Ловягин Никита Юрьевич
Ловягин Юрий Никитич

Основы работы в Unix-подобных операционных системах

Учебное пособие

Часть I.

Введение в семейство Unix-подобных операционных систем

Санкт-Петербург
2020

Оглавление

Об этом пособии

Пособие посвящено изучению основных принципов *Unix*-подобных операционных систем.

В пособии изложены необходимые понятия и даны представления об организации системы; описана работа в оболочке командной строки, основные команды; принципы и методы написания сценариев оболочки. Основная часть пособия построена по принципу от простого к сложному, с приведением необходимых определений, пояснений и примеров с целью получения читателем фундаментальных знаний и базовых навыков по темам курса. Приложение представляет собой обзор основных понятий и возможностей по дополнительным темам, связанным с администрированием *Unix*-подобных систем и серверов, и призвано стать базой для дальнейшего изучения данных тем. В сносках приводится дополнительная или уточняющая информация, в которой могут использоваться понятия, вводимые в следующих разделах, поэтому при первом чтении их можно пропустить.

Определяемые термины приводятся **жирным шрифтом**, а ***жирным курсивом*** в скобках приводится их английский аналог. *Курсивный шрифт* служит для записи собственных имен программ и других наименований, а также для выделения важных замечаний, пояснений и переводов. `Прямой моноширинный шрифт` служит для записи команд, их аргументов и иных данных, предназначенных для ввода в командную строку или в файлы сценариев, в том числе при описании синтаксиса команд, а также для примеров терминального вывода программ. При этом *моноширинный курсив* используется для описания читателю возможных элементов синтаксиса команд, которые при вызове команды должны быть заменены на соответствующие, понятные компьютеру, конструкции. Моноширинный текст, записанный в рамке, предназначен для ввода в файлы (сценариев и другие).

В пособии используются следующие сокращения:

ОС — операционная система;

ПО — программное обеспечение;

ФС — файловая система.

Глава 1. Ключевые понятия

§1.1. Данные и компьютеры

Компьютер (*англ.* ***computer*** — *вычислитель*) буквально следует понимать, как устройство для осуществления вычислений. Одно из возможных определений компьютера, наиболее близкое ко взгляду на компьютер с точки зрения программирования — устройство для выполнения операций над данными в соответствии с инструкцией.

Все действия, совершаемые компьютерами сводятся к операции над некоторыми данными: чтение (с клавиатуры, с сенсорного экрана, с камеры, с диска), запись (в память, на диск), передача (на другой компьютер, на внешнее устройство), отображение (на бумаге, на экране) и др. Наибольшее распространение в современном мире имеют **цифровые компьютеры**, оперирующие с данными, представленными в виде чисел. Под самим термином **данные** (*англ.* ***data***) понимается **информация** (*англ.* ***information***), представленная в пригодной для обработки форме (при этом для одной и той же информации такая форма для человека и для компьютера может существенно отличаться).

Инструкция (*англ.* ***instruction***) в определении компьютера представляет собой последовательность действий (команд), которые необходимо выполнить. В данном случае это понятие можно понимать двояко: с одной стороны есть команды, выдаваемые компьютеру пользователем или оператором (например, воспроизвести видео, запустить игру, распечатать документ и т.п.), а с другой — последовательность действий самого устройства, которые выполняются для достижения затребованного результата — команд, исполняемых непосредственно самим компьютером, то есть **машинных команды** или **машинных инструкций** (*англ.* ***machine instructions***), которые находятся "внутри" компьютера и исполняются автоматически. В этом смысле исполнение команд, выдаваемых человеком, по сути есть исполнение особых последовательностей внутренних команд. Таким образом, принято считать, что всякий компьютер представляет собой комбинацию двух продуктов:

- **твердый продукт** (*англ.* ***hardware***) — **аппаратное обеспечение** (аппаратура), собственно техника, устройство;
- **мягкий продукт** (*англ.* ***software***) — **математическое (программное) обеспечение**, **ПО** — собственно наборы команд и инструкций в соответствии с которым работает устройство.

§1.2. Архитектура компьютера

С точки зрения концепции устройства компьютера, *из того, как видна его аппаратная организация со стороны программной*, следует выделить ряд компонент, которые могут являются отдельными физическими частями (устройствами) или быть составной частью иных устройств. Также на физическом уровне выделяется много других отдельных компонент, но в данном курсе в первую очередь имеет значение именно взгляд с программной стороны.

- **Процессор** (*англ.* ***processor***), **центральный процессор**, **Центральное Процессорное Устройство**, **ЦПУ** (*англ.* ***Central Processor Unit, CPU***) — устройство, производящее собственно выполнение машинных команд, а также арифметические вычисления и логические операции над числами. Термин "центральный" относится к собственно "главному" процессору компьютера, в то время как другие составные части компьютера как физического объекта могут содержать собственные процессоры, например, **графический процессор** (*англ.* ***Graphic Processor Unit, GPU***) у видеокарты, собственные процессоры жестких дисков и др.

- **Постоянная память**, **Постоянное Запоминающее Устройство**, **ПЗУ** (*англ.* ***Read Only Memory, ROM****, буквально — память, доступная только для чтения*)[1] — память, в которой записывается программа, запускаемая при включении компьютера; для обозначения соответствующего класса программ используется термин "**встроенное программное обеспечение**" (*англ.* ***firmware*** *— программный продукт, предоставляемый фирмой-производителем*). На современных персональных компьютерах используются специфические для аппаратуры программы, объединенные под стандартном ***UEFI*** (*Unified Extensible Firmware Interface — интерфейс расширяемого встроенного ПО*), ранее использовалась встроенная программа *BIOS*[2]. Различные типы компьютеров могут иметь различное встроенное программное обеспечение, следующее иному или не следующее никакому стандарту (например, встроенное ПО мобильных телефо-

[1] Не следует путать ПЗУ с долговременными носителями информации, являющимися в данной классификации устройствами ввода-вывода. Также следует учесть, что на современных компьютерах данная память де-факто является доступной не только для чтения, но и перезаписываемой, что позволяет обновить версию встроенной программы.

[2] Термин *BIOS* нередко используется как обобщенно-нарицательный для обозначения всех встроенных программ настольных и портативных компьютеров, но строго говоря называть *UEFI* "разновидностью" *BIOS* некорректно.

нов, маршрутизаторов и др.).

- **Оперативная память**, (*англ.* ***Random Access Memory, RAM*** — *память с произвольным доступом*)[1], **Оперативное Запоминающее Устройство**, **ОЗУ** — память, в которой во время работы компьютера хранится код и данные программ, исполняемых процессором. Обычно данная память энергозависимая, то есть данные в ней стираются при отключении питания (это обусловлено возможность обеспечить многократно более высокую скорость работы с данными при той же стоимости).
- **Устройства ввода-вывода**, (*англ.* ***Input/Output devices***), или отдельно **устройства ввода** и **устройство вывода** — устройства, осуществляющие получение информации от компьютера (вывод) или передачу информации компьютеру (ввод). Информация может передаваться во внешнюю среду или прочитываться из нее, позволяя, например, обмениваться данными с человеком (клавиатуры, мониторы, мыши, принтеры, камеры, сканеры, датчики и др.), передаваться другому компьютеру (сетевые устройства, USB-интерфейс и др.), сохраняться внутри устройства (накопители данных — жесткие и гибкие диски, карты памяти, SSD накопители и др.). Технически данные устройства являются отдельными компьютерами со своими процессорами, постоянной и оперативной памятью, со своим набором исполняемых машинных команд, со своим вводом-выводом. Например, видеокарта содержит графический процессор и графическую память, обрабатывающие графические изображения в соответствии с посылаемыми центральным процессором программами для графического процессора и выводящими результат обработки на монитор, который тоже является отдельным компьютером и содержит свой процессор для обработки сигнала и вывода его на дисплей. Накопители данных также являются устройствами ввода-вывода со своим процессором, оперативной памятью, встроенным ПО.

Под **архитектурой компьютера** (*англ.* ***computer architecture***) понимается набор правил, описывающих организацию и функционал ком-

[1] Термин несколько неточный, т.к. под памятью с произвольным доступом можно понимать любое запоминающее устройство, к каждой ячейке которого есть доступ на чтение и запись, в том числе жесткие диски и SSD накопители, а не только оперативную память. Более точным термином является *англ.* ***Primary Memory*** — **первичная память** компьютера, к которой у процессора есть прямой произвольный доступ. Такой и только такой произвольный доступ предполагает термин RAM, в то время как операции с данными на накопителях информации являются вводом-выводом. В любом случае, термин RAM используется повсеместно именно для обозначения оперативной памяти.

пьютера. Более узкий термин — **архитектура процессора** (*англ.* ***processor architecture***), включающий в себя математическое устройство конкретного типа процессоров: набор машинных команд, которые может исполнять данный процессор, а также имеющийся набор **регистров** (*англ.* ***registry***), в которых осуществляется управление и арифметико-логическое вычисления. Регистр представляет собой сверхбыструю ячейку памяти, физически находящуюся непосредственно в процессоре. Аппаратно выполнение вычислений осуществляется в регистрах процессора. Среди машинных команд имеются, например, команды сложения значений в двух регистрах, добавления значения регистра к ячейке памяти и наоборот. В некоторых архитектурах могут присутствовать и более сложные команды, например, сложение значений в двух ячейках памяти с записью в третью, однако подобные машинные команды не являются атомарными аппаратными операциями и все равно выполняются в центральном процессоре.

Многообразие компьютерных возможностей сводится к следующим аппаратным действиям: арифметические и логические вычисления в процессоре, прямая передача данных из оперативной памяти в процессор и обратно, обмен данными с другим компьютером. Устройства ввода-вывода также могут управлять аппаратно-механической частью, работающей со внешней средой — например, печатающей головкой принтера, флеш-памяти и т.д., и получать данные о ее состоянии.

Процессоры различных архитектур могут быть несовместимы между собой — попытка исполнить машинный код, понятный одной архитектуре, приведет к ошибке на другой, также архитектура может быть расширенной версией более старой архитектуры, тогда код второй может быть исполнен на первой, но не наоборот. Например, архитектуры ARM (процессоры мобильных устройств) и x64 (процессоры настольных и портативных компьютеров) несовместимы между собой, в то время как архитектура x64 является расширением архитектуры x86.

§1.3. Машинный код, компиляция и интерпретация

Как было отмечено выше, компьютер может исполнять только **машинный код** (*англ.* ***machine code***). Этот код — двоичный код или его несколько более удобный и экономичный шестнадцатеричный эквивалент — крайне труден для восприятия, а тем более создания человеком. Для машинных команд существует мнемоническая запись, **мнемокод** (*англ.* ***mnemocode***) — буквенное представление машинных команд. Например, вместо двоичного

```
10100011 00000001 00000000 00000000 00000000
```

или шестнадцатеричного

```
A3 01 00 00 00
```

будет записано

```
MOV [0001], EAX
```

что понять куда легче — записать значение, находящееся в регистре `EAX` (арифметический регистр) в ячейку памяти с номером 1 (данный пример приведен для архитектуры x86).

Программирование с помощью мнемокодов требует ручного вычисления адресов данных и точек перехода в коде. Последнее — особо сложная задача, так как требует знания сколько байт занимает соответствующий двоичный код. Язык **ассемблера** (*англ.* ***assembler*** — *сборщик*) позволяет оперировать с метками — автоматически вычисляемыми адресами, а также некоторыми дополнительными возможностями, но по-прежнему основная часть кода записывается с помощью архитектурно-специфичных машинных команд.

