они стали основным рынком и направлением компьютерной техники, что своевременно поняла фирма Microsoft, став монополистом на этом рынке. В 1985 году появился процессор 386, который был достаточно мощным и функциональным для запуска Unix, однако версии Unix, доработанной для этого процессора долгое время не было. Такой версией Unix могла стать BSD, но в то время Университет Беркли был занят судебной тяжбой с AT&T, которая продлилась до 1994 года. Поэтому на призыв финского студента Линуса Торвальдса о создании с нуля ядра Unix-подобной операционной системы для IBM PC с процессором Intel 386 откликнулись многие опытные разработчики. Версия 0.01 ядра Linux вышла в сентябре 1991 и содержала чуть более 10 тысяч строк кода, а в марте 1995 года вышла версия 1.2, содержащая уже 300 тысяч строк кода. Сейчас исходный код ядра Linux содержит более 15 миллионов строк кода.

Linux является продолжателем идей и традиций Unix, и в настоящее время фактически является самой популярной и распространенной Unix-подобной системой[2]. Код ядра Linux продолжает развиваться, в том числе при поддержке крупных компаний, таких как IBM, Google и др. Прикладные и системные программы так же развиваются силами, как компаний, так и добровольцев. Все это позволяет не сомневаться в уверенном будущем Linux.

Достаточно заметным явлением последних нескольких лет стало появление операционной системы Android, которая использует ядро Linux, но переопределяет ее пользовательский и отчасти программный интерфейс. Строго говоря, Android не является Unix. В то же время многие наработки из проекта Android были добавлены и в основную ветку разработки ядра Linux, так что от появления Android Linux только выигрывает.

Дистрибутивы Linux

Строго говоря, термином Linux обозначается только **ядро** операционной системы. Для нормальной работы с компьютером требуется еще много различных программ – командный интерпретатор, графическая среда рабочего стола, дополнительные драйверы устройств, текстовый процессор, электронная таблица и т.п. Все эти программы в комплексе вместе с ядром Linux составляют **дистрибутив Linux** – набор программного обеспечения готовый для конечной установки на пользовательское оборудование. В дистрибутиве часто присутствуют программы для настройки системы и установки программного обеспечения. Так же дистрибутив поставляется с собственной документацией. Кроме этого может осуществляться поддержка пользователей данного дистрибутива по электронной почте или телефону (поддержка часто является платной). Программы, входящие в каждый конкретный дистрибутив могут быть как свободными, так и не свободными.

Разработать дистрибутив Linux может любой желающий, однако популярные дистрибутивы разрабатываются коммерческими фирмами (Fedora, Ubuntu), или крупными сообществами при поддержке коммерческих фирм (Debian). Обычно дистрибутив рассчитан на наилучшее удовлетворение потребностей определенного круга пользователей. Существуют дистрибутивы для корпоративного применения (Red Hat Linux,

[2] На эту роль так же может претендовать операционная система Mac OS X

Open SUSE), для домашних пользователей (Ubuntu), для опытных пользователей Linux (ArchLinux) и для обычных пользователей новичков (тот же Ubuntu), для применения при решении научных задач (Scientific Linux) и для детей (DoudouLinux).

На момент разработки курса существует порядка шестисот дистрибутивов Linux, около ста из которых поддерживаются в актуальном состоянии. Последние данные о популярности различных дистрибутивов Linux, основанные на статистике их скачивания, можно найти на сайте www.distrowatch.com.

Разработка собственного дистрибутива – это очень большая работа, поэтому новые дистрибутивы Linux, как правило, разрабатывают на основе какого-либо из уже существующих дистрибутивов. Более того, большинство дистрибутивов Linux основаны на одном из двух дистрибутивов – это либо *Debian*, либо *Red Hat*.

На момент разработки курса наиболее скачиваемыми дистрибутивами являются дистрибутивы Mint, Mageia, Ubuntu, Fedora, OpenSUSE и Debian.

Начинающему пользователю Linux можно рекомендовать использовать дистрибутивы группы Debian (сам Debian, Ubuntu, Mint) как наиболее распространенные. С точки зрения материала, рассматриваемого в данном курсе серьезной разницы между ними нет.

Достаточно много программ, входящих в большинство дистрибутивов Linux, разработаны в рамках проекта GNU, поэтому более корректным является термин **GNU/Linux**.

Области применения Linux

Теперь давайте обсудим, каковы основные области применения Linux, исходя из его особенностей, сильных и слабых сторон.

Важнейшей особенностью Linux является его гибкая конфигурируемость. Мы можем включать или не включать в ядро те или иные компоненты в зависимости от используемых аппаратных средств и специфики решаемой задачи. В каких случаях это нам может быть необходимо?

Во-первых, если мы хотим использовать текущую аппаратную конфигурацию максимально эффективно (максимизируется объем или сложность решаемых задач).

Во-вторых, если мы хотим решить текущую задачу в системе с минимальными требованиями к аппаратным ресурсам.

Первому случаю соответствуют две различные области, где успешно применяется Linux - **суперкомпьютеры** и **сетевые серверы**. Здесь достаточно сказать, что на момент написания этих строк под управлением Linux работают более 90% крупнейших суперкомпьютеров мира (по данным сайта www.top500.org). Так же набор программного обеспечения LAMP (Linux, Apache, MySQL, PHP или Perl или Python) очень широко используется на сетевых серверах (в данном случае web-серверах).

Второй случай – это **встроенные системы** – специализированные вычислительные системы, предназначенные для решения какой-либо прикладной задачи, и, как правило, имеющие ограниченные аппаратные ресурсы. Скорее всего, у вас есть маршрутизатор. Вы можете быть почти уверены, что он работает под управлением Linux. В этой области Linux, пожалуй, доминирует, т.к. может работать на машине с очень ограниченными по современным понятиям ресурсами.

“А как же настольные компьютеры?” – наверняка уже думаете вы. Здесь пока успехи Linux не так впечатляющи. Мы с вами уже говорили, что Unix создавался как операционная система для программистов, а не для обычных пользователей. Поэтому на **настольных компьютерах** пока Linux все еще остается инструментом энтузиастов, т.к. требует достаточно глубоких начальных знаний для работы. Тем не менее, такие фирмы как Canonical, производитель дистрибутива Ubuntu, много делают для того, чтобы облегчить для пользователя переход на Linux. Существенным достоинством Linux как операционной системы для настольных компьютеров и рабочих станций является бесплатность, как самой операционной системы, так и прикладных программ для нее. Стоимость Windows сравнительно невелика, а вот программы для нее, таких как офисный пакет, среды разработки, пакеты САПР, могут стоить достаточно дорого. Поэтому, при наличии необходимых бесплатных аналогов для Linux, выбор последней является вполне обоснованным. Особенно это справедливо для программистов, т.к. для Linux существует достаточно большое число качественных и бесплатных инструментов разработки.

Графический интерфейс Unix-подобных систем. Графическая система X Window. Среды рабочего стола GNOME и KDE.

Графический интерфейс в Linux и других Unix-подобных системах включает в себя несколько взаимодействующих друг с другом программных компонентов. Каждый из этих компонентов выполняет свои функции, и для каждого из них может существовать несколько реализаций. Основными компонентами графического интерфейса Unix-подобных систем являются:

- графическая система X Window,
- менеджер окон,
- библиотека графических элементов,
- графическая среда рабочего стола.

Графическая система X Window представляет доступ другим программам к аппаратным графическим ресурсам – экрану монитора, мыши, клавиатуре по специально оговоренному интерфейсу (X-протоколу). Более того, графическая система X Window позволяет программе, запущенной на одном компьютере (X-клиенте) использовать для взаимодействия с пользователем графические средства другого компьютера (X-сервера). На данный момент практически повсеместно используется 11-я версия системы X Window, разработанная еще в 1987 году. Однако в нескольких ведущих дистрибутивах начата работа по переходу на более новую альтернативную систему Wayland, использование которой позволяет несколько упростить структуру графической подсистемы Linux.

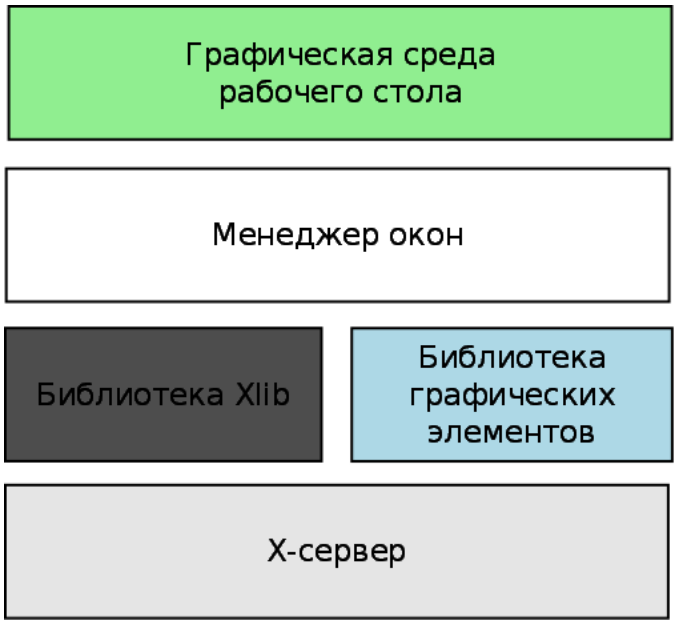

Рис. 1.1. Схема графической среды Linux

Менеджер окон работает поверх системы X Window. Он является посредником между данной системой и прикладными программами, которыми он управляет. Оконный менеджер реализует основные операции по работе с окнами - выполняет отрисовку основных элементов окна, позволяет изменять размер и положение окна во время выполнения программы, реализует возможность максимизации и минимизации окна, управляет тем, какое из окон является активным и получает поток ввода с клавиатуры от пользователя и т.п. Так же к функциям оконного менеджера относится поддержка нескольких виртуальных рабочих столов. Оконных менеджеров существует достаточно много. Некоторые из них реализуют часть функциональности среды рабочего стола.

Библиотеки графических элементов расширяют возможности стандартной библиотеки Xlib, которая содержит средства для отрисовки основных графических примитивов. Библиотеки графических элементов автоматизируют операции по использованию в прикладных программах распространенных графических элементов (виджетов) – кнопок, полей ввода и других графических элементов управления. Наиболее распространенными библиотеками графических элементов в Unix-подобных системах являются библиотеки Motif, Qt и GTK+.

Графическая среда рабочего стола является конечным приложением, которое, в итоге, реализует графический интерфейс пользователя, основанный на идее рабочего стола, используя механизмы остальных

компонентов графической подсистемы. Она обеспечивает единообразие, как внешнего вида окон других программ, так и реализует некоторые важные операции и элементы графического интерфейса (например, операцию Drag&Drop или поддержку буфера обмена). Как правило, среда рабочего стола включает панель задач, меню для запуска основных программ, менеджер входа в систему, программы для настройки, набор базовых программ.

Самыми распространенными средами рабочего стола являются GNOME, использующая библиотеку GTK+, и KDE, использующая библиотеку Qt. Список программ, поставляемых вместе с данными средами, представлен в табл. 1.1.

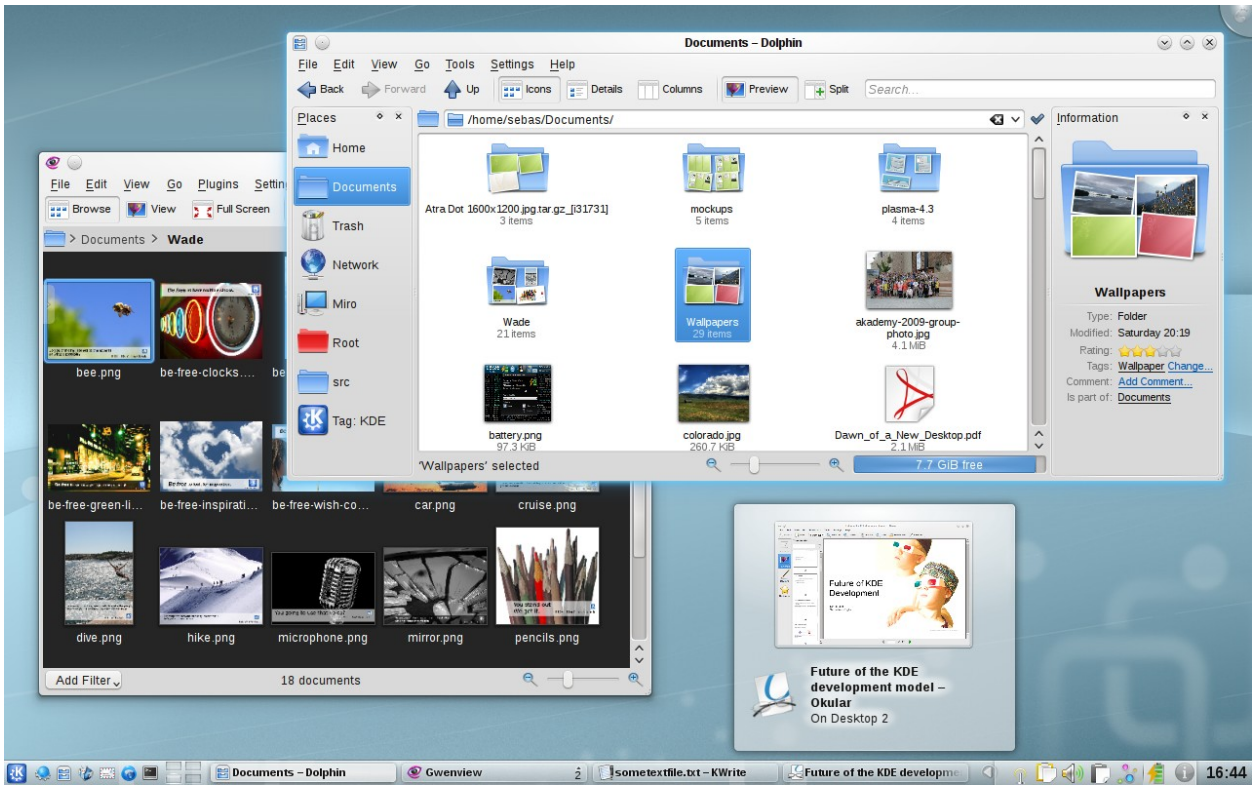

Рис. 1.2. Изображение рабочего стола KDE[3]

[3] www.kde.org

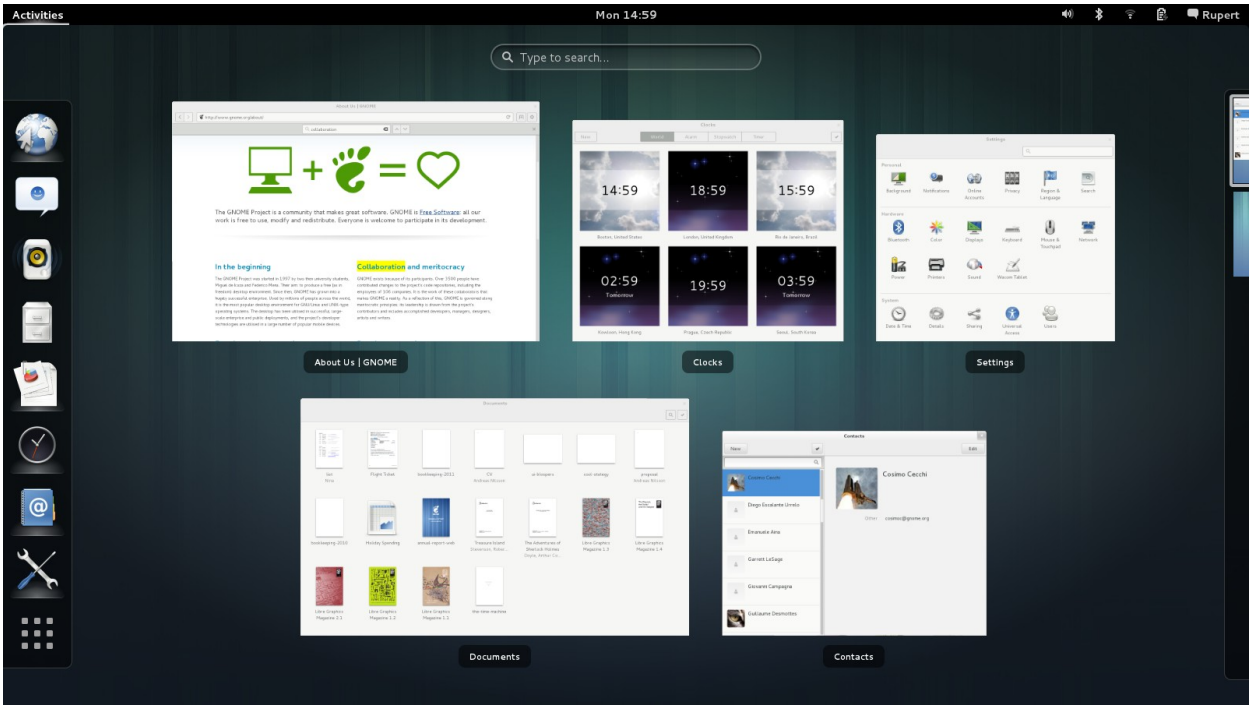

Рис. 1.3. Изображение рабочего стола GNOME 3[4]

Табл. 1.1. Основные программы, поставляемые с распространенными средами рабочего стола

Тип программы	Название программы в GNOME	Название программы в KDE
Файловый менеджер	Nautilus	Dolphin
Терминал	GNOME Terminal	Konsole
Простой текстовый редактор	Gedit	Kedit
Браузер	Epiphany	Konqeror
Почтовый клиент	Evolution	Kontact
Программа для мгновенного обмена сообщениями (IM)	Empathy	Kopete
Медиа-проигрыватель	Totem	Amarok
Офисный пакет	Официальный офисный пакет отсутствует, но дополнительно могут быть установлены программы текстовый редактор AbiWord, табличный процессор Gnumeric и др.	Koffice

[4] www.gnome.org

Глава 2. Командная строка Linux

Виды компьютерных интерфейсов

Одним из препятствий, возникающих перед начинающими пользователями Linux, является необходимость использования командной строки. Вначале она может казаться чем-то устаревшим и пугающим. Но со временем вы можете обнаружить, что командная строка – достаточно мощное средство, и начать ее использовать в тех ситуациях, в которых до этого использовали средства графического интерфейса.

Давайте сначала разберемся в том, какие человеко-машинные интерфейсы бывают в принципе, и попробуем сравнить их сильные и слабые стороны, а так же рассмотрим задачи в которых эти сильные и слабые стороны определяют выбор того или иного интерфейса.

Согласно [2], способ, которым вы выполняете какую-либо задачу с помощью какого-либо продукта, а именно совершаемые вами действия и то, что вы получаете в ответ и является **интерфейсом**.

Условно можно выделить следующие виды интерфейсов (такая классификация, конечно, является субъективной, но вполне соответствует целям повествования):

- ***Текстовый интерфейс пользователя*** (***интерфейс командной строки***, ***Command Line Interface***, ***CLI***). Оператор вводит команды в текстовом виде, компьютер выполняет их и отвечает тоже текстовыми сообщениями.
- ***Графический интерфейс пользователя с использованием мыши*** (***графический интерфейс пользователя***, ***Graphic User Interface***, ***GUI***). На экране компьютера отображается графическое изображение некоторой системы, оператор выполняет команды, указывая мышью на специальные зоны изображения (экранные кнопки, ссылки и т.п.).
- ***Графический интерфейс пользователя с использованием мультисенсорных панелей***. Отличается от предыдущего тем, что вместо мыши используется прямое указание пальцами или стилусом, а программа может обрабатывать действия сразу нескольких пальцев одновременно.

- ***Голосовой интерфейс пользователя*** (***Voice User Interface***). Пользователь отдает компьютеру команды голосом, а компьютер их интерпретирует и выполняет.

Два последних типа интерфейса начали широко внедряться относительно недавно и применяются в основном на портативных устройствах, таких как смартфоны и планшеты. Область их применения в настольных компьютерах пока только формируется, и мы с вами являемся свидетелями этих изменений.

Рассмотрим подробнее первые два типа интерфейса – интерфейс командной строки и графический интерфейс пользователя. Они проверены временем и показали свою эффективность. Как текстовый, так и графический интерфейсы имеют свои сильные и слабые стороны, однако не следует их противопоставлять. Эти два типа интерфейсов проявляют свои достоинства в разных типах задач и подходят для разных категорий пользователей, поэтому хорошо дополняют друг друга.

Графические интерфейсы обладают рядом достоинств, а именно:

- Такие интерфейсы отображают списки возможных вариантов действий и подсказки к ним, поэтому они существенно упрощают процесс изучения программы и для **пользователей-новичков являются предпочтительными**.
- Возможность реализации **принципа WYSIWYG** (What You See Is What You Get – что вижу, то и получаю). Этот принцип существенно упрощает работу в случае, если результатом работы является документ, имеющий непосредственное визуальное представление, например, текстовый документ или чертеж. В этом случае при редактировании пользователь сразу видит результат своих действий. В то же время не все объекты, создаваемые на компьютере, имеют непосредственное визуальное отображение. Например, если вы разрабатываете звуковой файл или программу, то получить преимущество от реализации принципа WYSIWYG не получится.

Сильными сторонами интерфейсов командной строки является следующее:

- Текстовые интерфейсы **более эффективны для опытных пользователей**. Когда команды уже достаточно хорошо изучены, на первое место выходит скорость и удобство работы. Нажатие клавиш на клавиатуре является более простой и быстрой

операцией, чем многократное позиционирование и нажатие кнопок мыши. Существующие исследования по измерению времени выполнения операций на компьютере [2] показывают, что нажатие клавиши на клавиатуре занимает примерно 0,2 с, позиционирование мыши – 1,1 с, а переключения между мышью и клавиатурой – 0,4 с. Таким образом, наиболее эффективным является интерфейс, использующий только клавиатуру, а не принуждающий пользователя переключаться от клавиатуры к мыши и обратно. Со временем, вы сами убедитесь, что набрать в терминале пару команд быстрее, чем перемещаться по каталогам в файловом менеджере.

- Интерфейсы командной строки предоставляют **широкие возможности для автоматизации**. Это одно из решающих достоинств. Графический интерфейс позволяет реализовать только те действия, которые в нем реализованы. Часто приходится повторять комбинации из нескольких действий. В командном интерфейсе вы можете записать соответствующие команды в скрипт и запускать его одной командой. Причем можно это делать не вручную, а автоматически в определенное время. Так же, скрипт может выполнять некоторые действия по анализу данных. Графический интерфейс не решит за вас, на какую кнопку нужно нажать, даже если данные известны заранее.
- Интерфейсы командной строки, как правило, **более лаконичны**.

Работая в Windows, где встроенные средства командного интерфейса заметно отстают[5] от графических инструментов, можно придти к выводу, что интерфейс командной строки является устаревшим и неудобным. Работая в Linux, вы почувствуете всю мощь командных интерфейсов.

Выбор того или иного интерфейса зависит от характера решаемой задачи. Если в рамках задачи имеет смысл мыслить в терминах существительных, то, вероятно, лучше использовать графический интерфейс. Если же в вашу задачу целесообразно описывать в основном в терминах-глаголах, стоит присмотреться к командным интерфейсам.

[5] Здесь идет речь о традиционной командной строке Windows, реализованной в файле cmd.exe. Начиная с Windows Vista, в Windows появился новый мощный инструмент командной строки – PowerShell, по возможностям сравнимый с командной строкойLinux. PowerShell не устанавливается при установке Windows, и должен быть установлен отдельно.

Для программистов язык команд является привычным. Вы ведь пишите код, правда? А блок-схемы вы когда чертили последний раз?

Дальше мы с вами подробно изучим командный интерфейс Linux, представляющий собой хороший пример мощного и выразительного командного интерфейса. Он проверен и доведен до состояния близкого к совершенству несколькими поколениями программистов во всем мире. Кстати, тот же командный интерфейс Unix присутствует и в Mac OS X.

Командные оболочки Linux. Оболочка bash

Командный интерфейс в Linux и других UNIX-подобных ОС реализуется специальной программной, называемой **командной оболочкой**, **shell**, **командным интерпретатором** или **командным процессором**.

Командная оболочка с точки зрения Linux представляет собой обычную прикладную программу, предназначенную для организации взаимодействия между пользователем и ядром операционной системы, причем это взаимодействие осуществляется путем ввода текстовых команд пользователем и их выполнения командной оболочкой. Еще одной областью применения командной оболочки является выполнение сценариев.

В Linux и других Unix-подобных операционных системах существует множество командных оболочек, и вы можете выбрать для использования ту, которая вам больше по душе.

Исторически первой появилась оболочка Борна – *Bourne Shell* (*shell*, *sh*), позволяющая выполнять интерактивные команды и сценарии. Она была разработана Стивеном Борном из Bell Labs.

Несколько позже в рамках BSD Unix появилась оболочка *C-shell* (*csh*), созданная в университете Беркли. Она для написания сценариев использовала синтаксис, приближенный к языку C.

Все остальные оболочки делятся на sh-совместимые и csh-совместимые, в зависимости от того синтаксиса, который они поддерживают.

В настоящее время наибольшее распространение получила оболочка *bash* (*Bourne Again Shell* – Возрожденная оболочка (Борна)) – усовершенствованная версия оболочки Борна, созданная в рамках проекта GNU. Она используется по-умолчанию в большинстве дистрибутивов Linux, а так же в других Unix-подобных операционных системах.

Еще одной распространенной sh-совместимой оболочкой является Z-*shell* (*zsh*), содержащей широкие интерактивные возможности в некоторых моментах превосходящие возможности bash.

Среди csh-совместимых оболочек достаточно распространенной является оболочка *tcsh*, которая часто используется в FreeBSD-системах.

Еще одной оболочкой, с которой вы можете столкнуться, является *BusyBox*. Она создавалась с целью минимизации размера исполняемого файла оболочки и применяется в основном во встроенных системах, например, в модемах и маршрутизаторах.

В курсе мы будем использовать оболочку **bash** как достаточно мощную, удобную и наиболее распространенную.

Структура команды

Как мы видели, работа в терминале представляет собой последовательность ввода команд. Каждая вводимая команда соответствует следующему формату:

```
команда –опции параметры
```

где опции – это некоторые текстовые слова или символы, уточняющие логику работы команды, а параметры, как правило, определяют к какому объекту команду нужно применить. Например, команда

```
ls -l /home/user/dir1/
```

выводит на экран расширенную информацию (`l` - long) о файлах, которые содержатся в каталоге `/home/user/dir1/`.

В строке команды сама команда, опции и параметры команды отделяются друг от друга разделителями, которыми по-умолчанию являются символы пробела и табуляции. Часто требуется в качестве одного параметра передать команде строку, содержащую пробелы. Если мы просто наберем

```
команда строка с пробелами
```

то введенная команда будет трактоваться как команда с тремя параметрами – “строка”, “с” и “пробелами”. Если требуется задать именно один строковый параметр, то исходную строку при наборе команды нужно заключить в двойные кавычки:

```
команда “строка с пробелами”
```

Для команды символы внешних кавычек прозрачны – они в команду не передаются, а только определяют границы параметра. Если внутри параметра должен присутствовать символ двойной кавычки, то его нужно записать в строке два раза подряд:

```
Команда “символ кавычки - “” задается дважды”
```

Имя команды, как правило, совпадает с именем исполняемого файла соответствующей программы и обычно содержит от двух до десяти символов. Причем команды, которые используются часто, имеют короткие имена, например `ls`, `cd`, `cp` и т.д., а реже используемые команды – более длинные имена.

Опции обычно состоят из одной буквы и задаются с предшествующим им знаком дефис. Можно одновременно указывать в команде несколько опций. При этом простые однобуквенные опции могут объединяться под одним знаком дефиса. Например, две следующие команды равнозначны:

```
ls -a -l
ls -al
```

При задании опций заглавные и строчные буквы различаются, например, команда `ls -R` выдает список файлов, содержащихся в текущем каталоге и всех его подкаталогах, а команда `ls -r` выдает список файлов, содержащихся в текущем каталоге и выданных в обратном алфавитном порядке.

Опции так же могут задаваться не буквой, а цифрой, но такая практика является не очень распространенной. Например, следующая команда выводит календарь на текущий, предыдущий и следующий месяц:

```
cal -3
```

Некоторые опции кроме активизации некоторого режима требуют задания вместе с ними параметра или параметров (параметров опций). Параметры опций являются обязательными. Например, команда

```
grep –f file1 –v file2
```

выводит список строк из файла `file2`, которые не соответствуют шаблонам, содержащимся в файле `file1`. Здесь `file1` является параметром опции `-f`.

Так же некоторые опции могут задаваться не одной буквой, а целым словом. В этом случае они предваряются не одним, а двумя дефисами. Например, опция `--help` практически в любой команде выдает справку по ее использованию, например:

```
wc --help
```

Если в списке опций встречается знак двойного дефиса `--` без последующего имени опции, то это указывает на окончание списка опций. Все, что будет содержаться дальше в строке команды, считается параметром даже в том случае, если начинается с символа дефиса.

Терминал. Приглашение командной строки. Обычный пользователь и суперпользователь

Для ввода команд используются специальные программы-терминалы. Например, в графической среде GNOME присутствует программа GNOME Terminal. Данную программу можно запустить через меню “Приложения/Стандартные/Терминал”. В дальнейшем мы будем подразумевать, что используется именно эта программа. Вид окна программы GNOME Terminal показан на рис 2.1.

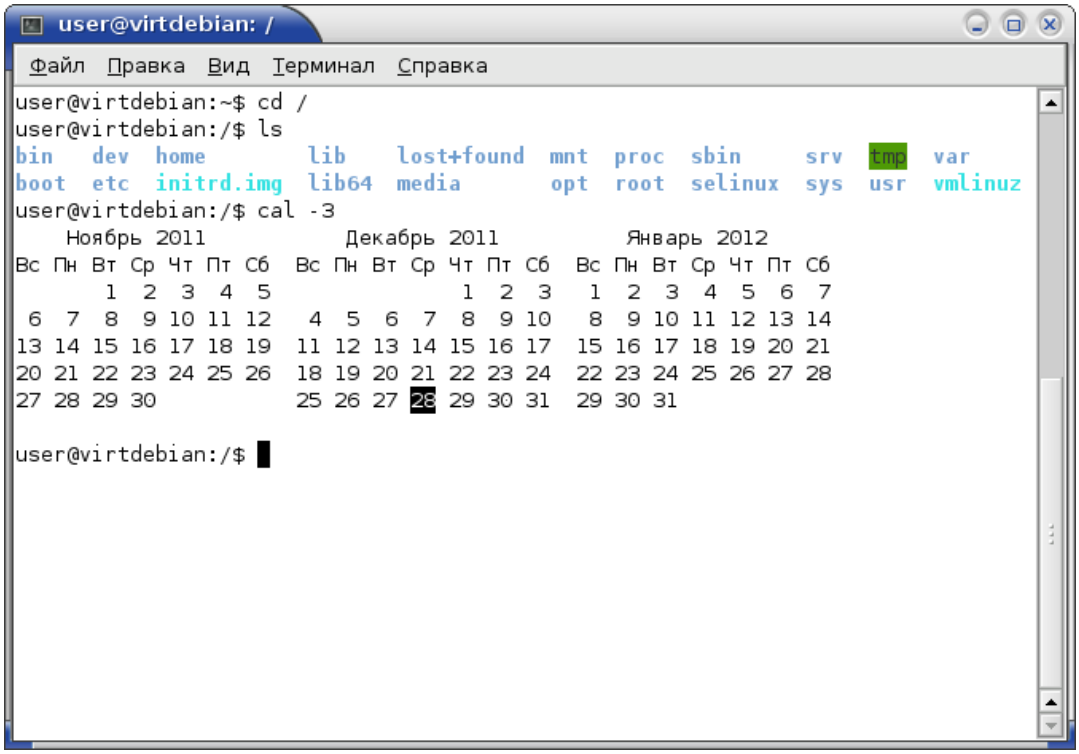

Рис. 2.1. Окно программы GNOME Terminal

При запуске в окне терминала на экране отобразится текстовая строка, заканчивающаяся символом $ и мигающий курсор справа от него. Это приглашение к вводу команд, которое означает, что вы можете вводить команды.[6]

Например, наберите `ls` и нажмите клавишу `Enter`. При этом на экран будет выведен список файлов, содержащихся в текущем каталоге, и после него на экране опять появится приглашение к вводу. Можно вводить следующую команду.

Вид приглашения к вводу может настраиваться и зависит от того, какой пользователь вводит команды. В Linux, как и в любой другой Unix-среде, один пользователь выделяется особо. Это пользователь с номером 1, называемый **администратором**, **суперпользователем** или **пользователем root**. Суперпользователь отличается от остальных тем, что только суперпользователю разрешается выполнять ряд действий по настройке системы, например, задавать параметры сетевых устройств или подключать и

[6] Далее в книге во всех листингах команды, которые вводит пользователь, будут начинаться со знака доллара “$”, чтобы отличить их от вывода команды. При наборе с клавиатуры знак “$” вводить не нужно. Он только обозначает приглашение к вводу команды. Для обозначения ввода команд, требующих прав суперпользователя будет использовано приглашение “#” или команда sudo.

отключать дисковые устройства. Часть команд, предназначенных для настройки системы, недоступны обычным пользователям. При работе в режиме суперпользователя с компьютером можно выполнить любую операцию, в том числе и такую, результат которой может быть необратимым, поэтому работать все время от имени суперпользователя считается небезопасным. В режим суперпользователя входят на короткое время для выполнения необходимых операций.

Для того чтобы можно было определить, к каком режиме находится система – в режиме обычного пользователя или в режиме суперпользователя, приглашение к вводу команд в этих режимах делают различным. В режиме обычного пользователя приглашение к вводу заканчивается символом `$`, а в режиме суперпользователя приглашение к вводу заканчивается символом `#`.

Команды su и sudo

Как мы обсуждали ранее, часть команд (тех, что предназначены для настройки системы) являются недоступными для обычных пользователей и могут выполняться только администратором системы, имеющим имя `root` (он же суперпользователь). Примером такой команды, которую мы будем использовать в этом разделе, является команда `ifconfig`[7], предназначенная для настройки сетевых интерфейсов компьютера. Если запустить данную команду без параметров, то она выведет на экран текущие настройки сетевых интерфейсов.

Для выполнения команд, требующих прав пользователя `root` можно закончить текущую сессию и начать новую, зарегистрировавшись как `root`. Однако делать это всякий раз, когда нужно выполнить привилегированную команду очень неудобно, потому что при новой регистрации придется каждый раз выполнять настроечные действия типа открытия нужного каталога, извлечения необходимой информации и т.п. Работать все время как пользователь `root` так же неудачное решение, поскольку это небезопасно. Выход можно найти в использовании двух специальных команд `su` и `sudo`, позволяющих временно получить права пользователя root.

[7] Описанная здесь ситуация является специфичной для дистрибутива Debian. Тем не менее, она является хорошим примером для пояснения работы команды `sudo`

Команда su позволяет временно подменить идентификатор текущего пользователя на идентификатор пользователя root. По-умолчанию остальные настройки пользователя остаются неизменными. После ввода команды su система запрашивает пароль пользователя root. Введите пароль (на экране он не отображается) и если вы не ошиблись при вводе пароля, то на экране отобразится приглашение к вводу команд. В конце приглашения стоит знак #, а это значит, что вы – суперпользователь и можете вводить привилегированные команды! Для возврата в режим обычного пользователя после выполнения необходимых команд нужно ввести команду exit.

```
$ ifconfig
bash: ifconfig: команда не найдена
$ su
Пароль:
# ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:46:af:57
          inet       addr:10.0.2.15              Bcast:10.0.2.255
Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe46:af57/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:40 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1510 (1.4 KiB)  TX bytes:6653 (6.4 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:732 (732.0 B)  TX bytes:732 (732.0 B)

# exit
exit
$
```

Недостатком подхода, связанного с использованием команды su, является то, что каждый пользователь, которому может понадобиться

выполнить привилегированную команду, должен знать пароль пользователя root. Естественно с точки зрения безопасности это плохое решение. Исправить этот недостаток можно используя команду sudo. Команда sudo отличается от команды su двумя моментами.

Во-первых, она по-умолчанию не переходит в отдельный режим ввода привилегированных команд, а выполняет только одну такую команду. И эта команда указывается в качестве параметра в команде sudo:

```
$ sudo команда параметры
```

Если вам нужно выполнить несколько привилегированных команд, то можно перейти в специальный режим подобно тому, как это делает команда su. Режим ввода привилегированных команд запускается командой

```
$ sudo -i
```

Во-вторых, при выполнении команды требуется ввести не пароль пользователя root, а свой собственный пароль. Кроме этого команда sudo является более гибкой, чем команда su в том смысле, что можно разрешить пользователю выполнять не все, а только некоторые привилегированные команды. Причем такая настройка выполняется для каждого пользователя. Сами настройки команды sudo содержатся в файле /etc/sudoers. С форматом данного файла можно ознакомиться, выполнив команду man sudoers (команда man рассматривается в следующем пункте). Если вы хотите использовать команду sudo, то вам нужно сначала добавить сведения о вашей учетной записи в файл /etc/sudoers.

Пример использования команды sudo:

```
$ ifconfig
bash: ifconfig: команда не найдена
$ sudo ifconfig
[sudo] password for user:
eth0      Link encap:Ethernet  HWaddr 08:00:27:46:af:57
          inet       addr:10.0.2.15             Bcast:10.0.2.255
Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe46:af57/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:171 errors:0 dropped:0 overruns:0 frame:0
```

```
          TX packets:152 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:111865 (109.2 KiB)  TX bytes:22407 (21.8 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:732 (732.0 B)  TX bytes:732 (732.0 B)
$
```

В целом можно считать, что команда `su` является устаревшей, и рекомендовать везде использовать более мощную и безопасную команду `sudo`. Достоинства команды `sudo` состоят в том, что она позволяет не сообщать обычным пользователям пароль администратора и делегировать им для выполнения конкретной задачи минимально необходимые полномочия.

Команды man и info

Очень полезными командами являются команды `man` и `info`. Эти команды предназначены для вывода справочной информации.