Языки программирования высокого уровня (*англ.* ***high-level programming language***) — *Си*, *Python* и др. — позволяют писать инструкции, специфические для данного языка, но не привязанные к архитектуре[1], и позволяют нагляднее, выразительнее представить программный код, что облегчает процесс программирования.

Процесс получения машинного кода из **исходного кода** (*англ.* ***source code***) — текстового файла, содержащего код, написанный с помощью человекочитаемых мнемокодов, на ассемблере или языке программирования высокого уровня — называется **трансляцией** (*англ.* ***translation*** — *перевод*), а соответствующая программа — **транслятором** (*англ.* ***translator*** — *переводчик*). Существует два вида трансляции исходного кода — **компиляция** (*англ.* ***compilation***) и **интерпретация** (*англ.* ***interpretation***), а соответствующие программы называются **компилятор** (*англ.* ***compilator***) и **интерпретатор** (*англ.* ***interpreter***).

Компилятор переводит исходный код в машинный однократно, после этого данный код может быть исполнен на любом компьютере с соответствующей архитектурой. Примером компилируемых языков программирования служат *Си*, *Ассемблер* и др. Интерпретатор переводит файл исходного кода в машинный пошагово, фактически исполняет его. Исполнение программы, написанной на таком языке, требует наличие у поль-

[1] Кроме того, акт исполнения такой инструкции решает более сложные задачи, чем одна машинная команда.

зователя интерпретатора для соответствующей архитектуры[1]. Примеры интерпретируемых языков — *Python*, *PHP* и др. Существует и смешанный подход: компиляция в особый байт-код, не зависящий от архитектуры, с его последующей интерпретацией при исполнении программы. Примеры — языки *JAVA*, *C#* и др.

§1.4. Функции и библиотеки

Как уже было отмечено, процесс написания машинного кода очень трудоемкий, поскольку машинные команды выполняют лишь простейшие операции. Для выполнения более сложных действий часто приходится иметь дело с исполнением одного и того же (или известным образом зависящего от обрабатываемых данных) кода. Для исключения написания повторяющегося кода разработан аппарат **функций** (*англ.* ***function***)[2].

Идейно всякая функция представляет собой решение некоторой задачи, преобразование входных данных в выходные определенным образом. Входные данные могут передаваться как из программы или другой функции, так и приходить с устройства ввода, аналогично с выходными данными. Исполнение функции из программы или другой функции называется **вызовом функции** (*англ.* ***function call***). На самом деле исполнение всякой программы можно рассматривать как вызов одной из ее функций (особой, выделенной функции), поэтому процесс выполнения программы можно представить как вызовы функций из функций, пока дело не дойдет до машинных кодов. Вызов функции — часто встречающееся в процессе работы компьютера явление. Компьютеры содержат соответствующие машинные инструкции для быстрой передачи управления в функцию, возврата из нее и обмена входными и выходными данными с ней.

Для удобства программирования готовые функции объединяют в **библиотеки** (*англ.* ***function library***) — особые файлы, содержащие наборы функций, которые, в зависимости от способа построения, могут быть включены в код программы или оставаться отдельными файлами, поставляемыми вместе с ней или в качестве отдельного программного продукта[3].

[1]Кроме несовместимости архитектур существует несовместимость операционных систем, поэтому нужно наличие компилятора и интерпретатора для данной архитектуры и данной операционной системы. Это поясняется в следующих параграфах.

[2]Близкие между собой, но не тождественные термины — функция, **процедура** (*англ.* ***procedure***), **метод** (*англ.* ***method***), **подпрограмма** (англ. ***subroutine, routine***) — не вдаваясь в различия между языками программирования можно считать синонимами и понимать под ними набор программных инструкций, выделенный в самостоятельную единицу. Следуя традиции системного программирования, мы используем термин “функция” как обобщающий.

[3]Такие файлы имеют расширение `.dll` в ОС *Windows* и `.so` в ОС *GNU/linux*.

Всякий язык программирования высокого уровня снабжается стандартной библиотекой функций (что и является одним из основных аппаратов высокоуровневых инструкций этих языков, решающих за один шаг более сложные задачи, чем одна машинная инструкция). Также библиотеки функций, облегчающие процесс программирования, создаются и распространяются как отдельные программные продукты.

§1.5. Интерфейс

Важным для компьютерных технологий термином является **интерфейс** (*англ.* ***interface***) — набор программных и/или аппаратных средств, с помощью которых компоненты компьютерных систем взаимодействуют между собой.

Различают:

- **аппаратный интерфейс** (*англ.* ***hardware interface***) — интерфейс взаимодействия аппаратных устройств, интерфейс подключения устройств к компьютеру, соединения компьютеров между собой и т.д. (например, интерфейс *USB*, сетевые интерфейсы *Ethernet* и *Wi-Fi*, интерфейс *SATA* для подключения *SSD*-накопителей и жестких дисков, интерфейс *PCI-E* для подключения видеокарт, интерфейс *HDMI* для подключения мониторов и др.);
- **программный интерфейс** (*англ.* ***software interface***) — интерфейс взаимодействия приложений (например операционной системы и программы, программы и встраиваемого модуля и др.);
- **пользовательский интерфейс** (*англ.* ***user interface***) — интерфейс взаимодействия пользователей и компьютера (командный, текстовый, графический и др.).

Среди видов программных интерфейсов выделяют **интерфейс программирования приложений** (*англ.* ***Application Programming Interface***, ***API***, *иногда переводится как "интерфейс прикладного программирования"*) — интерфейс функций и связанных конструкций, которые можно использовать при написании программы, а также **двоичный интерфейс приложений** (*англ.* ***Application Binary Interface***, ***ABI***) — интерфейс, регламентирующий формат исполняемых файлов и библиотек. Всякая библиотека предоставляет некоторый интерфейс программи-

Исполняемые файлы (*англ.* ***executable***) программ имеют расширение `.exe` в ОС *Windows* и обычно не имеет расширения в ОС *GNU/linux*. И библиотеки и исполняемые файлы содержат машинный код функций, исполняемый файл отличается тем, что в нем имеется выделенный участок кода, которому передается управление при запуске.

рования приложений — набор функций и структур (типов, способов кодирования) данных. Двоичный интерфейс приложений регулирует, как именно записываются функции в файлах библиотек и программ, как осуществляется их вызов и т.д.

Среди типов пользовательских интерфейсов следует выделить два основных — **графический** (*англ.* ***Graphical User Interface, GUI***) и **интерфейс командной строки** (*англ.* ***Command Line User Interface, CLI***). Первый позволяет пользователю взаимодействовать с компьютером путем манипуляции с графическими изображениями на экране посредством устройства ввода (мышью, сенсором, возможно клавиатурой). Можно также выделить **текстовый пользовательский интерфейс** (*англ.* ***Text-based User Interface, TUI***), который отличается от графического тем, что объекты интерфейса представлены не в виде изображений, а в виде текстовых символов — собственно букв и формально текстовых символов **псевдографики** (*англ.* ***box-drawing charactes, line-drawing charactes***), отображающих не буквы, а, например, линии, прямоугольники и т.п.[1]. Интерфейс командной строки представляет собой взаимодействие с компьютером путем ввода строк — команд — и получении текстового ответа после выполнения требуемого действия или сообщения об ошибке.

Связанные с интерфейсом командной строки термины — **консоль** (*англ.* ***console***) и **терминал** (*англ.* ***terminal***) — устройства для взаимодействия человека с компьютером, содержащее в себе дисплей и клавиатуру (иногда и другие устройства ввода-вывода)[2]. Поэтому интерфейс командной строки также называют **консольным интерфейсом** (*англ.* ***console interface***), а программу для работы с консольными приложениями из графического интерфейса — **эмулятором терминала** (*англ.* ***terminal emulator***).

Интерфейс командной строки имеет два больших преимущества по сравнению с графическим: потребляет многократно меньше ресурсов (процессорного времени, оперативной и долговременной памяти), а также позволяет автоматизировать действия. Автоматизация относится как к воз-

[1]Примерами программ с таким интерфейсом могут служить файловые менеджеры *Far Manager* для *Windows* и *Midnight Commander* для *Unix*-подобных систем. Существуют и игры, реализованные посредством псевдографики, например, первые версии игры *Tetris* для *DOS*.

[2]Активно развивались и использовались до 1980-х – 1990-х. Терминал в общем виде — оконечное устройство, в данном случае пользовательское. Консоль — разновидность терминала, которая поддерживала именно интерфейс командной строки. У одного большого компьютера могло быть несколько терминалов для одновременной работы нескольких пользователей. Терминалы были как без дисплея (использовался телетайп), так и с текстовым и даже графическим дисплеем.

можности выполнить за одну команду большое количество манипуляций, так и к созданию **сценариев** (*англ.* ***script, скрипт***). Примером ситуации, требующей автоматической обработки большого количества файлов, может служить задача массового переименования файлов по шаблону. В графической среде ее решение требует ручного переименования каждого файла или поиск готовой программы, которая может быть недоступна или ограничена в функционале, средства командной строки позволяют решить за несколько команд в практически любой ситуации. Сценарий — программа особого рода, представляющая собой список команд, подлежащих выполнению последовательно автоматически, а не по нажатию на клавишу *Enter* после ввода каждой команды вручную. Под файлом сценария также понимают исполняемый текстовый файл, содержащий список команд. Интерфейс командной строки используется системными администраторами, продвинутыми пользователями и часто является единственным интерфейсом на серверных компьютерах, особенно в случае, когда к нему возможен только удаленный доступ.

Фактически, исполнение команд пользователя в интерфейсе командной строки или команд сценария представляет собой исполнение программы на некотором интерпретируемом языке программирования. Поэтому программу, обеспечивающую такой интерфейс, называют **командным интерпретатором** (*англ.* ***command-line interpreter***) или **интерпретатором сценариев** (*англ.* ***script interpreter***), а если такая программа представляет собой интерфейс операционной системой — **командной оболочкой** (*англ.* ***command shell***) операционной системы, а соответствующие сценарии — **сценариями оболочки** (*англ.* ***shell script***)[1]. Используется также термин **командный процессор** (*англ.* ***command processor***).

§1.6. Операционные системы

Операционная система, **ОС** (*англ.* ***operating system, OS***) — программное обеспечение, осуществляющее управление аппаратным оборудованием компьютера, исполняемыми программами и взаимодействие с пользователем. Дать точное однозначное определение операционной си-

[1]Программу для любого интерпретируемого языка можно считать сценарием, но не любой интерпретатор — командным интерпретатором (т.к. интерпретатор может не поддерживать консольный интерфейс ввода команд). Любую командную оболочку можно считать командным интерпретатором, но не любой командный интерпретатор можно использовать в качестве командной оболочки, т.к. основная задача оболочки ОС — работа с программами и файлами, в то время как не всякий интерпретируемый язык имеет удобные средства для этого.

стемы сложно, более того, существуют научные и политические дискуссии и споры о том, что считать операционной системой, где заканчивается операционная система, а где начинаются другие программы, среди которых выделяют **системные** (*англ.* ***system program***), предназначенные для управления операционной системой и компьютером, и **прикладные** (*англ.* ***application program***), предназначенные для решения пользовательских задач. Фактически данное решение обычно принимает производитель соответствующего программного продукта.

С теоретической точки зрения операционную систему следует понимать как ПО, решающее определенный класс задач:

- **абстрагирование аппаратного обеспечения** (*англ.* ***hardware abstraction***)[1];
- запуск программ и управление запущенными программами;
- управление **ресурсами** (*англ.* ***resource***) компьютера (оперативной памятью, процессорным временем и др.);
- предоставление базового пользовательского интерфейса.

Данные задачи решаются посредством **системных вызовов** (*англ.* ***system call***) — особых функций операционной системы, а также системной библиотеки функций. Таким образом, всякая операционная система предоставляет интерфейс программирования приложений (API), а также регламентирует формат приложений и библиотек (ABI). Заметим, что код приложений и библиотеки состоит не только из вызовов функций и системных вызовов, но и машинного кода, поэтому для полной совместимости приложений в двоичном формате на различных устройствах с различными ОС требуется совпадение архитектуры, API и ABI операционных систем. В данном контексте используется также термины **аппаратная** и **программная платформы** соответственно (*англ.* ***hardware platform*** *и* ***platform***), и просто (компьютерная) платформа, как объединяющий все характеристики типа среды выполнения программного обеспечения. В то же время исходный код с языка высокого уровня может быть переведен

[1] Часто встречающиеся в программировании термины "**абстракция**" и "**абстрагирование**" (*англ.* ***abstraction***) означают переход от реальных, физических, аппаратных, **низкоуровневых** (*англ.* ***low-level***) средств к программным, математическим, **высокоуровневым** (*англ.* ***high-level***) средств. При этом **слои** или **уровни** абстракции (*англ.* ***abstraction layer***) — это набор функциональных или программных возможностей (или прогаммный интерфейс), реализованный аппаратно или посредством возможностей абстракций более низкого уровня. Именно в этом смысле языки программирования высокого и низкого уровня называются таковыми — первые предоставляют набор высокоуровневых инструкций, абстрагированных от более низкоуровневых системных и еще более низкоуровневых аппаратных.

в двоичный для различных архитектур и операционных систем, если существует транслятор для соответствующей платформы, а исходный код программы использует только функции языка программирования, но не библиотек операционной системы и не содержит низкоуровневых аппаратных (машинных) команд.

Абстрагирование аппаратного обеспечения — замещение прямого обращения к аппаратуре компьютера системными вызовами. Решает две основные задачи.

- *Предоставление удобного и унифицированного программного интерфейса для разнородных аппаратных устройств.* Например, файл — абстракция, реализованная на уровне операционной системы. Обращение с файлами удобнее, чем непосредственно с носителями информации. Кроме того, при обращении программы к файлу не важно, где находится этот файл — на жестком диске, на карте памяти или в сетевом хранилище. Несмотря на разные аппаратные команды для работы с этими устройствами, файлы на них прикладными программами обрабатываются одинаково[1].
- *Обеспечение безопасности.* Например, при обращении к файлу операционная система проверяет, есть ли у пользователя право на доступ к данному файлу. Прямое обращение к носителю информации, который ничего не знает о файлах и пользователях, а работает исключительно с адресуемыми ячейками, исключает такую возможность[2].

Другой пример абстрагирования — виртуальная память. При нехватке оперативной памяти ОС может записать часть данных работающей программы на долговременный носитель (в файл или раздел т.н. “подкачки”) и освободить место для другой программы, а в дальнейшем “поменять их местами”. Для программ это происходит прозрачно (т.е. “незаметно” без изменений в коде программы, разве что пользователь может заметить потерю производительности). Безопасность виртуальной памяти заключается в изоляции данных работающих программ — они не могут прочитать

[1] Унификация реализуется с помощью **драйверов устройств** (*англ.* ***device driver***), являющихся частью ОС или предоставляемых производителем оборудования, и обеспечивающих передачу специфичных для оборудования команд при одинаковых запросах от ОС.

[2] Безопасность обеспечивается поддержкой на уровне центрального процессора двух режимов работы — **пользовательского** (*англ.* ***user mode***) и **режима ядра**, также называемого привилегированным или режимом супервизора (*англ.* ***kernel mode***). В первом режиме запрещены небезопасные машинные команды, в частности прямое обращение к оборудованию, во второй может перейти только операционная система, точнее ее ядро (см. ниже), обеспечивающее интерфейс системных вызовов.

данные в памяти друг друга без разрешения ОС (ее администратора). Код программы содержит обращение не к физическим адресам оперативной памяти, а к виртуальным, конвертируемым в физические процессором в соответствии правилами, задаваемыми операционной системой. Набор виртуальных адресов, доступных программе, называется *виртуальным адресным пространством*.

Запуск программ приводит к созданию **процессов** (*англ.* ***process***) — единиц операционной системы, условно представляющих программу во время ее исполнения. *Программа* как продукт представляет собой статический набор файлов, среди которых могут быть исполняемые, библиотеки, встраиваемые модули, текстовые и двоичные вспомогательные и конфигурационные файлы самой программы (изображения, видео, документация и др.), причем исполняемых файлов может быть несколько, запуск каждого из них приводит к выполнению определенного действия. Поэтому точнее будет сказать, что процесс — запущенный экземпляр исполняемого файла. Строго говоря, программа может создать не один процесс, более точное определение процесса — абстракция операционной системы, представляющий собой набор команд (машинных инструкций), подлежащих исполнению. В современных ОС процессы исполняются в раздельных виртуальных адресных пространствах памяти. С процессами связываются рабочие файлы и другие элементы среды исполнения процесса данной операционной системы.

Операционная система решает ряд подзадач управления процессами.

- Создание, завершение и приостановку процессов.
- Распределение процессам процессорного времени[1], выделение виртуальной и оперативной памяти и др. аппаратных и программных вычислительных ресурсов.

Именно процесс, как динамическая единица, последовательность исполняющихся команд, производит вычисления и иные операции над данными, осуществляет системные вызовы — в том числе для ввода и вывода, а также для создания других процессов. Создание и завершение процесса — системный вызов. Любой процесс может создать другой процесс, используя соответствующий системный вызов, тем самым инициировав начало исполнения заданной последовательности инструкций. Действительно, запуск исполняемого файла по сути представляет собой создание процесса исполнения инструкций из этого файла, осуществляемый, напри-

[1]Обычно в современных ОС исполняется куда больше процессов, чем имеется процессоров, поэтому происходит периодическое и довольно частое — каждые несколько десятков миллисекунд — переключение между ними, что приводит к кажущемуся одновременному их выполнению, **многозадачности** (*англ.* ***multitasking***).

мер, средствами пользовательского интерфейса операционной системы. Процесс, инициировавший создание процесса, по отношению к созданному процессу называется **родительским** (*англ.* ***parent***), а созданный процесс по отношению к родительскому — **дочерним** (*англ.* ***child***) процессом.

Часть операционной системы, обеспечивающая абстрагирование оборудования и управление оборудованием, ресурсами и процессами, называется **ядром** (*англ.* ***kernel***). Пользовательский интерфейс предоставляется с помощью **оболочки** (*англ.* ***shell***) операционной системы. Она может быть ее составной частью или отдельным программным продуктом.

Схематически роль операционной системы можно изобразить следующим образом (компоненты ОС выделены курсивом, компоненты ядра — жирным шрифтом):

<table>
<tr><td colspan="2">Пользователь</td></tr>
<tr><td>Прикладные программы</td><td>Оболочка ОС</td></tr>
<tr><td colspan="2">Системные библиотеки (API)</td></tr>
<tr><td colspan="2">Системные вызовы</td></tr>
<tr><td colspan="2">Управление аппаратными ресурсами и процессами</td></tr>
<tr><td colspan="2">Слой абстракции аппаратного обеспечения</td></tr>
<tr><td colspan="2">Оборудование компьютера</td></tr>
</table>

§1.7. История семейства ОС *Unix* и стандарт *POSIX*

ОС ***Unix***[1] (“юникс”) была разработана в 1970-х как операционная система для компьютеров серии *PDP* фирмы *DEC* сотрудниками компании *Bell Labs*, подразделении компании *AT&T* (Кеном Томпсоном, Деннисом Ритчи, Дугласом Макилроем и др.). Кеном Томпсоном и Деннисом Ритчи также разработан язык программирования *Си* для написания кода ОС *Unix* и приложений под него. Данная система изначально была многопользовательской и многозадачной. Система распространялась платно, имела закрытый исходный код.

В 1980-х годах компьютеры *PDP* устаревают и уходят с рынка. Система *Unix* остается качественной, удобной и привычной в администрировании и использовании, но не может быть перенесена на новое оборудование

[1] Uniplexed Information and Computing Service — Объединенная информационно и вычислительная служба, в противоположность разрабатываемой ранее, но не получившей широкое распространение ОС Multics, Multiplexed Information and Computing Service — Мультиплексная информационная и вычислительная служба. Сочетание CS в сокращении *UNICS* было заменено на омофоничное X.

напрямую, в том числе из-за закрытости исходного кода, что ставит перед разработчиками задачу разработки новой операционной системы с аналогичным функционалом.

В 1983 году Ричард Столлман начинает разработку ***GNU***[1] — свободную *Unix*-подобную операционную систему. Значительная часть программного обеспечения, включая компилятор языка *Cu* и ряд утилит, была разработана, однако ядро операционной системы, *GNU Hurd*, так и не было создано.

В 1991 году Линус Торвальдс опубликовал ядро ***Linux***[2] ("линукс"), которое, совместно с программным обеспечением *GNU*, позволило реализовать рабочую операционную систему. Операционная система *GNU* и ядро *Linux* вместе составляют ОС, известную, как ***GNU/Linux***. В дальнейшем в силу открытости исходного кода на базе *GNU/Linux* различными компаниями и энтузиастами было разработано большое количество т.н. **дистрибутивов** (*англ.* ***Linux distribution, distro***) — операционных систем, включающих в себя *GNU/Linux* и другое программное обеспечение[3].

С 1980-х годов разрабатываются и другие системы, реализующие функционал *Unix* — *BSD*, *Solaric*, *MacOS* и др., среди которых как свободные, так и частично или полностью закрытые. Имеются и операционные системы, основанные на ядре *Linux*, но не использующие приложения *GNU*, например *Android* и некоторые **встроенные** (*англ.* ***embedded***) системы для различных устройств. В некоторых из них в качестве реализации основных команд и приложений оболочки используется *BusyBox*, в других таким приложения могут вообще отсутствовать. Таким образом, появилось целое семейство схожих, но различных операционных систем, известное как семейство *Unix*-подобных операционных систем или семейство ОС **nix*[4]. В настоящее время ОС данного семейства, в особенности основанных на ядре *Linux*, применяются очень широко: в персональных настольных и портативных компьютерах, мобильных устройствах, встроенных системах бытовых приборов и компьютерной техники, на серверах,

[1] *GNU* читается как "гну" (антилопа гну — талисман данной ОС) и расшифровывается как *GNU is Not a Unix*, *GNU — это не Unix* (подобные "рекурсивные аббревиатуры" получили популярность в названиях многих программ).

[2] От имени создателя — Linus и буквы X в названии *Unix*

[3] Нередко говорят о ОС *Linux*, что по указанным причинам не является корректным. Обобщенно можно говорить об ОС *GNU/Linux*, ОС на базе *GNU/Linux* или использовать названия конкретных дистрибутивов — ОС *Fedora*, ОС *Ubuntu* и др.

[4] В начале 1980-х также была разработана операционная система *CP/M* для компьютеров фирмы *IBM*, которая стала основой для семейства однозадачных и однопользовательских ОС *DOS*, и семейства ОС *Windows*, где была внедрена многозадачность, а затем и многопользовательский доступ.

суперкомпьютерах, космических станциях и др. оборудование. В целях унификации разнообразия систем этого семейства был разработан стандарт ***POSIX*** (*Portable Operating System Interface — переносимый интерфейс операционных систем*[1].). Данный стандарт регламентирует команды оболочки ОС и системные вызовы ОС[2]. Строгое следование стандарту *POSIX* позволяет создавать переносимые приложение на языке *Cи* и сценарии выполнения действий посредством оболочки.