Команда `man` (сокращение от англ. *manual* – руководство, справочник) предназначена для доступа к встроенным руководствам (так называемым `man`-страницам). Формат запуска команды:

```
$ man [раздел] имя [[раздел] имя …]
```

Здесь и далее будем указывать необязательные параметры команд в квадратных скобках. Многоточие показывает, что в команде можно указывать не только один параметр `имя`, но и несколько таких параметров (в этом случае страницы будут отображаться по очереди).

Параметр `имя` задает имя команды, утилиты или функции для которой нужно отобразить руководство.

Например, чтобы отобразить руководство по команде man нужно набрать команду

```
$ man man
```

Так как могут существовать несколько объектов с одинаковыми именами (и часто реально существуют), то нужен какой-то механизм определения, о каком именно объекте идет речь. Для этого все страницы разбиты на *разделы*, а в пределах одного раздела каждое имя является уникальным. Раздел задается в команде *номером*. Возможные номера разделов перечислены в табл. 2.1. Например, команда `man 1 kill` отображает справку по **команде** `kill`, а команда `man 2 kill` отображает справку по **функции** (системному вызову) `kill`.

При указании `man`-страницы номер раздела часто записывается в круглых скобках. Например, только что обсуждались страницы `KILL(1)` и `KILL(2)`.

По умолчанию команда `man` просматривает все разделы по порядку.

При отображении справки в команде `man` вы входите в специальный режим данной команды. Выйти из просмотра справочной страницы и вернуться в режим ввода команд можно нажав клавишу `q`.

Табл. 2.1. Список разделов man-страниц

Номер раздела	Описание раздела
1	Исполняемые программы или команды
2	Системные вызовы (функции, предоставляемые ядром)
3	Библиотечные вызовы (функции, предоставляемые программными библиотеками)
4	Специальные файлы устройств (обычно находящиеся в каталоге `/dev`)
5	Форматы стандартных файлов системы, например `/etc/passwd`
6	Игры и прочие несистемные программы
7	Разное (включает пакеты макросов и соглашения), например, `man(7)`, `groff(7)`
8	Команды администрирования системы (обычно, запускаемые только суперпользователем)
9	Процедуры ядра (раздел для системных программистов)

Команда `info` похожа на команду `man`. При запуске в команде `info` тоже указывается имя команды, для которой требуется вывести справку, например:

```
$ info ls
```

Основное отличие ее от команды `man` состоит в том, что информация в команде `info` отображается в гипертекстовом виде. Это несколько отличается внешне от всем знакомых `html`-страниц, но смысл, в общем, тот же.

Ссылки в `info`-страницах обозначаются символом *звездочки*. Для перехода по ссылке нужно перевести на нее курсор и нажать клавишу `Enter`. При нажатии клавиши `P` осуществляется переход к предыдущей странице, при нажатии на `N` – переход к следующей странице, а при нажатии клавиши `U` – переход на страницу уровнем выше.

Некоторые полезные возможности при работе в консоли

В заключение главы рассмотрим средства терминала, позволяющие облегчить выполнение рутинных операций и создающие некоторое удобство в работе.

Терминал позволяет использовать при работе как клавиатуру, так и мышь, а так же использовать меню программы. Например, вы можете выделить команду мышью, скопировать ее с помощью меню, а потом вставить в строку приглашения к вводу.

Так же полезной возможностью является использование вкладок. Вы можете открыть несколько вкладок и выполнять несколько сеансов работы с терминалом параллельно. Это может быть полезно, например, при отладке нескольких параллельно выполняющихся и взаимодействующих друг с другом программ.

Одной из полезных возможностей терминала является возможность работы с *историей команд*. При вводе команд с помощью клавиш управления курсором *вверх* и *вниз* можно вызвать одну из ранее введенных команд не вводя ее повторно. Особенно удобно то, что история команд не теряется при завершении сеанса работы с терминалом или выключении компьютера и ведется для каждого пользователя.

Еще одной полезной возможностью терминала является *автодополнение*. Довольно часто одним из параметров команды или самой командой является имя файла содержащегося в текущем каталоге. Если это

имя длинное, то набирать его каждый раз достаточно утомительно и при наборе легко ошибиться. Но терминал может здесь нам помочь. Достаточно набрать несколько первых символов имени файла и нажать клавишу `Tab`. Терминал дополнит имя файла автоматически, насколько это возможно. Если таких файлов несколько, то имя будет дописано до конца общей части. При следующем нажатии `Tab` будет выведен список всех возможных вариантов автодополнения.

Автодополнение так же работает и для имен команд.

Наконец, чтобы записать в командную строку какое-либо слово, уже содержащееся на экране, нужно выделить это слово мышью и щелкнуть средней клавишей мыши (на большинстве современных моделей мышей средняя клавиша совмещена с колесиком, но это колесико можно не только крутить, но и нажимать).

Глава 3. Файлы и каталоги

Типы файлов в Linux

Файл является одним из центральных понятий Linux и Unix. Одним из основных принципов построения системы является принцип "*Everything is file*" (*Все является файлом*). Смысл этого принципа состоит в том, что любая операция, связанная с обработкой и хранением некоторых данных, так или иначе, является файловой операцией. За счет этого достигается единообразие и лаконичность программных интерфейсов - очень большое число операций с различными объектами выполняется с помощью четырех файловых операций – open, close, read, write.

Рассмотренный принцип приводит к тому, что типов файлов в Linux довольно много. Их шесть:

- обычный файл,
- каталог,
- символьная или мягкая ссылка,
- файл устройства,
- канал,
- сокет.

Обычный файл (*file*) соответствует тому пониманию файла, к которому мы привыкли, работая в Windows – это обозначенная некоторым именем последовательность данных, которые хранятся на диске (устройстве хранения данных).

Каталог (*директория*, *directory*) формально тоже считается файлом, содержащим данные о файлах, хранящихся в нем. По смыслу понятие каталога аналогично понятию каталог в Windows.

Символьная или *мягкая ссылка* (*символическая ссылка*, *symbolic link*, *simlink*) во многом похожа на ярлык Windows. Но предназначена она не для быстрого запуска программ, а для исключения дублирования и более оптимального использования места на диске. Часто бывает ситуация, когда один и тот же файл требуется записать в несколько папок на диске одновременно, например, для того, чтобы он был доступен для различных программ. В этом случае нам и пригодятся символьные ссылки. Файл мы запишем только в одном месте, а во всех остальных создадим символьные

ссылки на него, которые будут содержать имя исходного файла и автоматически выполнят перенаправление операций ввода-вывода на исходный файл. Кроме символьных существуют еще жесткие ссылки, не являющиеся отдельным типом файлов. Понятие "*жесткая ссылка*" соответствует ситуация, когда несколько файлов ссылаются на одно и то же место на диске. У жестких ссылок есть одно ограничение по сравнению с символьными ссылками – они могут ссылаться на файл, находящийся только в том же разделе диска, что и сама ссылка. Символьные ссылки не имеют такого ограничения.

Файл устройства представляет собой способ обращения к различным устройствам с помощью файловых операций. Файлы устройств отображаются в каталоге `/dev` дерева каталогов Linux. Существуют два типа устройств – *символьные* (*character*) и *блочные* (*block*), работа с которыми происходит по-разному. Файл устройства не обязательно соответствует какому-либо реальному устройству. Он может соответствовать и виртуальному устройству, реализуемому программно. Примером такого устройства служит `/dev/random`, которое генерирует псевдослучайное число при обращении к нему. Файл устройства представляет собой интерфейс для взаимодействия с драйвером устройства.

Канал (*pipe*) представляет собой специальный механизм для обмена данными между процессами (исполняющимися программами), которые выполняются на одной машине. Видом файлов являются так называемые *именованные каналы* (*named pipes*), которые отображаются в файловой системе для того, чтобы процесс мог обратиться к нужному каналу.

Сокет (*socket*) похож на канал в том смысле, что тоже является механизмом для обмена данными между двумя разными процессами. Однако сокет имеет два важных отличия от канала – процессы чаще всего запущены на разных компьютерах, а обмен данными осуществляется по сети с использованием стека протоколов TCP/IP.

Дерево каталогов. Монтирование файловых систем

Одной из особенностей файловой системы Linux является отсутствие на уровне пользователя такого объекта как "диск". В Linux имеется единое дерево каталогов, вид которого определяется следующим образом. Основу дерева каталогов составляет корневое устройство (аналог диска `C:` в

Windows). Все остальные устройства отображаются в одном из каталогов корневой файловой системы. Операция, заключающаяся во включении файловой системы, находящейся на устройстве, в общее дерево каталогов называется *монтированием* устройства. Каталог, в котором будет отображаться содержимое монтируемого устройства, называется *точкой монтирования*. После завершения работы с некоторыми устройствами, например с флеш-дисками нужно выполнить операцию, противоположную монтированию – *демонтирование*.

Монтирование может выполняться вручную администратором системы с помощью команды `mount` или выполняться автоматически при загрузке системы на основе файла настроек `/etc/fstab`.

Рассмотрим пример. Пусть у нас с вами есть компьютер с двумя жесткими дисками, первый из которых разбит на два раздела (см. рис. 3.1. и 3.2.).

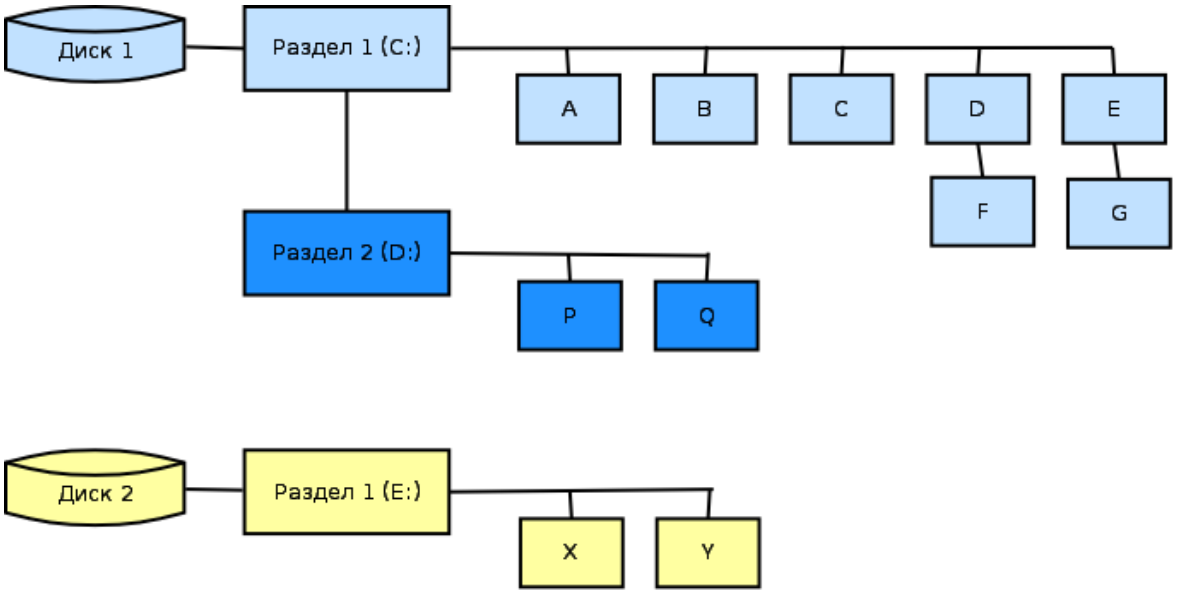

Рис. 3.1. Пример структуры файлов и каталогов в Windows

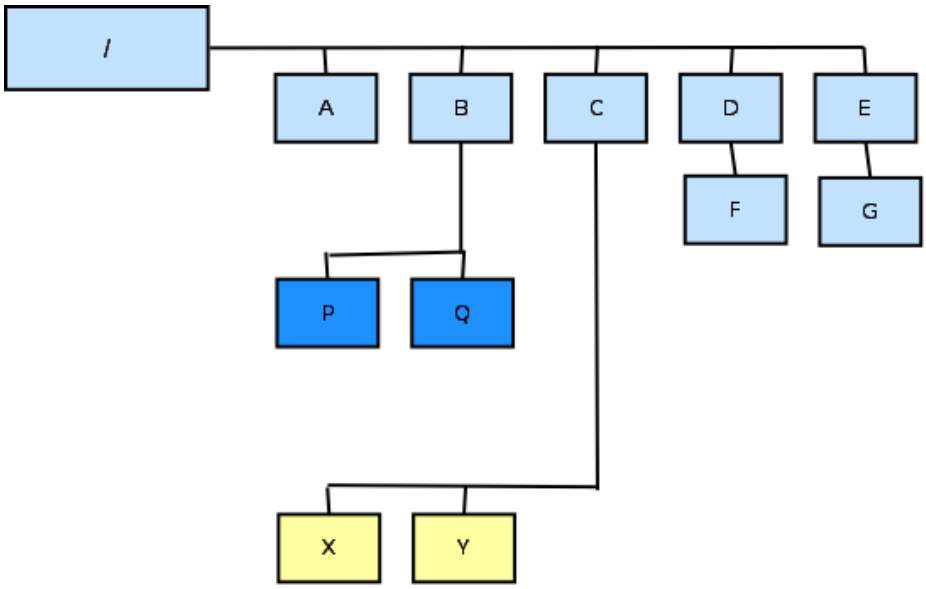

Рис. 3.2. Та же структура файлов и каталогов в Linux

В файловой системе Linux первый раздел первого диска (он обычно называется `hda1` или `sda1`) будет корневой файловой системой. Все остальные разделы – монтируются в папки корневого раздела. В примере второй раздел первого диска (он обычно называется `hda2` или `sda2`) смонтирован в каталог `/B`, а второй диск с единственным разделом (он обычно называется `hdb1` или `sdb1`) смонтирован в каталог `/C`. Полный путь к папке `X` будет "`/C/X`".

В рассмотренном примере устройства монтировались в некоторые абстрактные каталоги `B` и `C`. На самом деле расположение точек монтирования в Linux достаточно регламентировано. Это же справедливо и для имен и назначения многих других каталогов. Общая структура дерева каталогов в Linux схожа на разных компьютерах. Рассмотрим назначение основных каталогов в Linux.

Родительским для всех остальных каталогов является так называемый корневой каталог, обозначаемый знаком дроби "`/`".

Каталог `/bin` содержит основные системные утилиты, необходимые в однопользовательском режиме, а так же при обычной работе всем пользователям.

Каталог `/boot` содержит файлы загрузчика, например, образ ядра Linux.

Каталог `/dev` содержит файлы устройств.

Каталог /etc содержит общесистемные конфигурационные файлы и конфигурационные файлы программ.

Каталог /home содержит домашние каталоги пользователей. При обычной работе чаще всего используется этот каталог.

Каталог /lib содержит файлы общесистемных разделяемых библиотек и модулей ядра.

Каталог /media содержит точки монтирования для сменных носителей, таких как CD и DVD-диски, flash-накопители и т.п.

Каталог /mnt содержит точки монтирования для временных разделов.

Каталог /opt содержит дополнительные пакеты приложений.

Каталог /proc содержит виртуальную файловую систему proc, которая отображает в виде структуры файлов и каталогов различную информацию о работе системы.

Каталог /root является домашним каталогом пользователя root.

Каталог /sbin содержит основные системные программы для администрирования и настройки системы.

Каталог /sys является точкой монтирования еще одной виртуальной файловой системы sys, являющейся расширением proc.

Каталог /usr содержит большинство пользовательских приложений и утилит, использующихся в многопользовательском режиме.

Каталог /var содержит изменяемые файлы, такие как файлы регистрации, временные файлы и т.п.

Шаблоны имен файлов

При задании имени файла в Linux можно использовать *шаблоны имен файлов*. Они задаются с помощью специальных символов. Самыми простыми спецсимволами в шаблонах являются символы “*” и “?”.

Символ “*” заменяет несколько любых символов, в том числе и ни одного. Так шаблону “*” соответствует любое имя файла, а шаблону “a*b” соответствуют имена “ab”, “a1b”, “ahtb”, “a_file_b” и т.п.

Символ “?” в шаблоне заменяет один любой символ в имени файла. Например, шаблону “a?b” соответствуют имена файлов “aab”, “abb”, “axb”, “a8b” и т.п.

Если мы хотим, чтобы на определенном месте в имени файла стоял не любой символ, а символ из некоторого набора, то нужно использовать спецсимволы – квадратные скобки. Такому шаблону соответствует любое имя файла, в котором на месте квадратных скобок стоит любой символ, указанный в шаблоне внутри скобок. Например, шаблону “`a[abc]b`” соответствуют имена файлов “`aab`”, “`abb`” и “`acb`”. В квадратных скобках можно не только перечислять все возможные символы, но и указать их диапазон. Например, шаблон “`file[0-9]`” определяет файлы, имя которых начинается со слова “`file`”, за которым идет цифра.

Наконец, если после открывающей скобки стоит символ “^” или “!”, то в квадратных скобках далее указываются символы, которые **не** могут встречаться в имени файла. Например, шаблон “`file[^0-9]`” определяет файлы, имя которых начинается со слова “`file`”, за которым идет не цифра.

Удобным инструментом для изучения шаблонов имен файлов является команда `echo`, выводящая на экран значения переданных ей параметров. Если параметр команды будет содержать шаблон имен файлов, то перед выводом на экран будет выполнено раскрытие данного шаблона. Ниже приведен пример использования команды `echo` совместно с шаблонами.

```
$ echo ca*
case_op.sh cash.sh
$ echo for?.sh
for1.sh for2.sh for3.sh for4.sh
$ echo for[12].sh
for1.sh for2.sh
$ echo for[^12].sh
for3.sh for4.sh
```

Кроме шаблонов в именах файлов существуют так же специальные символы для обозначения каталогов. Так символ “`.`” (точка) означает текущий каталог, символы “`..`” (две точки) означают каталог, являющийся родительским по отношению к текущему каталогу, а символ “~” (тильда) означает домашний каталог текущего пользователя.

С символом “`.`” связано одно **исключение** из шаблонов. Дело в том, что в Linux *скрытые файлы* имеют имена, начинающиеся с точки. Поэтому

имена файлов, начинающиеся с точки, считаются соответствующими шаблону в том случае, если символ точки указан в шаблоне явно.

Команда ls

Первой командой, которую изучают начинающие пользователи Linux, чаще всего является команда `ls`. Она выводит на экран содержимое заданного каталога и имеет формат:

```
$ ls [флаги] [каталог]
```

Если каталог не указан, то отображается содержимое текущего каталога. Примеры использования команды:

```
$ ls /home/user/Prog
file.c fileout
$ ls
inst.txt ls.txt Prog
```

Команда содержит множество ключей, которые определяют:

- в каком формате выводить информацию,
- какую именно информацию об этих объектах нужно вывести,
- как упорядочить выводимую информацию.

За то, каким образом выводить содержимое каталога отвечают опции `-1`, `-m`, `-x`, `-C` и `-l`.

Если в команде `ls` указан ключ `-1` (цифра один), то имена файлов выдаются по одному в строке.

Если указан ключ `-m`, то файлы выдаются в компактном виде через запятую.

Ключ `-x` задает формат вывода в несколько колонок с сортировкой по горизонтали.

Ключ `-C` задает формат вывода в несколько колонок с сортировкой по вертикали.

```
$ ls
example2.txt  example.o    file100.txt  file20.txt  fileb.txt  textfile2
```

```
example.c     example.out  file1.txt    file.a.txt  textfile
$ ls -1
example2.txt
example.c
example.o
example.out
file100.txt
file1.txt
file20.txt
file.a.txt
fileb.txt
textfile
textfile2
$ ls -m
example2.txt, example.c, example.o, example.out, file100.txt, file1.txt,
file20.txt, file.a.txt, fileb.txt, textfile, textfile2
$ ls -x
example2.txt  example.c   example.o  example.out  file100.txt  file1.txt
file20.txt    file.a.txt  fileb.txt  textfile     textfile2
$ ls -C
example2.txt  example.o    file100.txt  file20.txt  fileb.txt  textfile2
example.c     example.out  file1.txt    file.a.txt  textfile
```

Следующая группа ключей отвечает за то, какая именно информация будет выводиться на экран. По умолчанию выводятся только имена файлов. Наверно самым полезным ключом команды `ls` является ключ `-l` (сокр. от *long format*). Если его указать в команде, то будет выведена *расширенная информация* о файлах, содержащая кроме имени файла, еще его права доступа, тип, дату и время создания, имя владельца, имя группы.

```
$ ls -l
итого 44
-rw-r--r-- 1 user user 22 Янв 26 14:26 example2.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.c
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.o
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.out
-rw-r--r-- 1 user user 22 Янв 26 14:25 file100.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file1.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file20.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file.a.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 fileb.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 textfile
-rw-r--r-- 1 user user 22 Янв 26 14:26 textfile2
```

Еще одним важным ключом является ключ `-a` (сокр. от *all*). Если он указан, то в список файлов, о которых выводится информация, включаются скрытые файлы (начинающиеся с точки). Довольно часто команду `ls` вызывают с ключами `-l` и `-a`.

```
$ ls -al
итого 52
drwxr-xr-x 2 user user 4096 Янв 26 14:29 .
drwxr-xr-x 3 user user 4096 Янв 26 14:24 ..
-rw-r--r-- 1 user user   22 Янв 26 14:26 example2.txt
-rw-r--r-- 1 user user   22 Янв 26 14:25 example.c
-rw-r--r-- 1 user user   22 Янв 26 14:25 example.o
-rw-r--r-- 1 user user   22 Янв 26 14:25 example.out
-rw-r--r-- 1 user user   22 Янв 26 14:25 file100.txt
-rw-r--r-- 1 user user   22 Янв 26 14:25 file1.txt
-rw-r--r-- 1 user user   22 Янв 26 14:25 file20.txt
-rw-r--r-- 1 user user   22 Янв 26 14:25 file.a.txt
-rw-r--r-- 1 user user   22 Янв 26 14:25 fileb.txt
-rw-r--r-- 1 user user   22 Янв 26 14:25 textfile
-rw-r--r-- 1 user user   22 Янв 26 14:26 textfile2
```

Еще одним важным ключом команды `ls` является ключ `-R` (сокр. от *recursive*). Если он указан, то выводится содержимое не только для данного каталога, но и для всех его подкаталогов.

Третья группа ключей отвечает за сортировку списка выводимых файлов. Это ключи `-f`, `-t`, `-v`, `-S`, `-X`, `-r`. По умолчанию файлы упорядочиваются по алфавиту.

Если в команде указан ключ `-f`, то файлы выводятся в том порядке, в котором они записаны на диск. Никакая сортировка не выполняется.

Если в команде указан ключ `-t` (сокр. от *time*), то файлы сортируются по времени последнего изменения.

```
$ ls -lt
итого 44
-rw-r--r-- 1 user user 22 Янв 26 14:26 textfile2
-rw-r--r-- 1 user user 22 Янв 26 14:26 example2.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.c
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.o
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.out
```

```
-rw-r--r-- 1 user user 22 Янв 26 14:25 fileb.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file.a.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file100.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file20.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file1.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 textfile
```

Если в команде указан ключ `-v` (сокр. от *version*), то файлы сортируются по номеру версии, который должен содержаться в имени файла. Номера версий сравниваются как числа.

```
$ ls -lv
итого 44
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.c
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.o
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.out
-rw-r--r-- 1 user user 22 Янв 26 14:26 example2.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file.a.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file1.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file20.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file100.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 fileb.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 textfile
-rw-r--r-- 1 user user 22 Янв 26 14:26 textfile2
```

Если в команде указан ключ `-S` (сокр. от *size*), то файлы сортируются по размеру файла. При этом самые большие файлы окажутся в начале списка.

Если в команде указан ключ `-X`, то файлы сортируются по расширениям имени файла (части имени файла после последней точки). Файлы без расширения оказываются в начале списка.

```
$ ls -lX
итого 44
-rw-r--r-- 1 user user 22 Янв 26 14:25 textfile
-rw-r--r-- 1 user user 22 Янв 26 14:26 textfile2
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.c
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.o
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.out
-rw-r--r-- 1 user user 22 Янв 26 14:26 example2.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file100.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file1.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file20.txt
```

```
-rw-r--r-- 1 user user 22 Янв 26 14:25 file.a.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 fileb.txt
```

Наконец ключ `-r` (сокр. от *reverse*) меняет порядок сортировки на обратный. Данный ключ может объединяться с одним из предыдущих ключей.

```
$ ls -lr
итого 44
-rw-r--r-- 1 user user 22 Янв 26 14:26 textfile2
-rw-r--r-- 1 user user 22 Янв 26 14:25 textfile
-rw-r--r-- 1 user user 22 Янв 26 14:25 fileb.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file.a.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file20.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file1.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 file100.txt
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.out
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.o
-rw-r--r-- 1 user user 22 Янв 26 14:25 example.c
-rw-r--r-- 1 user user 22 Янв 26 14:26 example2.txt
```

Команды cd и pwd

Команда `cd` изменяет текущий каталог (сокр. от *change directory*). Формат команды:

```
$ cd [каталог]
```

Если необязательный параметр `каталог` отсутствует, то выполняется переход в домашний каталог текущего пользователя. Например, для пользователя `user` будет выполнен переход в каталог `/home/user`.

Примеры:

`$ cd /usr` - перейти в каталог `/usr`,

`$ cd ..` - перейти в родительский каталог (находящийся в иерархии каталогов на один уровень выше),

`$ cd` - перейти в рабочий каталог пользователя.

`$ cd –` - перейти в предыдущий каталог.

Вывести на экран текущий каталог можно с помощью команды `pwd` (сокр. от *print working directory*). Эта команда не имеет параметров.

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

Команды mkdir и rmdir

Для создания одного или нескольких новых каталогов используется команда `mkdir`, имеющая формат:

```
$ mkdir [ключи] имя_каталога …
```

Например, следующая команда создает в текущем каталоге подкаталоги `dir_a`, `dir_b`, `dir_c`:

```
$ mkdir dir_a dir_b dir_c
```

Используя ключ `–m` можно при создании каталога сразу задать ему права доступа, которые подробно рассматриваются в конце данной главы. Так, следующая команда создает каталог `dir1` и устанавливает к нему полный доступ для всех категорий пользователей:

```
$ mkdir -m 777 dir1
```

Если в команде `mkdir` задать ключ `–p`, то можно сразу создать цепочку вложенных друг в друга подкаталогов (недостающие родительские каталоги будут создаваться автоматически). Например:

```
$ mkdir -p dir1/dir2/dir3
```

Для удаления каталогов предназначена команда `rmdir`. Она удаляет один или несколько пустых каталогов. Формат команды:

```
$ rmdir [ключи] каталог …
```

Например, следующая команда удаляет в текущем каталоге пустые подкаталоги `dir1` и `dir2`:

```
$ rmdir dir1 dir2
```

Если задать в команде ключ `-p`, то она позволяет удалять вложенные друг в друга пустые подкаталоги. Например, следующая команда удалит целую иерархию указанных в ней пустых каталогов:

```
$ rmdir -p ./dir1/dir2/dir3
```

Команда cat

Еще одной полезной командой является команда `cat`. Несмотря на свое название, полученное от англ. *catenate* – объединить, она позволяет не только объединять файлы. Формат команды:

```
$ cat [ключи] имя_файла …
```

Команду `cat` можно использовать для:

1. Вывода файла на экран

```
$ cat file
```

2. Ввода файла с клавиатуры. В конце ввода нужно нажать `Ctrl+D`.

```
$ cat > file
```

3. Объединения нескольких файлов в один.

```
$ cat file1 file2 file3 > file123
```

Например:

```
$ cat file1
This is file1
$ cat file2
This is file2
$ cat file1 file2 > file3
$ cat file3
This is file1
This is file2
```

Команды mv и cp

Переименование или перемещение файла выполняется командой `mv`, которая может быть задана в одном из двух форматов:

```
$ mv [флаги] исходный_файл файл_назначения
$ mv [флаги] исходный_файл… каталог_назначения
```

Если последний параметр команды является именем каталога (второй вариант формата команды), то команда `mv` перемещает указанные исходные файлы в этот каталог.

В противном случае если задано только два файла (формат 1) `исходный_файл` будет переименован в `файл_назначения`. Если `файл_назначения` существует к моменту запуска команды, то он удаляется.

Если `исходный_файл` и `файл_назначения` находятся в одной файловой системе, то `исходный_файл` не переписывается, а изменяются только ссылки в каталогах.

Если `исходный_файл` и `файл_назначения` находятся в разных файловых системах, то `исходный_файл` сначала копируется, а потом удаляется.

Примеры использования команды:

Переименовать `file1.txt` в `file1.c`.

```
$ mv file1.txt file1.c
```

Переместить файлы `file1.txt`, `file2.txt` и `file3.c` в каталог `dir`.

```
$ mv file1.txt file2.txt file3.c dir
```

Команда `cp` позволяет выполнить копирование указанного файла под новым именем или копировать несколько файлов под старыми именами в указанный каталог. Команда может быть задана в одном из двух форматов:

```
$ cp [флаги] файл1 файл2
$ cp [флаги] файл… каталог_назначения
```

Если последним параметром команды является имя каталога, то в данный каталог копируются указанные в команде файлы с сохранением их имен.

В противном случае, если в команде указаны два файла, то `файл1` копируется под именем `файл2`.

Примеры использования команды:

Копирование файла `file5.txt` под новым именем `file5.bak` в текущем каталоге:

```
$ cp file5.txt file5.bak
```

Копирование всех файлов из каталога `oldlog` в каталог `archive`.

```
$ cp oldlog/* archive
```

Если в команде задать ключ `-l`, то вместо копирования файлов будут созданы их жесткие ссылки (это распространяется только на обычные файлы, а не на каталоги).

Если же задать ключ `-s`, то вместо копирования файлов будут созданы их символьные ссылки.

Полезным ключом команды является ключ `-r` (сокр. от *recursive*). Он активизирует рекурсивный режим копирования, когда вместе с каждым копируемым каталогом копируются вложенные в него файлы и подкаталоги.

Например, следующая команда выполняет копирование каталога `dir1` и всех содержащихся в нем файлов и подкаталогов в каталог `target`:

```
$ cp -r dir1 target
```

Команда rm

Команда `rm` позволяет удалить указанные файлы или каталоги. Она имеет формат:

```
$ rm [флаги] файл…
```

В ОС Linux удаленный файл восстановить чаще всего **невозможно**, а иногда (если очень повезет) – крайне затруднительно! Поэтому перед удалением, нужно быть абсолютно уверенным, что этот файл нужно удалить.

Например, следующая команда удаляет файл `file.c`:

```
$ rm file.c
```

Достаточно мощным, но в то же время опасным ключом команды является ключ `-r`, включающий режим рекурсивного удаления каталогов, позволяющий удалять каталоги вместе с вложенными в них файлами и подкаталогами. Например, следующая команда удаляет каталог `prog` и все его подкаталоги:

```
$ rm -r prog
```

Особенно **опасной** является команда “`rm -rf .*`”. Если ее запустить с правами пользователя `root`, то она удалит все файлы на диске без возможности восстановления (ключ -f указывает на то, что не нужно запрашивать подтверждение пользователя при удалении файлов). Это происходит из-за особенностей функционирования шаблона “`.*`”. На прямом или замаскированном использовании этой команды основано несколько распространенных розыгрышей неопытных пользователей Linux (если случайное удаление всех файлов на диске можно назвать розыгрышем).

Никогда не запускайте эту команду на своем компьютере!

Еще одним достаточно важным ключом команды `rm` является ключ `-i`, который активизирует так называемый интерактивный режим работы команды. В этом режиме перед удалением каждого из файлов пользователю выдается запрос на подтверждение удаления данного файла. Если ответ пользователя не был утвердительным, то файл не удаляется.

Работа со ссылками

Для создания ссылок на файлы в Linux предназначена команда `ln`. Она может создавать как жесткие, так и символьные ссылки. Формат команды:

```
$ ln [ключи] имя_файла [имя_ссылки]
$ ln [ключи] имя_файла … каталог
```

Первый вариант команды создает (по-умолчанию - в текущем каталоге) ссылку с именем `ссылка` на заданный файл с именем `имя_файла`.

Второй вариант команды создает в заданном `каталоге` ссылки на указанные файлы, причем имена файлов ссылок будут совпадать с именами исходных файлов.

По-умолчанию создается жесткая ссылка. Если требуется создать символьную ссылку, то в команде нужно указать ключ `-s`.

Рассмотрим следующий пример. В нем для файла `file1.txt` сначала создается жесткая ссылка `file_hard`, а потом создается символьная ссылка `file_soft`.

```
$ ls -l
итого 4
-rw-r--r-- 1 user user 132 Map 23 14:39 file1.txt
$ cat file1.txt
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
$ ln file1.txt file_hard
$ ls -l
итого 8
-rw-r--r-- 2 user user 132 Map 23 14:39 file1.txt
-rw-r--r-- 2 user user 132 Map 23 14:39 file_hard
```

```
$ cat file_hard
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
$ ln -s file1.txt file_soft
$ cat file_soft
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
$ ls -l
итого 8
-rw-r--r-- 2 user user 132 Map 23 14:39 file1.txt
-rw-r--r-- 2 user user 132 Map 23 14:39 file_hard
lrwxrwxrwx 1 user user   9 Map 23 14:40 file_soft -> file1.txt
```

В примере видно, что работать с файлом мы можем, используя любую его ссылку. Причем жесткая ссылка практически ничем не отличается от обычного файла, а символьная является отдельным типом файла, указанным в выводе списка файлов первой буквой “`l`”.

Понятия владелец файла и права доступа к файлу

В Linux каждый файл имеет своего **владельца**. Им обычно является пользователь, создавший этот файл. Однако владелец файла может быть изменен администратором системы, поэтому предыдущее правило не является универсальным. Владельца файла можно посмотреть в расширенном листинге команды `ls`. Каждый файл в Linux имеет только одного владельца.

Владелец файла определяет **права доступа** к файлу. Для каждого файла могут быть заданы: право на чтение, право на запись и право на выполнение. Причем указанные права задаются отдельно для трех категорий пользователей:

1. для владельца файла,
2. для пользователей, входящих в так называемую группу владельца (это ограниченный круг доверенных пользователей владельца, которых он сам определяет, включая в группу пользователей, имя которой совпадает с именем владельца),
3. для всех остальных пользователей.

Обычно владелец файла имеет максимальные по сравнению с другими пользователями права, а остальные пользователи имеют минимальные права доступа.

Основные права доступа часто задают в виде трех восьмеричных цифр, например `764`. Первая цифра (`7`) определяет права владельца файла, вторая (`6`) – права группы владельца, а третья – права остальных пользователей. Каждая восьмеричная цифра прав складывается из трех двоичных разрядов – права на исполнение (в разряде единиц), права на запись (в разряде двоек) и права на чтение (в разряде четверок).

Таким образом, рассмотренные права `764` означают полный доступ (чтение, запись и выполнение) для владельца файла, право на чтение и запись для группы владельца и право на чтение для остальных пользователей.

Для каталогов права доступа носят особый смысл: право на чтение позволяет просматривать содержимое каталога, право на запись позволяет удалять или перемещать файлы в каталоге, а право на выполнение вместе с правом на чтение – осуществлять поиск в каталоге.

Кроме перечисленных есть еще три **дополнительных бита прав доступа**. Это так называемые **SUID-**, **SGID-** и **sticky**-биты. Они хранятся в старшей тройке битов прав доступа.

Биты SUID и SGID могут назначаться исполняемым файлам (программам). Часто для нормальной работы программа требует повышенных прав доступа, например для обращения к системным ресурсам. В то же время давать всем пользователям программы повышенные права не желательно. Преодолеть это противоречие можно с помощью битов SUID и SGID. Если установлен бит SUID (сокр. от англ. Set User ID), то программа при запуске получит не права запустившего ее пользователя, а права владельца программы. Если же установлен бит SGID (сокр. от англ. Set Group ID), то программа будет запущена с правами группы владельца.

Бит sticky первоначально создавался для того, чтобы пометить часто используемые программы. Такие программы не выгружались из памяти после завершения их работы, и следующий их запуск происходил быстрее.

В настоящее время sticky-бит потерял свое первоначальное значение. Он имеет следующий смысл: в каталоге, для которого установлен sticky-бит удалять файлы могут только владелец файла и суперпользователь.

В цифровом виде SUID имеет значение `4` в старшей тройке битов, SGID – `2`, sticky – `1`. Например, права `4744` означают полный доступ для

владельца, доступ только для чтения для группы владельца и остальных пользователей и установленный SUID-бит.

Установка прав доступа к файлу и команда chmod

Для изменения прав доступа к файлу предназначена команда `chmod`. Формат команды:

```
$ chmod [ключи] режим файл …
```

Среди ключей команды достаточно полезным является `-R`, активизирующий рекурсивный режим работы команды – права устанавливаются не только для родительского каталога, но и для его файлов и подкаталогов.

Основной интерес в команде представляет способ задания параметра режим. Он может задаваться в двух форматах – числовом и символьном.

Права в **числовом формате** задаются так же, как мы рассматривали в предыдущем разделе.

Например, следующая команда устанавливает полные права доступа к файлам `file1.txt`, `file2.txt` и `prog.c` для всех категорий пользователей:

```
$ chmod 777 file1.txt file2.txt prog.c
```

Вторая команда устанавливает файлу `report.txt` полные права для владельца файла, права на чтение и запись для группы владельца и право на чтение для остальных пользователей и, кроме этого, устанавливает SUID - бит:

```
$ chmod 4764 report.txt
```

В целом указание прав в числовом виде удобнее, когда нужно задать сразу все права доступа. Если же нужно изменить один или несколько битов в существующих правах – удобнее задавать права в **символьном формате**.

Символьная форма описания режима прав доступа задается в следующем формате:

```
[ugoa] [+-=] [rwxstugo]
```

Первая часть строки режима определяет, для кого изменяются права доступа – для владельца (u), группы владельца (g), остальных пользователей (o) или всех пользователей (a).

Вторая часть строки режима определяет, как изменяются права доступа – они добавляются к уже имеющимся (+), устанавливаются заново (=) или удаляют часть имеющихся прав (-).

Третья часть строки режима задает новые права доступа для пользователя, заданного первой строкой: r – чтение, w – запись, x – выполнение, s – SUID- или SGID-биты, t – sticky-бит.