В реальности *Unix*-подобные операционные системы не всегда точно следуют стандарту (хотя эти отклонения обычно некритичны), а также могут существенно расширять возможности, регламентированные в *POSIX*. Таким образом, например, если в программе используются *GNU*-специфичные или *Linux*-специфичные расширения, программа не сможет быть скомпилирована и запущена на *BSD* или *MacOS*.

Информация, приведенная в дальнейшей части данного пособия относится преимущественно к *GNU/Linux*, но бóльшая часть приведенных конкретных решений является *POSIX*-совместимой или может быть адаптирована под другие *Unix*-подобные системы.

§1.8. Файловая система

Под **файлом** (*англ.* ***file***) понимают поименованную упорядоченную совокупность байтов. Файлы объединяются в **каталоги** (*англ.* ***directory***, *директории*) — списки, находящихся в них файлов и других каталогов[3] — **подкаталогов** (*англ.* ***subdirectory***) данного каталога, который по отношению к подкаталогам называется — **родительским** (*англ.* ***parent directory***). В каждом каталоге может присутствовать только один файл или каталог с заданным именем, хотя имена разных файлов в разных каталогах могут повторяться. В *Linux* имена файлов регистрозависимы, то есть `file`, `File` и `FILE` — разные имена файлов. Каталоги образуют **дерево каталогов** (*англ.* ***directory tree***) — иерархическую структуру, начинающуюся с **корневого каталога** (*англ.* ***root directory***). Под **именем файла** (*англ.* ***file name***) чаще всего понимается имя файла в каталоге, в котором этот файл содержится, а всю иерархию каталогов, начиная с

[1] Буква “X” в аббревиатуре *POSIX* пришла от *Unix*

[2] Точнее, функции API, которые в реализации могут быть более высокоуровневыми, чем системные вызовы.

[3] Введенный в ОС *Windows* термин **папка** (*англ.* ***folder***) также начал использоваться в графических приложениях *Linux*. Вообще говоря, в *Windows* папки и каталоги не тождественны: например, каталога “Компьютер” не существует, хотя такая папка имеется, папка “Рабочий стол” — верхняя в дереве папок, но не является таковой в дереве каталогов и т.д.

корневого называют **путем к файлу** (*англ.* ***path to file***) (или каталогу). Имя файла вместе с путем также называется **полным именем** (*англ.* ***full name***) или **полным путем** (*англ.* ***full path***).

Используется так же термин **расширение** (*англ.* ***extension***) файла — в *Unix* это часть имени файла, следующая после последней точки[1]. Хотя расширение и является частью имени, часто под именем файла с пользовательской точки зрения понимают именно часть имени без расширения. Расширение не несет формальной нагрузки для ФС в *Unix*-подобных ОС, но призвано информировать пользователя и программы о формате файла. Например, расширение `.so` — библиотека, `.c` — файл исходного кода на языке *Cu*, `.txt` — текстовый файл и т.д. По этой причине иногда слово "тип" (формат) по отношению к файлу используется как синоним к слову расширение, а сами расширения выступают как синонимы имен форматов файла[2].

Термин **файловая система**, **ФС** (*англ.* ***file system, FS***) может относиться к

- файловой системе конкретного носителя информации или его раздела[3] — способе представления файлов на носителе; примерами таких файловых систем служат *NTFS* и *FAT* для *Windows*, *ext4* и *btrfs* для *Linux* и другие; у файловой системы носителя есть свой корневой каталог и дерево каталогов, не зависящие от ОС и компьютера, к которому подключен раздел;

- файловой системе операционной системы в целом, в *Linux* называемой **виртуальной файловой системой**, (*англ.* ***Virtual File System, VFS***), представляющей собой абстракцию для доступа к файлам, совокупность файловых систем всех подключенных разделов, структура которой зависит от ОС, ее настроек и текущего состояния системы. Доступ к файлам из приложений осуществляется по уникальным именам файлов в виртуальной файловой системе, то есть по именам, включающим путь к файлу.

В *Unix*-подобных ОС *VFS* представляет собой иерархию каталогов, начиная с корневого, имя которого `/`. Уникальное имя файла формиру-

[1] Расширение может отсутствовать — в этом случае точки в имени файла нет вообще. Файл может заканчиваться точкой, тогда расширение считается пустым, но такое используется крайне редко.

[2] В силу регистрозависимости, расширение, записанное заглавными буквами, может быть не воспринято программой, обрабатывающей файлы соответствующего типа, но записанного только строчными буквами.

[3] а также сетевого ресурса (например, NFS), образа раздела, записанного в виде файла в другой файловой системе, или любого другого "источника" файлов, видимого в ОС как ФС

ется путем перечисления каталогов в иерархии, также разделяемых символом `/`. В качестве корневого каталога *VFS* устанавливается корневой каталог дисковой ФС выбранного носителя (раздела носителя) информации. Другие разделы **монтируются** (*англ.* ***mount***) в некорневые каталоги *VFS* — **точки монтирования** (*англ.* ***mount point***) (как правило, пустые), при этом содержимое корневого каталога монтируемого раздела становится содержимым точки монтирования[1]. В большинстве случаев ОС для настольных компьютеров способны автоматически монтировать и размонтировать из графической оболочки сменные носители. Монтирование производится при загрузке в соответствии с настройками ОС, начальная настройка устанавливается при установке ОС.

Unix — система, основанная на принципе "все есть файл". Этот принцип полностью или частично поддерживается современными *Unix*-подобными системами: все данные, устройства и другие объекты пользователя, программ и самой ОС доступны в виде файлов, в том числе дисковые разделы. Таким образом, кроме обычных файлов в *Linux* существуют файлы других типов, вот некоторые из них:

Знак	Название	Описание
`-` или `f`	**обычный** (**регулярный**) файл (***regular file***)	
`d`	**каталог** (***directory***)	считается разновидностью файла
`b`	**блочное устройство** (***block device***)	накопители данных и их разделы (в частности, содержащие ФС, которые можно подмонтировать)
`c`	**символьное устройство** (***character device***)	другие устройства (мышь, клавиатура, экран и т.д.)
`l`	**символическая ссылка** (***symbolic link***)	содержит в себе относительный или абсолютный путь к другому файлу или каталогу, при обращении к ссылке операции будут фактически производиться с тем файлом (если он существует)
`p`	**канал** (***pipe***)	предназначен для обмена информацией между процессами
`s`	**сокет** (***socket***)	предназначен для обмена информацией между процессами по сети

[1] Такое понятие, как "буква диска" отсутствует.

Различают символические и **жесткие** (*англ.* ***hard link***) ссылки. При обращении к ссылке (чтении, дозаписи и перезаписи данных) операция совершается с тем файлом, на который указывает ссылка. Символические ссылки — особый тип файла, содержащие строку (путь) к другому файлу. Символическая ссылка может быть как на файл на том же носителе, так и вообще на любой файл любого типа (в т.ч. каталог) файловой системы ОС, в том числе несуществующий в данный момент (тогда при попытке работы с содержимым по ссылке произойдет ошибка). Жесткие ссылки представляют собой ссылку на файл той же ФС раздела, что и исходный файл. С символическими ссылками их роднит процесс создания (с помощью команды `ln`), но они не представляют собой особый тип файла, поэтому в таблице выше не указаны. Фактически жесткие ссылки не являются ссылками, точнее один и тот же файл (в смысле последовательности байт) раздела может быть доступен под разными именами в разных каталогах этого раздела. Тогда все эти имена файлов считаются жесткими ссылками на один и тот же файл и изменение содержимого по любому из имен будет выглядеть как изменение содержимого каждого из этих файлов.

Наиболее корректно будет сказать, что каждый файл файловой системы раздела (как последовательность байт) имеет уникальный номер. Запись о файле в каталоге представляет собой имя файла и ссылку на уникальный номер этого файла в файловой системе. Такая ссылка и считается жесткой ссылкой на данный файл, на каждый файл может быть одна или несколько жестких ссылок в дереве каталогов раздела[1]. Ссылки, особенно символические, на практике используются довольно часто. Изучение работы с каналами и сокетами лежит за рамками данного пособия.

Всякий файл снабжается рядом **атрибутов** (*англ.* ***attribute***) — дополнительных данных (метаданных), включающих в себя в зависимости от файловой системы и операционной системы, например, такую информацию как тип файла, права доступа, владельца файла, время создания, последнего доступа и последнего изменения файла, размер файла и др.[2].

Следует отметить, что файлы, чьи имена начинаются с “.”, считаются служебными и по умолчанию скрываются при просмотре содержимого каталога и в файловых менеджерах.

В *Linux* принята жесткая система каталогов, поддерживаемая создателями дистрибутивов и программ для них. В корневом каталоге *Linux*

[1] Жесткие ссылки на каталоги не допускаются, исключение — каждый каталог содержит жесткие ссылки “.” и “..” — на себя и родительский каталог соответственно.

[2] Эта информация относится к собственно файлу раздела, а не записи о нем в каталоге, поэтому является общей для всех жестких ссылок на один и тот же файл.

типично содержатся следующие каталоги:

имя	назначение
`bin`	исполняемые файлы прикладных программ (обычно ссылка на `/usr/bin`)
`boot`	файлы загрузчика ОС
`etc`	файлы настройки ОС
`home`	домашние каталоги пользователей

`lib`
`lib32`
`lib64`
	библиотеки (обычно ссылки на `/usr/lib`, `/usr/lib32`, `/usr/lib64`)

`opt`	файлы программ сторонних производителей (не данного дистрибутива)
`proc`	особая ФС для отображения запущенных процессов
`root`	домашний каталог администратора
`run`	временные файлы, создаваемые при работе систем для пользователя и приложений
`sbin`	исполняемые файлы системных программ (обычно ссылка на `/usr/sbin`)
`sys`	особая ФС, отображающая работающие элементы ядра операционной системы (драйверы, устройства и т.д.)
`tmp`	временные файлы, создаваемые программами для своих нужд
`usr`	файлы программ (исполняемые, библиотеки, изображения, документация и др.)
`var`	различные файлы ОС и приложений, создаваемые при работе, обычно для длительного хранения

Чаще всего `run` и `tmp` — файловые системы, данные которых хранятся в оперативной памяти и уничтожаются при выключении компьютера, `dev`, `proc` и `sys` — особые файловые системы, отражающие состояние управления компьютером ядром ОС. Каталоги `bin`, `lib` и `sbin` сохранены в исторических целях, когда `usr` мог быть отдельным (а иной раз и сетевым, общим для всех компьютеров предприятия для экономии места) разделом: в него устанавливались “большие” пользовательские приложения, в то время как небольшие, но критические для запуска и исправления проблем загрузки ОС приложения устанавливались в данные каталоги.

По той же причине домашний каталог администратора `root` не находится внутри `home`[1]. Обычно домашний каталог пользователя — подкаталог `home`, чье имя совпадает с именем пользователя, например `/home/user`. Рядовой пользователь имеет право записи только в свой домашний каталог, при этом права чтения файлов в домашних каталогах других пользователей нет.