Примеры задания прав доступа в символьном формате:

Установка файлу f1 прав доступа на чтение и запись для всех пользователей:

```
$ chmod a=rw f1
```

Установка для файлов f1 и f2 полных прав для владельца, прав на чтение и выполнение для группы владельца и права на чтение для остальных пользователей.

```
$ chmod u=rwx,g=rx,o=r f1 f2
```

Добавление группе владельца файла f1 права на запись данного файла.

```
$ chmod g+w f1
```

Установка SUID-бита для программы

```
$ chmod u+s prog
```

Установка SGID-бита для программы

```
$ chmod g+s prog
```

Глава 4. Команды Linux для работы с процессами

Понятие процесса

Процесс является одним из фундаментальных понятий Linux. Несколько упрощенно процесс можно определить как **программу, находящуюся в состоянии исполнения**.

Кроме, исполняемого кода каждый процесс содержит **собственное адресное пространство**, в котором содержатся **данные** процесса, а так же набор **ресурсов**, например, таких как **открытые им файлы**.

Linux, как и практически все современные операционные системы, является **многозадачной** операционной системой, т.е. позволяет выполнять много процессов одновременно. В реальности количество одновременно выполняющихся процессов ограничено числом ядер центрального процессора. Операционная система позволяет имитировать запуск большего числа процессов, выделяя им процессорное время по очереди небольшими частями – т.н. **квантами**.

При этом никакого гарантированного порядка в очередности выполнения процессов нет. Планированием очередности выполнения процессов занимается часть ядра, называемая планировщиком. В Linux возможно использование одного из нескольких существующих планировщиков – CFS, BFS, O(1), и все они используют разные принципы работы [3, 4].

Естественно одна и та же программа может одновременно исполнятся в нескольких процессах. Кроме этого, несколько процессов могут разделять одни и те же ресурсы, например, открытые файлы или данные.

Каждый процесс может породить дополнительные процессы. При этом процесс, запустивший новый процесс, называется **родительским**, а новый процесс по отношению к создавшему его процессу называется **дочерним**. Процессы, порожденные одним и тем же родительским процессом, называются **братьями** (или **сестрами**). Такое подробное рассмотрение “родственных” связей между процессами вызвано тем, что такие процессы, как правило, разделяют одни и те же ресурсы, в частности – открытые файлы и каналы.

В Linux реализована четкая **иерархия** процессов в системе. Каждый процесс в системе имеет всего одного родителя и может иметь один или более порожденных процессов.

При загрузке первым запускающимся после запуска ядра процессом является процесс `init`, выполняющий загрузку других процессов в соответствии со скриптом, расположенным в файле `/etc/inittab`. Процесс `init` имеет идентификатор, равный 1. От него порождаются все остальные процессы. Процесс `init` никогда не завершается.

Атрибуты процесса. Состояния процесса

Каждый процесс в ОС Linux характеризуется следующим набором атрибутов:

Идентификатор процесса (**P**rocess **ID**entifier - **PID**). Каждый процесс в системе имеет уникальный числовой идентификатор. Обычно процессы, запущенные позднее, имеют большее значение PID. Значение PID используется в ряде команд для задания процесса.

Идентификатор родительского процесса (**P**arent **PID** - **PPID**). Данный атрибут процесс получает во время своего запуска. Он используется для получения статуса родительского процесса.

Реальный и эффективный идентификаторы пользователя (**UID**, **EUID**) и **группы** (**GID**, **EGID**). Данные атрибуты процесса говорят о его принадлежности к конкретному пользователю и группе пользователей.

Реальные идентификаторы совпадают с идентификаторами пользователя, который запустил процесс, и группы, к которой он принадлежит.

Эффективные идентификаторы показывают, от имени какого пользователя был запущен процесс. Они могут не совпадать с реальными идентификаторами, если для запускаемой программы установлен бит SGID или SUID.

Права доступа процесса к ресурсам операционной системы определяются *эффективными идентификаторами*.

Для управления процессом используются реальные идентификаторы. Все идентификаторы передаются от родительского процесса к дочернему.

Приоритет (**priority**) и **относительный приоритет** (**уступчивость**) процесса (**nice**). Приоритет определяет количество процессорного времени,

которое выделяется процессу для выполнения. Пользователь может изменять значение только относительного приоритета (`nice`), который является своеобразной поправкой к основному значению приоритета и может изменяться в диапазоне от `-20` до 19. Здесь меньшее значение соответствует более высокому приоритету. Например, если мы зададим значение `nice` равным `20`, то процесс будет выполняться заметно реже остальных процессов, если же зададим значение `-19`, то чаще. Устанавливать отрицательное значение относительного приоритета процесса может только пользователь `root`. Термин nice происходит от англ. *nice – вежливый*, *тактичный*. Процессы с большим значением `nice` "тактично" по отношению к другим процессам выполняются реже. Значение относительного приоритета наследуется дочерними процессами от родительского процесса.

Процесс может в текущий момент находиться в одном из следующих **состояний**:

- *Runable* (R) – процесс выполняется или готов к выполнению,
- *Sleeping* (S) – процесс находится в состоянии ожидания некоторого внешнего события (например, завершения операции ввода-вывода) и не выполняется. При получении сигнала[8] процесс временно прерывает ожидание для обработки сигнала. Поэтому такие процессы можно завершить до окончания операции ввода-вывода.
- *Uninterruptible* (D) – состояние аналогично предыдущему с той разницей, что ожидание не прерывается для обработки сигналов. Такой процесс нельзя завершить до окончания операции ввода-вывода.
- *Stopped* (T) – процесс остановлен. Процесс переходит в это состояние когда получает соответствующий сигнал. Так же в этом состоянии находится процесс, если производится его отладка.
- *Zombie* (Z) – процесс-"зомби". Этот процесс завершил свое выполнение и почти полностью выгружен из памяти. Единственная информация, которая хранится о данном процессе

[8] Сигнал является способом оповещения процесса о системных событиях. Понятие сигнала рассмотрено в данной главе чуть позднее.

– *код его завершения*[9], значение которого может потребоваться родительскому процессу. После запроса кода завершения родительским процессом, процесс-“зомби” полностью удаляется из памяти.

Запуск процесса переднего плана и в фоновом режиме

Когда мы запускаем программы обычным способом, они запускаются как программы **переднего плана** (**foreground**). Это значит, что после запуска такой программы мы видим на терминале данные, которые она выводит, и можем вводить с клавиатуры данные, запрашиваемые программой. Приглашение к вводу команд, выводимое командной оболочкой, появится на экране только после завершения такой команды.

Это удобно, если программа подразумевает интерактивный режим взаимодействия с пользователем. Если же программа выполняет какие-либо операции, не требующие реакции пользователя в течение длительного времени, например, сложные расчеты или операции обмена данными с диском, то имело бы смысл позволить пользователю во время работы таких программ вводить другие команды. Это можно сделать, запустив команду в **фоновом режиме** (**background**). Для запуска команды в фоновом режиме в конце команды нужно указать знак `&`. В этом случае на экране сразу появится приглашение к вводу следующей команды. Так же в фоновом режиме имеет смысл запускать программы с графическим интерфейсом, чтобы можно было параллельно с их работой вводить другие команды в консоли. На рис. 4.1 показан запуск демонстрационной программы `xeyes` в фоновом режиме.

[9] Код завершения – это целое число, которое передается после завершения процесса его родительскому процессу. Как правило, код завершения используется для информирования о возникших ошибках или их отсутствии.

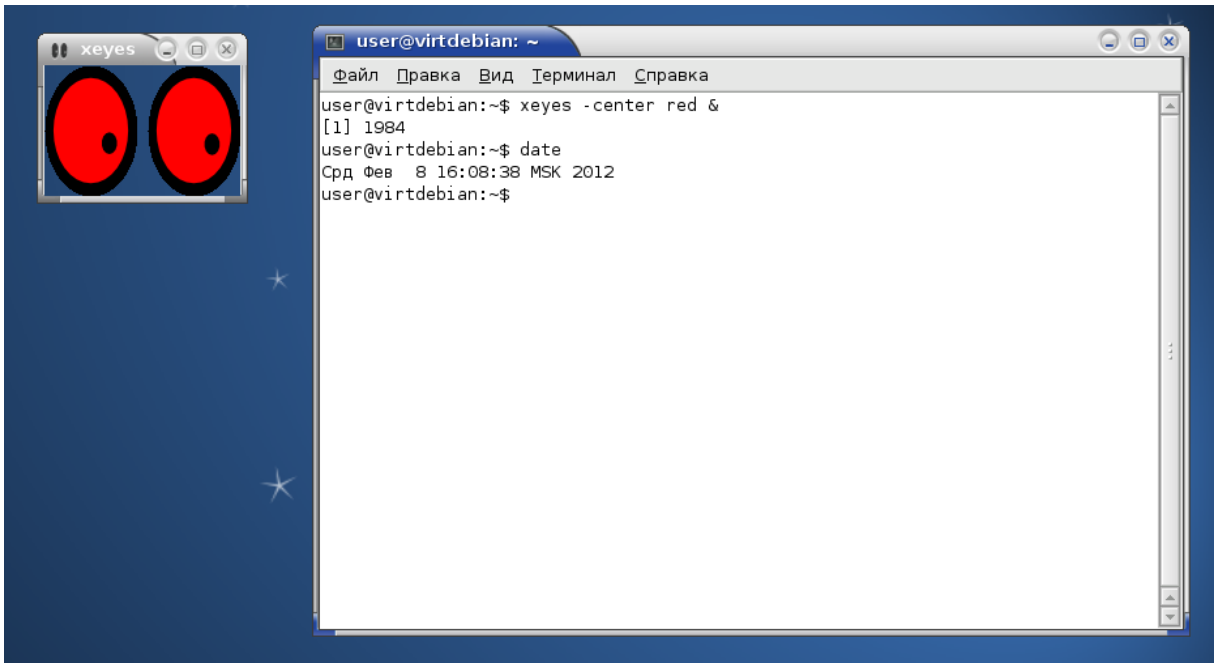

Рис. 4.1. Запуск демонстрационной программы `xeyes` в фоновом режиме

На рисунке видно, что после запуска программы на экране отобразилась надпись “`[1] 1984`”, после которой вывелось приглашение к вводу следующей команды. Здесь `1984` – это идентификатор процесса (PID) программы `xeyes`, а `1` – номер задания (программы, запущенной командной оболочкой в фоновом режиме).

Для просмотра текущих заданий используется команда `jobs`, которая вводится без аргументов. Рассмотрим вывод команды:

```
$ jobs
[1]   Running                 xeyes -center red &
[2]+  Stopped                 sleep 100
[3]-  Running                 gedit &
```

Команда выводит список заданий. Для каждого задания отображаются его номер в квадратных скобках и текущее состояние – `Running` (выполняется) или `Stopped` (Остановлено). Так же два задания помечены знаками “+” и “-”. Задание, которое помечено знаком “+”, является заданием, которому будут адресоваться команды управления заданиями (которые мы рассмотрим далее) по-умолчанию. Знаком “−” помечается задание, которое станет заданием по-умолчанию после того как завершится задание, помеченной знаком “+” в настоящий момент.

При работе с заданиями есть две комбинации клавиш, влияющие на их выполнение. Первая комбинация – это `Ctrl+C`, которая завершает выполнение программы, выполняющейся на переднем плане. Вторая комбинация – `Ctrl+Z`, переводящая программу, выполняющуюся на переднем плане в приостановленное состояние.

Есть еще две специальные команды для работы с заданиями – `bg` и `fg`. Они имеют следующий формат:

```
$ bg [номер_задания]
$ fg [номер_задания]
```

Команда `bg` продолжает выполнение приостановленной задачи в фоновом режиме. Команда `fg` переводит фоновую задачу на передний план (если задача приостановлена – то ее выполнение возобновляется).

Таким образом, если нам нужно перевести задачу с переднего плана в фоновый режим, нужно нажать `Ctrl+Z` и ввести команду `bg`. Если нужно вернуть задачу из фонового режима на передний план – нужно ввести команду `fg`.

Команда ps

Команда `ps` выводит информацию о запущенных процессах. Общий формат команды:

```
$ ps [ключи]
```

Эта команда имеет, пожалуй, самый запутанный синтаксис из всех команд Linux. Она пытается одновременно поддерживать сразу три варианта опций: опции UNIX, опции BSD, опции GNU. Часто ключи команды дублируют друг друга.

Рассмотрим несколько вариантов использования команды `ps`.

Если запустить команду без параметров, то она выведет список выполняющихся процессов, которые запущенны текущим пользователем и ассоциированы с текущим терминалом.

```
$ ps
```

```
  PID TTY          TIME CMD
 2171 pts/0    00:00:00 bash
 2191 pts/0    00:00:00 ps
```

В приведенном выводе команды `ps` содержатся данные о двух процессах. Если вы запустите терминал и наберете команду `ps`, то увидите примерно такой же вывод. По умолчанию выводятся сокращенные данные о процессах, которые содержат всего четыре колонки. Первая колонка, как видно из названия “`PID`” содержит идентификатор процесса. Вторая колонка содержит имя терминала, ассоциированного с данным процессом. Третья колонка “`TIME`” содержит данные о потребляемых процессом ресурсах центрального процессора. Четвертая колонка “`CMD`” содержит команду, с помощью которой был запущен процесс. Как видно из листинга, запущенные процессы, соответствующие оболочке `bash` и команде `ps`, ресурсов процессора практически не потребляют. Это достаточно ожидаемый факт, поскольку обе программы практически не занимаются вычислениями, а в основном ожидают реакции от файловой системы или клавиатуры и не загружают центральный процессор.

Если мы хотим просмотреть список всех запущенных процессов, то нужно задать в команде ключ `-A` или `-e` (в листинге приведены только начало и конец выдачи)

```
$ ps -A
  PID TTY          TIME CMD
    1 ?        00:00:00 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 migration/0
    4 ?        00:00:00 ksoftirqd/0
    5 ?        00:00:00 watchdog/0
...
 2035 ?        00:00:01 gnome-terminal
 2036 ?        00:00:00 gnome-pty-helpe
 2037 pts/0    00:00:00 bash
 2231 ?        00:00:00 flush-8:0
 2239 pts/0    00:00:00 ps
```

Как видно из вывода, большинство процессов не связаны ни с каким терминалом. Это графические приложения и так называемые *процессы-демоны* (запущенные все время, пока запущена операционная система, и выполняющие различные служебные функции).

Более полный вывод можно получить, выполнив команду `ps -Al`:

```
$ ps -Al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 -   509 ?      ?        00:00:00 init
1 S     0     2     0  0  80   0 -     0 ?      ?        00:00:00 kthreadd
1 S     0     3     2  0 -40   - -     0 ?      ?        00:00:00 migration/0
1 S     0     4     2  0  80   0 -     0 ?      ?        00:00:00 ksoftirqd/0
5 S     0     5     2  0 -40   - -     0 ?      ?        00:00:00 watchdog/0
...
0 R  1000  2035     1  0  80   0 - 37866 -       ?          00:00:05 gnome-
terminal
0 S  1000  2036  2035  0  80   0 -   476 ?      ?        00:00:00 gnome-pty-
helpe
0 S  1000  2037  2035  0  80   0 -  1622 -      pts/0    00:00:00 bash
1 S     0  2249     2  0  80   0 -     0 ?      ?        00:00:00 flush-8:0
0 R  1000  2267  2037  0  80   0 -   916 -      pts/0    00:00:00 ps
```

В приведенном листинге для каждого процесса дополнительно выводятся: текущее состояние процесса (`S`), идентификатор пользователя, с которым связан процесс (`UID`), идентификатор родительского процесса (`PPID`), приоритет и относительный приоритет процесса (соответственно `PRI` и `NI`).

Еще более полный вывод дает следующая команда

```
$ ps -AlF
```

Наконец, указав ключ `--forest` мы получим отображение иерархии процессов:

```
$ ps -A --forest
  PID TTY          TIME CMD
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00  \_ migration/0
    4 ?        00:00:00  \_ ksoftirqd/0
    5 ?        00:00:00  \_ watchdog/0
    6 ?        00:00:00  \_ events/0
...
  936 ?        00:00:00 gdm3
```

```
 942 ?        00:00:00  \_ gdm-simple-slav
 949 tty7     00:00:30      \_ Xorg
1648 ?        00:00:00      \_ gdm-session-wor
1763 ?        00:00:00          \_ x-session-manag
1823 ?        00:00:00              \_ ssh-agent
1837 ?        00:00:00              \_ seahorse-agent
1847 ?        00:00:00              \_ gnome-power-man
1855 ?        00:00:00              \_ metacity
...
2035 ?        00:00:08 gnome-terminal
2036 ?        00:00:00  \_ gnome-pty-helpe
2037 pts/0    00:00:00  \_ bash
2287 pts/0    00:00:00      \_ ps
```

Иерархию процессов в более наглядной форме отображает специальная команда `pstree`.

Команда top

В Linux кроме команды `ps` данные о запущенных процессах отображает команда `top`. Данная команда так же выводит данные о загрузки процессора и памяти компьютера. Команда при запуске создает собственную интерактивную среду с возможностью ввода команд, в которой информация периодически обновляется. На рис. 4.2. показан характерный вывод команды `top`.

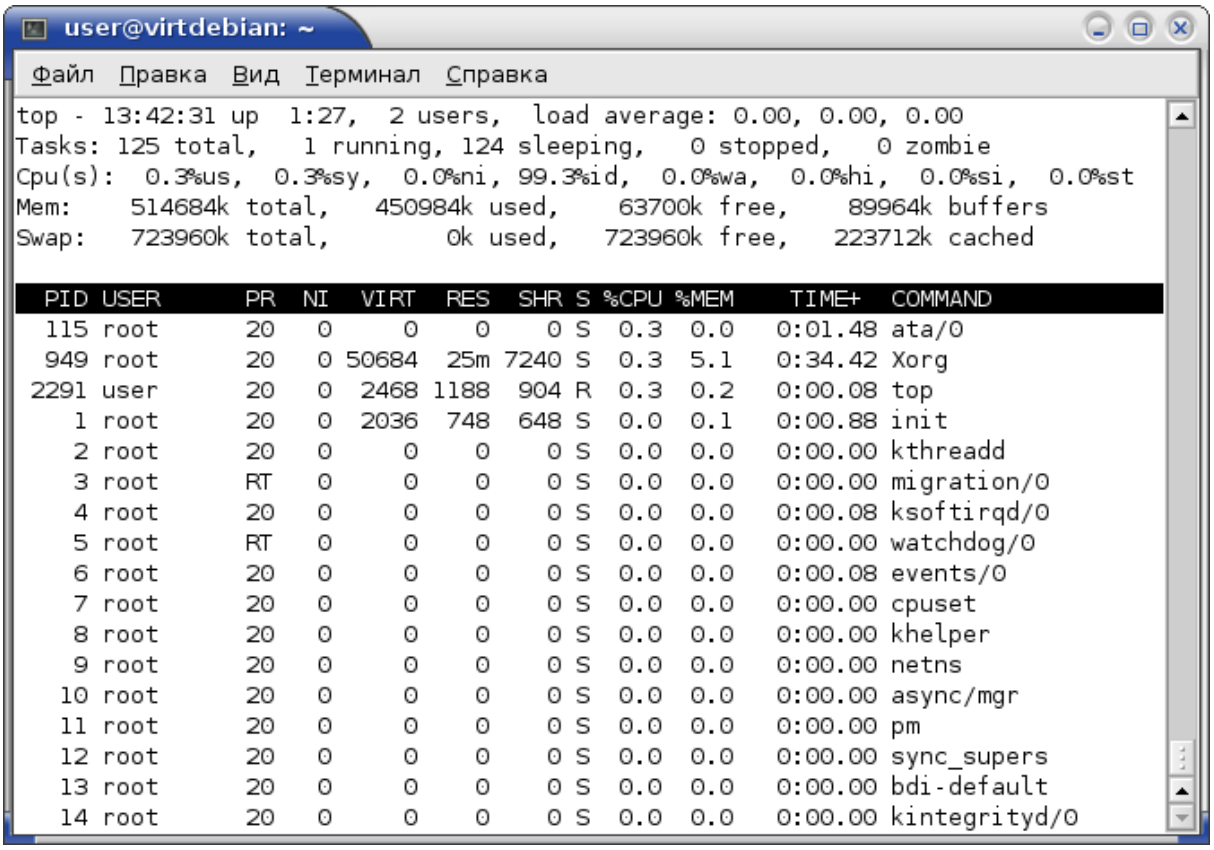

Рис. 4.2. Среда команды `top`

При выполнении утилиты `top` в верхней части окна отображается текущее время, время, прошедшее с момента запуска системы, число пользователей в системе, число запущенных процессов и число процессов, находящихся в разных состояниях, данные об использовании ЦПУ, памяти и раздела подкачки.

Далее идет таблица, характеризующая отдельные процессы. Число строк, отображаемых в этой таблице, определяется размером окна. Список процессов может быть отсортирован по используемому времени ЦПУ (по умолчанию), по использованию памяти, по PID. Для каждого процесса может выводиться следующая основная информация:

- `PID` – идентификатор процесса,
- `USER` – имя пользователя, от имени которого (и с правами которого) запущен процесс,
- `PR` – приоритет процесса,
- `NI` – относительный приоритет процесса,
- `%CPU` – процент используемого процессом процессорного времени,
- `%MEM` – процент физической памяти, используемой процессом,

• VIRT – объем занятой процессом виртуальной памяти (в килобайтах).

• S – состояние процесса.

• COMMAND – имя программы или команды.

В утилите top можно вводить команды с клавиатуры в интерактивном режиме. Наиболее полезные команды:

h – отображение справки по работе с программой,

k – отправка сигнала процессу. При этом требуется дополнительно ввести идентификатор процесса, а затем номер или название отправляемого сигнала (по умолчанию используется TERM или 15),

M – сортировать процессы по объёму занятой ими памяти (поле %MEM). Сверху отображаются процессы, занимающие больше памяти,

P – сортировать процессы по занятому ими времени CPU (поле %CPU; этот метод используется для сортировки по умолчанию). Сверху отображаются процессы, занимающие больше процессорного времени,

u – вывести процессы заданного пользователя, имя которого будет запрошено командой. Необходимо ввести имя пользователя, а не его UID. Если вы не введете никакого имени, будут показаны все процессы;

i – вывести информацию только о работающих в данный момент процессах (по умолчанию выводятся все процессы, даже спящие). Повторное использование этой команды вернёт вас назад к списку всех процессов.

r – эта команда используется для изменения динамического приоритета выбранного процесса. При выполнении команды требуется ввести значение идентификатора процесса и новое значение динамического приоритета процесса,

q – выход из утилиты top.

Команда at

Одной из часто встречаемых задач при администрировании системы является запуск команд в заданное время. Можно, например, запустить в полночь скрипт для резервного копирования данных или скачать с сервера обновления программы. Для запуска программ по расписанию в Linux предназначена системная служба (демон) cron. Интерфейс для работы с этой системной службой реализован командой at. Формат команды:

```
$ at [ключи] время
```

Команда at читает список выполняющихся команд из стандартного потока ввода или из указанного файла. Если команды вводятся пользователем с клавиатуры, то после завершения ввода нужно нажать Ctrl+D.

Введенные команды будут выполнены однократно в указанное время. Время может задаваться в различных форматах. Простейший формат – ЧЧ:ММ, например – 13:15. Так же при задании времени могут использоваться слова noon, midnight и приставки AM и PM для указания времени дня. Кроме времени может указываться дата выполнения команд.

Для того чтобы список выполняемых команд считывался из файла, нужно в команде at указать ключ -f и задать имя файла в качестве параметра.

При использовании команды at следует помнить, что команды будут выполняться демоном cron, и поэтому не будут связаны ни с каким терминалом, и не будут привязаны к графической среде. Не нужно ждать появления каких-либо сообщений на терминале или запуска программ XWindow. Факт запуска команд проще всего отследить по изменениям в файловой системе, как это показано в следующем примере:

```
$ echo 'ls > /home/user/test/attestfile' | at 12:45
warning: commands will be executed using /bin/sh
job 6 at Wed Apr 13 12:45:00 2011
$ date
Срд Апр 13 12:42:19 MSD 2011
$ cat /home/user/test/attestfile
cat: /home/user/test/attestfile: Нет такого файла или
каталога
$ date
Срд Апр 13 12:45:12 MSD 2011
$ cat /home/user/test/attestfile
attestfile
myscr
test2
testtextfile
```

В данном примере видно, что на 12:45 запланирована команда, которая записывает список файлов текущего каталога в файл attestfile.

В 12:42 файл с таким именем еще не существует, а в 12:45 – он уже создан.

Команда nohup

Когда мы выходим из командной оболочки, она посылает всем запущенным из нее программам сигнал на завершение работы. Если же мы хотим, чтобы программа продолжила выполняться после завершения командной оболочки, то запускать такую программу нужно с помощью специальной команды nohup. С помощью данной команды запускаются различные системные службы. Формат команды nohup:

```
$ nohup команда [ключи] &
```

Если стандартный поток ввода команды связан с терминалом, то он перенаправляется на устройство /dev/null, а стандартный вывод, связанный с терминалом, направляется в файл nohup.out, расположенный в домашнем каталоге пользователя.

Команды nice и renice

Ранее мы с вами обсуждали понятие относительного приоритета. С помощью задания относительного приоритета можно повысить или понизить приоритет некоторого процесса относительно других процессов. Относительный приоритет может задаваться целым числом от -20 (повышает приоритет процесса) до +19 (понижает приоритет процесса). Задать относительный приоритет некоторой команды при ее запуске можно с помощью команды nice, имеющей формат:

```
$ nice [ключи] [команда [аргументы_команды]]
```

Основным ключом команды nice является ключ -n, который задает значение относительного приоритета запускаемой команды, по умолчанию используется значение 10. Например:

```
$ nice -n 5 egrep –f pattern_file text_file
```

Команда `renice` позволяет изменить относительный приоритет уже работающего процесса. Формат команды:

```
$ renice отн_приоритет [ключи]
```

Ключи команды задают процессы, для которых изменяется относительный приоритет. Возможно три варианта:

`-p ID …` - процесс задается одним или несколькими идентификаторами процесса,

`-g ID …` - процесс задается одним или несколькими идентификаторами групп пользователей,

`-u username …` - процесс задается одним или несколькими именами пользователей.

Сигналы. Команды kill и killall

В Linux существует механизм оповещения процессов об асинхронных событиях, называемый **сигналами**. События могут быть как внешними по отношению к процессу, например, когда пользователь нажал `Ctrl+C`, так и внутренними, например деление на ноль. Сигналы обрабатываются процессами *асинхронно*, т.е. при получении сигнала операционная система прерывает нормальное выполнение процесса и передает управление обработчику сигнала.

Когда процесс получает сигнал, он должен его обработать одним из трех способов:

- просто игнорировать сигнал,
- захватить и обработать сигнал,
- выполнить действие по-умолчанию.

Передать процессу сигнал может или операционная система, или другая программа, или пользователь с помощью специальной команды.

Описание основных сигналов приведено в табл. 4.1.

Табл. 4.1. Описание основных сигналов Linux

Имя	Номер	Описание	Действие по-

сигнала	сигнала[10]		умолчанию
SIGBUS	7	Аппаратная ошибка или ошибка выравнивания	Завершиться с созданием дампа ядра
SIGCHLD	17	Завершился дочерний процесс	Игнорировать
SIGCONT	18	Процесс продолжил выполняться после того, как был остановлен	Игнорировать
SIGFPE	8	Арифметическое исключение	Завершиться с созданием дампа ядра
SIGHUP	1	Управляющий терминал процесса был закрыт (чаще всего это связано с тем, что пользователь выходит из системы)	Завершиться
SIGILL	4	Процесс попытался выполнить недопустимую функцию	Завершиться с созданием дампа ядра
SIGINT	2	Пользователь ввел символ прерывания (Ctrl+C)	Завершиться
SIGKILL	9	Завершение процесса, которое невозможно захватить	Завершиться
SIGPWR	30	Сбой питания	Завершиться
SIGSEGV	11	Нарушение доступа к памяти	Завершиться с созданием дампа ядра
SIGSTOP	19	Приостановить выполнение процесса	Остановиться
SIGTERM	15	Завершение процесса с возможностью захвата	Завершиться
SIGTSTP	20	Пользователь ввел символ приостановки (Ctrl+Z)	Остановиться

Основной командой, с помощью которой можно отправить сигнал процессу является команда `kill`. Как видно из названия, в большинстве случаев она используется для завершения процесса. Формат команды:

[10] Номера некоторых сигналов могут различаться на разных платформах. Необходимо уточнять номера сигналов с помощью команды man.

```
$ kill [ключи] идентификатор_процесса …
```

Если ключи в команде не заданы, то процессу посылается сигнал TERM (15), который приводит к его корректному завершению.

Для посылки процессу сигнала используется следующий синтаксис:

```
$ kill -s сигнал идентификатор_процесса …
```

Сигнал указывается по имени или по номеру.

Получить список доступных сигналов можно с помощью команды:

```
$ kill -l
```

Пример использования команды:

```
$ xeyes -center red&
[1] 1967
$ ps
  PID TTY          TIME CMD
 1951 pts/0    00:00:00 bash
 1967 pts/0    00:00:00 xeyes
 1968 pts/0    00:00:00 ps
$ kill 1967
$ ps
  PID TTY          TIME CMD
 1951 pts/0    00:00:00 bash
 1969 pts/0    00:00:00 ps
[1]+  Завершено      xeyes -center red
```

Команда kill чаще всего используется для завершения процессов. Процесс может завершаться с помощью сигналов TERM или KILL.

Сигнал TERM выполняет программное завершение процесса, получив его, процесс может выполнить действия, необходимые для корректного завершения – сохранить несохраненные данные, освободить системные ресурсы и т.п. Сигнал TERM является предпочтительным способом

завершения. Это так называемое “мягкое завершение”. Однако процесс может переопределить сигнал TERM и даже его игнорировать.

Если же процесс “завис” и не реагирует на сигнал TERM, то единственным способом его завершения является посылка сигнала KILL. В этом случае процесс завершается операционной системой в любом случае. Это так называемое “жесткое завершение”. Казалось бы – этот вариант лучше, так как работает всегда. Однако недостатком сигнала KILL является то, что при этом несохраненные процессом данные будут потеряны, поэтому использовать его нужно только в крайнем случае.

Часто одна команда запускает сразу несколько процессов. Чтобы их завершить (или же послать им некоторый сигнал) нужно сначала изучить родственные связи процессов с помощью команды ps, а потом завершить каждый из них командой kill. То же самое можно выполнить с помощью одной команды killall. Она позволяет завершить все процессы (в общем случае – послать им сигнал), запущенные одной командой. Формат команды killall:

```
$ killall [ключи] имя_команды
```

Для посылки процессу сигнала используется следующий синтаксис:

```
$ killall -s сигнал имя_команды
```

Так же, как и в команде kill, если сигнал не указан, то процессу посылается сигнал TERM.

Пример:

```
$ xeyes -center red &
[1] 2225
$ killall -s SIGTERM xeyes
[1]+ Завершено xeyes –center red
```

Если указать в команде ключ –r, то имя команды можно задать в виде шаблона в форме регулярного выражения.

Глава 5. Перенаправление ввода-вывода и команды-фильтры

Стандартные потоки ввода-вывода. Перенаправление ввода-вывода команд

При запуске каждый процесс получает три так называемых потока ввода-вывода. Это стандартный поток ввода (поток с номером 0), стандартный поток вывода (поток с номером 1) и стандартный поток сообщений об ошибках (поток с номером 2).

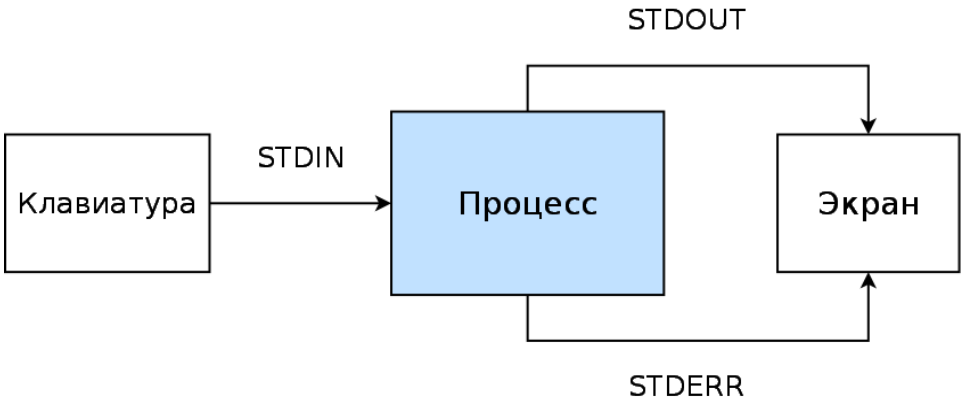

Рис. 5.1. Потоки ввода-вывода процесса

Ввод данных, выполняемый в программе, производится из *стандартного потока ввода* (*standard input*, *stdin*), которым в большинстве случаев является консоль, т.е. пользователь вручную вводит данные с клавиатуры.

Вывод данных, выполняемый в программе, производится в *стандартный поток вывода* (*standard output*, *stdout*). Как правило, этот поток выводится на экран.

Если в программе возникают ошибки, то соответствующие диагностические сообщения выводятся в отдельный поток – стандартный поток сообщений об ошибках (*standard errors*, *stderr*). Он тоже в большинстве случаев выводится на экран.

Полезной особенностью потоков ввода-вывода является то, что их можно **перенаправлять** – направлять на другое устройство или в файл.

Таким образом, мы можем сохранить данные выводимые программой в файл или организовать передачу данных из одной программы в другую.

Перенаправления ввода-вывода указываются в команде при запуске программы с помощью символов >, <, >>. В команде можно указать сразу несколько перенаправлений.

Основными видами перенаправлений являются:

1. Направить данные из файла в стандартный поток ввода программы:

```
команда < файл
```

2. Направить данные из стандартного потока вывода команды в файл (если файл уже существует, то он перезаписывается):

```
команда > файл
```

3. Направить данные из стандартного потока вывода команды в файл (если файл уже существует, то новые данные дописываются в конец файла):

```
команда >> файл
```

4. Направить данные из стандартного потока сообщений об ошибках в файл (если файл уже существует, то он перезаписывается):

```
команда 2> файл
```

5. Направить данные из стандартного потока сообщений об ошибках команды в файл (если файл уже существует, то новые данные дописываются в конец файла):

```
команда 2>> файл
```

6. Направить стандартный поток сообщений об ошибках в стандартный поток вывода (точнее в то же месте, куда направлен стандартный поток вывода):

```
команда 2>&1
```

Теперь рассмотрим примеры перенаправления стандартных потоков ввода-вывода.

```
$ ls -lR > dir-tree.list
```

Данная команда создает файл `dir-tree.list`, содержащий список дерева подкаталогов для текущего каталога.

```
$ sort < unsortedlines > sortedlines
```

Команда читает текстовые строки из файла `unsortedlines`, выполняет их сортировку, и записывает результат в файл `sortedlines`.

```
$ ps * > psout.txt 2> pserr.txt
```

Перенаправить стандартный поток вывода команды `ps` в файл `psout.txt`, а стандартный файл сообщений об ошибках – в файл `pserr.txt`.

```
$ ps -aF > file 2>&1
```

Перенаправить стандартный поток вывода команды `ps` в файл `file`, а стандартный поток сообщений об ошибках в стандартный поток вывода (который уже перенаправлен в файл `file`), т.е. тоже в файл `file`.

Важным частным случаем перенаправления является перенаправление стандартного вывода команды на специальное устройство `/dev/null`. Все поступающие на данное устройство данные уничтожаются. Данное перенаправление используется в том случае, когда вывод команды не требуется выводить на экран или сохранять.

Конвейеры. Команды-фильтры

Достаточно распространенной является ситуация, когда стандартный поток вывода одной программы нужно направить на стандартный поток ввода другой программы. В принципе это можно сделать с помощью использования в перенаправлениях промежуточного файла – сначала записать в файл данные, выводимые первой программой, а затем считать из файла данные для ввода во вторую программу.

Но в Linux есть способ лучше – использование так называемых *конвейеров* (*каналов*, *pipe*). Для организации конвейера нужно между именами команд указать знак | (вертикальная черта):

```
команда1 | команда2
```

В этом случае данные, поступающие в стандартный поток вывода команды 1, направляются в стандартный поток ввода команды 2. Все необходимые операции по пересылке данных и синхронизации данного процесса выполняются операционной системой автоматически.

Пример конвейера:

```
$ ls -a1 | wc -l
```

Данная команда выводит на экран количество файлов в текущем каталоге (включая скрытые файлы). При этом команда `ls` формирует список файлов, а команда `wc` – считает количество строк в сформированном списке.

В конвейер можно объединить и более двух команд:

```
команда1 | команда2 | команда3 | … | командаn
```

С помощью конвейеров можно объединить несколько команд для решения общей составной задачи. В Linux есть ряд команд специально предназначенных для использования внутри конвейеров. Такие команды называются командами-фильтрами. Команды-фильтры принимают некоторые данные из стандартного потока ввода, производят их обработку, и выдают результат в стандартный поток вывода.

Далее мы рассмотрим несколько распространенных команд-фильтров Linux.

Использование конвейеров и команд-фильтров позволяет с помощью комбинации стандартных средств Linux решать нестандартные задачи.

Отметим, что во всех командах-фильтрах данные передаются в обычном **текстовом формате**.

Команда wc

Одной из самых простых команд-фильтров является команда `wc`. Она считает и выводит на экран количество *байтов* (*byte*), *символов* (*character*), *слов* (*word*), *строк* (*lines*) в заданном тексте. Количество байтов и символов в общем случае различаются, т.к. один символ может представляться несколькими байтами. По умолчанию текст считывается из стандартного потока ввода. Так же можно считать текст из файла, указав его имя в параметре команды.

Формат команды: `$ wc [ключи] [файл]…`

Рассмотрим пример:

```
$ cat textfile
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
Ведь были ж схватки боевые,
Да, говорят, еще какие!
Недаром помнит вся Россия
Про день Бородина!
$ wc textfile
  7  26 305 textfile
```

Команда `wc` вывела, что файл `textfile` содержит 7 строк текста, 26 слов и 305 байтов.