§1.9. Вход в систему, графический и текстовый терминал

Текстовый интерфейс командной строки является первичным для большинства *Unix*-подобных систем, предоставляя всю полноту работы с ядром ОС, настройкой и обслуживанием ОС, работой с файловыми системами и файлами, а также является интерфейсом, в который изначально загружается ОС. По понятным причинам, варианты дистрибутивов ОС, рассчитанных на пользователей, настроены на автоматическое подключение графического режима и графического интерфейса. В большинстве случаев есть возможность переключиться в текстовый режим, например нажатием сочетания клавиш *Ctrl*+*Alt*+одной из функциональных клавиш — это сочетание производит переключение между независимыми **виртуальными терминалами** (*англ.* ***virtual terminal***) или **виртуальными консолями** (*англ.* ***virtual console, VC***)[2], среди которых есть как текстовые, так и графические, на которых могут работать разные пользователи. Одному пользователю, как правило, разрешается работать в нескольких текстовых виртуальных терминалах, работа в двух графических терминалах может вызвать конфликт и не рекомендуется. При нахождении в текстовом терминале обычно для переключения на другой терминал достаточно нажать *Alt*+функциональная клавиша, в графическом обычно

[1] В настоящее время `usr` и `home` по-прежнему могут быть отдельными и сетевыми разделами, причем загрузка и устранение проблем загрузки происходит не за счет приложений, установленных в ОС, а благодаря особой инициализирующей ФС — ее файл-образ находится внутри каталога `boot`, он загружается в память сразу после загрузки ядра, после чего запускается сценарий загрузки с него. Ядро также находится в каталоге `boot`. При проблемах загрузки администратор может войти в командную строку инициализирующей ФС и решить их.

[2] Хотя эти терминалы и работают на реальном оборудовании, их считают виртуальными, так как они представляют собой программную единицу ОС, работающую поверх оборудования. Они независимы в том смысле, что на них могут содержаться разные изображения, запущены разные программы, вводиться разные команды — виртуальные терминалы представляют собой виртуализацию нескольких физических терминалов, одновременно подключенных к компьютеру.

дополнительно требуется нажимать *Ctrl*[1]. Переключение в текстовый терминал бывает полезно при наличии проблем с графическим из-за сбоя или нехватки ресурсов. Также работа с интерфейсом командной строки возможна из эмулятора терминала. При входе в текстовый терминал и запуске эмулятора терминала будет запущена оболочка, установленная для данного пользователя по умолчанию.

В текстовых терминалах (виртуальных консолях) возможно "пролистывать" вывод к более ранним (ушедшим на верх) строкам и обратно с помощью сочетаний *Shift+PageUp* и *Shift+PageDown*, однако буфер в них не очень большой (несколько экранов). Графические эмуляторы терминала, как правило, имеют полосу прокрутки и возможность настройки размера буфера.

Unix — изначально многопользовательская система, поэтому за редким исключением системы настраиваются на то, чтобы при запуске (при открытии как текстового, так и графического виртуального терминала) проводить **аутентификацию** (*англ.* ***authentication***) пользователей — процедуру проверки подлинности пользователя. Чаще всего осуществляется путем ввода **логина** (*англ.* ***login***) — открытого уникального имени пользователя — и **пароля** (*англ.* ***password***) — секретному слову пользователя, но возможны и другие варианты, например использование электронного ключа или смарт-карты. Среди всех пользователей выделяется пользователь с логином `root` — это администратор системы, имеющий полный доступ к системе. Остальные пользователи — "обычные", **непривилегированные** (*англ.* ***normal***, ***regular user***) пользователи — имеют ограниченный доступ к файлам и оборудованию[2].

Следует отметить, что из соображений безопасности даже работа системного администратора или единственного пользователя домашнего компьютера должна осуществляться из ограниченного аккаунта. Привилегированный аккаунт используется только для выполнения отдельных действий, требующих вмешательство в систему. Во многих

[1]Сочетания клавиш для различных систем, дистрибутивов и настроек могут изменяться. Типично наличие 8 виртуальных терминалов, то есть активны сочетания *Alt+F1*, *Alt+F2*, ..., *Alt+F8*, но это также может быть изменено. Некоторые терминалы могут содержать экраны загрузки ОС и быть недоступны для работы, графический терминал может быть запущен "поверх" текстового, с сохранением содержимого последнего, но без возможности доступа.

[2]Связанные с аутентификацией понятия — **идентификация** (*англ.* ***identification***) — присвоение пользователю уникального идентификатора, по которому система будет "узнавать" аутентифицированного пользователя, и **авторизация** (*англ.* ***authorization***) — получение пользователем разрешение на выполнение действия. Уникальный идентификатор представляет собой целое число, причем именно пользователь с идентификатором 0 является администратором системы (`root`).

системах вход в привилегированный аккаунт (особенно в графическом режиме) запрещен настройками, переключение в него осуществляться специальными средствами, описанными ниже.

Вопросы для самопроверки.

1. *Что такое машинная команда?*
2. *Что такое машинный код?*
3. *Что такое архитектура процессора?*
4. *Опишите понятие исходного кода программы.*
5. *В чем состоит различие между интерпретатором и компилятором?*
6. *Что такое функция (подпрограмма)?*
7. *Дайте описание понятия "Интерфейс".*
8. *Опишите, что такое интерфейс программирования приложений (API).*
9. *Какие существуют виды пользовательского интерфейса?*
10. *Опишите процесс работы с командной строкой.*
11. *Что такое командный интерпретатор?*
12. *Что такое эмулятор терминала?*
13. *Что представляет собой сценарий оболочки?*
14. *Что такое операционная система?*
15. *Какие задачи решает операционная система?*
16. *Что представляют собой системные вызовы?*
17. *Приведите примеры абстрагирования аппаратного обеспечения.*
18. *Что такое ядро операционной системы?*
19. *Что такое процесс?*
20. *Что такое оболочка операционной системы?*
21. *Опишите различия между программными продуктами Unix, Linux, GNU.*
22. *Что такое POSIX?*
23. *Что такое виртуальная файловая система?*
24. *Опишите результат монтирования файловой системы.*

25. Чем отличается символическая ссылка от жесткой?

26. Какой логин имеет пользователь с административными полномочиями?

Глава 2. Работа в оболочке ОС семейства *Unix*

§2.1. Оболочка командной строки, встроенные и внешние команды, переменные окружения

Стандартная оболочка *POSIX* имеет название *Bourne Shell*, команда для ее запуска — `sh`, поэтому данную оболочку часто называют просто *Shell* или *Sh*. В *Linux* и других системах часто используется другая оболочка — ***GNU Bash*** — *Bourne Again SHell*, поддерживающая все возможности оболочки *POSIX Shell*, но существенно их расширяющая. Данное пособие посвящено преимущественно оболочке *Bash*.

При запуске оболочка выводит на экран **приглашение командной строки** (*англ.* ***command prompt***) — строку, означающую готовность к приему команд. Содержание приглашения настраиваемо, там может отображаться имя пользователя, имя компьютера, текущий каталог (см. ниже), дата, время и другая информация. Обычно приглашение завершается символом `$` или `%`, если пользователь является обычным пользователем, и символом `#`, если выполнен вход от администратора. В данном пособии при описании формата и запуска команд оболочки мы будем использовать в качестве приглашения оболочки одиночный символ `$`, если описываемая команда может быть исполнена как от обычного пользователя, так и от администратора и `#`, если требуются административные привилегии. Например

```
$ cal
```

команда, выводящая на экран календарь на текущий месяц, не требует административных привилегий,

```
# mount /dev/sda1 /mnt/hdd
```

команда, монтирующая разделы в ФС (в данном случае первый раздел первого жесткого диска в каталог `/mnt/hdd`), требует административных привилегий.

Любая команда оболочки после ввода и нажатия на клавишу *Enter* приводит к немедленному выполнению некоторого действия. Команды оболочки бывают **внутреннми** (*англ.* ***internal command***), называемые также **встроенными** (*англ.* ***embedded***), и **внешними** (*англ.* ***external***

command). Внутренние команды исполняются самой оболочкой (встроены в нее), внешние команды представляют собой исполняемые файлы: всякий исполняемый файл может быть выполнен как команда оболочки. В общем случае для исполнения файла нужно указать путь к этому файлу в качестве команды. Если файл расположен в каталоге, перечисленном в переменой окружения `PATH`, то данный файл будет найден и запущен даже без указания пути к нему, то есть просто по записи имени файла в качестве команды[1].

Переменные окружения или **переменные среды** (*англ.* ***environment variables***)[2] — пары вида имя–значение, где имя и значения представляют собой строки символов. Система предоставляет набор переменных окружения всякому работающему процессу, при этом процесс может их изменять и передать запускаемому из него процессу, но не повлиять на переменные процесса, запустившего данный.

Просмотреть значение переменной окружения можно с помощью команды ***echo***, в общем виде выводящей на экран указанный текст. Чтобы текстом стало значение переменной необходимо предварить ее имя знаком $. Например,

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

вывод значение переменной `PATH`. В переменной `PATH` пути поиска должны разделяться символом двоеточия. Изменить значение переменной можно указав ее имя (без знака доллара) и знак равенства, после чего значение (отсутствие пробелов до и после знака равенства является существенным).

```
$ PATH=/usr/bin:/usr/sbin:/opt/bin
$ echo $PATH
/usr/bin:/usr/sbin:/opt/bin
```

Того же эффекта — добавления каталога к существующему списку — можно было достичь, используя старое значение переменной прямо в команде:

```
$ PATH=$PATH:/opt/bin
$ echo $PATH
/usr/bin:/usr/sbin:/opt/bin
```

Суть в том, что Bash сначала раскрывает значения переменных, а потом исполняет команду. То есть, если в переменной PATH записана строка `/usr/bin:/usr/sbin`, то команда

```
$ echo PATH=$PATH
```

[1] Внутренняя команда `type` оболочки *Bash* позволят определить является ли заданная команда внутренней, внешней или другим объектом.

[2] Термин относится именно к переменным (чего?) окружения или среды, а не переменности самого окружения или среды. Среда в свою очередь выступает синонимом к слову “окружение”, а не обозначением дня недели.

раскрывается в

```
$ echo PATH=/usr/bin:/usr/sbin
```

а затем исполняется, что и приводит к выводу

```
PATH=/usr/bin:/usr/sbin
```

а команда

```
$ PATH=$PATH:/opt/bin
```

раскрывается в

```
$ PATH=/usr/bin:/usr/sbin:/opt/bin
```

что и приводит к записи требуемого значения переменной при исполнении.

В приведенных примерах в имени и значении переменной не было символов пробела "␣". Использование пробелов и других специальных символов (кроме букв и цифр) в имени переменной не допускается. Наличие подобных символов в значении переменной — довольно типичная ситуация. При присвоении значения переменной наличие пробела после знака равенства будет воспринято оболочкой специальным образом. Чтобы избежать этого, необходимо заключить значение (или хотя бы его часть, содержащую пробел) в двойные кавычки, например,

```
$ PATH="$PATH:/opt/path with spaces/bin" или
$ PATH=$PATH:"/opt/path with spaces/bin"
```

В целом оболочка устроена так, чтобы различия в запуске внутренних и внешних команд не ощущалось. Если указан не путь к исполняемому файлу, а просто имя команды, то оболочка сначала пытается найти такую внутреннюю команду, если таковой не имеется — перебирает все каталоги в переменной `PATH` (слева направо, без захода в подкаталоги), при нахождении такого файла он сразу же запускается (т.е. каталоги, которые указаны раньше имеют приоритет, а внутренняя команда имеет приоритет над внешней). Если файл не найден, выводится сообщение о несуществующей команде[1]. Для запуска исполняемого файла одноименного внутренней команде или выборе конкретного файла из нескольких в разных каталогах `PATH` нужно указывать путь к нему.

Внутренние и внешние команды — утилиты командной строки — выводят результат своей работы на экран (например *echo*) и завершают свою работу, только после этого оболочка позволит ввести следующую команду. Подобным образом можно запускать и графические приложения из оболочки командной строки, запущенной в графическом эмуляторе терминала — просто указав его имя (или полный путь при необходимости) — однако, если не предпринять дополнительных мер, в оболочке нельзя будет ввести следующую команду до завершения работы с графическим

[1]Некоторые системы могут предложить установить программу, предоставляющую соответствующую команду, если таковая имеется, и ее установщик доступен в сети.