Намного чаще команда `wc` используется внутри конвейера, например:

```
$ cat textfile | wc
      7      26     305
```

С помощью ключей команды можно управлять тем, какие именно данные нужно вывести. Так ключ `-l` или `--lines` выводит только количество строк текста, ключ `-w` или `--words` выводит количество слов, ключ `-m` или `--chars` выводит количество символов, а ключ `-c` или `--bytes` выводит количество байтов.

```
$ cat textfile | wc -l
```

```
7
$ cat textfile | wc -w
26
$ cat textfile | wc -m
171
$ cat textfile | wc -c
305
```

Еще одним полезным ключом команды `wc` является ключ `-L`. При задании этого ключа выводится количество символов в самой длинной строке текста.

```
$ cat textfile | wc -L
29
```

Команда tee

Еще одной небольшой, но полезной командой является команда `tee`. Команда `tee` принимает данные из стандартного потока ввода, передает их без изменения на стандартный вывод и одновременно записывает их в один или несколько указанных файлов.

Формат команды: `$ tee [ключи] файл…`

Казалось бы, для чего это может понадобиться? Ответ прост – команда используется для отладочных целей внутри конвейеров. Она позволяет просмотреть данные одного из промежуточных этапов конвейера.

Например, пусть нам нужно составить команду, выводящую на экран количество файлов в текущем каталоге. Выполняем команду:

```
$ ls -1 | wc -l
```

Однако выводимое число не совпадает с реальным числом файлов. Как же найти ошибку? Нужно просмотреть список файлов. Для этого включаем внутрь конвейера команду `tee`.

```
$ ls -1 | tee testfile | wc -l
```

Анализ файла `testfile` показывает, что в списке отсутствуют скрытые файлы. Добавляем в команду `ls` ключ `-a` и получаем нужный результат.

```
$ ls -1a | wc -l
```

Команды tail и head

Команды `head` и `tail` выполняют очень простую задачу – выводят несколько первых (команда `head`) или несколько последних (команда `tail`) строк заданного текста. Текст берется из стандартного потока ввода или из заданного файла.

Формат команд:

```
$ head [ключи] [файл]
$ tail [ключи] [файл]
```

По умолчанию выводится `10` строк текста, но можно указать и другое значение, задав ключ команды `-n` и число.

```
$ cat textfile | head -n 3
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
$ cat textfile | tail -n 3
Да, говорят, еще какие!
Недаром помнит вся Россия
Про день Бородина!
```

Так же можно сделать запись команды более компактной, указав количество выводимых строчек в качестве ключа:

```
$ cat textfile | tail -1
Про день Бородина!
```

```
$ cat textfile | head -2
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
```

Комбинируя эти две команды, можно вывести заданный интервал строк файла, например с 3 по 5:

```
$ cat textfile | head -5 | tail -3
Французу отдана?
Ведь были ж схватки боевые,
Да, говорят, еще какие!
```

Имя файла может быть указано в качестве параметра команд `tail` и `head`:

```
$ head -2 textfile
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
```

Здесь есть один нюанс, состоящий в том, что после запуска команды в файл могут быть дописаны новые данные. Если с командой `head` все просто, то в случае команды `tail` не понятно, как следует поступить – вывести последние строчки, имеющиеся в файле, или ждать появления вновь записанных строк файла.

По умолчанию команда `tail` выводит имеющиеся строки.

Кроме этого имеется специальный режим слежения за файлом, когда после вывода имеющихся строк команда `tail` так же будет выводить вновь дописанные в файл строки. Режим слежения за файлом активизируется специальным ключом `-f`.

Команда sort

Как вы уже наверно догадались, команда `sort` выполняет сортировку строк в тексте.

Формат команды: `$ sort [ключи] [файл]…`

Если в команде не указан файл, то исходный текст берется из стандартного потока ввода.

Если указано несколько файлов – то исходным текстом является объединение этих файлов.

```
$ sort textfile
Ведь были ж схватки боевые,
Да, говорят, еще какие!
Москва, спаленная пожаром,
Недаром помнит вся Россия
Про день Бородина!
Скажи-ка, дядя, ведь не даром
Французу отдана?
```

По умолчанию команда `sort` выполняет простейший вариант сортировки, когда ключом сортировки является вся строка.

Однако ключ сортировки может быть и другим. Например, следующая команда отсортирует строки по вторым словам в них.

```
$ sort -k 2 textfile
Ведь были ж схватки боевые,
Да, говорят, еще какие!
Про день Бородина!
Скажи-ка, дядя, ведь не даром
Французу отдана?
Недаром помнит вся Россия
Москва, спаленная пожаром,
```

Мы получили только самое общее представление о команде `sort`. Это достаточно мощная и гибкая утилита, содержащая множество ключей настройки, влияющих на процесс сортировки.

Большинство опций команды `sort` отвечают на один из вопросов:

- Как выполняется сравнение?
- Как выводится результат?
- Что именно в строках сравнивается?

Для начала будем считать, что сравниваются строки целиком.

Ответ на вопрос “*Как выполняется сравнение*” задается указанием в строке команды следующих ключей: `-b`, `-d`, `-n`, `-f` и др.

С ключом `-b` (сокр. от *blank*) незначащие (например, пробелы) символы в начале строки будут игнорироваться при сравнении.

С ключом `-d` (сокр. от *dictionary*) при сравнении будут учитываться только пробелы, буквы и цифры.

С ключом `-n` (сокр. от *numeric*) сравниваться будут содержащиеся в строках числа (причем сравниваются именно их числовые значения).

С ключом `-f` верхний и нижний регистр символов при сравнении не различаются.

Если никакой из указанных ключей не указан, то будет выполняться обычное текстовое сравнение.

Способ вывода результата определяется наличием или отсутствием в команде ключа `-r` (сокр. от *reverse*). Если в команде указан ключ `-r`, то результаты будут выводиться в обратном порядке.

```
$ sort -r textfile
Французу отдана?
Скажи-ка, дядя, ведь не даром
Про день Бородина!
Недаром помнит вся Россия
Москва, спаленная пожаром,
Да, говорят, еще какие!
Ведь были ж схватки боевые,
```

При сравнении считается, что каждая строка состоит из полей, ограниченных символами-разделителями. По умолчанию разделителями являются символы пробела и табуляции. Однако пользователь может задать свой символ-разделитель полей, указав данный символ в опции `-t` (сокр. от *transition*).

При сравнении строк могут сравниваться части строк – так называемые ключи сортировки. Причем ключей сортировки может быть несколько. Тогда сначала сравниваются первые ключи сортировки, если они равны, то сравниваются вторые ключи сортировки и т.д.

Ключ сортировки задается опцией `-k` (сокр. от *key*) в виде

```
-k POS1[,POS2]
```

Здесь `POS1` и `POS2` задают соответственно начало и конец ключа сортировки и имеют формат `F[.C] [OPTS]`, где `F` – номер текстового поля в строке, `C` – номер символа в текстовом поле, а `OPTS` – параметры сравнения. Параметры сравнения задаются уже знакомыми нам значениями ключей упорядочивания.

Для того чтобы окончательно разобраться с заданием ключей сортировки, рассмотрим следующий пример. Пусть у нас имеется файл, содержащий значения ip-адресов вида `192.168.X.X`. Требуется выполнить сортировку строк файла в порядке возрастания ip-адресов.

Очевидно, что строки файла состоят из четырех полей. Разделителем полей является символ точка. Значимыми при сравнении являются третье и четвертое поле (первые два заданы заранее и не меняются). Поэтому первым ключом сортировки будет третье поле, а вторым ключом сортировки будет четвертое поле. Так же следует учесть, что сравнивать нужно не текст, содержащийся в полях, а соответствующие ему числовые значения. Таким образом, команда сортировки ip-адресов будет иметь вид, показанный в листинге:

```
$ cat iplist
192.168.1.100
192.168.1.19
192.168.1.75
192.168.2.1
192.168.0.5
$ sort -t . -k 3n -k 4n iplist
192.168.0.5
192.168.1.19
192.168.1.75
192.168.1.100
192.168.2.1
```

Здорово, правда? Всего одна строчка! До знакомства с утилитами Linux для решения такой задачи вы бы наверняка стали писать программу на C++ или другом аналогичном языке. Она состояла бы примерно из сотни строчек.

Кроме сортировки команда `sort` может выполнить проверку, является файл отсортированным, или не является. Для выполнения такой проверки нужно в команде указать ключ `-c` (сокр. от *check*). Если текст не отсортирован, то будет выведена первая строка, которая не соответствует порядку сортировки. Если же текст отсортирован, но на экран ничего не выводится.

```
$ sort -c textfile
sort: textfile:2: неправильный порядок: Москва,
спаленная пожаром,
```

Команда uniq

Команда `uniq` предназначена для исключения в тексте повторяющихся строк. При этом исходный текст должен быть отсортирован.

Формат команды: `$ uniq [ключи] [входной_файл [выходной_файл]]`

Вид информации, выдаваемой командой, определяется заданными ключами. С ключом `-c` (сокр. от *count*) перед каждой выводимой строкой указывается количество ее появлений в исходном тексте. С ключом `-d` (сокр. от *duplicate*) выводятся только повторяющиеся строки. С ключом `-u` (сокр. от *unique*) наоборот выводятся только неповторяющиеся строки. С ключом `-i` (сокр. от *ignore case*) символы в верхнем и нижнем регистрах при сравнении не различаются.

```
$ cat textfile1
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
Ведь были ж схватки боевые,
Да, говорят, еще какие!
Недаром помнит вся Россия
Про день Бородина!
```

```
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
Французу отдана?
$ sort textfile1 | uniq
Ведь были ж схватки боевые,
Да, говорят, еще какие!
Москва, спаленная пожаром,
Недаром помнит вся Россия
Про день Бородина!
Скажи-ка, дядя, ведь не даром
Французу отдана?
$ sort textfile1 | uniq -c
      1 Ведь были ж схватки боевые,
      1 Да, говорят, еще какие!
      2 Москва, спаленная пожаром,
      1 Недаром помнит вся Россия
      1 Про день Бородина!
      2 Скажи-ка, дядя, ведь не даром
      3 Французу отдана?
$ sort textfile1 | uniq -u
Ведь были ж схватки боевые,
Да, говорят, еще какие!
Недаром помнит вся Россия
Про день Бородина!
$ sort textfile1 | uniq -d
Москва, спаленная пожаром,
Скажи-ка, дядя, ведь не даром
Французу отдана?
```

Команда cut

Команда cut выбирает отдельные байты, символы или поля из строк исходного текста и направляет их в стандартный поток вывода.

Формат команды: `$ cut ключи [файл]…`

Что именно нужно выделять из строк задается одним из ключей: `-b` (байты - *bytes*), `-c` (символы - *characters*) или `-f` (поля - *fields*) после которых идет параметр `LIST`, задающий какие именно элементы нужно выделить. В команде может присутствовать только один из перечисленных ключей. Значение параметра `LIST` задает один диапазон или несколько диапазонов, разделенных запятыми. Каждый из диапазонов записывается в виде:

`N` – N-й байт, символ или поле. Отсчет ведется с 1,

`N-` – байты, символы или поля, начиная с N-го и до конца строки,

`N-M` – байты, символы или поля с N-го по M-го включительно,

`-M` – байты, символы или поля с 1-го до M-го включительно.

В команде `cut` в качестве разделителя полей по умолчанию используется символ табуляции. Другой разделитель полей можно задать с помощью ключа `-d` (сокр. от *delimiter*).

В качестве примера попробуем выделить из расширенного вывода команды `ls` только имена файлов и права доступа.

```
$ ls -l
-rw-r--r-- 1 user user  64 Янв 10 18:45 iplist
-rw-r--r-- 1 user user 305 Янв 10 14:40 textfile
-rw-r--r-- 1 user user 468 Янв 11 10:58 textfile1
$ ls -l | cut -c 1-10,43-
-rw-r--r-- iplist
-rw-r--r-- textfile
-rw-r--r-- textfile1
```

Команда cmp

Команда `cmp` сравнивает два файла побайтово, и, если они различаются, то выводит информацию о первом байте и строке, где было обнаружено различие.

Формат команды: `$ cmp [ключи] файл1 [файл2 [пропуск1 [пропуск2]]]`

Команда может начать выполнять сравнение файлов не с их начала, а с некоторой позиции в каждом из файлов. Параметры `пропуск1` и `пропуск2` как раз и задают число байтов, которые пропускаются в каждом из файлов перед тем, как начать выполнять сравнение.

Пример:

```
 $ cat textfile
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
Ведь были ж схватки боевые,
Да, говорят, еще какие!
Недаром помнит вся Россия
Про день Бородина!
$ cat textfile2
Скажи-ка, дядя, ведь не даром
Москва, сожженная пожаром,
Французу отдана?
$ cmp textfile textfile2
textfile textfile2 различаются: байт 70, строка 2
```

Команда diff

Команда `diff` так же как и команда `cmp` выполняет сравнение двух файлов. Но если команда `cmp` отвечает на вопрос “*Отличаются файлы или нет?*”, то команда `diff` отвечает на вопрос “*Чем именно отличаются файлы?*”.

Формат команды: `$ diff [опции] старый-файл новый-файл`

Задача, которую решает программа diff при программировании встречается довольно часто. Вы (или кто-то другой) внесли в файл исходного кода изменения, но, так как это было неделю назад, забыли какие именно. Команда diff позволяет сравнить старую и новую версии файла и выяснить, какие изменения были произведены.

Возьмем некоторый текстовый файл, внесем туда изменения, а именно: удалим, вторую строку, добавим строку после пятой и изменим четвертую строку. Проверяем:

```
$ cat textfile
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
Ведь были ж схватки боевые,
Да, говорят, еще какие!
Недаром помнит вся Россия
Про день Бородина!
$ cat textfile_new
Скажи-ка, дядя, ведь не даром
Французу отдана?
Ведь были ж схватки боевые, строку мы изменили
Да, говорят, еще какие!
Эта строка добавлена!
Недаром помнит вся Россия
Про день Бородина!
$ diff textfile textfile_new
2d1
< Москва, спаленная пожаром,
4c3
< Ведь были ж схватки боевые,
---
> Ведь были ж схватки боевые, строку мы изменили
5a5
> Эта строка добавлена!
```

Рассмотрим вывод команды. Видно, что вывод состоит из блоков, начинающихся загадочными строками “2d1”, “4c3”, “5a5”, а за ними следуют строки из файлов, предваряемые символами “<” и “>”. На самом деле ничего загадочного в начале блоков нет. “a” означает *added* – в исходный файл была добавлена строка или фрагмент, “d” означает *deleted* – из исходного файла была удалена строка, а “c” означает *changed* – строка

была изменена. Числа слева и справа от символов указывают номера строк в исходном и измененном файлах.

Ключи команды задают способы сравнения строк, и форматы вывода. Ниже показан вывод команды `diff` с ключами `-u` и `--side-by-side`, облегчающими восприятие выводимой информации.

```
$ diff -u textfile textfile_new
--- textfile    2012-01-10 14:40:05.000000000 +0400
+++ textfile_new2012-01-18 11:26:26.000000000 +0400
@@ -1,7 +1,7 @@
 Скажи-ка, дядя, ведь не даром
-Москва, спаленная пожаром,
 Французу отдана?
-Ведь были ж схватки боевые,
+Ведь были ж схватки боевые, строку мы изменили
 Да, говорят, еще какие!
+Эта строка добавлена!
 Недаром помнит вся Россия
 Про день Бородина!

$ diff --side-by-side textfile textfile_new
Скажи-ка, дядя, ведь не даром         Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,       <
Французу отдана?                      Французу отдана?
Ведь были ж схватки боевые,      |    Ведь  были  ж  схватки  боевые,
строку мы изменили
Да, говорят, еще какие!               Да, говорят, еще какие!
                                 >    Эта строка добавлена!
Недаром помнит вся Россия             Недаром помнит вся Россия
Про день Бородина!                    Про день Бородина!
```

В заключение следует сказать, что вывод команды `diff` часто используется командой `patch`, позволяющей из старого файла и списка изменений получить новый файл. Это полезно при обновлении исходных текстов больших программ, например, ядра Linux, и позволяет передавать не все файлы целиком, а только изменения.

Команда egrep

Наконец мы подошли к изучению, пожалуй, самой мощной утилиты из рассмотренных, – утилиты `egrep`. Она предназначена для поиска строк в тексте, которые соответствуют одному или нескольким шаблонам.

Возможные форматы запуска команды:

```
$ egrep [ключи] шаблон [файл…]
$ egrep [ключи] [-e шаблон | -f файл] [файл…]
```

Анализируемый текст берется из указанного файла или, если файл не указан, то из стандартного потока ввода.

Шаблон задается так называемым регулярным выражением, которые мы рассмотрим чуть позже. В простейшем случае таким шаблоном может быть искомая в тексте подстрока или слово. Рассмотрим простейший пример использования утилиты `egrep`:

```
$ cat textfile
Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
Ведь были ж схватки боевые,
Да, говорят, еще какие!
Недаром помнит вся Россия
Про день Бородина!
$ cat textfile | egrep ведь
Скажи-ка, дядя, ведь не даром
$ cat textfile | egrep -i ведь
Скажи-ка, дядя, ведь не даром
Ведь были ж схватки боевые,
```

Мы видим, что в первом примере ищутся строки, содержащие слово “ведь”, при этом учитывается регистр символов и выводится только одна строка. Во втором случае с ключом `-i` (сокр. от *ignore case*) регистр символов не учитывается, и выводятся две строки.

При анализе совпадений с шаблоном алгоритм работы команды уточняется следующими ключами.

Уже рассмотренный ключ `-i` указывает на то, что при сравнении регистр символов не должен учитываться.

Ключ `-v` активизирует режим обратного поиска совпадений – при этом выводятся строки, **не** совпадающие с шаблоном.

```
$ cat textfile | egrep -iv ведь
Москва, спаленная пожаром,
Французу отдана?
Да, говорят, еще какие!
Недаром помнит вся Россия
Про день Бородина!
```

С ключом `-w` (сокр. от *word*) будут выводиться строки, в которых совпадения являются целыми словами, а не частями слов.

С ключом `-x` будут выводиться строки, в которых с шаблоном совпадает вся строка, а не часть строки.

С ключом `-c` (сокр. от *count*) команда `egrep` будет выдавать не сами строки, соответствующие шаблону, а только количество совпадений.

```
$ cat textfile | egrep -c ведь
1
$ cat textfile | egrep -ic ведь
2
```

Так же ряд опций включает выдачу вместе с совпадающей строкой дополнительной информации, например, с ключом `-n` (сокр. от *number*) дополнительно выдаются номера строк.

```
$ cat textfile | egrep -in ведь
1:Скажи-ка, дядя, ведь не даром
4:Ведь были ж схватки боевые,
```

Если нам нужно найти совпадения в тексте с одним из нескольких шаблонов, то мы можем записать эти шаблоны в файл и указать этот файл в качестве параметра в ключе `-f` (сокр. от *file*). В следующем примере набор шаблонов сохранен в файле `patterns`.

```
$ cat patterns
дядя
Французу
Россия
```

```
$ egrep -f patterns textfile
Скажи-ка, дядя, ведь не даром
Французу отдана?
Недаром помнит вся Россия
```

Регулярные выражения

Регулярные выражения (**regular expressions**, **regexp**) представляют собой формальный язык описания шаблонов для поиска в тексте.

Если вы изучали теорию формальных грамматик, то регулярные выражения покажутся вам очень знакомыми. Регулярные выражения в теории формальных грамматик описываются формальными грамматиками самого простого типа (согласно иерархии Хомского), которые так и называются – регулярные.

Если вы никогда не сталкивались с чем-то похожим на регулярные выражения, то будьте готовы к тому, что их понимание, возможно, потребует от вас некоторых усилий. Язык формальных выражений является *декларативным* и этим отличается от большинства других компьютерных языков, которые вы изучали. Декларативные языки описывают то, *как должен выглядеть результат*, а не то, *как этот результат получить* (второй способ описания характерен для *императивных* языков).

Поиск текста с помощью регулярных выражений был впервые реализован Кеном Томпсоном в редакторе `qed` [5], затем постепенно это механизм вошел в большинство языков и фактически регулярные выражения стали стандартным средством для поиска в тексте.

Теперь от общих замечаний перейдем к рассмотрению самого языка регулярных выражений.

Регулярные выражения состоят из *обычных символов*, таких как буквы или цифры, и *метасимволов*, имеющих специальное значение.

Любой обычный символ в шаблоне должен совпадать с таким же символом в строке. В этом мы уже могли с вами убедиться в предыдущем пункте.

Значение различных метасимволов приведено в табл. 5.1.

Табл. 5.1. Значение метасимволов в регулярных выражениях

Метасимвол	Значение
\c	Отменяет все специальные значения символа c
^	Начало строки
$	Конец строки
.	Произвольный одиночный символ
[…]	Любой из символов указанного подмножества. При задании допустимы диапазоны, например [a-z] – задает любую латинскую строчную букву.
[^…]	Любой символ, не входящий в указанное множество. Допустимы диапазоны
r*	Ноль или больше вхождений выражения r
r+	Одно или больше вхождений выражения r
r?	Ноль или одно вхождение выражения r
r1r2	Выражение r1 следует за выражением r2
r1|r2	Любое из двух выражений – r1 или r2
(r)	регулярное выражение r (скобки позволяют группировать подвыражения)
r{n}	Ровно n вхождений выражения r
r{n,}	n или более вхождений выражения r
r{,m}	До m вхождений выражения r
r{n,m}	От n до m вхождений выражения r

Метасимволы ^ и $ позволяют “привязать” шаблон к началу и концу строки соответственно. Поэтому их иногда называют *символами-якорями*. В следующем листинге шаблон “один” выводит все строки файла, содержащие данное слово. Шаблон “^один” выводит все строки файла, в начале которых стоит слово “один”. Шаблон “один$” выводит все строки файла, в конце которых стоит слово “один”. И наконец, третий шаблон “^один$” выводит строки, состоящие только из этого слова (т.к. перед началом слова должно быть начало строки, а после конца слова – конец строки).

```
$ cat simpletext
один, два, три
два, три, один
три, один, два
один
```

```
$ egrep один simpletext
один, два, три
два, три, один
три, один, два
один
$ egrep ^один simpletext
один, два, три
один
$ egrep один$ simpletext
два, три, один
один
$ egrep ^один$ simpletext
один
```

Еще одним важным метасимволом является точка. Она обозначает любой символ, который может стоять на этом месте. Например, шаблону “о..н” будут соответствовать “один”, “овен”, “о12н” и т.п.

Если же мы хотим, чтобы в некотором месте стоял не любой символ, а к примеру, только буква, или только гласная буква, то нужно воспользоваться метасимволом *квадратные скобки*. На месте данного метасимвола может стоять любой символ из стоящих внутри квадратных скобок и никакой другой. Например, шаблону “г[ои]д” соответствуют слова “год” и “гид”, и не соответствуют слова “гад”, “г2д” и т.п.

Внутри квадратных скобок можно вместо перечисления всех символов указать их диапазон. Например, цифра описывается шаблоном “[0-9]”, строчная латинская буква описывается шаблоном “[a-z]”, а заглавная или строчная русская буква – шаблоном “[а-яА-Я]”.

Если после первой квадратной скобки поставить знак ^, то смысл данного метасимвола меняется на противоположный. Такому выражению будет соответствовать любой символ, кроме перечисленных в квадратных скобках. Например, шаблон “[^0-9]” означает “любой символ, кроме цифры”.

Метасимволы *, +, ? задают количество возможных повторений стоящего перед ними подвыражения.

Так шаблон “аб*в” описывает выражения “ав”, “абв”, “аббв”, “аббббв” и т.д.

Шаблон “аб+в” описывает выражения “абв”, “аббв”, “аббббв” и т.д.

А шаблон “аб?в” описывает выражения “ав” и “абв”.

Похожий смысл несут метасимволы фигурные скобки `{}`.

Еще одной важной возможностью регулярных выражений является группировка подвыражений с помощью круглых скобок.

Например, шаблон “`аб*`” описывает строки “`а`”, “`аб`”, “`абб`”, “`аббб`” и т.д.

Шаблон “`(аб)*`” описывает строки “”(пустая строка), “`аб`”, “`абаб`”, “`абабаб`” и т.д.

Ну и, наконец, метасимвол вертикальная черта `|` означает – одно из двух возможных выражений. Например, сочетание “русская *буква или две цифры*” описывается шаблоном “`([а-я])|([0-9]{2})`”

Примеры использования регулярных выражений

В предыдущем разделе мы с вами рассмотрели “азбуку” регулярных выражений. Теперь попробуем использовать их для решения некоторых практических задач.

Сначала рассмотрим несколько простых примеров, связанных с обработкой вывода команды `ls`.

Сначала выведем строки для всех подкаталогов текущего каталога. Как известно, в расширенном выводе команды `ls` первый символ строки определяет тип файла и для каталогов он равен “`d`”, поэтому соответствующая команда имеет вид:

```
$ ls -l | egrep ^d
```

Так же мы можем вывести строки для файлов нескольких типов, например, для каталогов и символьных ссылок:

```
$ ls -l | egrep ^[dl]
```

Выведем только строки для файлов с расширением “`.c`”. Здесь нужно учесть два момента. Во-первых, расширение файла стоит в самом конце строки, поэтому в конце шаблона должен стоять метасимвол-якорь `$`. Во-вторых, символ точки является метасимволом, поэтому для отключения его

специального значения нужно его экранировать символом “\”. Получаем команду:

```
$ ls -l | egrep \.c$
```

Наконец, выведем все файлы, которые могут быть запущены на исполнение владельцем файла. Для этого файл должен иметь соответствующее право на исполнение “x” и не быть каталогом (для каталогов этот атрибут прав доступа имеет другой смысл):

```
$ ls -l | egrep ^[^d]..x
```

Изучая предыдущие примеры, можно подумать, что регулярные выражения используются всегда только вместе с утилитой `egrep`. Однако это не так. Другие утилиты Linux так же могут поддерживать задание шаблонов для поиска в виде регулярных выражений. Одной из таких утилит является утилита `find`, которая позволяет использовать регулярные выражения при поиске файлов. Рассмотрим следующий пример. Пусть у нас есть цифровой фотоаппарат, который сохраняет отснятые снимки в файлах с именами вида `DSCN3518.JPG` или `DSCN0157.JPG` (такую структуру имен файлов имеют в частности некоторые модели фотоаппаратов Nikon). Следующая команда позволяет найти все файлы фотографий, снятых данным фотоаппаратом, в текущем каталоге и его подкаталогах:

```
$ find -regextype posix-egrep -regex ".*DSCN[0-9]{4}\.JPG"
./test/DSCN0123.JPG
```

Рассмотрим еще несколько примеров регулярных выражений, предназначенных для поиска в тексте или валидации данных определенного вида.

Рассмотрим сначала несколько шаблонов, определяющих целые числа в различных системах счисления.

В двоичной: `(\+|-)?[01]+`.

В восьмеричной: `(\+|-)?[0-7]+`.

В десятичной: `(\+|-)?[0-9]+`.

В шестнадцатеричной: `(\+|-)?[0-9a-fA-F]+`.

Вещественные числа в десятичном виде описываются шаблоном:

```
(\+|-)?[0-9]+(\.[0-9]+)?(e|E(\+|-)[0-9]+)?
```

Идентификатор в большинстве языков программирования (последовательность латинских букв, цифр и знаков подчеркивания, начинающаяся с буквы или знака подчеркивания):

```
[a-zA-Z_][a-zA-Z_0-9]*
```

В заключение данного раздела рассмотрим два более сложных примера, взятых автором из замечательной книги Брайана Кернигана и Роба Пайка “UNIX. Программное окружение” [6].

В первом примере нужно найти в файле со списком английских слов те слова, гласные буквы в которых идут в последовательности “a-e-i-o-u”. Шаблон для поиска достаточно длинный, поэтому для удобства он сохранен в файле alphvowels. Сам шаблон достаточно прост и не требует дополнительных пояснений.

```
$ cat alphvowels
^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u[^ae
iou]*$
$ egrep –f alphvowels wordsfile
abstemious abstemiously abstentious
achelious acheirous acleistous
affectious annelidous arsenious
arterious bacterious caesious
facetious facetiously fracedinous
majestious
```

Второй пример ищет все английские слова из шести букв в файле словаря, буквы в которых упорядочены по алфавиту:

```
$ cat monotonic
^a?b?c?d?e?f?g?h?i?j?k?l?m?n?o?p?q?r?s?t?u?v?w?x?y?z?$
$ egrep –f monotonic wordsfile | egrep ’^......$’
abdest acknow adipsy agnosy almost
befist behint beknow bijoux biopsy
```

```
chintz dehors dehort deinos dimpsy
egilops ghosty
```

Поиск производится в два этапа. На первом этапе отбираются все слова, буквы в которых упорядочены по алфавиту. На втором этапе из них отбираются слова, состоящие из шести букв. Шаблон для поиска слов с буквами, упорядоченными по алфавиту, достаточно длинный, поэтому он предварительно сохранен в файле `monotonic`.

Будем считать, что буквы в слове не повторяются. Если буквы в слове упорядочены по алфавиту, то мы можем получить это слово вычеркиванием части букв из алфавитной последовательности. То есть каждая буква в алфавитной последовательности либо присутствует в слове один раз, либо не присутствует вообще. Поэтому получаем шаблон, приведенный в листинге.

Глава 6. Командные файлы и язык shell

Общие сведения о языке shell

В предыдущей главе мы с вами рассмотрели, как с помощью конвейеров можно объединять несколько программ для решения общей задачи. Однако средств простой передачи данных между программами, которую реализуют конвейеры, для решения многих практических задач хватает не всегда. Дополнительные средства здесь нам предоставляет язык shell, содержащий ряд возможностей, характерных для алгоритмических языков программирования, – переменные и выражения, операторы ввода-вывода, условия, циклы, функции, – все это есть в языке shell. Программы на языке shell называются **сценариями** или **скриптами**. В этой главе рассмотрен вариант языка shell, реализованный в командной оболочке **bash**, которая является самым распространенным командным интерпретатором в Unix-подобных системах.

В то же время язык shell предназначен для других задач, нежели обычные языки программирования, такие как C, C++ и др. Shell – это своеобразный “раствор” для скрепления крупных строительных блоков, которыми являются стандартные утилиты Linux и программы пользователя. Только представьте – операторами shell являются целые программы, как стандартные, так и разработанные пользователем!

Если вам нужно написать программу, которая будет выполнена один раз, или быстро создать прототип программы для проверки какой-то идеи, то shell и другие скриптовые языки – ваш естественный выбор. Основная задача языка shell – *автоматизация* повторяющихся рутинных операций при работе с компьютером и его настройке. Скрипты инициализации системы написаны на языке shell.

Сильная сторона языка shell – небольшое время разработки программ. Часто одну и ту же задачу можно решить, написав за несколько дней программу на C++ размером в несколько тысяч строк кода, или написав за полчаса скрипт на shell, содержащий 30 строк кода.

В то же время согласно [7] язык shell лучше **не использовать** для решения следующих типов задач:

- для ресурсоемких задач, особенно когда важна скорость исполнения (поиск, сортировка и т.п.),

- для задач, связанных с выполнением математических вычислений, особенно это касается вычислений с плавающей запятой, вычислений с повышенной точностью, комплексных чисел,
- для кросс-платформенного программирования,
- для сложных приложений, когда структурирование является жизненной необходимостью (контроль за типами переменных, прототипами функций и т.п.),
- для целевых задач, от которых может зависеть успех предприятия,
- когда во главу угла поставлена безопасность системы, когда необходимо обеспечить целостность системы и защитить ее от вторжения, взлома и вандализма,
- для проектов, содержащих компоненты, очень тесно взаимодействующие между собой,
- для задач, выполняющих огромный объем работ с файлами,
- для задач, работающих с многомерными массивами,
- когда необходимо работать со структурами данных, такими как связанные списки или деревья,
- когда необходимо предоставить графический интерфейс с пользователем (GUI),
- когда необходим прямой доступ к аппаратуре компьютера,
- когда необходимо выполнять обмен через порты ввода-вывода или сокеты,
- когда необходимо использовать внешние библиотеки,
- для проприетарных, "закрытых" программ (скрипты представляют из себя исходные тексты программ, доступные для всеобщего обозрения).

Операторы shell как команды. Командные операторы

При написании программ на языке shell следует понимать, что операторы языка shell с точки зрения интерпретатора являются *обычными командами* с присущим им синтаксисом – в каждом из операторов кроме названия (*имя команды*) могут быть *ключи* и *параметры*. Так же важным моментом является то, что все данные представляются в *текстовом виде*.

При выполнении можно объединить несколько команд в одной строке с помощью одного из нескольких *командных операторов*.

Самым простым командным оператором является оператор “*точка с запятой*”. Его формат следующий:

```
команда1 ; команда2
```

При вводе такой составной команды сначала будет выполнена `команда1`, после ее завершения будет выполнена `команда2`. Т.е. фактически символ `;` является обычным разделителем команд.

Еще одним командным оператором является символ `&`:

```
команда1 & команда2
```

В этом случае выполняется `команда1`, не дожидаясь ее завершения сразу начинает выполняться `команда2`.

Еще два оператора `&&` и `||` могут запускать или не запускать вторую команду в зависимости от кода завершения первой команды. Формат их следующий:

```
команда1 && команда2
команда1 || команда2
```

В обоих случаях сначала будет запущена `команда1`.

Если код завершения[11] `команды1` равен `0` (значение *истина*), то в операторе `&&` будет запущена `команда2`.

Если код завершения `команды1` не равен `0` (значение *ложь*), то в операторе `||` будет запущена `команда2`.

В некотором смысле командные операторы `&&` и `||` являются аналогами логических операций “*и*” и “*или*” и реализуют сокращенный способ вычисления выражения, когда общее значение полностью определяется первым операндом.

[11] Код завершения представляет собой целое число, передающееся после завершения программы той программе, которая запустила первую. Как правило, в коде завершения сообщается об ошибках во время выполнения программы или об отсутствии ошибок.

Пример скрипта на shell

Скрипт на языке shell – это обычный текстовый файл. Первой строкой скрипта должна стоять строка, указывающая на командный процессор, предназначенный для обработки скрипта. Например, для командной оболочки bash эта строка будет иметь вид (для других оболочек нужно изменить путь к исполняемому файлу интерпретатора):

```
#!/bin/bash
```

Кроме этого файл скрипта должен иметь права на его исполнение. Их можно задать, например, следующей командой:

```
$ chmod u+x scriptfile
```

В заключение этого раздела рассмотрим пример небольшого скрипта:

```
#!/bin/bash
# Вывод текстового сообщения
echo ‘Пример сценария.’
echo ‘Количество файлов в текущем каталоге:’
# Вывод количества файлов в текущем каталоге
ls -a | wc –l
```

Приведенный скрипт выводит количество файлов в текущем каталоге. Рассмотрим его.

Строки, начинающиеся с символа “#” являются комментариями. Первая строка, начинающаяся с символов “#!”, как мы уже рассмотрели ранее, задает путь к командному интерпретатору.

Команда `echo` выводит текстовую строку на экран. Последняя строка содержит команду, которая выводит количество строк в текущем каталоге.

Теперь давайте выполним скрипт. Для этого нужно ввести соответствующее ему имя файла с указанием каталога (текущего):

```
$ ./myscript.sh
Пример сценария.
```

```
Количество файлов в текущем каталоге:
5
```

Еще одним вариантом команды, запускающей скрипт, является команда

```
$ bash имя_скрипта
```

Или для нашего примера:

```
$ bash myscript.sh
```

Второй вариант может быть полезен в том случае, если нужно выполнить скрипт с указанием какого-либо ключа интерпретатора при запуске интерпретатора команд. Подробнее эту возможность мы будем рассматривать в пункте, посвященном отладке сценариев.

Пользовательские переменные. Ввод-вывод

При решении любой нетривиальной программной задачи требуется где-то сохранять промежуточные значения данных. Поэтому без переменных никак не обойтись. Есть они и в языке shell. Кроме переменных, которые создает сам пользователь, существуют так же и системные переменные, которые мы рассмотрим в следующих разделах.

В языке shell переменные **не имеют типа**. Все значения хранятся **в строковом виде**. Числовые значения тоже хранятся в строковом виде и обрабатываются с помощью специальных команд.

Так как переменные не имеют типа то, специально *описывать переменные не нужно*. Нужно просто присвоить переменной значение:

```
var=1
```

Важным моментом здесь является то, что данное выражение нужно писать **без пробелов**. Если вы напишите присвоение как ”`var = 1`”, то оно будет трактоваться как запуск команды с именем `var` и параметрами = и `1`.

Еще раз повторю – **пробелы до и после знака равенства недопустимы**!

Имена переменных, так же, как и в большинстве других языков программирования, представляют собой последовательности букв и цифр, начинающиеся с буквы. При этом на правах буквы может использоваться знак подчеркивания.

Чтобы получить значение переменной, перед ее именем нужно указать знак `$`, например:

```
var2=$var
```

Вывести значение переменной можно с помощью команды `echo`:

```
$ echo $var2
1
```

Однако рассмотренный вариант вывода значения переменной в большинстве случаев на практике использовать не стоит. Рекомендуется всегда заключать значения переменных в двойные кавычки:

```
$ echo “$var2”
1
```

Это необходимо в тех случаях, когда значение используется в качестве параметра какой-либо команды. Если значение содержит разделители, например, пробелы, то оно будет интерпретировано, как несколько параметров команды. Если же мы заключим значение в двойные кавычки, то оно будет интерпретировано, как один параметр.

Использованная нами команда `echo`, как мы уже увидели, выводит заданные ей параметры на экран. По умолчанию она переводит курсор на следующую строку (т.е. является аналогом функции WriteLn в языке Pascal). Если же нам нужно оставить курсор в той же строке (т.е. использовать аналог функции Write в Pascal), то нужно задать в команде `echo` ключ `-n`.

Ввести значение переменной можно с помощью команды `read`.

```
$ cat echoread.sh
#!/bin/bash
echo -n "Введите значение переменной A:"
```

```
read a
echo -n "A = "
echo "$a"
$ ./echoread.sh
Введите значение переменной A:100
A = 100
```

С помощью рассмотренных команд можно выполнять ввод данных из файла и вывод данных в файл. Естественно здесь нам поможет перенаправление ввода-вывода.

Следующий пример демонстрирует вывод в файл:

```
$ cat file_out1.sh
#!/bin/bash
echo "Если вы в своей квартире -" > outfile.txt
echo "Лягте на пол, три-четыре!" >> outfile.txt
echo "Выполняйте правильно движения." >> outfile.txt
$ ./file_out1.sh
$ cat outfile.txt
Если вы в своей квартире -
Лягте на пол, три-четыре!
Выполняйте правильно движения.
```

Первая операция вывода перенаправляется со знаком > и перезаписывает файл `outfile.txt`, а остальные операции вывода перенаправляются со знаком >> и дописывают информацию в конец файла.

Ввод выполняется аналогично. Однако при перенаправлении ввода отсутствует операция <<. Поэтому каждая новая операция чтения с перенаправлением ввода будет выполнять чтение с начала файла. Исправить этот недостаток можно, объединив несколько операций чтения в один блок с помощью фигурных скобок, и перенаправив ввод-вывод целого блока операторов.

```
$ cat file_in1.sh
#!/bin/bash
{
```

```
read a
read b
} < outfile.txt
echo "Из файла считаны строки:"
echo "1) $a"
echo "2) $b"
$ ./file_in1.sh
Из файла считаны строки:
1) Если вы в своей квартире -
2) Лягте на пол, три-четыре!
```

Перенаправлять ввод-вывод можно не только для блока операторов, но и для операторов проверки условия и операторов циклов.

Еще одним специальным видом перенаправления ввода-вывода является *встроенный документ* (*here document*). Встроенные документы обычно используются с интерактивными программами, которые ожидают ввода команд от пользователя. Встроенный документ позволяет разместить вводимые данные прямо в тексте скрипта. Встроенные документы имеют следующий синтаксис:

```
интерактивная_команда << ОГРАНИЧИТЕЛЬ
данные1
данные2
…
ОГРАНИЧИТЕЛЬ
```

После имени команды ставится знак << за которым идет последовательность символов – ограничитель. Далее все строки до ограничителя считаются вводимыми данными.

Пример: многострочный вывод с помощью команды `cat`:

```
$ cat here_doc1.sh
#!/bin/bash
cat << END_DOC
Это первая строка многострочного вывода
Это вторая
```

```
Это третья
Это последняя
END_DOC
$ ./here_doc1.sh
Это первая строка многострочного вывода
Это вторая
Это третья
Это последняя
```

В качестве еще одного примера рассмотрим скачивание файла с `tftp`-сервера. В данном примере запускается tftp-клиент, который имеет собственную интерактивную командную оболочку. Данные, содержащиеся во встроенном документе скрипта, представляют собой команды, которые вводятся в интерактивной командной оболочке tftp-клиента.

```
$ cat here_doc2.sh
#!/bin/bash
tftp << DELIMITER
connect 127.0.0.1
get example
quit
DELIMITER
$ ./here_doc2.sh
tftp> tftp> Received 22 bytes in 0.0 seconds
tftp> $
```

Позиционные и специальные параметры

При запуске любой программы из командной строки мы можем передавать ей ключи и параметры. Точно так же мы можем передавать ключи и параметры скрипту при его запуске. Доступ к значениям ключей и параметров командной строки в скрипте осуществляется с помощью специальных переменных, называемых *позиционными параметрами*. Позиционные параметры имеют имена, совпадающие с номером соответствующего параметра - `$1`, `$2` и т.д. Если номер параметра состоит из двух цифр, то их нужно взять в фигурные скобки, например `${10}`. В

параметре $0 передается имя скрипта. Следующий листинг является иллюстрацией к использованию позиционных параметров

```
$ cat pospar.sh
#!/bin/bash
echo "Параметр номер 0:" "$0"
echo "Параметр номер 1:" "$1"
echo "Параметр номер 2:" "$2"
echo "Параметр номер 3:" "$3"
echo "Параметр номер 4:" "$4"
echo "Параметр номер 5:" "$5"
$ ./pospar.sh -n --help par1 file_5
Параметр номер 0: ./pospar
Параметр номер 1: -n
Параметр номер 2: --help
Параметр номер 3: par1
Параметр номер 4: file_5
Параметр номер 5:
```

Кроме позиционных параметров, есть еще один вид встроенных переменных, имеющих специальные значения – специальные параметры. Во время выполнения служебные параметры заменяются некоторым значением. Правила замены служебных параметров приведены в табл 6.1.

Табл 6.1. Правила замены специальных параметров

Специальный параметр	Правило замены
*	Заменяется списком всех позиционных параметров, представленным в виде одной строки
@	Заменяется списком всех позиционных параметров, представленным в виде совокупности нескольких строк
#	Заменяется числом позиционных параметров
?	Заменяется значением кода завершения последнего из выполнявшихся на переднем плане процессов
$	Заменяется идентификатором процесса (PID) оболочки
!	Заменяется идентификатором процесса (PID) последней из выполняющихся фоновых команд

0	Заменяется именем оболочки или запускаемого скрипта. Если bash запускается для выполнения командного файла, $0 равно имени этого файла. В противном случае это значение равно полному пути к оболочке
_(подчеркивание)	Заменяется последним аргументом предыдущей из выполнявшихся команд

Пример:

```
$ echo $0
/bin/bash
$ echo $$
2511
$ echo $?
0
```

Переменные окружения

Еще одним видом специальных переменных являются *переменные окружения*. Они осуществляют доступ к значениям параметров, операционной системы, командной оболочки и различных прикладных программ.

Значения некоторых важных переменных окружения приведены в табл. 6.2.

Табл. 6.2. Значения основных переменных окружения

Переменная окружения	Значение
HOME	Домашний каталог пользователя
LOGNAME	Имя текущего пользователя
PATH	Содержит список каталогов, разделенных двоеточием, в которых интерпретатор будет искать программу, если пользователь при запуске последней явно не указал путь к ней
PS1	Внешний вид приглашения командной строки
SHELL	Путь к командному интерпретатору по умолчанию для

	текущего пользователя
PWD	Текущий рабочий каталог
RANDOM	При каждом чтении значения переменной пользователь получает псевдослучайное число

Пример:

```
$ cat env_view.sh
#!/bin/bash
echo "Домашний каталог   : " "$HOME"
echo "Текущий каталог    : " "$PWD"
echo "Имя пользователя   : " "$LOGNAME"
echo "Путь поиска        : " "$PATH"
echo "Приглашение        : " "$PS1"
echo "Командная оболочка : " "$SHELL"
echo "Случайное число    : " "$RANDOM"
echo "Случайное число    : " "$RANDOM"
echo "Случайное число    : " "$RANDOM"
$ ./env_view.sh
Домашний каталог   :  /home/user
Текущий каталог    :  /home/user/LinuxBook
Имя пользователя   :  user
Путь поиска        :  /usr/local/bin:/usr/bin:/bin:
/usr/local/games:/usr/games
Приглашение        :
Командная оболочка :  /bin/bash
Случайное число    :  1578
Случайное число    :  18104
Случайное число    :  23315
```

Просмотреть значения всех установленных переменных окружения можно с помощью команды `env` без параметров.

Значения переменных окружения передаются каждой программе при ее запуске. Если мы запускаем некоторую программу из своего скрипта, и для этой программы нужно установить некоторые специфические переменные окружения, то это можно сделать командой экспорт.

```
$ export CROSS_COMPILE=arm-linux-
```

В данной строке создается переменная окружения `CROSS_COMPILE`, имеющая значение `arm-linux-`. Надо отметить, что команда export влияет на значения переменных окружения только для текущего процесса и для его дочерних процессов.

Скобки и кавычки

Ранее мы с вами говорили о том, что все значения в языке shell хранятся в текстовом виде. При задании строковых констант важным моментом является то, в какие скобки или кавычки они заключены. В зависимости от вида скобок и кавычек строковая константа обрабатывается тем или иным образом. В языке shell строковые константы могут заключаться:

- в двойные кавычки ",
- в одинарные кавычки (апострофы) ',
- в обратные кавычки (данный символ расположен на клавиатуре на клавише "тильда" ~) `.

Двойные кавычки (**нестрогие кавычки**) предназначены для объединения в один параметр строк, содержащих пробелы. При этом если строка содержит подстроку `$имя_переменной`, то в это место строки подставляется значение данной переменной. Одиночные подстановки значения переменных так же рекомендуется заключать в двойные кавычки.

Например:

```
$ str="кавычки"
$ echo "двойные $str"
двойные кавычки
```

Так же внутри двойных кавычек специальный смысл носят символы обратная кавычка (`) и обратный слеш (\).

Одинарные кавычки (**строгие кавычки**) в точности передают в качестве параметра то значение, которое в них заключено. Никакие подстановки не выполняются.

Например:

```
$ echo 'одинарные $str'
одинарные $str
```

Обратные кавычки предназначены для выполнения команд. Строка, заключенная в обратные кавычки заменяется выводом команды, содержащейся в обратных кавычках.

Например:

```
$ filenum=`ls -a1 | wc -l`
$ echo "В текущем каталоге $filenum файлов."
В текущем каталоге 13 файлов.
```

Вместо обратных кавычек также можно использовать символы `$(…)`, которые легче читаются. Данная форма является более предпочтительной.

```
$ echo "В текущем каталоге $(ls -a1 | wc -l) файлов."
В текущем каталоге 13 файлов.
```

Выражения

Как мы с вами уже обсуждали, все переменные в shell хранят значение в текстовом виде, поэтому если нам нужно вычислить значение числового выражения, то нужно использовать специальную команду – команду `expr`. В параметрах команды передается выражение, которое нужно вычислить.

Например:

```
$ expr 3 + 5
8
$ expr 3 - 5
-2
$ expr 3 \* 5
15
```

```
$ expr 3 / 5
0
$ expr 3 % 5
3
$ expr 3 = 5
0
$ expr 3 \< 5
1
$ expr 3 \>= 5
0
```

При использовании команды `expr` операнды нужно отделять друг от друга пробелами, так как они являются параметрами команды. Символы операций традиционны для языков с С-подобным синтаксисом, кроме того, что числовая проверка на равенство обозначается одним знаком “=”, а не двумя. Для тех, кто не знаком с языком С, знак “`%`” соответствует операции остаток от деления. Некоторые символы в знаках операций приходится *экранировать символом “`\`”* (или *заключать в кавычки*). Например, если просто написать символ “*”, то он будет трактоваться как шаблон имен файлов, а не как знак операции умножения.

Более удобной формой записи команды `expr` являются двойные круглые скобки `$((…))`, которые являются более наглядными. При их использовании отделять все части выражения друг от друга пробелами не обязательно:

```
$ echo $(((2+3)*5))
25
```

В целом возможности встроенных команд bash в области вычислений достаточно ограничены. Однако в Linux существуют калькуляторы `bc` и `dc`, позволяющие выполнять вычисления значений с плавающей запятой с достаточной степенью точности, и реализующие интерфейс командной строки. Они могут успешно применяться внутри сценариев.

Оператор if. Команда test

Синтаксис условного оператора `if` в языке shell подобен аналогичным операторам в большинстве других языков. Этот формат следующий:

```
if список_команд1
  then список_команд2
  else список_команд3
fi
```

где `список_команд1`, `список_команд2`, `список_команд3` – это последовательности команд, разделенные запятыми и оканчивающиеся точкой с запятой или символом новой строки. Так же эти последовательности могут быть заключены в фигурные скобки: `{список_команд}`.

Семантика оператора имеет некоторую специфику. Само понятие оператора в shell несколько условно. Строго говоря, в языке shell вообще нет операторов, а есть только команды. Данная конструкция представляет собой четыре команды – `if`, `then`, `else` и `fi`.

Оператор `if` работает следующим образом. Он проверяет коды завершения, возвращаемые командами из `список_команд1`. Если в этом списке несколько команд, то проверяется код завершения последней команды списка. Если значение кода завершения равно `0` (успешное завершение), то будут выполняться команды из `список_команд2`; если это значение ненулевое, будут выполнены команды из `список_команд3`. Код завершения, возвращаемый таким составным оператором `if`, совпадает с кодом завершения, выдаваемым последней командой выполняемой последовательности.

Обратите внимание, что *нулевое* значение кода завершения команды соответствует значению “*истина*”, а *ненулевое* значение кода завершения соответствует значению “*ложь*”.

Оператор `if` также может включать в себя вложенные ветвления. Полный его формат следующий:

```
if список_команд
    then список_команд
  [ elif список_команд
      then список_команд ]
    ...
  [ else список_команд ]
```

```
fi
```

Здесь в квадратных скобках указаны необязательные элементы.

В качестве условия в операторе `if` практически всегда используется команда `test`, которая специально предназначена для анализа условий. Формат команды `test`:

```
test выражение
```

Так же команда `test` имеет альтернативную долее наглядную форму записи:

```
[ выражение ]
```

Команда `test` анализирует переданное ей в качестве параметра выражение, и если оно *истинно*, то команда завершается с кодом `0`. Если же выражение *ложно*, то команда завершается с кодом `1`.

Команда test может выполнять проверку нескольких типов простых условий:

- проверки, связанные с файлами,
- проверки, связанные со строками,
- арифметические сравнения.

Синтаксис перечисленных проверок простых условий приводится соответственно в табл. 6.3, 6.4 и 6.5.

Табл. 6.3. Проверки, связанные с файлами, в команде test

Выражение	Значение выражения
`-a file` или `-e file`	Верно, если файл с именем `file` существует
`-b file`	Верно, если `file` существует и является специальным файлом блочного устройства
`-c file`	Верно, если `file` существует и является специальным файлом символьного устройства
`-d file`	Верно, если `file` существует и является каталогом
`-f file`	Верно, если файл с именем `file` существует и

	является обычным файлом
`-g file`	Верно, если файл с именем `file` существует и для него установлен бит смены группы (SGID)
`-h file` или `-L file`	Верно, если файл с именем `file` существует и является символической ссылкой
`-k file`	Верно, если файл с именем `file` существует и для него установлен "sticky" -бит
`-p file`	Верно, если файл с именем `file` существует и является именованным каналом (FIFO)
`-r file`	Верно, если файл с именем `file` существует и для него установлено право на чтение
`-s file`	Верно, если файл с именем `file` существует и его размер больше нуля
`-u file`	Верно, если файл с именем `file` существует и для него установлен бит смены пользователя (SUID)
`-w file`	Верно, если файл с именем `file` существует и для него установлено право на запись
`-x file`	Верно, если файл с именем `file` существует и для него установлено право на исполнение
`-O file`	Верно, если файл с именем `file` существует и его владельцем является пользователь, на которого указывает эффективный идентификатор пользователя
`-G file`	Верно, если файл с именем `file` существует и принадлежит группе, определяемой эффективным идентификатором группы
`-S file`	Верно, если файл с именем `file` существует и является сокетом
`-N file`	Верно, если файл с именем `file` существует и изменялся с тех пор, как был последний раз прочитан.
`file1 -nt file2`	Верно, если файл `file1` имеет более позднее время модификации, чем `file2`
`file1 -ot file2`	Верно, если файл `file1` старше, чем `file2`
`file1 -ef file2`	Верно, если файлы `file1` и `file2` имеют одинаковые номера устройств и индексных

	дескрипторов (inode)

Табл. 6.4. Проверки, связанные со строками, в команде test

Выражение	Значение выражения
`-z string`	Верно, если длина строки равна нулю
`-n string`	Верно, если длина строки не равна нулю
`string1 == string2`	Верно, если строки совпадают. Вместо == может использоваться =
`string1 !== string2`	Верно, если строки не совпадают
`string1 < string2`	Верно, если строка `string1` лексикографически предшествует строке `string2`
`string1 > string2`	Верно, если строка `string1` лексикографически стоит после строки `string2`

Табл. 6.5. Арифметические сравнения целых чисел (положительных и отрицательных) в команде `test`

Выражение	Значение выражения
`arg1 -eq arg2`	Верно, если числа `arg1` и `arg2` равны
`arg1 -ne arg2`	Верно, если числа `arg1` и `arg2` не равны
`arg1 -lt arg2`	Верно, если число `arg1` меньше числа `arg2`
`arg1 -le arg2`	Верно, если число `arg1` меньше или равно числу `arg2`
`arg1 -gt arg2`	Верно, если число `arg1` больше числа `arg2`
`arg1 -ge arg2`	Верно, если число `arg1` больше или равно числу `arg2`

Из рассмотренных элементарных условий можно составлять и сложные условия с помощью операций И, ИЛИ, НЕ.

Операция НЕ имеет формат:

```
!(выражение)
```

Операция И имеет формат:

```
выражение -a выражение
```

Операция ИЛИ имеет формат:

```
выражение -o выражение
```

В языке shell есть две вспомогательные команды `true` и `false`. Команда `true` всегда завершается с кодом `0`, а команда `false` всегда завершается с кодом `1`.

Теперь рассмотрим несколько примеров применения оператора `if` и команды `test`.

Сначала простой пример. В нем пользователя просят ввести имя файла. Если введена пустая строка, то выводится соответствующее сообщение. Далее проверяется, что файл существует. И в конце проверяется, что файл обладает правами на чтение и выполнение и, соответственно, может быть запущен.

```
#!/bin/bash
echo -n “введите имя файла: ”
read filename

if [ -z “$filename” ]
then
  echo “Введена пустая строка!”
  exit
fi

if [ ! -f “$filename” ]
then
  echo “Файл $filename не существует!”
  exit
fi

if [ -x “$filename” -a -r “$filename” ]
  then
```

```
    echo "Файл $filename может быть запущен на
исполнение"
  fi
```

Еще один пример скрипта на работу с файлами и строковыми значениями приведен ниже. В данном скрипте использован знак двоеточия ":", который в shell имеет смысл "пустая команда".

```
#!/bin/bash
# выполнение операций с файлами
if [ "$1" == "-help" ]
then
  # Получение справки
  echo "Формат вызова команды:"
  echo " Запуск файла на выполнение"
  echo " filescr -run файл"
  echo " Удаление файла"
  echo " filescr -delete файл"
  echo " Получение справки"
  echo " filescr -help"
  exit
fi

# Проверка наличия файла
if [ -f "$2" ]
then :
else
  echo "Файл $2 не существует!"
  exit
fi

if [ "$1" == "-run" ]
then
  # Запуск файла
  if [ -x "$2" ]
  then :
  else
```

```
        echo "Вы не обладаете правами на запуск файла
$2!"
        exit
      fi

      echo "Запускаем файл $2!"
      "$2"
      exit
    fi

    if [ "$1" == "-delete" ]
    then
      # Удаление файла
      rm -f "$2"
      echo "Файл $2 удален!"
      exit
    fi
```

Оператор case

Оператор `case` выполняет выбор одного из множества вариантов действий в зависимости от значения некоторого выражения. Он является аналогом оператора `case` в языке Pascal или оператора `switch` в C-подобных языках. Формат оператора

```
case выражение in
 [(] шаблон [ | шаблон ] ... )
    список_команд
    ;;
...
esac
```

Оператор `case` вычисляет значение выражения, а потом последовательно сравнивает его значение с заданными шаблонами. После нахождения первого совпадения выполняются команды, стоящие после этого шаблона и до знака `;;`. После первого совпадения дальнейшие сравнения не производятся.

Оператор `case` завершается с кодом `0`, если не было ни одного совпадения. Если совпадение было найдено, то код завершения равен коду завершения последней выполненной команды.

Шаблон задается **не** в форме *регулярного выражения*, как можно было бы подумать, а в форме *шаблона файла*. Рассмотрим различные варианты шаблонов на примере, в котором анализируется первый параметр командной строки, переданный скрипту:

```
$ cat case_op.sh
#!/bin/bash
case "$1" in
  a)
    echo "Это буква a" ;;
  [bcd])
    echo "Это буква b, c или d" ;;
  e | f)
    echo "Это буква e или f" ;;
  [0-9])
    echo "Это цифра" ;;
  [^xyz])
    echo "Это буква, но не x, y или z" ;;
  ??)
    echo "Это две буквы" ;;
  a*)
    echo "Это слово на букву a" ;;
  yes)
    echo "Это слово yes" ;;
  *)
    echo "Это другое значение" ;;
esac
$ ./case_op.sh a
Это буква a
$ ./case_op.sh b
Это буква b, c или d
$ ./case_op.sh e
Это буква e или f
$ ./case_op.sh 5
Это цифра
```

```
$ ./case_op.sh p
Это буква, но не x, y или z
$ ./case_op.sh no
Это две буквы
$ ./case_op.sh all
Это слово на букву a
$ ./case_op.sh yes
Это слово yes
$ ./case_op.sh end
Это другое значение
```

Оператор for

Оператор `for` в языке shell несколько отличается от одноименных аналогов в алгоритмических языках программирования. Он перебирает значения из заданного списка и для каждого из значений выполняет некоторую последовательность команд. Поэтому аналогом данного оператора в других языках является не оператор `for`, а оператор `foreach`. Формат оператора `for` следующий:

```
for переменная [ in список_слов ]
do
  список_команд
done
```

Так же как в операторе `if` команды в списке команд разделяются запятыми или знаком перевода строки.

Пример:

```
$ cat for1.sh
#!/bin/bash
for i in a b c
do
  echo "Обрабатывается слово $i"
done
$ ./for1.sh
Обрабатывается слово a
```

```
Обрабатывается слово b
Обрабатывается слово c
```

Если в команде опущен список слов, то тело цикла выполняется для каждого из позиционных параметров, заданных при запуске команды.

Достаточно часто в качестве списка слов используется шаблон имени файла. В этом случае тело цикла выполняется для всех файлов, соответствующих шаблону.

```
$ cat for2.sh
#!/bin/bash
for filename in *
do
  echo "Найден файл $filename"
done
$ ./for2.sh
Найден файл case_op.sh
Найден файл for1.sh
Найден файл for2.sh
Найден файл for3.sh
Найден файл textfile
Найден файл textfile1
Найден файл textfile2
Найден файл textfile_new
```

И еще один похожий пример:

```
$ cat for3.sh
#!/bin/bash
for filename in for?.sh
do
  echo "Найден файл $filename"
done
$ ./for3.sh
Найден файл for1.sh
Найден файл for2.sh
Найден файл for3.sh
```

В некоторых случаях требуется, чтобы оператор for работал так же, как и в традиционных языках программирования. Для этого существует специальная команда `seq`, которая выводит последовательность чисел, лежащих между двумя заданными числами, являющимися параметрами команды. В следующем примере выполняется перебор чисел от `1` до `4`. При запуске данного скрипта на экран будет выведено содержимое файлов `for1.sh`, `for2.sh`, `for3.sh`, `for4.sh`.

```
#!/bin/bash
for i in $( seq 1 4 )
do
  echo "Файл for""$i"".sh"
  cat "for$i.sh"
  echo "=================="
done
```

В заключение еще один пример использования оператора `for`:

```
#!/bin/bash
# вывод списка исполняемых файлов в текущем каталоге
echo "Список исполняемых файлов в каталоге $PWD"
for file in *
do
  if [ -x "$file" ]
  then
    echo " $file"
  fi
done
```

Операторы while и until

Обычный цикл с предусловием в языке shell реализуется операторами `while` и `until`. Оператор `while` имеет формат:

```
while список_команд1
do
```

```
  список_команд2
done
```

Перед каждой итерацией цикла выполняется `список_команд1`. Если код завершения последней команды в нем равен `0` (условие истинно), то выполняется тело цикла, представленное списком команд 2. Если код завершения последней команды в списке команд 1 не равен нулю, то цикл завершается.

Ниже в качестве примера приведен скрипт, выполняющий ввод строк из файла и вывод их на экран. Так же в данном примере показано перенаправление ввода-вывода для цикла.

```
$ cat while.sh
#!/bin/bash
while read a
do
  echo $a
done < outfile.txt
$ ./while.sh
Если вы в своей квартире -
Лягте на пол, три-четыре!
Выполняйте правильно движения.
```

Оператор `until` очень похож на оператор `while`. Он имеет формат:

```
until список_команд1
do
  список_команд2
done
```

Отличие оператора `until` от оператора `while` состоит в том, что `список_команд1` задает не условие продолжения цикла, а условие его окончания. Т.е. когда последняя выполненная команда в списке команд 1 завершится с кодом `0`, цикл завершится.

Обратите внимание – цикл `until` проверяет условие **перед** выполнением очередной итерации, а не после ее завершения как принято, например, в языке Pascal.

Операторы break и continue

В shell есть и два традиционных оператора, связанных с циклами, - операторы `break` и `continue`. Формат их следующий

```
break [n]
continue [n]
```

Оператор `break` завершает выполнение цикла и выходит из него. Оператор `continue` прерывает выполнение текущей итерации цикла и начинает следующую итерацию.

Удобным отличием языка shell от многих других языков является наличие необязательного числового параметра, указывающего уровень вложенности циклов, на который действует оператор. Так, если у вас два вложенных цикла, и вам, находясь во внутреннем цикле, нужно прервать выполнение внешнего цикла, то можно просто написать:

```
break 2
```

Странно, почему во многих других языках, например в Pascal или C нет этой простой, но очень полезной возможности.

Функции

Функции так же присутствуют в языке shell, правда, обладая несколько урезанной функциональностью по сравнению с другими языками программирования. Внешне функции в языке shell напоминают *определенные пользователем команды*.

Формат определения функции следующий:

```
function имя_функции()
{
  список_команд
}
```

или

```
имя_функции()
{
  список_команд
}
```

Слово `function` в определении функции необязательно, но его рекомендуется использовать.

Если мы хотим вызвать функцию, то нужно просто написать:

```
имя_функции
```

Сама функция при этом должна быть определена в скрипте до места ее первого вызова.

При вызове функции *отдельный процесс не запускается*, т.е. функция выполняется в том же процессе.

Функции в языке shell могут быть *рекурсивными*, т.е. вызывать сами себя.

Важным моментом является *передача параметров* в функцию и *возврат* из нее *значения*.

В большинстве языков программирования требуется описывать имена и типы параметров функции заранее. В языке shell нет типов данных, поэтому параметры функции описывать не нужно. Более того, мы можем передать в функцию различное количество параметров.

При вызове функции значения параметров дописываются после имени функции через разделитель. Внутри тела функции переданные ей параметры заменяют собой позиционные параметры скрипта. Соответственно доступ к параметрам функции осуществляется через специальные переменные `$1`, `$2` и т.д., а само количество параметров передается в переменной `$#`. При этом позиционный параметр `$0` не меняется. После возврата из функции позиционные параметры приобретают старое значение, которое они имели до вызова функции.

Возможность возврата значения из функции несколько ограничена. Функция возвращает значение в виде *кода завершения*. Код завершения можно задать явно с помощью оператора `return`, имеющего формат:

```
return [значение]
```

При вызове данной команды так же выполняется выход из функции.

Если код завершения в функции не задан явно, то он равен коду завершения последней выполненной команды.

Недостатком способа возврата значения функции через код завершения является то, что максимальное положительное значение, которое можно вернуть из функции равно `255`.

Следующий пример демонстрирует определение функции, вызов функции, передачу параметров и возврат значения:

```
$ cat func_1.sh
#!/bin/bash

# Определение функции func
function func()
{
  echo "Вызвана функция func()"
  echo "Всего передано параметров $#"
  echo "Первый параметр равен $1"
  echo "Второй параметр равен $2"
  return 5
}

# Вызов функции func
func 1 "два"
echo "Функция возвратила значение $?"
$ ./func_1.sh
Вызвана функция func()
Всего передано параметров 2
Первый параметр равен 1
Второй параметр равен два
Функция возвратила значение 5
```

Внутри функции можно задавать локальные переменные – переменные, которые будут доступны только внутри данной функции. Синтаксис определения локальной переменной следующий:

```
locale переменная=значение
```

В качестве примера, рассмотрим функцию, вычисляющую факториал заданного числа и выводящую значение на стандартный вывод:

```
$ cat func_2.sh
#!/bin/bash
function fuctorial()
{
  local res=1
  for i in $(seq 1 $1)
  do
    res=$(($res*$i))
  done
  echo "Факториал числа $1 равен $res."
}

fuctorial 1
fuctorial 2
fuctorial 3
fuctorial 4
fuctorial 5
$ ./func_2.sh
Факториал числа 1 равен 1.
Факториал числа 2 равен 2.
Факториал числа 3 равен 6.
Факториал числа 4 равен 24.
Факториал числа 5 равен 120.
```

Отладка скриптов на shell

При разработке больших скриптов на shell может возникнуть ситуация, когда:

- запускаются и выдают множество сообщений о синтаксических ошибках,
- запускаются, но работают не так, как предполагалось,
- запускаются и выполняют свою основную задачу, но с побочными эффектами [7].

В каждом из этих случаев требуется выполнить отладку скрипта.

К сожалению, по возможностям средств поиска ошибок и отладки язык shell отстает от популярных сред разработки. Тем не менее, некоторые инструменты для отладки `bash`-скриптов имеются.

Во-первых, есть несколько специальных режимов командного интерпретатора `bash`, позволяющих разобраться с тем, что происходит со скриптом во время выполнения. Они задаются специальными ключами при запуске командного интерпретатора `bash`. Рассмотрим применение этих ключей на примере следующего скрипта:

```
$ cat func_2.sh
#!/bin/bash
function fuctorial()
{
  local res=1
  for i in $(seq 1 $1)
  do
    res=$(($res*$i))
  done
  echo "Факториал числа $1 равен $res."
}

fuctorial 1
fuctorial 2
fuctorial 3
fuctorial 4
fuctorial 5
```

При запуске `bash` с ключом `-n` сам скрипт не выполняется, а только проверяется наличие синтаксических ошибок. Однако некоторые ошибки при этом не определяются.

```
$ bash -n func_2.sh
```

Последняя команда ничего не вывела, т.к. проверяемый скрипт не содержит синтаксических ошибок.

При запуске `bash` с ключом `-v` каждая команда перед выполнением выводится на экран, что позволяет проследить последовательность выполнения команд в скрипте:

```
$ bash -v func_2.sh
#!/bin/bash
function fuctorial()
{
  local res=1
  for i in $(seq 1 $1)
  do
    res=$(($res*$i))
  done
  echo "Факториал числа $1 равен $res."
}

fuctorial 1
seq 1 $1)
seq 1 $1
Факториал числа 1 равен 1.
fuctorial 2
seq 1 $1)
seq 1 $1
Факториал числа 2 равен 2.
fuctorial 3
seq 1 $1)
seq 1 $1
Факториал числа 3 равен 6.
fuctorial 4
seq 1 $1)
seq 1 $1
Факториал числа 4 равен 24.
fuctorial 5
```

```
seq 1 $1)
seq 1 $1
Факториал числа 5 равен 120.
```

При запуске `bash` с ключом `-x` в краткой форме выводятся результаты выполнения каждой команды, что позволяет проконтролировать правильность получения промежуточных результатов:

```
$ bash -x func_2.sh
+ fuctorial 1
+ local res=1
++ seq 1 1
+ for i in '$(seq 1 $1)'
+ res=1
+ echo 'Факториал числа 1 равен 1.'
Факториал числа 1 равен 1.
+ fuctorial 2
+ local res=1
++ seq 1 2
+ for i in '$(seq 1 $1)'
+ res=1
+ for i in '$(seq 1 $1)'
+ res=2
+ echo 'Факториал числа 2 равен 2.'
Факториал числа 2 равен 2.
+ fuctorial 3
+ local res=1
++ seq 1 3
+ for i in '$(seq 1 $1)'
+ res=1
+ for i in '$(seq 1 $1)'
+ res=2
+ for i in '$(seq 1 $1)'
+ res=6
+ echo 'Факториал числа 3 равен 6.'
Факториал числа 3 равен 6.
+ fuctorial 4
+ local res=1
```

```
++ seq 1 4
+ for i in '$(seq 1 $1)'
+ res=1
+ for i in '$(seq 1 $1)'
+ res=2
+ for i in '$(seq 1 $1)'
+ res=6
+ for i in '$(seq 1 $1)'
+ res=24
+ echo 'Факториал числа 4 равен 24.'
Факториал числа 4 равен 24.
+ fuctorial 5
+ local res=1
++ seq 1 5
+ for i in '$(seq 1 $1)'
+ res=1
+ for i in '$(seq 1 $1)'
+ res=2
+ for i in '$(seq 1 $1)'
+ res=6
+ for i in '$(seq 1 $1)'
+ res=24
+ for i in '$(seq 1 $1)'
+ res=120
+ echo 'Факториал числа 5 равен 120.'
Факториал числа 5 равен 120.
```

Кроме описанных специальных режимов командного интерпретатора существует программа *Bash Debugger*, позволяющая выполнить сценарий в режиме пошаговой отладки. В *Debian* для ее установки нужно установить пакет `bashdb`. Эта программа реализует командный интерфейс пошагового отладчика `gdb`, который будет рассмотрен в следующих разделах курса. Она позволяет выполнять пошаговую отладку скрипта, просматривать значения переменных, устанавливать точки останова и т.п. Для запуска отладки скрипта нужно набрать одну из двух следующих команд:

```
bash --debugger имя_скрипта
```

или

```
bashdb имя_скрипта
```

Рассмотрим пример отладки нашего скрипта. Для удобства восприятия все команды отладчика вводятся в полном формате и далее выделены жирным шрифтом:

```
$ bash --debugger func_2.sh
Bourne-Again Shell Debugger, release 4.0-0.4

Copyright 2002, 2003, 2004, 2006, 2007, 2008, 2009 Rocky Bernstein
This is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.

(/home/user/LinuxBook/func_2.sh:12):
12:  fuctorial 1
bashdb<0> help
Available commands:
  / debug enable help next show step+ untrace
  alias delete eval history print signal tbreak up
  break disable examine info pwd skip trace watch
  commands display file kill quit source tty where
  condition down frame list restart step unalias
  continue edit handle load set step- undisplay

Readline command line editing (emacs/vi mode) is available.
Type "help" followed by command name for full documentation.
bashdb<1> step
(/home/user/LinuxBook/func_2.sh:3):
3:   {
bashdb<2> step
(/home/user/LinuxBook/func_2.sh:4):
4:     local res=1
bashdb<3> step
(/home/user/LinuxBook/func_2.sh:3):
3:   {
bashdb<(4)> step
(/home/user/LinuxBook/func_2.sh:5):
5:     for i in $(seq 1 $1)
bashdb<5> step
```

```
(/home/user/LinuxBook/func_2.sh:7):
7:       res=$(($res*$i))
bashdb<6> step
(/home/user/LinuxBook/func_2.sh:9):
9:     echo "Факториал числа $1 равен $res."
bashdb<7> print $res
1
bashdb<8> step
Факториал числа 1 равен 1.
(/home/user/LinuxBook/func_2.sh:13):
13:  fuctorial 2
bashdb<9> step
(/home/user/LinuxBook/func_2.sh:3):
3:   {
bashdb<10> step
(/home/user/LinuxBook/func_2.sh:4):
4:     local res=1
bashdb<11> step
(/home/user/LinuxBook/func_2.sh:3):
3:   {
bashdb<(12)> step
(/home/user/LinuxBook/func_2.sh:5):
5:     for i in $(seq 1 $1)
bashdb<13> step
(/home/user/LinuxBook/func_2.sh:7):
7:       res=$(($res*$i))
bashdb<14> step
(/home/user/LinuxBook/func_2.sh:5):
5:     for i in $(seq 1 $1)
bashdb<15> step
(/home/user/LinuxBook/func_2.sh:7):
7:       res=$(($res*$i))
bashdb<16> step
(/home/user/LinuxBook/func_2.sh:9):
9:     echo "Факториал числа $1 равен $res."
bashdb<17> print $res
2
bashdb<18> quit
```

Глава 7. Управление пакетами и менеджеры пакетов

Понятие о программных пакетах. Менеджеры пакетов

Работая в Windows, мы с вами часто сталкиваемся с установкой новых программ. При этом мы сами должны найти и скачать установочные файлы программы, запустить процесс установки, много раз нажать “Далее” и в конце в идеале получить установленную программу, готовую к работе. В неидеальном мире реальных программ в середине процесса можно получить сообщение вида “Новая программа не установится, пока не будут установлены программы A, B и C”. Например, однажды при установке Turbo Delphi автору пришлось предварительно поставить целых пять программ, прежде чем главная программа согласилась установиться.

Если задуматься, то все описанные действия легко автоматизируются, и единственным препятствием для этого является необходимость контроля фактов установки программы со стороны её хозяина (правообладателя). В Linux большая часть программного обеспечения является свободным, и процесс установки программ автоматизирован с помощью менеджеров пакетов.

Менеджер пакетов – это набор инструментальный средств, предназначенных для автоматизации процесса установки, обновления, конфигурирования и удаления программ. Устанавливаемые программные файлы при этом формируются в минимальные, относительно самостоятельные наборы, называемые **программными пакетами**.

Основными функциями менеджера пакетов являются:

- поддержка установки, обновления и удаления пакетов,
- обеспечение целостности пакетов,
- контроль зависимостей между пакетами,
- поиск среди доступных пакетов.

Характерной чертой современных менеджеров пакетов является то, что доступные пакеты хранятся в локальных или сетевых централизованных хранилищах пакетов, так же называемых **репозиториями**. Осуществляется централизованное сопровождение этих хранилищ.

Самыми распространенными менеджерами пакетов являются:

- `dpkg` – используется в дистрибутивах Debian, Ubuntu, Mint и др.

- `rpm` – используется в дистрибутивах Red Hat, Fedora, SuSe и др.

Одной из важнейших задач менеджера пакетов является отслеживание и обработка **взаимосвязей** (**зависимостей**) между пакетами. Рассмотрим возможные взаимосвязи между двумя пакетами A и B. Существуют следующие основные виды взаимосвязей между пакетами:

- Если для нормального функционирования пакета A требуется, чтобы на компьютере был обязательно установлен пакет B, то говорят, что пакет A **зависит** (**depends**) от пакета B, а пакет B **предоставляет** (**provides**) некоторую функциональность пакету A. При этом часто дополнительно требуется, чтобы версия пакета B была не меньше заданной.
- Если пакет A не может работать, при установленном в системе пакете B, то говорят, что пакет A **конфликтует** (**conflicts**) с пакетом B.
- Если файлы пакета A удаляются и/или замещаются файлами пакета B, то говорят, что пакет B **заменяет** (**replaces**) пакет A.