окном. Чтобы избежать этого, следует указать знак & в конце команды, например,

```
$ leafpad &
```

для запуска простого текстового редактора. Тогда окно редактора запустится в фоне (дочерний процесс создается как **фоновый** (*англ.* ***background***), то есть работающий параллельно процессу оболочки без ее приостановки), а в командной строке можно будет продолжать работу — ввод новых команд.

Завершение работы в оболочке осуществляется с помощью команды

```
$ exit
```

При этом, если данный конкретный процесс оболочки не был запущен из другой оболочки, то это приведет к закрытию окна эмулятора терминала или выходу из системы, если оболочка была запущена в виртуальной консоли. Также будут завершены все фоновые дочерние процессы данной оболочки (что может привести к потере несохраненных данных в них).

§2.2. Рабочий каталог и относительные пути

У каждого работающего процесса есть **рабочий** или **текущий** каталог (*англ.* ***current working directory***) — при запуске процесса он устанавливается в текущий каталог процесса, запустившего процесс, но может быть изменен в любое время самим процессом с помощью соответствующего системного вызова (что никак не повлияет на текущий каталог исходного процесса). Сменить текущий каталог в оболочке можно с помощь команды `cd`, указав в качестве аргумента нужный каталог[1]. Например,

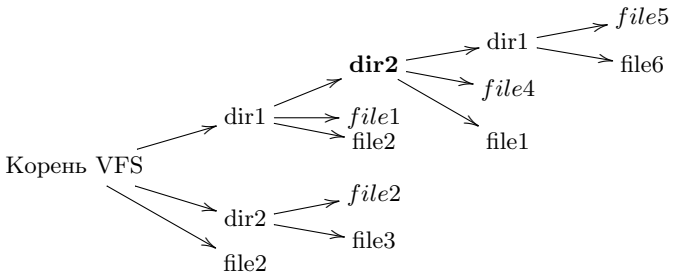
```
$ cd /path/to/dir
```

При этом часто приглашение командной строки оболочки настроено отображать текущий каталог, а команда `pwd` выводит его на экран в явном виде, например,

```
$ pwd
/path/to/dir
```

Кроме задания полного пути (имени) файла, начиная с корневого — **абсолютного пути** (*англ.* ***absolute path***) — можно указывать **относительный** (*англ.* ***relative***) путь. При этом, если путь не начинается с `/`, то он считается относительным. В относительном пути можно использовать имена каталогов "`.`" — ссылка на текущий каталог, и "`..`" — ссылка на родительский каталог. Рассмотрим пример структуры файлов и каталогов.

[1] Команда `cd` является внутренней, так как изменение текущего каталога в процессе исполнения внешней команды не скажется на текущем каталоге оболочки.



Пусть текущий каталог — `/dir1/dir2` (выделен жирным). Тогда относительным путем к файлу `file4` в нем будет `file4` или `./file4`, к `file5` в подкаталоге `dir1` — `dir1/file5` или `./dir1/file5`, к файлу `file1` в каталоге `/dir1` — `../file1`, к файлу `file2` в каталоге `/dir2` — `../../dir2/file2` и т.д. (упомянутые файлы выделены курсивом).

Хотя в большинстве случаев программы "понимают" относительный путь, начинающийся с имени файла или подкаталога, иногда требуется указывать `./` перед ним — например, чтобы запустить программу, находящуюся в текущем каталоге следует указывать

```
$ ./file4
```

а не

```
$ file4
```

так как текущий каталог обычно не указывается в переменной окружения `PATH`, последнее приведет к сообщению о неизвестной команде.

Просмотреть текущий каталог можно с помощью команды `pwd`, также текущий каталог является значением переменой окружения `PWD`.

§2.3. Структура команд оболочки, справочная система

Команды оболочки могут иметь **аргументы** (*англ.* ***arguments***), а также **опции** (*англ.* ***options****, используется также термин ключи*), которые в свою очередь могут иметь или не иметь *аргумент*. Аргументы предназначены для передачи информации о том, что должна обработать команда, а опции влияют на то, как именно команда будет работать. Опции бывают короткие (из одной буквы) и длинные (из нескольких букв). Короткие опции предваряются одиночным символом дефиса "`-`", а длинные — двойным, "`--`". Иногда у короткой опции есть длинный (полный) синоним.

Строго говоря, аргументы команды представляют собой строки. Команде (и всякому процессу) передается **массив** (*англ.* ***array***) строк-аргу-

ментов[1]. В командной строке при запуске аргументы указываются после команды и разделяются пробелами.

Аргументы считаются опциями, если начинаются с дефиса. Условимся называть **параметрами**[2] те аргументы, которые не являются опциями (*англ.* ***non-option argument***). Опции и другие параметры могут следовать в произвольном порядке. Параметры могут быть обязательными и необязательными, опции могут быть обязательными или взаимоискючающими — если указана одна из опций, другая не может присутствовать или игнорируется. Опции, начинающиеся с одиночного дефиса, являются однобуквенными[3], при этом, если после дефиса указано несколько букв, то это эквивалентно указанию каждой из этих опций. Опции, начинающиеся с двойного дефиса являются многобуквенными.

Следует отметить, что у программ (внешних команд), не следующих стандарту *POSIX* и стилю *GNU*, могут быть собственные стили синтаксиса аргументов, но здесь речь пойдет о типичной для командной оболочке стилистике.

Рассмотрим вышесказанное на примере синтаксиса команды `ls`. Данная команда выводит содержимое каталога — список файлов и каталогов в нем — на экран. Ее параметры и опции могут быть записаны так[4]

```
ls [-Aadl1] [--help] [--version] [--sort={time|size|extension}] [--] [файл...]
```

Параметр команды обозначен как "`[`*`файл...`*`]`": взятие в квадратные скобки означает, что аргумент необязательный, а троеточие указывает на то, что файл может быть не один, а несколько (они должны быть разделены пробелами). Напомним, что текст, представленный прямым моноширинным шрифтом может вводиться в компьютер "как есть", а моноширинный курсив использован для обозначения смысла значения аргумента, подлежащего подстановке на фактическое значение.

Под "файлом" в списке аргументов понимаются имена файлов (не обязательно "обычных", т.е. это могут быть и каталоги и другие типы файлов, если не оговорено иное). Имена могут быть заданы с помощью абсолютного или относительного пути. Для команды `ls` параметры — имена файлов, информацию о которых нужны вывести, или имена каталогов,

[1] Массив — упорядоченный набор **элементов**, доступных по их номеру, называемому **индексом** (*англ.* ***index***).

[2] Часто термин "параметр" используется и как синоним слову "аргумент". В английском языке отдельного термина для аргументов, не являющихся опцией, не используется, а под "параметром" (*англ.* ***parameter***) понимается любой объект, который содержит значение, в т.ч. переменная. Опции также иногда называют **ключами**.

[3] Состоят из одной буквы или цифры.

[4] Здесь приведен неполный список расширенной *GNU*-версии команды `ls`. За исчерпывающим и точным описанием всегда следует обращаться к документации.

содержимое которых нужно вывести. Можно заметить, что у команды `ls` нет обязательных аргументов. Если список файлов отсутствует, команда выводит содержимое текущего каталога.

Также следует отметить, что если имя файла (или каталога) содержит пробел, то оболочка воспримет его как разделитель аргументов, поэтому

```
$ ls a b
```

это вывод информации о объекте с именем "`a`" и о объекте с именем "`b`". Чтобы вывести информацию об объекте с именем "`a b`" необходимо заключить его в кавычки

```
$ ls "a b"
```

или предварить пробел символом "\":

```
$ ls a\ b
```

Пробелы в кавычках или предваренные символом обратного слеша считаются частью аргумента, а не разделителем аргументов. Это называется **экранированием** (*англ.* ***escaping***) специальных символов, то есть "превращением" их из специальных в воспринимаемые буквально.

Указание длинной опции `--help` приводит к выводу справки о формате команды, опция `--version` — версии программы. Наличие одной из этих опций приводит к игнорировнанию остальных (вывод содержимого каталога не производится) но это в синтаксисе выше не отражено.

Длинная опция `--sort` может иметь один из трех значений — `time`, `size`, `extension`, означающие соответственно 3 различных режима сортировки: по времени последнего изменения файла, по размеру файла и по расширению файла. В описании формата вызова факт обязательности одного из вариантов указывается перечислением этих вариантов в фигурных скобках с разделением вертикальной чертой. Если опция не указана, то сортировка производится по имени файла. Значение должно отделяться от опции знаком равно или пробелом, например

```
$ ls --sort=size /path/to/dir
$ ls --sort time
```

В случае пропуска значения опции можно получить неожиданный результат, так как последующий аргумент может быть воспринят в качестве этого значения (что может привести к неправильному поведению или выводу сообщения об ошибке в синтаксисе команды).

Короткие опции имеют следующий смысл:

- `-a` — выводить все файлы, в том числе скрытые (те, чьи имена начинаются на точку, без данной опции они не выводятся);
- `-A` — выводить все файлы, в том числе скрытые, кроме каталогов "`.`" (ссылка на текущий каталог) и "`..`" (ссылка на родительский каталог), которые в норме присутствуют в каждом каталоге и будут

отображены при использовании опции `-a`;

- `-1` — выводить информацию о каждом файле и каталоге в отдельной строке (без данной опции вывод производится в одну строку);
- `-l` — выводить не только имена файлов и каталогов, но и дополнительную информацию (атрибуты) — тип файла (каталог, обычный файл, ссылка и др.), права доступа, размер, дату последнего изменения, цель символической ссылки и др. (без данной опции выводится только список имен);
- `-d` — выводить информацию о каталоге, а не его содержимое[1].

Следует отметить, что размер каталогов, выводимый опцией `-l` и используемый для сортировки, не означает суммарный размер всех файлов каталога. Это размер записи содержимого каталога в файловой системе — служебная и вряд ли полезная пользователю информация. Для просмотра суммарного размера следует использовать команду `du`.

Короткие опции можно указывать вместе с одним дефисом, например вызовы

```
$ ls -l -d
```

и

```
$ ls -ld
```

эквиваленты (выводят подробную информацию о текущем каталоге, а не его содержимом).

Опция "`--`" имеет особый смысл — она разграничивает параметры (аргументы, не являющиеся опциями) и опции. Параметры и опции могут чередоваться. Имя файла может начинаться со знака дефиса. Если такой файл указан в качестве аргумента, команда будет считать его опцией. Например

```
$ ls -1 -list.txt
```

воспринимается как вызов `ls` без параметров, но с опциями `-1`, `-l`, `-i`, `-s`, `-t`, `-.`, `-t`, `-x` (сложно предсказать последствия, если в конкретной реализации программы `ls` или другой программы в подобной ситуации все опции окажутся действующими). Выйти из подобной ситуации можно, указав

```
$ ls -1 ./-list.txt
```

теперь `./-list.txt` — имя файла, а не список опций. Такой способ является пригодным для всех команд, но не универсальным для всех ситуаций. Например, если имя файла содержится в переменной, не всегда возможно

[1] Если аргументом является каталог, без данной опции выводится его содержимое, опция нужна, чтобы проверить наличие каталога или получить его атрибуты с помощью опции `-l`

и удобно определять, начинается ли оно с дефиса, является ли относительным путем или абсолютным и т.д.