Структура пакета deb

Программный пакет представляет собой архивный файл специального формата. Программные пакеты Debian, имеют расширение `.deb` и представляют собой архивный файл формата `ar`, внутри которого содержатся три других файла:

- `debian-binary` – текстовый файл, содержащий номер версии формата deb,
- `control.tar.gz` – архив, содержащий различную служебную информацию о пакете: описание пакета, версию пакета, сценарии, которые выполняются при установке и удалении пакета и др.
- `data.tar` – архив, содержащий устанавливаемые в системе файлы: выполняемые файлы, файлы настроек, страницы man/info и другие файлы. Данный файл чаще всего хранится в сжатом виде и в этом случае имеет имя `data.tar.gz`, `data.tar.bz2` и т.п.

Рассмотренная структура файла соответствует версиям формата deb, начиная с версии 0,93. Она может меняться, поэтому актуальное описание формата нужно смотреть на man-странице `deb(5)`.

Программы для работы с пакетами debian

Для работы с пакетами в Debian существует несколько программ.

Во-первых, это программа **dpkg**. Она выполняет набор базовых низкоуровневых операций с пакетами в виде deb-файлов. Она является утилитой достаточно низкого уровня, и самостоятельно используется сравнительно редко. В большинстве случаев пользуются другими программами, которые, в свою очередь, используют утилиту dpkg.

Во-вторых, это программа **apt**. Apt является основной утилитой, для работы с пакетами через командную строку. Она позволяет работать с сетевыми репозиториями (хранилищами) пакетов. Кроме этого данная утилита умеет работать с зависимостями пакетов. Используется достаточно часто. Может встраиваться в сценарии.

В-третьих, это программа **Synaptic**, которая является графическим интерфейсом в среде GNOME для управления программными пактами Debian. Пожалуй, это самая удобная программа для работы с пакетами для обычного пользователя.

В-четвертых, это программа **aptitude**. Она имеет командный интерфейс похожий на apt, а без параметров запускает псевдографический интерфейс для управления пакетами. Многие считают, что aptitude более удобный инструмент, чем apt.

Программа dpkg

В большинстве случаев для работы с пакетами следует использовать утилиту apt, которая работает с репозиториями пакетов. Тем не менее, возможна ситуация, когда, вы получите некоторую программу в виде deb-файла. В этом случае нужно использовать утилиту dpkg.

Для установки программного пакета используется следующий синтаксис:

```
$ sudo dpkg –i пакет…
```

Например:

```
$ sudo dpkg -i uclibc-crosstools-common_3.4.2-13_i386.deb
Выбор ранее не выбранного пакета uclibc-crosstools-common.
(Чтение базы данных ... на данный момент установлено 187466 файлов и каталогов.)
Распаковывается пакет uclibc-crosstools-common (из файла uclibc-crosstools-common_3.4.2-13_i386.deb)...
Настраивается пакет uclibc-crosstools-common (3.4.2-13) ...
```

Для удаления пакета нужно выполнить команду:

```
$ sudo dpkg –r пакет…
```

Также утилита dpkg может быть полезна для получения информации о пакете. Описание пакета можно получить, выполнив команду с ключом `-s`:

```
$ dpkg -s minicom
Package: minicom
Status: install ok installed
Priority: optional
Section: comm
Installed-Size: 1152
Maintainer: Martin A. Godisch <godisch@debian.org>
Architecture: i386
Version: 2.4-3
Depends: libc6 (>= 2.7), libncurses5 (>= 5.7+20100313)
Recommends: lrzsz
Description: friendly menu driven serial communication program
 Minicom is a clone of the MS-DOS "Telix" communication program. It emulates
```

```
 ANSI and VT102 terminals, has a dialing directory
and auto zmodem download.
```

Выполнив команду с ключом `-L` можно узнать список файлов, устанавливаемых в системе, содержащихся в данном пакете:

```
$ dpkg -L minicom
/.
/etc
/etc/minicom
/usr
/usr/bin
/usr/bin/minicom
/usr/bin/runscript
/usr/bin/xminicom
/usr/bin/ascii-xfr
/usr/share
/usr/share/doc
/usr/share/doc/minicom
/usr/share/doc/minicom/minicom.FAQ
/usr/share/doc/minicom/AUTHORS
…
```

Программа apt

Рассмотрим основные операции при работе с программой `apt`. Она состоит из нескольких утилит, из которых наиболее распространенной является утилита `apt-get`. Так же достаточно часто используется утилита `apt-cache`.

Программа apt работает с сетевыми и локальными хранилищами пакетов, которые перечисляются в файле `/etc/apt/sources.list`. Пример содержимого данного файла приведен ниже:

```
#

# deb cdrom:[Debian GNU/Linux 6.0.0 _Squeeze_ - Official
i386 DVD Binary-1 20110205-17:27]/ squeeze contrib main
```

```
deb http://ftp.ru.debian.org/debian/ squeeze main
deb-src http://ftp.ru.debian.org/debian/ squeeze main

# Line commented out by installer because it failed to
verify:
# deb http://security.debian.org/ squeeze/updates main
contrib
# Line commented out by installer because it failed to
verify:
# deb-src http://security.debian.org/ squeeze/updates main
contrib

deb http://ftp.ru.debian.org/debian/ squeeze-updates main
contrib
deb-src http://ftp.ru.debian.org/debian/ squeeze-updates
main contrib
```

В приведенном выше файле символ “#” в начале строки означает, что данная строка является комментарием (указанный в ней источник пакетов не используется).

Рассмотрим базовые операции работы с пакетами.

Поиск пакетов

Для поиска пакетов и просмотра информации о них используется утилита `apt-cache`.

С помощью данной утилиты мы можем выполнить поиск пакетов по ключевым словам. Для этого используется следующий синтаксис команды:

```
$ apt-cache search ключевые_слова
```

В результате будет выведен список пакетов, в названии или описании которых присутствую указанные ключевые слова.

Например, выполним поиск пакетов по названию программы `minicom` (программа-терминал, предназначенная для связи через последовательный порт):

```
$ apt-cache search minicom
```

```
cutecom - Graphical serial terminal, like minicom
modemu - Telnet services for communication programs
focal - Interpreter for FOCAL programming language
minicom - программа связи через последовательный порт с интерфейсом основанным на меню
```

Мы видим, что существует пакет с таким именем. Далее узнаем информацию о данном пакете.

Для просмотра основных параметров пакета используется команда

```
$ apt-cache show имя_пакета
```

Для пакета `minicom` будет выведена следующая информация:

```
$ apt-cache show minicom
Package: minicom
Priority: optional
Section: comm
Installed-Size: 1152
Maintainer: Martin A. Godisch <godisch@debian.org>
Architecture: i386
Version: 2.4-3
Depends: libc6 (>= 2.7), libncurses5 (>= 5.7+20100313)
Recommends: lrzsz
Filename: pool/main/m/minicom/minicom_2.4-3_i386.deb
Size: 308934
MD5Sum: 6aa8760da2cdf1ea1b8820876ff66167
SHA1: 12864a533695f536bbfb7e0cdbecaa7b236aa115
SHA256: e2f647b8d20187487f232c208fc750542bc7dc41b02569a0991c6ab5bb969ae7
Description-ru: программа связи через последовательный порт с интерфейсом основанным на меню
 Minicom -- это аналог программы связи "Telix" в MS-DOS. Она эмулирует
```

```
 терминалы ANSI и VT102, имеет телефонную книгу и
может получать файлы
 автоматически по протоколу zmodem.
Tag: hardware::modem, interface::text-mode,
role::program, scope::utility, uitoolkit::ncurses,
use::login
```

В листинге указано, что пакет имеет тип `optional`, т.е. не является обязательным и предназначен для решения каких-либо специальных (не базовых) задач, версия пакета 2.4-3, зависимости пакета – `libc6` и `libncurses5`, также приведено описание пакета.

Для просмотра зависимостей пакета существует и отдельная команда:

```
$ apt-cache depends имя_пакета
```

Для пакета `minicom` будут выведены следующие зависимости:

```
$ apt-cache depends minicom
minicom
  Зависит: libc6
  Зависит: libncurses5
  Рекомендует: lrzsz
```

Установка пакета

Установка пакета (или нескольких пакетов) выполняется командой:

```
$ sudo apt-get install список_пакетов
```

В качестве примера установим пакет `dia`, содержащий одноименный редактор диаграмм:

```
user@dlink-debian:~$ sudo apt-get install dia
Чтение списков пакетов... Готово
Построение дерева зависимостей
Чтение информации о состоянии... Готово
Будут установлены следующие дополнительные пакеты:
```

```
  dia-common dia-libs
НОВЫЕ пакеты, которые будут установлены:
  dia dia-common dia-libs
обновлено 0, установлено 3 новых пакетов, для удаления отмечено 0 пакетов, и 0 пакетов не обновлено.
Необходимо скачать 0 B/7 227 kB архивов.
После данной операции, объём занятого дискового пространства возрастёт на 27,3 MB.
Хотите продолжить [Д/н]? д
Выбор ранее не выбранного пакета dia-common.
(Чтение базы данных ... на данный момент установлено 148378 файлов и каталогов.)
Распаковывается пакет dia-common (из файла .../dia-common_0.97.1-7_all.deb)...
Выбор ранее не выбранного пакета dia-libs.
Распаковывается пакет dia-libs (из файла .../dia/dia-libs_0.97.1-7_i386.deb)...
Выбор ранее не выбранного пакета dia.
Распаковывается пакет dia (из файла .../d/dia/dia_0.97.1-7_i386.deb)...
Обрабатываются триггеры для desktop-file-utils ...
Обрабатываются триггеры для gnome-menus ...
Обрабатываются триггеры для gconf2 ...
Обрабатываются триггеры для hicolor-icon-theme ...
Обрабатываются триггеры для menu ...
Обрабатываются триггеры для man-db ...
Настраивается пакет dia-common (0.97.1-7) ...
Настраивается пакет dia-libs (0.97.1-7) ...
Настраивается пакет dia (0.97.1-7) ...
update-alternatives: используется `/usr/bin/dia-normal' для предоставления `/usr/bin/dia' (dia) в автоматический режим.
Обрабатываются триггеры для menu ...
```

Как видно из листинга, был установлен не только запрашиваемый пакет `dia`, но, кроме этого, автоматически были установлены пакеты, от которых он зависит.

Теперь можно запустить редактор диаграмм Dia, выбрав соответствующий пункт в подменю “Графика” меню “Приложения” (в главном меню GNOME).

Обновление пакета

Через некоторое время после установки пакета в репозиториях может появиться его новая версия, в которой реализованы новые функции, исправлены найденные ошибки и т.п. Поэтому время от времени нужно выполнять обновление программных пакетов.

Операция обновления пакетов выполняется в два этапа. Первый этап – это получение информации о наличии в репозиториях обновленных пакетов. Данная операция выполняется командой:

```
$ sudo apt-get update
```

Второй этап – это скачивание и установка обновленных пакетов. Он выполняется командой:

```
$ sudo apt-get upgrade
```

Если же нам нужно обновить только один определенный пакет, то можно воспользоваться уже рассмотренной командой `apt-get install`.

Удаление пакета

Для удаления программного пакета используется команда:

```
$ sudo apt-get remove список_пакетов
```

Для примера удалим установленный ранее пакет `dia`:

```
$ sudo apt-get remove dia
Чтение списков пакетов... Готово
Построение дерева зависимостей
Чтение информации о состоянии... Готово
```

```
    Следующие пакеты устанавливались автоматически и
больше не требуются:
      dia-libs dia-common
    Для их удаления используйте 'apt-get autoremove'.
    Пакеты, которые будут УДАЛЕНЫ:
      dia
    обновлено 0, установлено 0 новых пакетов, для
удаления отмечено 1 пакетов, и 0 пакетов не обновлено.
    После данной операции, объём занятого дискового
пространства уменьшится на 598 kB.
    Хотите продолжить [Д/н]? д
    (Чтение базы данных ... на данный момент
установлено 150653 файла и каталога.)
    Удаляется пакет dia ...
    Обрабатываются триггеры для man-db ...
    Обрабатываются триггеры для menu ...
```

В листинге команды указано, что пакеты, от которых зависел удаляемый пакет, `dia-libs` и `dia-common` больше не требуются. Однако в данном варианте команды не нужные больше зависимости автоматически не удаляются. Для их удаления нужно использовать следующую команду:

```
$ sudo apt-get autoremove dia
```

Особенностью рассмотренной команды `apt-get remove` является то, что она не удаляет файлы настроек указанных для удаления программ. Если вам нужно полностью удалить программу, включая файлы настроек, то можно использовать команду `apt-get purge`, которая имеет синтаксис, аналогичный двум предыдущим командам.

Графические средства для работы с пакетами. Программа Synaptic

Программа apt является достаточно мощным и удобным средством для работы с программными пакетами. Однако в Debian существуют специальные утилиты, позволяющие выполнять те же операции, но с помощью графического интерфейса. Это программа Synaptic и программа

Update Manager. Обе программы требуют для своей работы ввода пароля администратора.

Synaptic является графическим интерфейсом к apt в среде GNOME. Данная программа позволяет выполнять все перечисленные операции с пакетами, характерные для утилиты apt-get.

Окно программы Synaptic изображено на рис. 8.1. В правой части окна находится список доступных пакетов (вверху) и описание выбранного пакета (внизу). В списке пакетов установленные пакеты помечены слева зеленым квадратиком. С помощью списка в левой части окна можно ограничить число отображаемых в основном списке пакетов (например, только установленные пакеты, или только пакеты, относящиеся к категории “программирование” и т.п.).

В правой верхней части окна расположено поле ввода для поиска пакетов (Quick search), которое позволяет выполнить поиск пакетов по названию пакета или ключевым словам.

Для того чтобы установить пакет, нужно выполнить двойной щелчок мышью на соответствующем пункте в списке пакетов (при этом пакет будет помечен стрелочкой) и нажать кнопку “Применить”.

Вторая, из упомянутых программ, Update Manager (рис. 8.2), позволяет выполнить две операции – проверить наличие обновлений (Check), и обновить систему (Install Updates).

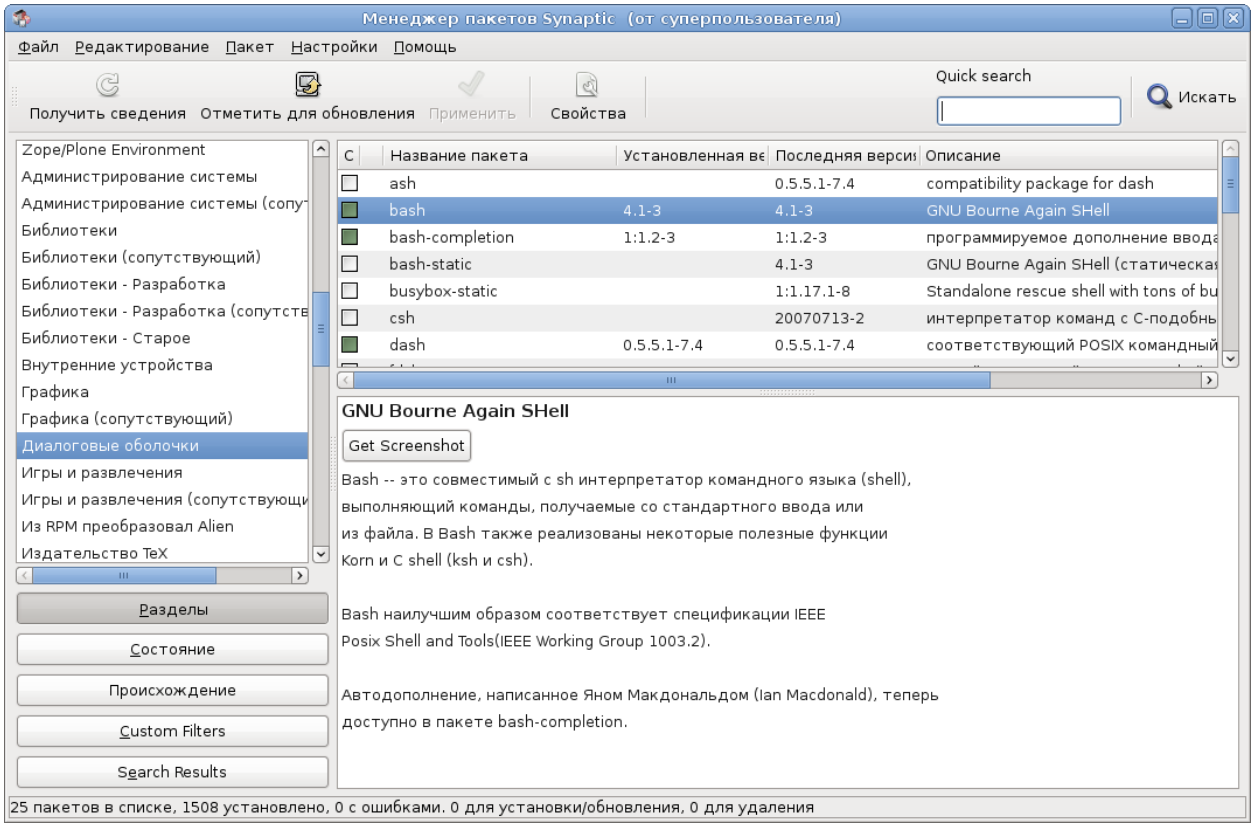

Рис. 8.1. Изображение главного окна программы Synaptic

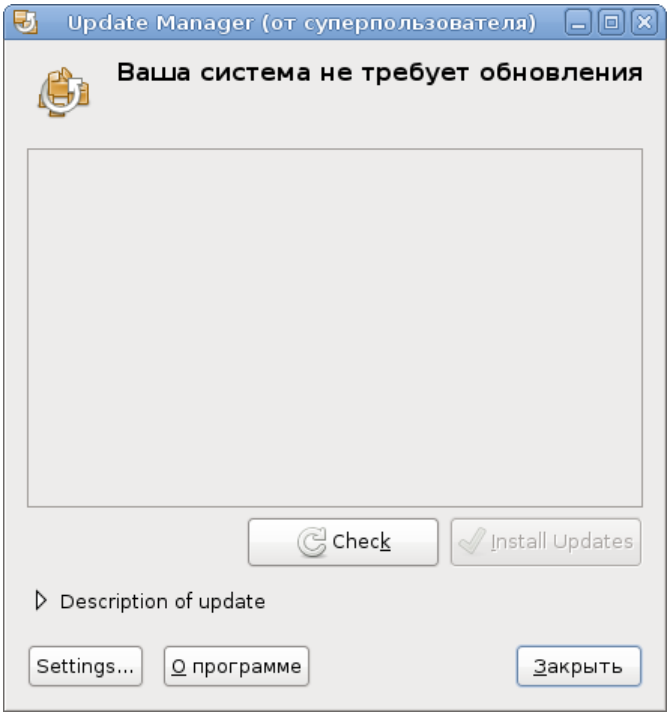

Рис. 8.2. Изображение окна программы Update Manager

Другие менеджеры пакетов - rpm, yum. Программа alien

Мы рассмотрели основу работы с менеджерами пакетов в Debian-подобных дистрибутивах. Также, достаточно большую группу дистрибутивов составляют дистрибутивы, которые используют менеджер пакетов RPM, разработанный для дистрибутива Red Hat. Подробно работу с этим менеджером пакетов мы рассматривать не будем. Дадим только общую информацию.

Пакеты в менеджере пакетов RPM представляют собой файлы с расширением `.rpm`, которые являются архивами. Для работы с такими файлами пакетов существует утилита `rpm`, по своим возможностям и синтаксису похожая на утилиту `dpkg`. Также существует утилита более высокого уровня `yum`, позволяющая обрабатывать зависимости между пакетами и работать с сетевыми репозиториями. В целом функциональность `yum` близка к функциональности `apt`. Кроме этого существует утилита `apt-rpm`, позволяющая работать с пакетами `rpm` с помощью утилиты `apt`.

Достаточно часто возникает ситуация, когда нужно преобразовать тип пакета, например, из пакета `rpm` получить пакет `deb`, и наоборот. Для решения таких задач предназначена утилита `alien`.

Для преобразования пакета `rpm` в пакет `deb` используется следующий синтаксис:

```
$ sudo alien --to-deb --scripts имя_пакета_rpm
```

При этом будет создан пакет с таким же именем, но в формате и с расширением `deb`. Ключ `--scripts` указывает на то, что нужно выполнить преобразование установочных скриптов.

Обратное преобразование (из `deb` в `rpm`) выполняется командой:

```
$ sudo alien --to-rpm --scripts имя_пакета_deb
```

Компиляция программ из исходных кодов

Устанавливать программы с помощью менеджеров пакетов очень удобно, однако может возникнуть ситуация, когда программа, либо ее последняя версия, отсутствует в хранилищах пакетов. В этом случае пользователю придется прибегнуть к самостоятельной компиляции программы из исходного кода и ее установке. Данный способ следует применять только при невозможности установки программы из репозитория пакетов, поскольку в этом случае пользователю придется самому отвечать за процесс установки и возможного удаления программы. При этом ему, возможно, придется самому отслеживать, какие файлы были установлены, и какие из них нужно удалить при удалении программы.

Обычная последовательность действий при компиляции программы из исходных кодов состоит в выполнении следующих трех этапов:

1) конфигурация программы,
2) компиляция программы,
3) установка программы.

Чаще всего эти три шага соответствуют следующей последовательности команд:

```
$ ./configure
$ make
$ sudo make install
```

Вроде бы все просто. Но это только в теории. На практике все сложнее, и вот почему. Во-первых, последовательность компиляции в каждом конкретном случае может несколько отличаться, поэтому перед компиляцией нужно сначала изучить файл `readme`. Во-вторых, первые два шага с достаточно большой степенью вероятности могут закончиться неудачей из-за отсутствия или несоответствия версий других программ, которые требуются для компиляции. Поэтому фактически процесс компиляции состоит в том, что приходится разбираться в возникающих ошибках и устранять их.

Рассмотрим пример. Выполним компиляцию и установку последней версии (на момент разработки курса 2.6.2) программы `minicom`. Данная программа предназначена для обмена данными через последовательные порты компьютера.

Скачаем файл с исходными кодами с сайта http://alioth.debian.org/projects/minicom и распакуем его в отдельный каталог.

Запускаем первую команду (приведена только часть вывода команды):

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
…
config.status: creating man/Makefile
config.status: creating lib/Makefile
config.status: creating src/Makefile
config.status: creating po/Makefile.in
config.status: creating minicom.spec
config.status: creating config.h
config.status: executing depfiles commands
config.status: executing po-directories commands
config.status: creating po/POTFILES
config.status: creating po/Makefile
```

Далее запускаем команду make (приведена только часть вывода команды):

```
$ make
make  all-recursive
make[1]:  Entering  directory  `/home/user/minicom/minicom-
2.6.2'
Making all in doc
make[2]:  Entering  directory  `/home/user/minicom/minicom-
2.6.2/doc'
make[2]: Цель `all' не требует выполнения команд.
make[2]:  Leaving  directory  `/home/user/minicom/minicom-
2.6.2/doc'
Making all in extras
```

```
    make[2]: Entering directory `/home/user/minicom/minicom-
2.6.2/extras'
    Making all in linux
    make[3]: Entering directory `/home/user/minicom/minicom-
2.6.2/extras/linux'
    …
    mv -f .deps/script.Tpo .deps/script.Po
    gcc -g -O2 -W -Wall -Wextra -std=gnu99 -o runscript
script.o sysdep1_s.o common.o ../lib/libport.a -lncurses
    gcc -DHAVE_CONFIG_H -I. -I.. -I../lib -
DCONFDIR=\"/usr/local/etc\" -
DLOCALEDIR=\"/usr/local/share/locale\" -g -O2 -W -Wall -Wextra
-std=gnu99 -MT ascii-xfr.o -MD -MP -MF .deps/ascii-xfr.Tpo -c -o
ascii-xfr.o ascii-xfr.c
    mv -f .deps/ascii-xfr.Tpo .deps/ascii-xfr.Po
    gcc -g -O2 -W -Wall -Wextra -std=gnu99 -o ascii-xfr
ascii-xfr.o ../lib/libport.a -lncurses
    make[2]: Leaving directory `/home/user/minicom/minicom-
2.6.2/src'
    make[2]: Entering directory `/home/user/minicom/minicom-
2.6.2'
    make[2]: Цель `all-am' не требует выполнения команд.
    make[2]: Leaving directory `/home/user/minicom/minicom-
2.6.2'
    make[1]: Leaving directory `/home/user/minicom/minicom-
2.6.2'
```

И, наконец, последняя команда (приведена только часть вывода команды):

```
    $ sudo make install
    Making install in doc
    make[1]: Entering directory `/home/user/minicom/minicom-
2.6.2/doc'
    make[2]: Entering directory `/home/user/minicom/minicom-
2.6.2/doc'
    make[2]: Цель `install-exec-am' не требует выполнения
команд.
    make[2]: Цель `install-data-am' не требует выполнения
команд.
    make[2]: Leaving directory `/home/user/minicom/minicom-
2.6.2/doc'
```

```
     make[1]:  Leaving  directory  `/home/user/minicom/minicom-
2.6.2/doc'
     Making install in extras
     make[1]:  Entering  directory  `/home/user/minicom/minicom-
2.6.2/extras'
     Making install in linux
     …
     make[2]:  Entering  directory  `/home/user/minicom/minicom-
2.6.2/src'
      /bin/mkdir -p '/usr/local/bin'
       /usr/bin/install   -c   minicom   runscript   ascii-xfr
'/usr/local/bin'
      /bin/mkdir -p '/usr/local/bin'
      /usr/bin/install -c xminicom '/usr/local/bin'
     make[2]:  Цель  `install-data-am'  не  требует  выполнения
команд.
     make[2]:  Leaving  directory  `/home/user/minicom/minicom-
2.6.2/src'
     make[1]:  Leaving  directory  `/home/user/minicom/minicom-
2.6.2/src'
     make[1]:  Entering  directory  `/home/user/minicom/minicom-
2.6.2'
     make[2]:  Entering  directory  `/home/user/minicom/minicom-
2.6.2'
     make[2]:  Цель  `install-exec-am'  не  требует  выполнения
команд.
     make[2]:  Цель  `install-data-am'  не  требует  выполнения
команд.
     make[2]:  Leaving  directory  `/home/user/minicom/minicom-
2.6.2'
     make[1]:  Leaving  directory  `/home/user/minicom/minicom-
2.6.2'
```

Все три шага компиляции и установки выполнены. Теперь мы можем проверить номер версии программы minicom, которая установлена в системе:

```
     $ minicom --version
     minicom версия 2.6.2 (откомпилирован Apr  5 2013)
     Copyright (C) Miquel van Smoorenburg.

     This program is free software; you can redistribute it
and/or
     modify it under the terms of the GNU General Public License
     as published by the Free Software Foundation; either version
```

```
2 of the License, or (at your option) any later version.
```

Как видим, номер версии равен `2.6.2`, т.е. программа успешно скомпилирована и установлена.

Ранее мы уже подчеркивали, что ручная компиляция и установка программ являются нежелательным. Одним из негативных факторов здесь является сложность учета установленных файлов при удалении установленной программы.

К счастью, для решения данной проблемы разработана программа `checkinstall`. Она выполняется вместо команды `make install` и позволяет вместо непосредственной установки в систему сначала создать пакет в одном из известных форматов, а потом установить созданный пакет с помощью пакетного менеджера. Тем не менее, созданный пакет предназначен только для использования на собственном компьютере и не предназначен для передачи другим пользователям.

Для создания пакета формата `deb` нужно выполнить следующую команду:

```
$ sudo checkinstall –D
```

В нашем случае мы получим:

```
$ sudo checkinstall -D

checkinstall 1.6.2, Copyright 2009 Felipe Eduardo Sanchez Diaz Duran
          Эта программа распространяется на условиях GNU GPL


The package documentation directory ./doc-pak does not exist.
Should I create a default set of package docs?  [y]: n

**************************************
**** Debian package creation selected ***
**************************************

Этот пакет был создан с использованием данных значений:
```

```
0 -  Maintainer: [ root@dlink-debian ]
1 -  Summary: [ Пакет для minicom после ручной компиляции. ]
2 -  Name:    [ minicom ]
3 -  Version: [ 2.6.2 ]
4 -  Release: [ 1 ]
5 -  License: [ GPL ]
6 -  Group:   [ checkinstall ]
7 -  Architecture: [ i386 ]
8 -  Source location: [ minicom-2.6.2 ]
9 -  Alternate source location: [  ]
10 - Requires: [  ]
11 - Provides: [ minicom ]
12 - Conflicts: [  ]
13 - Replaces: [  ]
```

Введите номер для изменения параметра или нажмите ВВОД для продолжения:

Installing with make install...

========== Результаты установки ==========

```
Making install in doc
make[1]: Entering directory `/home/user/minicom/minicom-2.6.2/doc'
make[2]: Entering directory `/home/user/minicom/minicom-2.6.2/doc'
make[2]: Цель `install-exec-am' не требует выполнения команд.
make[2]: Цель `install-data-am' не требует выполнения команд.
make[2]: Leaving directory `/home/user/minicom/minicom-2.6.2/doc'
…
make[1]: Entering directory `/home/user/minicom/minicom-2.6.2'
make[2]: Entering directory `/home/user/minicom/minicom-2.6.2'
make[2]: Цель `install-exec-am' не требует выполнения команд.
make[2]: Цель `install-data-am' не требует выполнения команд.
make[2]: Leaving directory `/home/user/minicom/minicom-2.6.2'
```

```
     make[1]: Leaving directory `/home/user/minicom/minicom-
2.6.2'

     ========================= Установка успешно завершена
======================

     Файлы копируются во временный каталог...OK

     Stripping ELF binaries and libraries...OK

     Сжимаются страницы руководства...OK

     Построение списка файлов...OK

     Собирается Debian-пакет...OK

     Устанавливается Debian-пакет...OK

     Удаляются временные файлы...OK

     Записывается пакет с резервной копией...OK
     OK

     Удаляется временный каталог...OK


     ************************************************************
**********

      Done. The new package has been installed and saved to

      /home/user/minicom/minicom-2.6.2/minicom_2.6.2-1_i386.deb

      You can remove it from your system anytime using:

           dpkg -r minicom

     ************************************************************
**********
```

Из листинга видно, что в результате выполнения команды `checkinstall` был создан пакет `minicom_2.6.2-1_i386.deb`, который был установлен в системе.

Глава 8. Компилятор GCC

Общие сведения о компиляторе GCC

В названии и тексте главы мы с вами будем использовать термин “компилятор GCC”. Однако это не совсем верно. GCC – это *набор компиляторов* с разных языков программирования, созданный и поддерживаемый Фондом свободного программного обеспечения в рамках проекта GNU. Первоначально аббревиатура GCC расшифровывалась как GNU C Compiler (Компилятор языка C проекта GNU). Позже GCC стали трактовать как **GNU Compiler Collection** (Набор компиляторов проекта GNU).

Разработка GCC начата Ричардом Столлманом в 1985 г. Первая версия была выпущена в 1987 г.

Компилятор достаточно сложная программная система, которая обладает множеством характеристик – набором поддерживаемых языковых конструкций, скоростью компиляции, оптимальностью получаемого кода по памяти и быстродействию, набором архитектур, для которых генерируется код и т.п. В зависимости от целей разработки компилятора приоритет может отдаваться тем или иным характеристикам (естественно ограничивая при этом оставшиеся характеристики). При разработке GCC основным приоритетом была *универсальность*, как по числу поддерживаемых языков программирования, так и по числу аппаратных архитектур, для которых выполняется генерация кода. GCC является лидером по количеству поддерживаемых аппаратных платформ. В настоящее время поддерживается более 60 аппаратных платформ, в числе которых как традиционные для персональных компьютеров `i386` и `ia64`, так и распространенные во встроенных применениях платформы `arm`, `avr`, `mips`, `powerpc` и др. Если в числе ваших приоритетов переносимость, то на GCC стоит обратить внимание.

Компилятор GCC может выполнять компиляцию программ, написанных на языках C, C++, Objective C, Fortran, Java, Ada и Go. Так же сторонними разработчиками поддерживаются дополнительные модули для компиляции программ на Pascal, Modula-2, Modula-3, Cobol, PL/1 и некоторых других языках.

Компилятор GCC является стандартным компилятором в Linux, а так же в некоторых других Unix-подобных операционных системах – FreeBSD, OpenBSD, OpenSolaris, MacOS X и др. Кроме этого, существует **MinGW** – портированная версия GCC для Microsoft Windows, которая кроме самого компилятора содержит интерфейс для взаимодействия с Windows API.

Компиляция простейшей программы

Рассмотрим простейшие приемы компиляции программ на языке C. Создадим файл `simple.c` со следующим содержанием:

```
#include <stdio.h>
int main( int argc, char* argv[] ){
  printf("Пример простейшей программы.\n");
  return 0;
}
```

Выполнить компиляцию данной программы можно командой

```
$ gcc simple.c
```

После выполнения данной команды в текущем каталоге появится файл с именем `a.out` – это скомпилированный исполняемый файл для нашей программы, которому было дано имя по умолчанию.

```
$ ls
a.out  simple.c
$ ./a.out
Пример простейшей программы.
```

Ясно, что удобней было бы дать исполняемому файлу какое-либо осмысленное имя. Это имя можно задать в параметре ключа `-o` в команде:

```
$ gcc simple.c -o simple.out
$ ls
a.out  simple.c  simple.out
$ ./simple.out
```

Пример простейшей программы.

Ясно, что реальные программы чаще всего состоят из нескольких файлов исходного кода. Перед тем, как перейти к рассмотрению компиляции сложных программ, рассмотрим сначала этапы компиляции программы на языке С.

Структура компилятора: препроцессор, компилятор, ассемблер, компоновщик

Когда мы набираем команду `gcc`, то этой командой мы запускаем только общий командный интерфейс компилятора, который потом вызывает служебные программы, выполняющие различные этапы компиляции. Детали процесса компиляции и параметры выполнения служебных программ можно посмотреть, запустив команду `gcc` с ключом `-v`.

Компиляция программы на языке С состоит из следующих этапов:

- обработка препроцессором,
- собственно компиляция,
- ассемблирование,
- компоновка.

Структура процесса компиляции программы на языке С, показана на рис. 8.1.

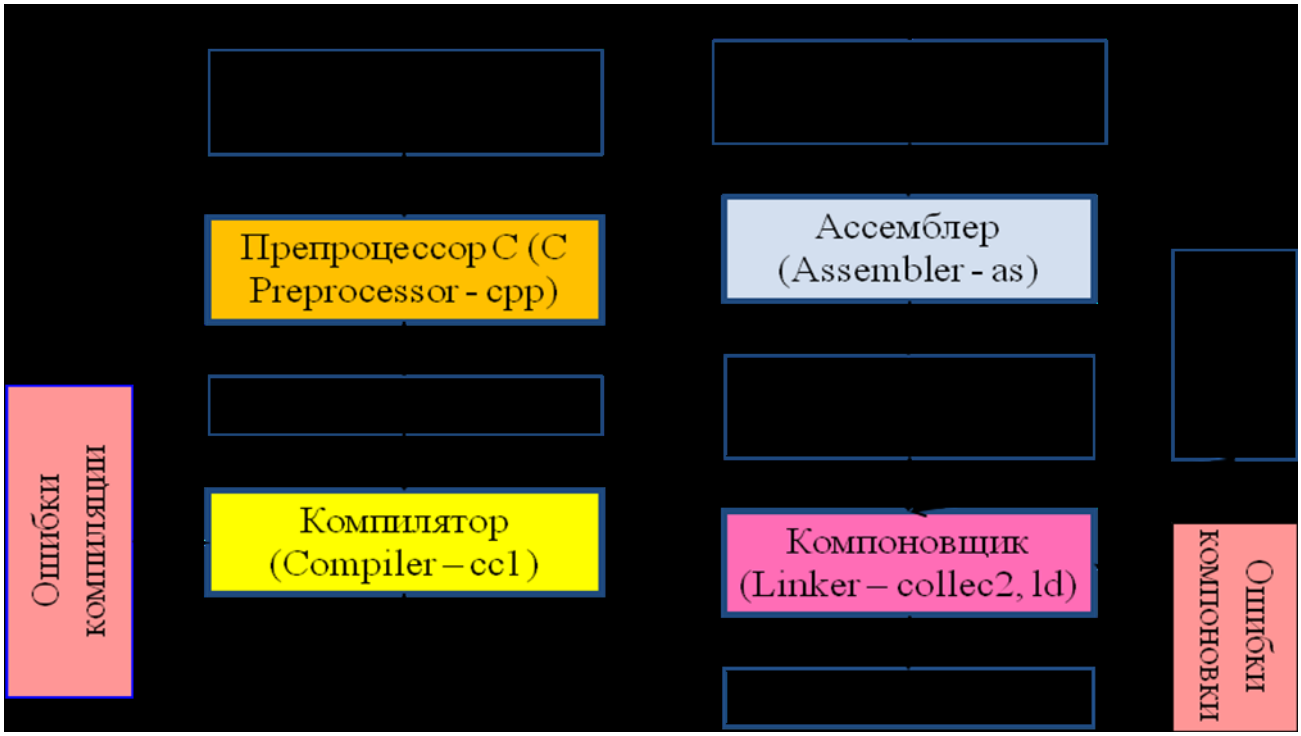

Рис. 8.1. Структура процесса компиляции программы на языке C в GCC

На рис. 8.1 видно, что кроме файлов исходного кода и исполняемых файлов, в процессе компиляции создаются еще файлы трех промежуточных типов. Типы файлов, с которыми может работать компилятор gcc при компиляции программ на языке C, описаны в табл. 8.1 [8].

Табл. 8.1. Типы файлов, с которыми работает gcc при компиляции программ на языке C

Расширение файла	Описание
`.a`	Статическая объектная библиотека (архив)
`.c`	Исходный код на языке C, подлежащий обработке препроцессором
`.h`	Исходный код заголовочного файла на языке C
`.i`	Исходный код на языке C не подлежащий обработке препроцессором. Файлы данного типа являются промежуточным результатом компиляции
`.o`	Объектный файл в формате, поддерживаемом компоновщиком. Файлы данного типа являются промежуточным результатом компиляции
`.s`	Ассемблерный код. Файлы данного типа являются промежуточным результатом компиляции

`.so`	Разделяемая объектная библиотека

Компилятор gcc позволяет нам выполнить не всю компиляцию, а только несколько первых этапов. При этом мы можем просмотреть промежуточные результаты.

Для того чтобы выполнить только *препроцессорную обработку*, нужно указать в команде ключ `-E`. В результате мы получим исходный код на языке C, прошедший обработку препроцессором, в нашем случае перед текстом программы будет добавлено содержимое заголовочного файла `stdio.h`. Полученный в результате файл `stage1.txt` получается достаточно объемным, в листинге приводится только его фрагмент.

```
$ gcc -E simple.c -o stage1.txt
$ cat stage1.txt
# 1 "simple.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "simple.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 313 "/usr/include/features.h" 3 4
# 1 "/usr/include/bits/predefs.h" 1 3 4
# 314 "/usr/include/features.h" 2 3 4
# 346 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 353 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 354 "/usr/include/sys/cdefs.h" 2 3 4
# 347 "/usr/include/features.h" 2 3 4
# 378 "/usr/include/features.h" 3 4
# 1 "/usr/include/gnu/stubs.h" 1 3 4
…
```

Если мы хотим выполнить два первых шага компиляции – *препроцессорную обработку* и *собственно компиляцию*, получив на выходе код программы на языке ассемблера, то нужно указать ключ `-S`.

```
$ gcc -S simple.c -o simple.s
$ cat simple.s
    .file   "simple.c"
    .section .rodata
    .align 4
.LC0:
    .string
"\320\237\321\200\320\270\320\274\320\265\321\200
\320\277\321\200\320\276\321\201\321\202\320\265\320\27
1\321\210\320\265\320\271
\320\277\321\200\320\276\320\263\321\200\320\260\320\27
4\320\274\321\213."
    .text
.globl main
    .type   main, @function
main:
    pushl   %ebp
    movl %esp, %ebp
    andl $-16, %esp
    subl $16, %esp
    movl $.LC0, (%esp)
    call puts
    movl $0, %eax
    leave
    ret
    .size   main, .-main
    .ident  "GCC: (Debian 4.4.5-8) 4.4.5"
    .section .note.GNU-stack,"",@progbits
```

Ну и если мы хотим выполнить три первых этапа компиляции без выполнения компоновки, получив в результате объектный файл, нужно указать в команде ключ `-c`.

```
$ gcc -c simple.c
$ ls
simple.c  simple.o
```

Последний вариант команды gcc (с ключом `-c`) используется достаточно часто.

Компиляция программ на языке С

После рассмотрения этапов компиляции программ на языке С мы можем вернуться к рассмотрению методов компиляции программ. Далее в этом пункте мы рассмотрим несколько типовых ситуаций, возникающих при компиляции.

1) Компиляция программы, состоящей из нескольких файлов исходного кода

Рассмотрим данную ситуацию на следующем примере.

```
$ cat msgfunc.c
#include <stdio.h>
void print_msg(){
  printf("Пример программы из нескольких файлов.\n");
  return;
}
$ cat main.c
#include "stdio.h"
void print_msg();
int main( int argc, char* argv[] ){
  print_msg();
  return 0;
}
$ gcc main.c msgfunc.c -o complex.out
$ ./complex.out
Пример программы из нескольких файлов.
```

Из листинга видно, что для компиляции программы из нескольких файлов достаточно указать все эти файлы в качестве параметров команды gcc. Однако рассмотренный вариант в некотором смысле является не самым удачным. Дело в том, что при этом выполняется перекомпиляция всех

программных файлов. Если же изменялись только некоторые из них, то удобнее применять другую методику компиляции – сначала получить для каждого файла объектный файл, затем выполнить компоновку объектных файлов. В этом случае при повторной компиляции нужно будет перекомпилировать только те файлы, которые изменялись.

```
$ gcc msgfunc.c -c
$ gcc main.c -c
$ gcc msgfunc.o main.o -o complex.out
$ ./complex.out
Пример программы из нескольких файлов.
```

2) Компиляция программы, использующей статические библиотеки[12]

Рассмотрим следующую программу (файл `sin.c`).

```
/*   sin.c   */
#include<stdio.h>
#include<math.h>
int main(int argc, char* argv[]) {
  float x, y;
  printf("x=? : ");
  scanf("%f", &x);
  printf("x = %f\n", x);
  y = sin(x);
  printf("sin(x) = %f\n", y);
  return 0;
}
```

В ней используется функция `sin()`, описанная в заголовочном файле `math.h`. Попробуем скомпилировать программу обычным способом:

```
$ gcc sin.c
/tmp/ccFEeYgH.o: In function `main':
```

[12] Способом, рассмотренным в этом разделе можно так же использовать разделяемые библиотеки

```
sin.c:(.text+0x48): undefined reference to `sin'
collect2: ld returned 1 exit status
```

Как видим, компиляция завершилась неудачей – компоновщик (`collect2`) не смог найти местонахождение функции `sin()`. Данная функция содержится в файле статической библиотеки `libm.a`. Для успешной компиляции программы нужно дать указать компилятору на использование данного файла библиотеки. Для этого существует специальный параметр `-l<имя>`, где `<имя>` - имя файла библиотеки без трех первых букв `lib` и расширения файла (по существующему соглашению все файлы статических библиотек начинаются с букв `lib`, поэтому их можно опустить). Таким образом, нам нужно указать в команде ключ `-lm`.

```
$ gcc sin.c -lm -o sin.out
$ ./sin.out
x=? : 3.1415926
x = 3.141593
sin(x) = 0.000000
```

3) Статические и разделяемые (динамические) библиотеки

Кроме исполняемых файлов с помощью компилятора gcc (и дополнительных утилит) мы можем скомпилировать еще два вида библиотек подпрограмм – *статические* библиотеки (`*.a`) и *разделяемые* (динамические) библиотеки (`*.so`).

Если мы хотим в своей программе использовать *статическую* библиотеку, то она нам понадобится уже на стадии компиляции. Мы уже рассматривали пример со статической библиотекой `libm.a`. Недостатком статических библиотек является то, что содержащийся в них код включается в каждый исполнимый файл, который использует библиотеку. Если одну и ту же библиотеку использует сразу несколько программ, то каждая из них содержит копию используемой библиотеки, что негативно отражается на потреблении памяти.

Если мы хотим использовать в программе *разделяемую* библиотеку, то она нам потребуется только на стадии выполнения программы. Важным достоинством разделяемых библиотек является то, что в памяти хранится

только одна копия разделяемой библиотеки вне зависимости от количества программ, которые с ней работают.

4) Компиляция статической библиотеки

Статическая библиотека подпрограмм, по сути, представляет собой набор объектных файлов, которые помещаются в единый архив. Соответственно нужно получить объектные файлы для включаемых в библиотеку подпрограмм, а потом сформировать из них статическую библиотеку с помощью специальной утилиты `ar`. Рассмотрим следующий пример. Пусть у нас есть два файла `filelib1.c` и `filelib2.c`, содержащие функции `func1()` и `func2()`. Создадим статическую библиотеку `libexample.a`.

```
$ cat filelib1.c
#include<stdio.h>
int func1( int i ){
  printf("Вызвана функция func1() из библиотеки libexample.a.\n");
  printf(" i=%d, i*i=%d\n", i, i*i);
  return i * i;
}
$ cat filelib2.c
#include<stdio.h>
int func1( int i ){
  printf("Вызвана функция func2() из библиотеки libexample.a.\n");
  printf(" i=%d, i+5=%d\n", i, i+5);
  return i + 5;
}
$ gcc -c filelib1.c filelib2.c
$ ar -r libexample.a filelib1.o filelib2.o
ar: creating libexample.a
$ ls
filelib1.c   filelib1.o   filelib2.c   filelib2.o
libexample.a
```

Как видим, в результате статическая библиотека `libexample.a` создана.

Заметим, что имя файла статической библиотеки традиционно начинается с трех букв `lib`. Как правило, большинство статических библиотек располагается в каталоге `/usr/lib/` .

Проверим, насколько работоспособна созданная нами библиотека. Для этого попробуем ее использовать в другой программе.

```
$ cat testlib.c
int func1( int );
int func2( int );
int main( int argc, char* argv[] ){
  func1(3);
  func2(3);
  return 0;
}
$ gcc testlib.c –lexample –L. -o testlib.out
$ ./testlib.out
Вызвана функция func1() из библиотеки libexample.a.
 i=3, i*i=9.
Вызвана функция func2() из библиотеки libexample.a.
 i=3, i+5=8.
```

Параметр `–L.` указанный в команде gcc определяет, что поиск библиотеки начинается с текущего каталога (на текущий каталог указывает точка, стоящая в параметре). Как видно из листинга, созданная статическая библиотека вполне работоспособна.

5) Компиляция разделяемой библиотеки

В предыдущем пункте нами была рассмотрена методика создания статических библиотек. Теперь создадим разделяемую библиотеку для того же самого примера. В нашем случае при компиляции файлов исходного кода нужно указать параметр `–fpic` (от Position Independent Code) для того, чтобы был скомпилирован код, работающий независимо от физического расположения кода и данных в памяти.

```
$ gcc -fpic -c filelib1.c filelib2.c
```

Далее при окончательной компиляции нужно указать параметр `-shared`, который определяет, что будет создана разделяемая библиотека.

```
$ gcc -shared filelib1.o filelib2.o -o libsharedexample.so
$ ls
filelib1.c filelib1.o filelib2.c filelib2.o libsharedexample.so
```

Как видно из листинга, разделяемая библиотека `libsharedexample.so` успешно создана.

6) Указание используемого стандарта языка C

В заключение данного раздела рассмотрим еще один параметр, касающийся компиляции программ на языке C. Это параметр `-std`, определяющий в соответствии с каким из стандартов языка C выполнять компиляцию. Формат параметра: `-std=стандарт`, где в качестве стандарта может стоять одна из следующих констант: `c89` (стандарт ISO C89), `c99` (стандарт ISO C99), `gnu89` (ISO C89 с некоторыми расширениями GNU), `c11` (стандарт ISO C11). Например, выполнить компиляцию файла `simple.c` в соответствии с требованиями стандарта C99 можно следующей командой:

```
$ gcc -std=c99 simple.c
```

Другие полезные опции компилятора

В этом разделе мы рассмотрим основные ключи компилятора, которые *не зависят от языка программирования* и используются достаточно широко.

1) Включение отладочной информации

Если вы собираетесь отлаживать программу, используя какой-либо отладчик, то в исполняемый файл необходимо включить *отладочную информацию*. Для этого в команде надо указать ключ `-g`.

Кроме этого существует несколько дополнительных ключей компилятора, определяющих, какой из существующих форматов отладочной информации нужно использовать. Например, при указании ключа `-ggdb3` будет сформирована отладочная информация в стандартном для системы формате, а так же расширенная отладочная информация в формате отладчика `gdb` (число 3 означает третий, наиболее полный, уровень отладочной информации).

2) Параметры оптимизации кода

При генерации исполняемого файла компилятор может тем или иным способом попытаться выполнить оптимизацию кода программы. Целями такой оптимизации могут быть:

- ускорение работы программы (за счет уменьшения числа выполняемых операторов и их перегруппировки),
- уменьшение объема используемой оперативной памяти,
- уменьшение размера исполняемого файла в байтах.

При использовании оптимизации необходимо помнить, что сама оптимизации состоит во внесении изменений в исходную программу, поэтому если вы собираетесь выполнять отладку, то никаких видов оптимизации выполнять, как правило, *не следует*. Если вы все же скомпилируете программу, включив туда отладочную информацию и выполнив оптимизацию, то результат выполнения программы под отладчиком вас, скорее всего, очень удивит. Поэтому лучше взять за правило:

Или отладка, или оптимизация, но не и то и другое одновременно.

Если вы хотите выполнить генерацию исполняемого файла без оптимизации, например, для того, чтобы получить ожидаемые результаты во время отладки, нужно указать ключ `-O0`. В прочем, данный ключ можно и не указывать, он действует по умолчанию.

Для того чтобы выполнить оптимизацию, нужно указать один из ключей: `-O1`, `-O2`, `-O3`, `-Os`.

Ключ `-O1` включает базовый набор методов оптимизации (оптимизация уровня 1).

Ключ `-O2` включает все методы оптимизации уровня 1, а так же ряд дополнительных методов.

Ключ `-O3` включает все методы оптимизации уровня 2, а так же некоторые другие методы.

При переходе от первого уровня оптимизации ко второму, и от второго к третьему время компиляции растет, а эффективность оптимизации увеличивается.

Кроме рассмотренных ключей оптимизации `-O1` - `-O3` существует еще ряд дополнительных ключей, которые указывают на использование какого-либо конкретного приема оптимизации.

Существует также еще один флаг `-Os`, позволяющий уменьшить объем исполняемого файла. Данный флаг активизирует все методы 2 уровня оптимизации (как при использовании флага `-O2`), добавляя к ним ряд методов, направленных специально на уменьшение объема исполняемого файла. Рассматриваемый флаг является полезным при разработке программ для встроенных систем, в которых место на устройстве хранения данных является достаточно ограниченным ресурсом.

3) Параметры выдачи предупреждений

Если вы попытаетесь выполнить компиляцию программы, которая содержит ошибки, то компилятор выдаст вам список ошибок. Даже если формально в программе нет ошибок, компилятор может выдавать *предупреждения* (*warnings*), сообщающие о том, что в программе применяются какие-либо не рекомендуемые средства или методы программирования, например, используются функции считающиеся небезопасными. В большинстве случаев программисту стоит обратить внимание на предупреждения компилятора, в некоторых случаях их можно игнорировать.

Программист может указывать компилятору, какие из предупреждений следует выдавать программисту. Обычно, ключ управляющий выдачей предупреждений некоторого типа имеет формат `-Wпредупреждение`. Например, ключ `-Wunused-function` указывает компилятору на выполнение проверки и выдачи предупреждения для функций, которые определены, но в программе не используются.

Достаточно часто используется ключ `-Wall`, указывающий на то, что должно выдаваться большинство существующих предупреждений. Еще больше предупреждений выдается при указании ключа `-Wextra`.

Компиляция программ на языке C++

Еще одним языком программирования, для компиляции программ на котором, широко применяется компилятор gcc, является язык программирования C++. При работе с программами на языке C++ добавляется еще два используемых формата файлов: файлы `*.cpp`, содержащие исходный код программы на языке C++ подлежащий обработке препроцессором, и файлы `*.ii`, содержащие исходный код программы на языке C++ не подлежащий обработке препроцессором.

Для компиляции программ на C++ у компилятора gcc существует отдельный командный интерфейс, запускаемый командой `g++`. В принципе, он не сильно отличается от основного интерфейса gcc. В g++ добавлено использование стандартной библиотеки языка C++ по умолчанию.

Ниже приведен пример компиляции простой программы:

```
$ cat simple.cpp
#include<iostream>
int main( int argc, char* argv[] ){
  std::cout << "Пример простой программы." << std::endl;
  return 0;
}
$ g++ simple.cpp -o simple.out
$ ./simple.out
Пример простой программы.
```

Структура компилятора gcc

Как упоминалось ранее, компилятор gcc поддерживает достаточно много языков программирования и целевых платформ. Естественно, для реализации данных требований нужно было адаптировать под них архитектуру компилятора. Самое интересное то, что, по сути, компилятор gcc

является расширяемым. И любой программист может добавить в компилятор gcc поддержку нового языка программирования или новой целевой платформы.

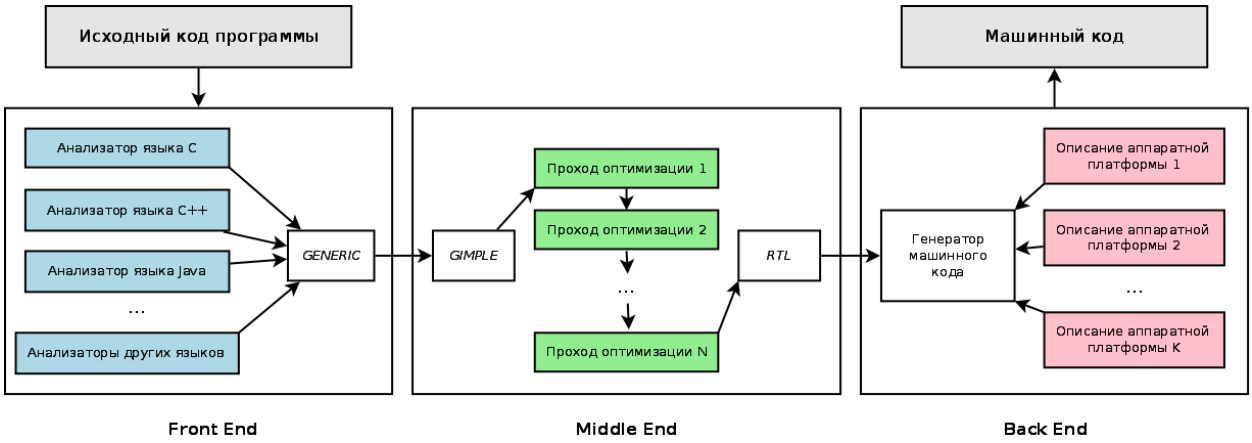

Рис. 8.2. Структура компилятора gcc

Структура компилятора gcc изображена на рис. 8.2. Компилятор gcc изначально построен таким образом, чтобы можно было сравнительно просто добавлять к ядру компилятора дополнительные модули, которые тем или иным образом расширяют его функциональность. Эти модули могут относиться к одному из трех типов:

- модули, которые реализуют поддержку новых языков программирования (показаны на рисунке голубым цветом),
- модули, которые реализуют новые методы оптимизации программ (показаны на рисунке зеленым цветом),
- модули, которые реализуют поддержку новой целевой платформы (показаны на рисунке розовым цветом).

В целом, в структуре компилятора gcc присутствуют три укрупненные части: Front End, Middle End и Back End.

Front End реализует все операции, которые зависят от языка программирования: лексический, синтаксический и семантический анализ и преобразование в промежуточную форму, представляющую собой синтаксическое дерево и называемую GENERIC.

Middle End реализует операции, которые не зависят ни от языка программирования, ни от целевой платформы. Сначала здесь выполняется преобразование программы в промежуточную форму, использующую структуру данных GIMPLE. После этого выполняется несколько проходов

оптимизации. В gcc может выполняться более 20 проходов оптимизации. После этого программа преобразуется в форму, удобную для генерации кода и называемую RTL.

Back End реализует операции, зависящие от целевой платформы. Здесь выполняется генерация машинного кода на основе используемого описания аппаратной платформы (Machine Description).

Глава 9. Утилита make

Понятие о системах сборки

В предыдущей главе мы с вами рассмотрели, как, используя компилятор gcc, можно скомпилировать сравнительно простые программы. При этом во многих случаях для компиляции программы нам требовалось выполнить не одну, а несколько команд. В сложных программах для их полной сборки может потребоваться выполнить десятки и даже сотни команд. Конечно, можно было бы записать эти команды в скрипт и выполнять его. Сложность такого метода состоит в том, что все команды должны идти в определенной последовательности, т.к. результаты выполнения некоторых из них необходимы для того, чтобы выполнить другие команды. Такой файл будет непросто написать, а еще сложнее – его сопровождать.

К счастью, есть метод лучше – использовать специализированные системы сборки. Системы сборки сами определят порядок выполнения команд. Они так же способны определить, какие файлы исходных текстов программы были изменены с момента последней компиляции, и перекомпилировать только их.

Можно было бы возразить, что современные IDE выполняют сборку автоматически. Однако система сборки является самостоятельным и весьма полезным инструментом, особенно для крупных проектов. Например, для языка Java существуют одни из лучших IDE – Eclipse и NetBeans. В то же время для Java существует и мощная система сборки – Apache Ant.

Кроме собственно сборки программы из исходных текстов системы сборки позволяю автоматизировать любые операции, которые требуется выполнять в порядке, определяемом зависимостями по данным между ними. Обычно системы сборки используются для:

- конфигурации программы – определения, какие именно функции сложной программы нужно включать в создаваемый исполняемый файл,
- компиляции программы из исходных текстов,
- установки программы - записи созданных программных файлов в целевые каталоги,

- удаления временных файлов, созданных при компиляции и т.п.

Утилита make

Исторически первой системой сборки является традиционная для Unix-подобных систем утилита make, первая версия которой написана сотрудником Bell Labs Стюартом Фельдманом в 1977 году. Несмотря на то, что у данной системы сборки есть определенные недостатки, она все еще применяется во многих существующих проектах, в том числе и таких сложных, как ядро Linux. Существует несколько реализаций make. В этом курсе мы будем подразумевать, что используется GNU make.

Утилита make обладает двумя важными характеристиками:

- она автоматически определяет порядок, в котором нужно выполнить команды для сборки программы, исходя из условия, что файлы, которые требуются для компиляции на каждом этапе, должны быть получены на предыдущих этапах,
- она предотвращает повторную компиляцию тех файлов, которые не изменялись с момента предыдущей компиляции.

Строго говоря, область применения утилиты make не ограничивается только задачей сборки программ – она может использоваться в любых задачах, в которых нужно обновлять файлы при изменении некоторых других файлов.

Введение в написание make-файлов

Для использования утилиты make нужно в рабочем каталоге программы создать файл с именем `Makefile`, содержащий инструкции по сборке программы в специальном формате. В этом случае сборка будет производиться простым вызовом команды `make`, возможно с некоторыми параметрами.

Makefile представляет собой текстовый файл и состоит из набора правил следующего формата:

```
цель: зависимости
    команда
```

команда
…

Каждое правило начинается с указателя *цель*. Цель может быть *именем файла*, который необходимо получить, а может просто обозначением некоторой операции, не совпадающим с именем файла (в этом случае говорят об *абстрактной цели*).

Далее после двоеточия через пробел перечисляются *зависимости* – определяющие действия, которые должны быть выполнены заранее для получения цели. Обычно в качестве зависимости указывается имя файла или имя другой цели.

После зависимостей указываются *команды*. Каждая команда указывается в своей строке.

При этом перед каждой командой в начале строки должен стоять знак **табуляции**. Это особенность программы make нужно запомнить и выполнять, т.к. без символа табуляции сборка не выполнится из-за ошибки.

Рассмотрим создание make-файла на примере сборки программы, состоящей из нескольких файлов, рассмотренной в предыдущей главе. Скопируем файлы sayhello.c и hellomain.c в отдельный каталог, а также создадим там `Makefile` следующего содержания:

```
hello: sayhello.o hellomain.o
    gcc sayhello.o hellomain.o -o hello
sayhello.o: sayhello.c
    gcc -c sayhello.c
hellomain.o: hellomain.c
    gcc -c hellomain.c
```

Работа с утилитой `make` показана в следующем примере:

```
$ ls
hellomain.c Makefile sayhello.c
$ make
gcc -c sayhello.c
gcc -c hellomain.c
gcc sayhello.o hellomain.o -o hello
```

```
$ ls
hello hellomain.c hellomain.o Makefile sayhello.c
sayhello.o
```

В приведенном примере все цели совпадали с именами реально существующих файлов. В общем случае это требование не является обязательным. В файле могут присутствовать и так называемые *абстрактные цели* или *псевдоцели*, т.е. цели не связанные с файлом на диске. Такие цели чаще всего предназначены для упорядочивания обработки других целей. Кроме того при использовании абстрактных целей в соответствующем правиле могут отсутствовать команды. Рассмотрим следующий пример. Добавим в предыдущий файл следующие правила:

```
clean:
    rm -f sayhello.o hellomain.o

build: clean hello
```

Правило `clean` удаляет временные файлы (в данном случае – созданные объектные файлы). В то же время файл с именем `clean` не создается – т.е. данная цель является абстрактной.

Правило `build` сначала приводит к выполнению правила `clean`, а затем приводит к выполнению правила `hello`. Таким образом, все файлы принудительно перекомпилируются. Данная цель так же является абстрактной. Команды в ней отсутствуют.

Проверьте результаты выполнения этого примера самостоятельно. Формат запуска команды make следующий:

```
$ make [-f имя_файла] [параметры] [цели].
```

При использовании абстрактных целей необходимо иметь в виду следующий момент. Внешне абстрактные цели никак не отличаются от имен файлов. Поэтому если в текущем каталоге окажется файл, имя которого совпадает с именем абстрактной цели, то это может привести к тому, что make-файл будет обрабатываться не так, как задумывал его разработчик. В утилите GNU Make есть способ указать, что цель является абстрактной. Для

этого нужно перечислить все абстрактные цели в строке, начинающейся с “.PHONY:”. Для рассмотренного файла такая строка будет иметь вид:

```
.PHONY: clean build
```

Алгоритм работы утилиты make

Важным моментом при работе программы make является то, что она сама определяет в каком порядке выполнять содержащиеся в файле инструкции. Более того, если указана главная цель, то не важно, в каком порядке в файле приведены остальные. Правила в файле можно произвольно менять местами, результат от этого не изменится. В этом можно убедиться на следующем примере. В нем файлы Makefile1 и Makefile2 содержат одинаковые правила, стоящие в разном порядке, в то же время они выдают одинаковый результат при их выполнении:

```
$ cat Makefile1
target3: target1 target2
    echo "target 3"

target1:
    echo "target 1"

target2:
    echo "target 2"
$ cat Makefile2
target2:
    echo "target 2"

target1:
    echo "target 1"

target3: target1 target2
    echo "target 3"

$ make -f Makefile1 target3
```

```
echo "target 1"
target 1
echo "target 2"
target 2
echo "target 3"
target 3
$ make -f Makefile2 target3
echo "target 1"
target 1
echo "target 2"
target 2
echo "target 3"
target 3
```

Теперь рассмотрим алгоритм работы утилиты `make` более формально. Описание основано на работе [9]

Если говорить математическим языком, то нужно построить ориентированный граф зависимостей, а потом обойти этот граф, начиная с некоторой начальной вершины таким образом, чтобы вершины зависимости в каждом правиле были пройдены раньше, чем основная вершина правила. Построение графа начинается с поиска некоторой начальной вершины, которая обходится последней, и представляет собой то правило, которое, в конечном счете, нужно выполнить.

Это начальное правило, определяющее главную цель, может определяться одним из двух способов.

Во-первых, главная цель может быть явно задана в команде `make`. В рассмотренном выше примере таким правилом являлось явно указанное нами правило `target3`.

Если главная цель в команде `make` указана не была, то в качестве главной цели выбирается цель из **первого** правила в make-файле.

После этого запускается на выполнение **рекурсивный алгоритм обработки правил**. Обработка одного правила включает в себя два этапа:

- обработка зависимостей,
- выполнение команд.

При этом выполнение команд производится не всегда, а только при выполнении условий, о которых будет сказано ниже.

Первый этап при обработке правил – *обработка зависимостей*. Здесь для каждой из зависимостей проверяется, является ли данная зависимость целью какого-либо другого правила. Если ответ на этот вопрос утвердительный, то найденное правило так же обрабатывается в соответствии с алгоритмом обработки правил.

Если зависимость не является целью другого правила, то проверяется, есть ли на диске файл, имя которого совпадает с именем зависимости. Если такой файл не найден, то процесс сборки завершается с выдачей сообщения об ошибке.

Второй этап при обработке правил – *выполнение команд*. Команды, содержащиеся в правиле, выполняются, если оказывается истинным одно из следующих условий:

- цель данного правила является абстрактной,
- цель данного правила является именем файла, и такого файла не существует,
- какая-либо из зависимостей данного правила является абстрактной целью,
- цель данного правила является именем файла, и какая-либо из зависимостей, являющихся именем файла, имеет более позднее время модификации, чем цель.

Стандартные имена целей

Ясно, что для того, чтобы пользоваться make-файлом, необходимо знать имена основных целей, определенных в нем. Поэтому для основных задач, которые требуется выполнить пользователю, имеет смысл использовать стандартные имена. В качестве стандартных используются имена целей, перечисленные в табл. 9.1. Использование перечисленных имен целей носит рекомендательных характер. В make-файле может быть реализована только часть рассмотренных целей.

Табл. 9.1. Стандартные имена целей make.

Цель	**Описание**
all	Сборка основных исполняемых файлов и библиотек.
clean	Удаление всех временных файлов, созданных на этапе сборки

configure	Запуск процедуры конфигурации программы – определения компонентов, которые необходимо включить в исполняемый файл.
distclean	Удаление временных файлов, созданных на этапах сборки и конфигурации
install	Установка созданных на этапе сборки файлов в соответствующие каталоги в системе[13].
uninstall	Удаление файлов из служебных каталогов, которые были размещены там командой make install.

Использование переменных в make

В утилите make существует механизм, аналогичный макросам в языке C. И хотя в make чаще используется термин “переменная”, по сути это все те же макросы. Мы будем использовать термин переменная. Рассмотрим, для чего же они нужны.

При сборке может потребоваться передавать компилятору те или иные параметры, например, нам потребуется скомпилировать программу с отладочной информацией, задав для `gcc` ключ `-g`. Ясно, что записывать и удалять вручную ключ `-g` во всех вызовах компилятора – неудачная идея. Для решения подобных задач в make предназначены переменные. Они позволяют определить переменную с этим ключом только в одном месте make-файла, а во всех местах, где требуется данный ключ, вставить значение этой переменной. Таким образом, переменные в make, как и макросы в C, - это способ борьбы с избыточностью.

В именах переменных в make часто используют только заглавные буквы, как и в макросах C, хотя это и не является обязательным.

Определить переменную в make-файле можно следующим способом:

```
переменная = выражение
```

[13] На практике нужно относиться к команде make install с некоторой осторожностью, поскольку корректно удалить или обновить установленные таким образом файлы можно не всегда. Рекомендуемым вариантом является установка программ с помощью пакетных менеджеров.

Важным моментом здесь является то, что объявление переменной – это самостоятельная конструкция. Другой такой самостоятельной конструкцией является правило. Объявлять переменные внутри правил нельзя.

Значение переменной можно получить с помощью выражения:

```
$(переменная)
```

Рассмотрим следующий пример.

```
$ cat Makefile_var
KEYS = -g
all: file1.o file2.o
    gcc $(KEYS) file1.o file2.o -o file.out
file1.o: file1.c
    gcc $(KEYS) -c file1.c -o file1.o
file2.o: file2.c
    gcc $(KEYS) -c file2.c -o file2.o
$ make -f Makefile_var
gcc -g -c file1.c -o file1.o
gcc -g -c file2.c -o file2.o
gcc -g file1.o file2.o -o file.out
```

В этом примере, если мы захотим скомпилировать программу без отладочной информации, то не нужно будет искать и редактировать все места, где вызывается команда `gcc`, а нужно будет только задать переменной KEYS пустое значение.

Важная особенность переменных в make – это то, что они являются глобальными. Не важно, в каком месте файла определена переменная, – она будет доступна во всем файле. При этом более позднее определение заменяет более раннее.

Предопределенные правила

Ясно, что писать в make-файле десятки практически одинаковых строк, подобно тем, что приведены ниже, занятие однообразное и рутинное.

```
file.o: file.c
```

```
gcc –c file.c
```

К счастью, эта задача легко автоматизируется с помощью предопределенных шаблонных правил. Ряд таких правил содержится в самой утилите make. С ними можно ознакомиться с помощью команды `make –p`.

Кроме этого, пользователи могут создавать свои *шаблонные правила*. Шаблонное правило должно содержать знак “%” в имени цели, но притом только один. Цели при этом рассматривается как шаблон имени файла. Знак “%” соответствует любой непустой подстроке, остальные символы должны совпадать. Например, шаблонное правило для компиляции файлов программ на языке С может иметь вид:

```
$ cat Makefile_pattern
%.o: %.c
    gcc -c $< -o $@
$ make -f Makefile_pattern file1.o
gcc -c file1.c -o file1.o
```

Видно, что мы создали шаблонное правило и с его помощью из файла программы на языке С получили объектный файл.

В командах последнего примера использованы несколько специальных переменных, которые описаны в табл. 9.2.

Табл. 9.2. Значение некоторых специальных переменных для использования внутри шаблонных правил

Переменная	Описание
$@	Имя файла цели правила
$<	Имя первой зависимости
$?	Имена всех зависимостей, которые являются более новыми, чем цель
$^	Имена всех зависимостей

Дополнительные возможности make

Мы рассмотрели только самые основные возможности утилиты make, которые, тем не менее, позволяют получить о ней первое представление. К

другим полезным возможностям make относятся, например, условные конструкции в make-файлах или встроенные функции. Однако их рассмотрение выходит за рамки данного курса. Дополнительную информацию по работе с самой распространенной версией make – GNU make можно получить в [9].

Недостатки make. Другие системы сборки

Утилита make используется достаточно широко. Исторически она является первой системой сборки, и по этой причине, стала стандартом де-факто. Тем не менее, она не лишена недостатков.

Первая группа недостатков связана с низкой производительностью make.

Вторая группа недостатков связана с некоторой запутанностью языка make. Здесь можно упомянуть и необходимость использования табуляции перед командами в правилах, и отсутствие строгого синтаксиса языка, и некоторую запутанность механизма работы с переменными, и просто отсутствие в языке множества полезных конструкций, например, циклов.

Для исправления недостатков make было разработано достаточно большое количество альтернативных систем сборки проектов. В качестве примеров можно привести системы:

- CMake, использующую синтаксис языка C,
- Apache Ant и NAnt, использующие язык XML,
- SCons, использующую синтаксис языка Python,
- Rake, использующую синтаксис языка Ruby.

Глава 10. Отладчики в Linux

Отладчики в Unix и Linux. Отладчик gdb

Любому, кто серьезно занимается программированием, ясно, что без использования отладчика разработать программу размером больше нескольких экранов практически невозможно. Отладчик – один из основных инструментов программиста. В командном интерфейсе Linux вполне логично, что отладчики так же имеют командный интерфейс. Мысль о том, что придется пользоваться отладчиком командной строки может не вызвать энтузиазма, однако на практике не все так страшно.

На сегодняшний день стандартом де-факто среди отладчиков в Unix-подобных операционных системах является отладчик gdb (GNU Debuger), разработанный в рамках проекта GNU. Данный отладчик позволяет отлаживать программы, разработанные на языках программирования C, C++, Go, Pascal, Basic, Ada, Objective-C и других языках. Он широко используется как сам по себе, так и другими отладчиками и средами разработки, которые, по сути, являются высокоуровневой надстройкой над gdb. В частности, с gdb могут взаимодействовать такие среды разработки, как Eclipse, NetBeans, Qt Creator, Lazarus и др. Данный отладчик, кроме прочего, позволяет выполнять отладку удаленно, т.е. отлаживать программу, которая выполняется не на рабочем компьютере программиста, а на другом компьютере, соединенном с первым через сеть. Отладчик gdb создавался для совместного использования с рассмотренным ранее компилятором gcc, вместе с которым и некоторыми другими программами он составляет так называемый toolchain (с англ. – цепочка инструментов) – набор инструментальных программ, позволяющих выполнять разработку программного обеспечения.

Общее описание и основные команды отладчика gdb

Отладчик gdb является одним из самых развитых отладчиков. Как и большинство других отладчиков, он позволяет:

- запустить программу, указав все необходимые параметры запуска,
- остановить выполнение программы при указанных условиях,

- изучить состояние программы в момент ее останова,
- изменять значения переменных и регистров во время выполнения программы, что позволяет экспериментировать с целью выявления и устранения ошибок в программе.

Отладчик gdb позволяет, как запустить программу из отладчика, так и подключиться к уже запущенному процессу.

Описание основных команд отладчика представлено в табл. 10.1.

Табл. 10.1. Основные команды отладчика gdb.

Имя и формат команды	Сокр.	Описание
Команды общего назначения		
`help`		вывести справку о работе с отладчиком gdb
`file имя_файла`		загрузить программу для работы под отладчиком (если программа не была указана как параметр при запуске gdb)
`list номер_строки`	`l`	вывести строку программы с заданным номером и 10 ближайших строк
`quit`	`q`	выход из отладчика
Выполнение и трассировка программы		
`run`	`r`	запустить программу на исполнение, в команде можно указывать параметры командной строки и перенаправления ввода/вывода
`start`		начать пошаговое выполнение
`next`	`n`	выполнить следующую строку с обходом функций. В команде может быть указано число выполняемых операторов
`step`	`s`	выполнить следующую строку с входом в функции. В команде может быть указано число выполняемых операторов
`return`		выйти из текущей функции,

		пропустив оставшуюся ее часть
`continue`	`c`	продолжить выполнение программы
`kill`	`k`	прекратить выполнение отлаживаемой программы
Работа с точками останова		
`break номер_строки`	`b`	установить точку останова в строке с заданным номером. В качестве аргумента могут выступать так же адрес или имя функции
`watch имя_переменной`	`wa`	установить точку останова по изменению значения заданной переменной
`info breakpoints`		отобразить информацию обо всех точках останова
`disable breakpoint номер_точки_останова`		временно отключить точку останова с заданным номером (номер отображается в команде `info breakpoints`)
`enable breakpoint номер_точки_останова`		включить точку останова с заданным номером (номер отображается в команде `info breakpoints`)
`delete breakpoint номер_точки_останова`		удалить точку останова с заданным номером (номер отображается в команде `info breakpoints`)
Работа с переменными		
`print имя_переменной`		показать значение заданной переменной
`display выражение`		добавить выражение или переменную в список вывода. После выполнения данной команды значение выражения или переменной будет выводиться после каждого останова или

		выполнения шага трассировки
set var имя_переменной=значение		изменить значение указанной переменной

Пример отладки программы

Рассмотрим работу с отладчиком gdb на примере отладки следующей простой программы (строки программы пронумерованы, для того, чтобы на них было проще ссылаться):

```
1 /* sin.c */
2 #include<stdio.h>
3 #include<math.h>
4 int main(int argc, char* argv[]){
5   float x, y;
6   printf("Введите значение x: ");
7   scanf("%f", &x);
8   printf("x = %f\n", x);
9   y = sin(x);
10  printf("sin(x)= %f\n", y);
11  return 0;
12 }
```

Первое, о чем нужно помнить при отладке программы, это то, что при компиляции программы нужно включить в исполняемый файл отладочную информацию, указав в команде ключ `-g`.