Большинство команд поддерживают универсальный способ — указание "`--`" в конце списка аргументов. Двойной дефис обеспечивает разделение опций и аргументов, не являющихся опциями: все, что следует после двойного дефиса считается параметрами.

```
$ ls -1 -- -list.txt
```

Указание неверных (несуществующих или с неверным значением) опций или неверного параметра (в данном случае — несуществующего файла или каталога) приводит к выводу соответствующего сообщения об ошибке.

Вызов `ls` *без параметров или* `ls -l` *— типичный способ просмотреть содержимого текущего (рабочего) каталога.*

Для изучения команд читателю рекомендуется знакомиться с документацией по команде и тестировать ее работу, исполняя на компьютере с различными комбинациями опций, желательно в "безопасном месте" — каталоге, в котором нет информации, которую было бы жалко повредить.

Ознакомиться с описанием команд (документацией) в большинстве случае можно с помощью опции `--help`*, а также, обращаясь ко встроенному справочному руководству* `man` *(от английского* ***manual*** *— руководство). Это команда, параметром которой является имя команды, справку по которой нужно посмотреть. При этом справка будет не просто выведена в терминал, а будет отображена в интерактивном просмотрщике. Интерфейс просмотрщика позволяет листать текст вверх и вниз с помощью стрелок, осуществлять поиск (клавиша "/" открывает строку поиска внизу, в которую можно ввести текст поиска и нажать* ***Enter*** *для поиска первого совпадения, поиск следующего совпадения осуществляется нажатием* ***n****). Для выхода из просмотрщика следует нажать* ***q****. Нажатие* ***h*** *приводит к переходу к справке о самом просмотрщике. Справку по внутренним командам оболочки* ***Bash*** *можно получить с помощью команды* `help`. Например

```
$ man ls
```

отобразит в просмотрщике справку по (внешней) команде `ls`, а

```
$ help cd
```

выведет на экран справку по (внутренней) команде `cd`.

§2.4. Работа с файлами и каталогами

Просмотреть содержимое текстового файла можно с помощью команды

```
$ cat файл...
```

содержимое указанных файлов будет выведено на экран.

Создание файла можно произвести с помощью команды

```
$ touch файл...
```

Вообще-то эта команда устанавливает дату и время последнего изменения файла в текущие (или на указанное в соответствующей опции), не изменяя содержимого файла. Однако если такой файл не существует, то создается пустой файл с указанным именем.

Создать текстовый файл с нужным содержимым также можно с помощью команды *cat*:

```
$ cat > файл
```

после чего ввести текст, завершить ввод нажатием сочетания клавиш *Ctrl+D. Если файл с указанным именем существует, его содержимое будет перезаписано.*[1]

Удаление файла производится командой

```
$ rm файл...
```

По умолчанию файлы удаляются сразу и без сообщения какой-либо информации, хотя система может быть настроена на запрос подтверждения удаления каждого файла или каталога. Чтобы его избежать, можно использовать опцию `-f` (*force*, принудительно).

Создание каталога выполняет команда

```
$ mkdir каталог...
```

Удаление каталога — команда

```
$ rmdir каталог...
```

Команда удаляет только пустые каталоги. Для удаления непустого каталога можно использовать команду *rm* с опцией `-r` (recursive, рекурсивно по дереву каталогов)

```
$ rm -r каталог...
```

При использовании этой команды следует проявлять осторожность, особенно при вызове с ключом `-f` — содержимое каталога будет удалено безвозвратно, и если по ошибке указать нужный каталог, то можно лишиться важных данных. Поэтому сначала следует проверить содержимое каталога с помощью `ls`[2].

Копирование файла производится с помощью команды

```
$ cp источник назначение
```

В качестве назначения может выступать каталог или файл (новое имя

[1] Замечание: для создания и изменения текстовых файлов во многих случаях удобнее пользоваться текстовыми редакторами, но различные команды позволяют автоматизировать и этот процесс, в чем есть свои преимущества.

[2] Консольные команды удаления не перемещают файлы в “корзину”, а удаляют их из ФС.

файла). Для копирования каталога следует вызвать

```
$ cp -r источник назначение
```

(рекурсивно) или, лучше,

```
$ cp -a источник назначение
```

для сохранения структуры (например, чтобы символические ссылки были скопированы как ссылки, а не получены целевые файлы ссылок, сохранились время последнего изменения и др.) Параметр назначения является обязательным, но, разумеется, скопировать в текущий каталог можно просто указав в качестве данного параметра точку.

Переименование файлов производится командой

```
$ mv источник назначение
```

Команда может как присвоить файлу или каталогу-источнику новое имя, так и перенести файл в другой каталог (если назначение является существующим каталогом).

Команды `rm`, `cp`, `mv` обычно работают молчаливо, но имеют опцию `-v`, позволяющую вывести список обрабатываемых файлов. Также имеется опция `-i`, которая приводит к интерактивному подтверждению операции над каждым файлом (при этом опция `-i` подавляется опцией `-f`).

Команда `ln` позволяет создавать ссылки

```
$ ln [-s] цель ссылка
```

Без опции `-s` создается жесткая ссылка, с опцией `-s` — символическая. Цель — конкретный файл той же ФС носителя для жесткой ссылки (может быть задан в виде относительного или абсолютного пути) или любой текст для символической ссылки. Это может быть абсолютный или относительный путь к файлу или каталогу в любом месте ОС, может быть вообще к несуществующему файлу (такая ссылка имеет право на существование, хотя целевой файл не будет достижим). Ссылка — имя создаваемой ссылки (т.е. файла типа “символическая ссылка”).

Для жестких ссылок при удалении исходного файла ссылка сохраняется как файл — жесткие ссылки друг с другом не связаны, но при изменении содержимого файла, это изменение “отразится” на содержимом всех ссылок (так как они ссылаются на одни и те же данные в ФС). Для символических ссылок удаление исходного файла приведет к появлению ссылки с недостижимой целью, перемещение ссылки при использовании относительных путей приведет к тому, что ссылка также будет указывать в несуществующее место[1].

[1]Хотя именно относительные пути используются чаще, так как символические ссылки чаще всего существуют в рамках одного поддерева каталогов системы, программы или пользовательской задачи и при перемещении всей системы сохраняются, в то время как абсолютные ссылки начнут указывать в несуществующее место.

§2.5. Шаблоны поиска файлов

Для того, чтобы обработать несколько файлов в одной команде можно использовать т.н. **шаблоны поиска**, известные также как **шаблоны подстановки** и **символы-джокеры** (*англ.* ***wildcard*** *— игральная карта, которая может быть использована для замены любой карты*). Это символы звездочка — "`*`" — и вопросительный знак — "`?`". Символ звездочка может быть заменен на любую последовательность символов, в том числе пустую, символ вопросительного знака — на любой непустой символ. Например, если набрать

```
$ cp *.txt /run/media/user/drive/
```

то все файлы с расширением `.txt` из текущего каталога будут скопированы в каталог `/run/media/user/drive/`, или, наоборот, вызов

```
$ mv /run/media/user/drive/*.txt .
```

скопирует все файлы с расширением `.txt` из указанного в текущий.

Обработка шаблонов поиска производится следующим образом: оболочка находит список файлов, удовлетворяющих шаблону, и подставляет их на место указанного шаблона в командную строку в качестве параметров. Таким образом, команда получает уже список файлов, а не шаблон. Если таких файлов нет, оболочка передаст шаблон команде "как есть" (и, скорее всего, команда, не найдя таких файлов, выдаст сообщение об ошибке).

Кроме вопросительного знака, выбирающего любой символ, существует возможность выбрать один символ из списка, для этого следует указать список символов в квадратных скобках. Например `[abc]` будет соответствовать символам "`a`", "`b`" и "`c`". В квадратных скобках можно указывать диапазоны, например, `[a-d]`, `[0-9]` и др., а также "отрицания" символов с помощью "`^`": `[^d]` соответствует любому символу, кроме "`d`".

В качестве примера выбора символа рассмотрим ситуацию, когда в каталоге имеются следующие файлы:

```
a1bx.txt
a1x.txt
a2q.txt
a3x.txt
a4x.txt
aq
aq.txt
aw.txt
ax.txt
```

Тогда под следующие шаблоны попадают указанные файлы:

Шаблон	соответствующие файлы
*	все файлы
.	все файлы с непустым расширением (т.е. кроме aq)
a*x.txt	a1bx.txt, a1x.txt, a3x.txt, a4x.txt, ax.txt
a?x.txt	a1x.txt, a3x.txt, a4x.txt
a[123]x.txt	a1x.txt, a3x.txt
a[1-3]x.txt	a1x.txt, a3x.txt
a?[^x].txt	a2q.txt
a*[^x^q].txt	aw.txt
a?[^x^q].txt	ни одного файла

Шаблоны подстановки могут использоваться и для указания каталогов.

Кроме шаблонов подстановки файлов, можно использовать фигурные скобки для создания перечислений последовательностей символов. Например `a{1x,2q,4x,5w}.txt` превратится в список `a1x.txt`, `a2q.txt`, `a4x.txt`, `a5w.txt` независимо от наличия таких файлов. Фигурные скобки можно использовать не только для построения списков файлов, но и для любых других списков. Внутри фигурных скобок можно использовать символы подстановки.

Поскольку символы подстановки раскрываются оболочкой, то для задания в качестве параметра имени файла, содержащего такие символы, данные символы, как и символ пробела, следует экранировать — с помощью обратного слеша "\", двойных кавычек или апострофов.

Следует отметить, что скрытые файлы — чьи имена начинаются с точки "`.`" — не попадают под шаблон, начинающийся со звездочки "`*`" и знака вопроса "`?`". Например "`.cfg`", "`.a`", "`.`" и "`..`" не попадут под шаблоны "`*`", "`*a`", "`?`", "`?a`", "`??`" и другие, но попадут под шаблоны "`.*`" (все указанные), `.?` ("`.a`" и "`..`") и др. Это поведение можно изменить настройками. В связи с этим бывает плохой практикой использование `*` и других шаблонов для затрагивания *всех* файлов, особенно при рекурсивном обходе. В последнем случае лучше указывать каталог, а не `*` для обхода. Сравним подходы:

```
$ cp -r * /path/to/dest
```

копирует только нескрытые файлы и подкаталоги текущего каталога, а также все файлы и каталоги (скрытые и нескрытые) в нескрытых подкаталогах текущего каталога,

```
$ cp -r . /path/to/dest
```

копирует все (скрытые и нескрытые) файлов и подкаталоги текущего ка-

талога и его подкаталогов, но не сам сам текущий каталог.

Следует отметить, что

```
$ cp -r *.txt /path/to/dest
```

приведет к копированию не всех файлов с расширением `txt` *в текущем каталоге и его подкаталогах, а только соответствующих файлов в текущем каталоге и всех таких подкаталогов, имена которых попадают под шаблон* `*.txt` *со всеми файлами и подкаталогами в них. Дело в том, что шаблон раскрывается оболочкой, а команда, в данном случае* `cp`*, получает уже готовый список файлов и каталогов текущего каталога с расширением* `txt`*. Файлы и каталоги она обработает, а затем войдет в переданные подкаталоги рекурсивно, обработав все файлы в нем.* Об исходном шаблоне команда не будет иметь никаких данных. Для осуществления желаемого действия следует использовать более тонкие методы поиска и составления списка файлов, например команду `find`. Также нежелательный результат можно получить при пропуске каталога назначения, например

```
$ cp /path/to/src/*.txt
```

не приведет к копированию всех файлов с расширением `txt` из каталога `/path/to/src` в текущий каталог: последний файл списка будет воспринят как точка назначения копирования. Если это каталог, а до него в списке обнаружено несколько файлов, то эти файлы будут скопированы в него, если это файл, а обнаружено ровно два файла с расширением `txt`, то произойдет перезапись содержимого второго файла первым файлом. В остальных ситуациях будет выведено сообщение об ошибке. Поэтому корректной будет команда

```
$ cp /path/to/src/*.txt .
```

Проверить, над какими именно файлами будет совершаться действие, можно, например, с помощью команды `echo`:

```
$ echo /path/to/src/*.txt
```

выведет именно тот список файлов, который будет передан оболочкой любой другой команде по указанному шаблону. Для наиболее аккуратного копирования всего содержимого каталога `/path/to/src` со всеми его подкаталогами в каталог `/path/to/dst` следует использовать именно

```
$ mkdir /path/to/dst
$ cd /path/to/dst
$ cp -a /path/to/src .
```

а не

```
$ cp -a /path/to/src /path/to/dst
```

так как последнее приведет к созданию каталога `/path/to/dst/src` и копированию структуры каталога в него.