```
$ gcc sin.c -g -lm -o sin.out
```

Теперь можно запустить отладчик следующей командой:

```
$ gdb sin.out
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
```

```
    This is free software: you are free to change and
redistribute it.
    There is NO WARRANTY, to the extent permitted by
law.  Type "show copying"
    and "show warranty" for details.
    This GDB was configured as "i486-linux-gnu".
    For bug reporting instructions, please see:
    <http://www.gnu.org/software/gdb/bugs/>...
    Reading symbols from
/home/user/Linux_book/gdb/sin.out...done.
    (gdb)
```

В команде запуска в качестве параметра указано имя отлаживаемого исполняемого файла. После ввода команды выдается информация о версии отладчика, авторских правах и условиях использования, а последняя строка `(gdb)` является приглашением отладчика к вводу команд.

Поставим точку останова в строке 9 с помощью команды `break`.

```
(gdb) break 9
Breakpoint 1 at 0x80484e4: file sin.c, line 9.
```

Теперь запустим программу на выполнение командой `run`.

```
(gdb) run
Starting program: /home/user/Linux_book/gdb/sin.out
Введите значение x: 3.5
x = 3.500000

Breakpoint 1, main (argc=1, argv=0xbffff4f4) at sin.c:9
9       y = sin(x);
```

В последнем листинге видно, что программа была запущена, был выполнен ввод значения `x` (введено значение 3.5) и эхопечать данного значения. После выполнения перечисленных действий программа была остановлена в точке останова в девятой строке.

Распечатаем фрагмент программы вокруг строки 9 с помощью команды `list` (выводится пять строк до и пять строк после).

```
(gdb) list 9
4   int main(int argc, char* argv[]) {
5     float x, y;
6     printf("x=? : ");
7     scanf("%f", &x);
8     printf("x = %f\n", x);
9     y = sin(x);
10    printf("sin(x) = %f\n", y);
11    return 0;
12  }
```

Выведем текущее значение переменной `x` с помощью команды `print`.

```
(gdb) print x
$1 = 3.5
```

Теперь изменим значение переменной с помощью команды `set`. Установим новое значение равным `-2` и опять выведем его на экран.

```
(gdb) set x=-2
(gdb) print x
$2 = -2
```

Далее выполним программу до конца пошагово с помощью команды `step`.

```
(gdb) step
10    printf("sin(x) = %f\n", y);
(gdb) step
sin(x) = -0.909297
11    return 0;
(gdb) step
12  }
(gdb) step
0xb7e78ca6    in    __libc_start_main    ()    from
/lib/i686/cmov/libc.so.6
(gdb) step
```

```
    Single stepping until exit from function
__libc_start_main,
    which has no line number information.

    Program exited normally.
```

Закончим работу с отладчиком:

```
(gdb) quit
```

На данном примере видно, что работа с отладчиком gdb, по сути, мало отличается от работы со встроенными отладчиками сред разработки. В то же время у gdb есть свои достоинства.

Мы уже говорили о возможности подключаться к уже работающему процессу, а так же выполнять отладку программы, выполняющейся на другом компьютере, например, во встроенной системе.

Так же, находясь в среде gdb, можно выполнять команды командной оболочки.

Еще одной полезной функцией gdb является возможность останова программы при изменении значения выражения.

Также gdb умеет при останове анализировать цепочку вызовов функций в стеке – можно проследить, каким путем вы попали в данную точку, начиная с функции main. Отладчик gdb умеет отображать не только текущие значения переменных, но и значения регистров и данных, содержащихся в заданных адресах памяти.

Более подробную информацию об отладчике gdb можно получить в [10].

Отладчик ddd. Отладка программы с использованием средств визуализации данных

В начале этой главы мы упоминали, что отладчик gdb может взаимодействовать с другими отладчиками, реализующими высокоуровневый интерфейс отладки. Такой способ построения программного обеспечения достаточно широко используется в Unix-подобных системах. Считается, что каждая программа должна реализовывать только одну функцию, но делать это хорошо. Решение же реальных задач

достигается с помощью взаимодействия нескольких программ, обменивающихся друг с другом текстовыми данными (командами). В нашей ситуации отладка считается одной функцией (которую реализует отладчик gdb), а реализация удобного интерфейса – другой функцией (которую реализуют другие отладчики).

Достаточно интересным отладчиком, реализующим высокоуровневый интерфейс для gdb, является отладчик ddd (Data Display Debugger). Он первоначально был разработан Доротеей Люткехаус (Dorothea Lutkehaus) в 1994 г. в рамках ее магистерской диссертации. Далее он разрабатывался программистами Фонда свободного программного обеспечения FSF.

Как видно из названия, отладчик ddd специализируется на удобном отображении текущих данных программы во время отладки. Когда вы работаете с отладчиком, ваша работа заключается, прежде всего, в том, что вы пытаетесь понять текущее состояние программы и определить, в каком месте программы ее состояние становится ошибочным. Состояние программы – это в первую очередь совокупность текущих значений ее переменных, которые требуется одновременно держать в голове. Поэтому отладчик ddd, позволяющий упростить восприятие данных, является достаточно полезным инструментом. Рассмотрим подробнее его функции.

Во-первых, в отладчике ddd реализован традиционный интерфейс отладчика, такой как в интегрированных средах разработки. На рис. 10.1 показан вид окна отладчика, в котором выполняется отладка программы из предыдущего примера. Окно отладчика разделено на три области. В верхней части окна располагается область отображения данных, в средней – область исходного кода программы, а в нижней части окна выводится консоль отладчика gdb.

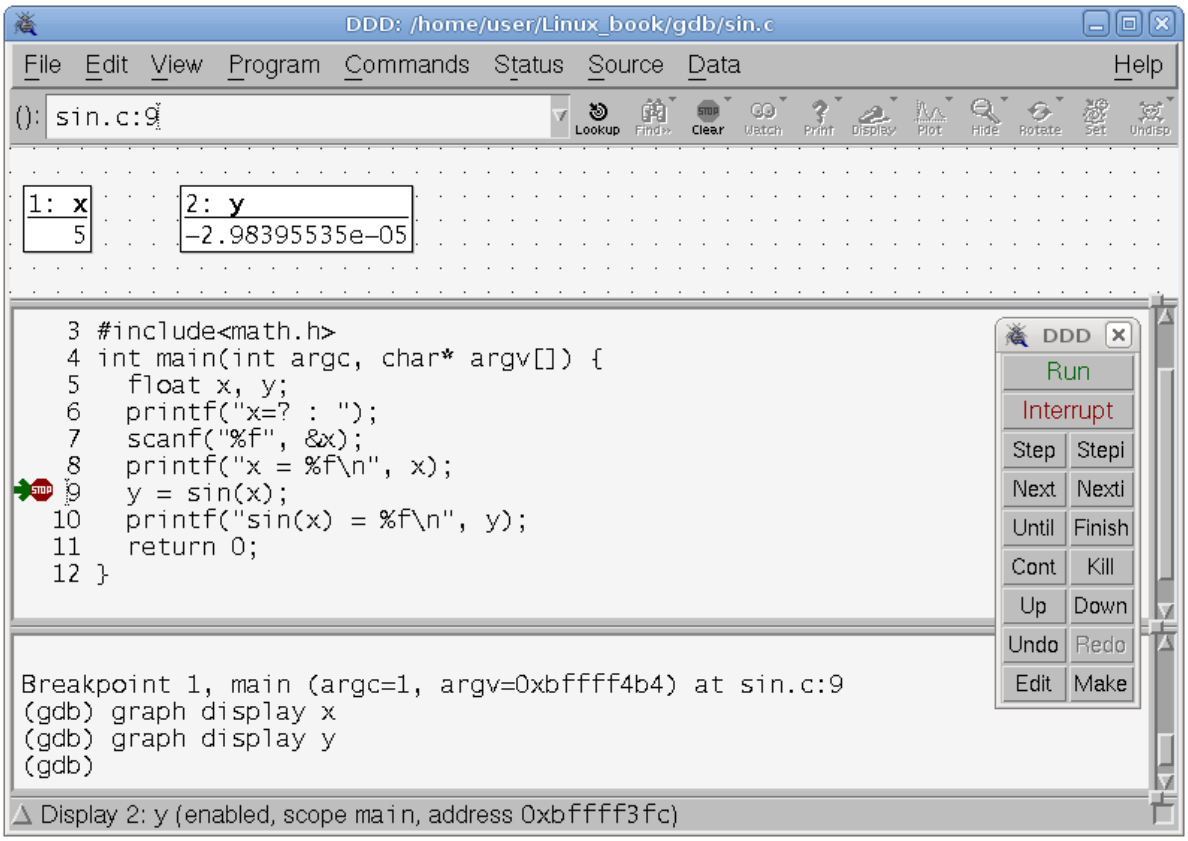

Рис. 10.1. Изображение окна отладчика ddd

Во-вторых, отладчик ddd позволяет отображать графически динамические структуры данных. Более того, отладчик ddd может автоматически обновлять изображение при пошаговой отладке. Поиск ошибок в работе с указателями является достаточно сложной задачей, и рассматриваемая функция отладчика является весьма полезной. Пример диаграммы для бинарного дерева, построенной отладчиком ddd, приведен на рис. 10.2.

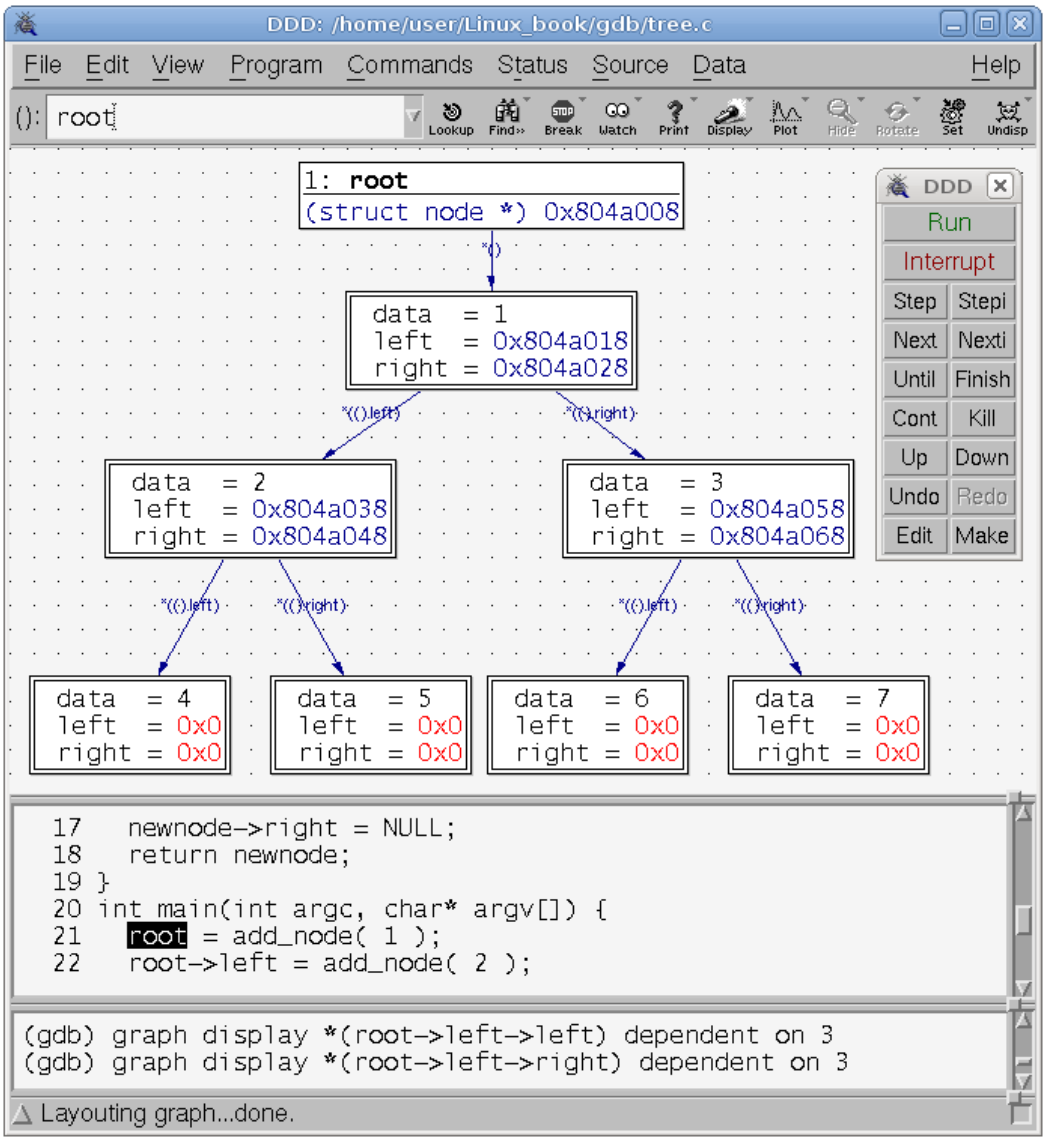

Рис. 10.2. Изображение динамических структур данных в окне отладчика ddd

В-третьих, отладчик ddd позволяет отображать данные, хранящиеся в массивах, в виде графиков (см. рис. 10.3). Если читатель занимался разработкой программ, предназначенных для выполнения научных и статистических расчетов, он по достоинству оценит и эту функцию.

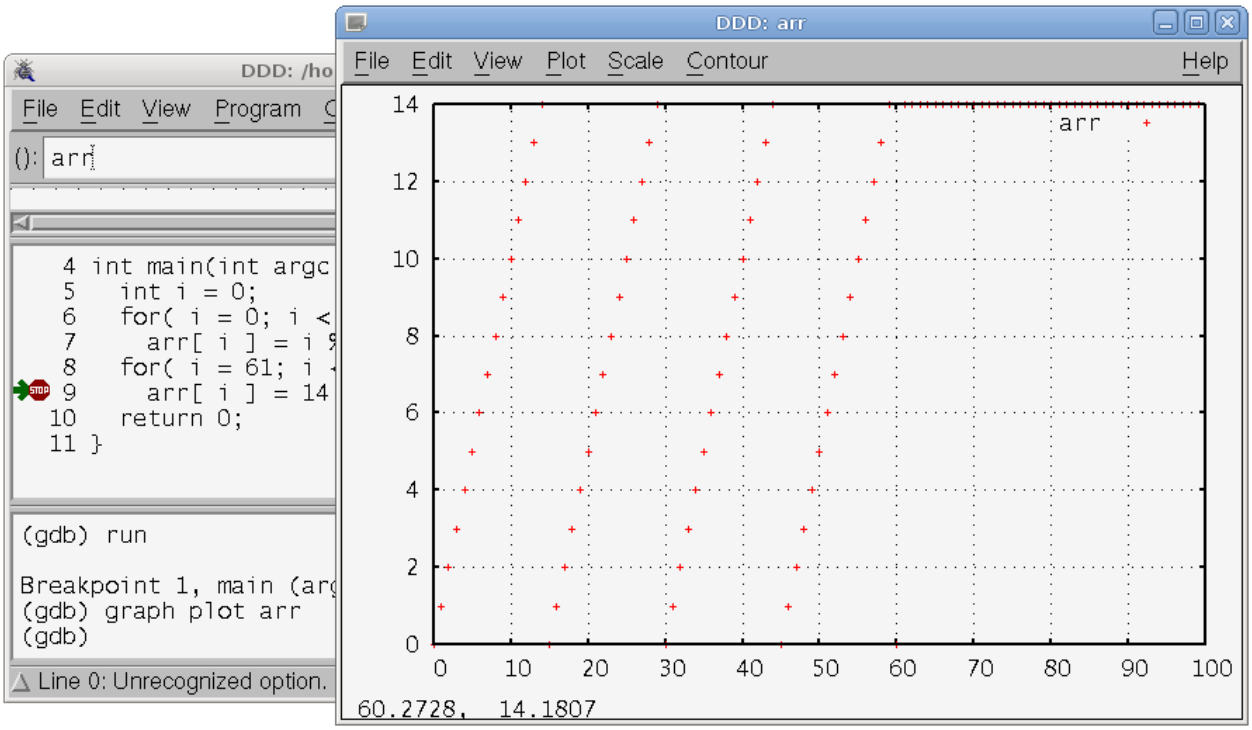

Рис. 10.3. Изображение массивов данных в виде графиков в окне отладчика ddd

Глава 11. Системы управления версиями

Введение в системы управления версиями

Если читатель сейчас только учится в университете, то, скорее всего, разрабатывает небольшие по объему программы, разработка ведется в одиночку, и ее длительность не превышает нескольких месяцев. Поэтому необходимость такого инструмента, как системы управления версиями, читателю может быть не очевидна. Чаще всего студенты попросту не знают о существовании систем такого рода, т.к. в существующие учебные программы они не входят.

Начав заниматься программированием профессионально, сталкиваешься с программами достаточно большого размера, уследить за всеми изменениями в которых, человек просто не в состоянии. Кроме того, реальные программы разрабатываются не в одиночку, а коллективно, и система, организующая совместную работу с исходным кодом программы нескольких программистов, становится просто необходимой. Не случайно умение работать с той или иной системой управления версиями присутствует как одно из требований практически в каждой вакансии программиста. Сейчас разработка серьезного программного обеспечения без использования систем управления версиями просто немыслима.

Системы управления версиями представляют собой достаточно сложные программные системы, которые пользователь может использовать по-разному. Они чрезвычайно полезны, но при неумелом использовании могут причинить и вред. Одного знания команд для эффективной работы с такими системами недостаточно, по мере приобретения опыта, вы можете выработать свою методику работы с системами управления версиями.

Система управления версиями (Version Control System, VCS) предназначена для отслеживания изменений в файлах исходного кода и других файлах проекта и управления этими изменениями. VCS организует хранение промежуточных версий файлов проекта, которые называются *версиями* или *ревизиями*.

В традиционных VCS существует центральное хранилище файлов, называемое *репозиторием*. Такие системы контроля версий называются *централизованными*. Пользователь может получить некоторую версию проектных файлов из центрального репозитория, внести в нее необходимые

изменения, а затем переслать изменения в центральный репозиторий. Примерами централизованных систем являются CVS (на данный момент устаревшая) и Subversion (SVN) (наиболее распространенная централизованная VCS).

В последнее время широкое распространение получили так называемые *распределенные* системы управления версиями. В таких системах отсутствует центральный репозиторий, а каждый пользователь имеет свою локальную версию репозитория. Текущая работа производится с локальным репозиторием, а крупные промежуточные версии синхронизируются с репозиториями других пользователей. Примерами распределенных систем контроля версий являются системы git, Bazaar и Mercurial.

Система управления версиями выполняет следующие две основные функции:

- автоматизирует работу с версиями рабочих файлов проекта – позволяет сохранить промежуточные версии файлов, вернуться к одной из предыдущих версий, сравнить, чем именно отличаются версии друг от друга,
- синхронизирует работу нескольких разработчиков над одним проектом – позволяет внести в свою версию рабочих файлов изменения, произведенные другими разработчиками, внести собственные изменения в центральный репозиторий. При этом объединение изменений внесенных разными разработчиками в один файл происходит автоматически (если это возможно). Таким образом, система управления версиями позволяет нескольким разработчиком одновременно вносить изменения в один и тот же файл исходного кода проекта.

Нужно заметить, что всю свою мощь системы управления версиями проявляют в том случае, если файлы проекта представлены в формате *текста*. В этом случае VCS может вносить в файл дополнительные фрагменты, удалять некоторые фрагменты или заменять один фрагмент другим.

В случае *бинарных файлов* все, что в большинстве случаев может сделать VSC – это заменить одну версию файла на другую версию файла (файл заменяется целиком).

Объединение нескольких различных изменений в одном файле легко выполняются автоматически в том случае, если они затрагивают различные фрагменты данного файла. Если же несколько изменений затрагивают один и тот же фрагмент файла, то возникает *конфликт* (*conflict*). В случае конфликта, разработчик должен будет выполнить операцию объединения изменений (получить окончательную версию) вручную.

Работа с системами управления версиями представляет собой следующую ежедневно повторяющуюся последовательность операций:

- загрузка изменений из репозитория, произведенных другими разработчиками и объединение их с рабочими файлами. Для данной операции в большинстве систем используется термин "*update*". Если же мы хотим загрузить версию проекта из репозитория, полностью заменив рабочие файлы, то данная операция обычно называется "*checkout*".
- внесение изменений в рабочую версию файлов на локальном компьютере,
- фиксация выполненных изменений в репозитории. Данная операция называется "*commit*". В русскоязычной литературе обычно пользуются русской калькой для данного слова – "*коммит*".

Кроме этих основных операций при разработке сложных проектов может возникнуть ситуация, при которой в репозитории нужно будет хранить фактически несколько параллельно развивающихся и связанных между собой вариантов проекта, например, стабильную, тестовую версию и версии, находящиеся в разработке. Такие отдельно разрабатываемые версии проектов называют **ветвями** (англ. **branch**). Соответственно операция ветвления проекта обычно называется "*branch*".

Так же может понадобиться объединить изменения, выполненные в двух ветвях проекта, получив в результате новую единую версию. Данная операция объединения изменений из нескольких ветвей обычно называется "*merge*".

Система управления версиями git

Далее мы рассмотрим несколько приемов работы с системой управления версиями **git**. Данная система в последнее время стала одной из самых распространенных. Разработку git инициировал Линус Торвальдс для использования в проекте разработки ядра Linux. Так же данная система используется в проектах Android, Drupal, Wine, jQuery, PHP и многих других. Git распространяется под лицензией GNU GPL 2.

Система управления версиями git является *децентрализованной* – она не имеет обязательного централизованного репозитория. Тем не менее, его роль может выполнять локальный репозиторий одного из разработчиков.

В этом разделе мы затронем только самые распространенные команды, которые позволяют представить, что такое система управления версиями. Мы будем использовать git, но, тем не менее, рассмотрим команды, характерные для большинства систем управления версиями. В то же время специфические возможности именно git мы рассматривать не будем, оставляя их на самостоятельное изучение читателю.

Первое что нужно сделать при работе с системой управления версиями – это создать хранилище исходного кода (репозиторий). Чаще всего при использовании git репозиторий копируется из другого репозитория (при совместной распределенной разработке какой-либо программы). Для этого применяется команда `git clone`. В данном случае мы в создадим новый пустой репозиторий командой `git init`.

```
$ git init
Initialized empty Git repository in /home/user/Linux_book/git/progexample/.git/
```

При этом в текущем каталоге появится скрытый каталог с именем “`.git`”, в котором будут храниться файлы репозитория.

Дальше нам нужно в репозитории создать проект, состоящий из нескольких файлов. Для этого создадим в текущем каталоге файлы sin.c и Makefile со следующим содержимым:

```
$ cat sin.c
/*   sin.c   */
#include<stdio.h>
#include<math.h>
int main(int argc, char* argv[]) {
```

```
  float x, y;
  printf("x=? : ");
  scanf("%f", &x);
  printf("x = %f\n", x);
  y = sin(x);
  printf("sin(x) = %f\n", y);
  return 0;
}
$ cat Makefile
all: sin.o
    gcc sin.o -o sin.out -lm

sin.o: sin.c
    gcc -c sin.c
```

Теперь добавим их в проект. Для этого нужно использовать команды `git add` и `git commit`:

```
$ git add sin.c
$ git add Makefile
$ git commit -m "Стартовая версия проекта"
[master (root-commit) d73b9b7] Стартовая версия проекта
 Committer: user <user@dlink-debian.dlink-domain>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

If the identity used for this commit is wrong, you can fix it with:
```

```
        git commit --amend --author='Your Name
<you@example.com>'

 2 files changed, 17 insertions(+), 0 deletions(-)
 create mode 100644 Makefile
 create mode 100644 sin.c
```

Здесь мы создали начальную версию проекта.

Попробуем изменить данную версию и внести изменения в репозиторий. Для этого отредактируем файл `sin.c` следующим образом:

```
/*   sin.c   */
#include<stdio.h>
#include<math.h>

float func(float var){
  float res = sin(var) + cos(var);
  return res;
}

int main(int argc, char* argv[]) {
  float x, y;
  printf("x=? : ");
  scanf("%f", &x);
  printf("x = %f\n", x);
  y = func(x);
  printf("func(x) = %f\n", y);
  return 0;
}
```

Выполним сборку проекта с помощью команды make.

Далее выполним следующую команду:

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be
committed)
```

```
    #     (use "git checkout -- <file>..." to discard
changes in working directory)
    #
    #   modified:   sin.c
    #
    # Untracked files:
    #     (use "git add <file>..." to include in what
will be committed)
    #
    #   sin.o
    #   sin.out
    no changes added to commit (use "git add" and/or
"git commit -a")
```

Последняя введенная команда `git status` сообщает нам, что в основной ветке проекта изменен, но не отмечен, как требующий обновления в репозитории файл `sin.c` (“changed but not updated”), так же в каталоге появились файлы `sin.o` и `sin.out`, которые пока не отслеживаются системой управления версиями (“untracked”).

В общем случае, файл, находящийся в рабочем каталоге, может находиться в одном из четырех состояний [11]:

- *untracked* – файл находится в рабочем каталоге, но не отслеживается системой управления версиями,
- *unmodified* – файл отслеживается системой управления версиями, но не изменялся с момента предыдущего коммита,
- *modified* – файл отслеживается системой управления версиями, изменен по сравнению с предыдущим коммитом, но не указан, как включаемый в следующий коммит,
- *staged* – файл отслеживается системой управления версиями, изменен по сравнению с предыдущим коммитом, и *проиндексирован* - указан, как включаемый в следующий коммит,

Состояния файлов в git и возможные переходы файла из одного состояния в другое показаны на рис. 11.1 [11].

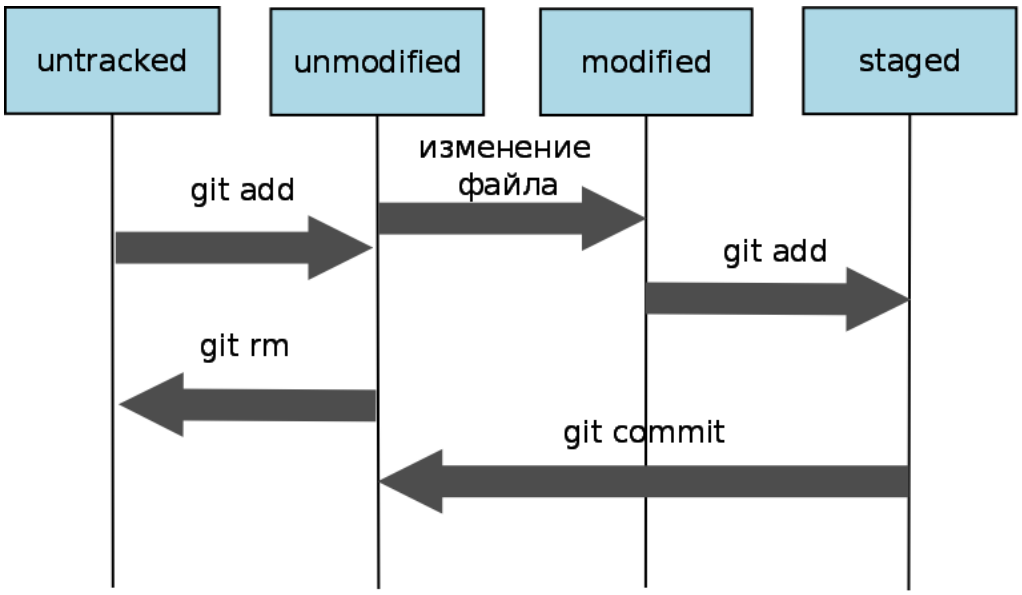

Рис. 11.1. Возможные состояния файла в git

После выполнения коммита все файлы, отслеживаемые системой контроля версий, будут находиться в состоянии unmodified. Внеся изменения в какой-либо файл, мы переводим его в состояние modified. Далее измененный файл можно проиндексировать - отметить как включаемый в новый коммит (перевести в состояние staged), выполнив команду `git add`.

Теперь создадим вторую версию проекта, внеся сделанные изменения в репозиторий с помощью следующих команд:

```
$ git add sin.c
$ git commit -m "вторая версия с функцией func()"
[master c04685f] вторая версия с функцией func()
 Committer: user <user@dlink-debian.dlink-domain>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com
```

```
    If the identity used for this commit is wrong, you
can fix it with:

        git commit --amend --author='Your Name
<you@example.com>'

     1 files changed, 8 insertions(+), 2 deletions(-)
```

Git сообщает нам, что во время выполнения последнего коммита был изменен один файл, в него были добавлены 8 строк кода и удалены 2 строки.

Теперь мы можем посмотреть историю версий проекта с помощью следующей команды:

```
$ git log
commit c04685f8a6c2e684276b50da9433d2709d77aff3
Author: user <user@dlink-debian.dlink-domain>
Date:   Fri Feb 8 12:50:21 2013 +0400

    вторая версия с функцией func()

commit d73b9b75daedd48b4139583fa0bc2adbd075c02b
Author: user <user@dlink-debian.dlink-domain>
Date:   Thu Feb 7 15:40:59 2013 +0400

    Стартовая версия проекта
```

Мы вкратце рассмотрели функциональность `git`, касающуюся работы с версиями проекта. Теперь нам нужно рассмотреть функции, выполняющие синхронизацию работы нескольких разработчиков. Допустим, к созданному нами проекту присоединился еще один программист, который собирается

работать над ним параллельно с первым. Разумеется, читателю далее придется имитировать работу этих двух программистов.

Войдем в систему от имени другого пользователя (в примере это будет `user2`) и создадим копию репозитория командой `git clone`.

```
$ git clone /home/user/Linux_book/git/progexample/
Cloning into progexample...
done.
```

Здесь в качестве параметра команды `git clone` нужно указать путь к созданному ранее пользователем `user` репозиторию. В результате, в текущем каталоге появится подкаталог `progexample` с файлами проекта и служебными файлами начального репозитория.

Дальше нам будет нужно внести в файл sin.c правки от имени пользователей `user` и `user2`, а потом выполнить слияние этих изменений в одну ветку проекта.

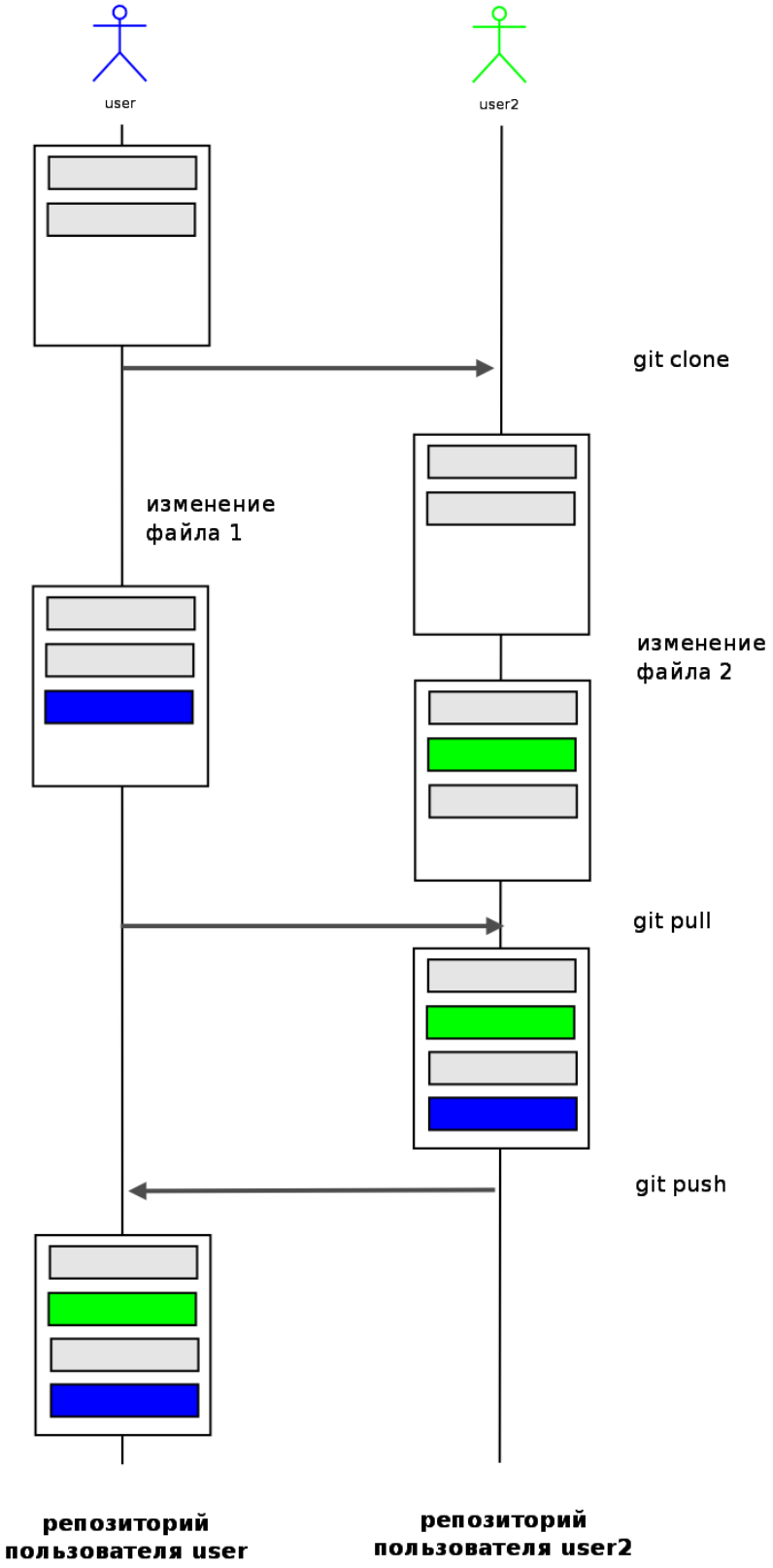

Рис. 11.2 Схема изменений в проекте из примера

Сначала доработаем функцию `func()` от имени пользователя `user2` так, как показано ниже, и выполним коммит.

```
/*   sin.c   */
#include<stdio.h>
#include<math.h>

float func(float var){
  float res = sin(var);
  int i;
  for ( i=0; i <= 5; i++)
   res = res + cos( var + i );
  return res;
}

int main(int argc, char* argv[]) {
  float x, y;
  printf("x=? : ");
  scanf("%f", &x);
  printf("x = %f\n", x);
  y = func(x);
  printf("func(x) = %f\n", y);
  return 0;
}
```

Далее внесем от имени пользователя `user` в его рабочую версию файла `sin.c` следующие изменения и тоже выполним коммит.

```
/*   sin.c   */
#include<stdio.h>
#include<math.h>

float func(float var){
  float res = sin(var) + cos(var);
  return res;
}

int main(int argc, char* argv[]) {
```

```
  float x, y;
  printf("x=? : ");
  scanf("%f", &x);
  printf("x = %f\n", x);
  y = func(x);
  printf("func(x) = %f\n", y);
  y = func(x + 5);
  printf("func(x + 5) = %f\n", y);
  y = func(x + 10);
  printf("func(x + 10) = %f\n", y);
  return 0;
}
```

Далее зайдем от имени пользователя `user2` и загрузим из центрального репозитория содержащиеся в нем изменения (мы их только что сами сделали от имени пользователя `user`) с помощью следующей команды:

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/user/Linux_book/git/progexample
   c04685f..4aec5a7  master     -> origin/master
Auto-merging sin.c
Merge made by recursive.
 sin.c |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

В выводе предыдущей команды видно, что при выполнении последней команды был изменен файл `sin.c`, в него были добавлены 4 новые строки кода. Теперь данный файл содержит:

```
/*   sin.c   */
#include<stdio.h>
#include<math.h>

float func(float var){
```

```
  float res = sin(var) + cos(var);
  return res;
}

int main(int argc, char* argv[]) {
  float x, y;
  printf("x=? : ");
  scanf("%f", &x);
  printf("x = %f\n", x);
  y = func(x);
  printf("func(x) = %f\n", y);
  y = func(x + 5);
  printf("func(x + 5) = %f\n", y);
  y = func(x + 10);
  printf("func(x + 10) = %f\n", y);
  return 0;
}
```

Как мы видим из последнего листинга, git **автоматически** объединил все изменения, внесенные пользователями `user` и `user2`. Причем, результат объединения именно такой, какой должен быть. Заметьте, что изменения вносили два разных пользователя, которые, в принципе, могли работать на двух разных компьютерах в разных точках мира.

Заметим, что последний листинг программы в приведенном виде доступен только пользователю `user2`. Для того чтобы отправить изменения в исходный репозиторий (которым пользуется пользователь `user`), нужно выполнить следующую команду:

```
$ git push origin master
```

Для успешного выполнения последней команды пользователь должен иметь разрешение на запись в каталоге исходного репозитория.

Данная глава является только самым общим введением в системы управления версиями и систему git в частности. Здесь не рассмотрены многие возможности системы. Тем не менее, автор надеется, что читатель

оценил полезность систем управления версиями, и это послужит толчком для их дальнейшего изучения.

Список литературы

1. Реймонд Э.С. Искусство программирования для Unix: Пер. с англ. – М: Издательский дом “Вильямс”, 2005. – 544. с.

2. Раскин Д. Интерфейс: новые направления в проектировании компьютерных систем. – Пер. с англ. СПб: Символ-Плюс, 2004. – 272 с. ил.

3. Лав Р. Разработка ядра Linux, 2-е издание. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2006. — 448 с.

4. Robert Love. Linux Kernel Development. Third Edition. Addison-Wesley. 2010. ISBN-13: 978-0-672-32946-3, ISBN-10: 0-672-32946-8

5. Фридл Дж. Регулярные выражения, 3-е издание.- Пер. с англ. - СПб.: Символ-Плюс, 2008.-608 с, ил.

6. Керниган Б., Пайк Р. UNIX. Программное окружение. – Пер. с англ. – СПб: Символ-Плюс, 2003. – 416 с., ил.

7. Купер М. Advanced Bash-Scripting Guide: Искусство программирования на языке сценариев командной оболочки. Ссылка интернет: http://www.opennet.ru/docs/RUS/bash_scripting_guide/

8. Гриффитс А. GCC. Настольная книга пользователей, программистов и системных администраторов: Пер. с англ. – К: ООО “Тид ДС”, 2004. – 624. с.

9. Игнатов В. Эффективное использование GNU make. Сайт Интернет http://www.opennet.ru/docs/RUS/gnumake/.

10. Столлман Р. и др. Отладка с помощью GDB. Сайт Интернет http://www.opennet.ru/docs/RUS/gdb/.

11. Scott Chacon. Pro Git. Сайт Интернет http://git-scm.com/book.