§2.6. Домашний каталог

Всякий пользователь имеет **домашний каталог** (*англ.* ***home directory***) — каталог, предназначенный для хранения рабочих файлов пользователя, пользовательских настроек программ и др. файлов, которые доступны пользователю для чтения и записи и недоступны другим пользователям. Обычно домашний каталог пользователя — `/home/`*`логин`*, где *`логин`* — логин данного пользователя (например, `/home/user`), домашний каталог администратора — `/root`.

При запуске оболочки текущим каталогом обычно становится домашний каталог пользователя.

В командах оболочки можно использовать символ `~` в качестве псевдонима домашнего каталога. Например,

```
$ cd ~
```

переходит в домашний каталог,

```
$ cd ~/Documents
```

переходит в подкаталог `Documents` домашнего каталога, а

```
$ cp ~/file.txt .
```

скопирует файл `file.txt` из домашнего каталога в текущий (или выдаст ошибку попытки копирования файла в себя, если текущий каталог — домашний).

Также домашний каталог может храниться в переменной окружения `HOME`.

§2.7. История команд и автодополнение

Оболочка *Bash* сохраняет историю команд. Для того, чтобы получить ранее исполненные команды следует в командной строке использовать стрелки вверх и вниз для пролистывания истории назад (к более ранним командам) и вперед (к более поздним) соответственно. Если перед именем команды добавить символ пробела, то команда в истории не сохранится:

```
$ mc
```

сохраняется в истории, а

```
$ ␣mc
```

не сохраняется в истории.

Еще одно удобное средство работы с командной строкой — **автодополнение** (*англ.* ***auto-completion***) команд и аргументов. Если в командной строке набрать часть (префикс) имени команды, а затем нажать клавишу *Tab*, то в случае, когда существует единственная команда с таким префиксом, оболочка автоматически завершит набор имени команды. Если

таких команд несколько, то при первом нажатии *Tab* оболочка издаст короткий звуковой сигнал (при наличии технической возможности), а при повторном нажатии выведет список возможных завершений префикса: пользователю следует донабрать команду и, возможно, повторить процедуру автодополнения. Например

```
$ rmd→Tab
```

может автоматически дополниться до

```
$ rmdir
```

в то время как

```
$ rm→Tab→Tab
```

превратится в

```
rm rmdir rmmod
$ rm
```

или даже более длинный список (в зависимости от количества подходящих установленных в системе команд; `rmd` может также не дополняться до `rmdir` однозначно).

Автодополнение работает и с аргументами-именами файлов: если такие файлы с указанным префиксом имени или полного или относительного пути существуют, то оболочка найдет их по нажатии *Tab*. Оболочка *Bash* также поддерживает автодополнение синтаксиса (опций и др. элементов) многих команд и возможности автодополнения могут быть расширены самими программами-командами при установке.

Еще одна полезная функция — возможность разбить команду на несколько строк. Для этого необходимо в конце строки набрать символ "\" и нажать *Enter*: тогда команда не будет исполнена, а вместо этого оболочка будет в новой строке ожидать ввода продолжения команды.

Символ "\" подавляет раскрытие служебного символа, например, "\␣" не считается символом разделителя аргументов, "\"" также будет восприниматься как актуальный символ "" (например, часть имени файла), а не как служебная кавычка, а сочетание *Enter* является подавлением символа перевода строки, поэтому оболочка обрабатывает его соответствующим образом.

§2.8. Файловый менеджер Midnight Commander

Файловый менеджер *Midnight Commander* (*MC*) — инструмент, с удобным интерфейсом для выполнения файловых операций. Он запускается с помощью команды `mc`. Его ключевые особенности следующие.

- Текстовый пользовательский интерфейс — возможна работа поверх практически любой консоли с минимальным расходом ресурсов; име-

ется поддержка мыши.

- Двухпанельный режим — в каждой из двух панелей отображаются списки файлов и подкаталогов открытых каталогов. Перемещение по списку осуществляется с помощью стрелок или путем поиска файла по префиксу имени. Вход в окно поиска выполняется сочетанием клавиш *Alt+s*, выход — двойное нажатие *Esc*[1]. Перемещение по каталогам — смена текущего каталога — производится естественным образом: курсор перемещается с помощью стрелок, с помощью клавиши *Enter* осуществляется вход в каталог или переход на каталог на уровень выше, если курсор находится на каталоге "..". Если нажать *Enter* на исполняемый файл программы, то программа будет запущена. Средства ОС и сам *MC* также поддерживают ассоциацию файлов с приложениями в зависимости от расширения — при нажатии на ассоциированный файл будет запущено приложение, в котором будет открыт данный файл. Переключение курсора между панелями осуществляется клавишей *Tab*.

- Большинство действий может быть выполнено с помощью коротких комбинаций горячих клавиш — это позволяет работать очень быстро (часто быстрее, чем вводить команды, и намного быстрее, чем с помощью мыши и меню). Например, клавиша *F5* позволяет скопировать файл или каталог под курсором в каталог, который открыт в другой панели, клавиша *F6* — переместить, клавиша *F8* — удалить, клавиша *F7* создает каталог, клавиша *F4* открывает файл для редактирования (как правило, во встроенном текстовом редакторе), клавиша *F3* — для просмотра. Переместить, скопировать или удалить группу файлов можно, выделив их с помощью клавиши *Insert* (она же снимает выделение, которое не пропадает при перемещении курсора), а также выбрать группу файлов для выделения с помощью клавиши "+" и снятия выделения с помощью клавиши "-". Клавиша "*" инвертирует выделение.

- Высокая интеграция с командной строкой: внизу экрана имеется командная строка, в которую можно вводить команды и запускать их, нажимая *Enter* (если содержимое строки непусто, то будет исполнена она, а не файл под курсором). Комбинация *Alt+H* открывает историю команд. Также в любой момент можно нажать комбинацию

[1]Клавиша *Esc* — отмена/выход — также используется для ввода функциональных клавиш, которые не всегда имеются на клавиатуре. Например, последовательность *Esc*, *1* равносильна клавише *F1*. Поэтому именно последовательное двукратное нажатие *Esc* интерпретируется как выход и отмена.

Ctrl+O, чтобы убрать панели и работать с командной строкой как обычно. Возвращает панели та же комбинация.

- Обработка архивов и сетевых файловых систем. Архивы большинства форматов (при наличии установленных программ, поддерживающих работу с ними) могут быть открыты как каталоги, нажатием клавиши *Enter*. Далее можно просматривать файлы в архиве, копировать файлы из архива и в архив. В меню панели можно подключиться к сетевым каталогам файлов и работать с ними теми же средствами. Невозможен только запуск команд в архивах и на нелокальных файловых системах.
- Компактный и удобный встроенный текстовый редактор *mcedit*. В нем имеется поиск и замена по горячим клавишам (*F7* и *F4* соответственно), возможность выделить одно или многострочный блок символов (*F3*), скопировать или переместить его (*F5*, *F6*) и другие возможности. Выход из редактора (и просмотрщика) — *Esc* или *F10*. Сохранение изменений при работе — *F2*.
- Говорящее меню настройки и возможностей. Внизу экрана имеется подсказка по горячим функциональным клавишам. С помощью клавиши *F9* можно войти в меню вверху экрана, в котором можно осуществлять различные действия над файлами и панелями, а также настраивать интерфейс. Рядом с пунктами меню указаны сочетания клавиш для быстрого доступа к ним. Свое меню имеется и у редактора. Рекомендуется изучать возможности файлового менеджера и редактора, пролистывая список пунктов меню и справочной системы, вызываемой по клавише *F1*.
- Выход из *Midnight Commander* — *F10* или команда `exit`.

§2.9. Смена пользователя и запуск с административными привилегиями

Для того, чтобы временно (в данной командной строке, а не в другом виртуальном текстовом терминале) войти от имени администратора можно использовать команду

```
$ su [-c команда] пользователь
```

Если пользователь не указан, подразумевается *root*, если не указана опция `-c`, то происходит запуск оболочки от имени данного пользователя. После ввода команды система запросит пароль пользователя (администратора или указанного) и только в случае его успешного ввода выполнит команду.

```
$ su
Password:
#
```

Также в системе может быть настроена команда *sudo*:

```
$ sudo [-u пользователь] команда
```

позволяющая выполнить команду от имени администратора. При этом, в отличие от *su*, данная команда запрашивает пароль пользователя, вызвавшего команду, а не пароль администратора. Именно данному пользователю должны быть предоставлены соответствующие полномочия, фактически делающие его администратором без сообщения пароля администратора. Такие полномочия можно отозвать, не меня пароль пользователя *root*[1]. Например, обе команды

```
$ sudo mc
```

и

```
$ su -c mc
```

запускают *Midnight Commander* от имени администратора, но в первом случае пользователю нужно вводить свой пароль, а во втором — пароль администратора. В первом случае в выполнении команды будет отказано, если у пользователя нет полномочия выполнять команду *mc* от имени администратора или введен неверный пароль пользователя, во втором — только если введен неверный пароль администратора. Требование ввода пароля пользователя является мерой безопасности при получении доступа к терминалу с запущенной командной оболочкой пользователя посторонним лицом, а также мерой, призванной повысить внимание пользователя при выполнении действий от имени администратора.

Для открытия административной оболочки с помощью *sudo* следует вызвать

```
$ sudo {-i|-s}
```

Отличие опций в том, что при использовании `-s` сохраняется среда пользователя (переменные окружения, текущий каталог, настройки оболочки и т.д.), а при указании `-i` используется среда администратора. Последний вариант предпочтительнее, если нужно исключить влияние пользовательских настроек и программ на административные действия, в том числе из соображений безопасности. Опцию `-i` можно указывать и при вызове команды (а не оболочки) с теми же целями. Вызов

```
$ sudo -i
```

[1]Вообще говоря, команда *sudo* может быть настроена на выполнение некоторых или всех команд без ввода пароля пользователя или с запросом пароля пользователя для отдельных пользователей и групп, что делает ее очень гибким инструментом в предоставлении ограниченных административных полномочий.

работает как

`$ su` с разницей в том, пароль какого аккаунта будет запрошен. Опция `-u` команды `sudo` позволяет выполнять команду от имени заданного пользователя, а не администратора (по-прежнему вводя свой пароль, а не пароль целевого пользователя).

§2.10. Пользователи и права доступа

Всякий пользователь снабжается **уникальным идентификатором** (*англ.* ***user identifier, uid***) — целым числом, по которому идентифицируются пользователи и права доступа пользователей (в момент аутентификации логину сопоставляется данный номер, который затем используется системой). Всякий пользователь должен быть членом хотя бы одной **группы** (*англ.* ***group***) — **основной** (*англ.* ***primary*** — ***первичной***) группы пользователя. Пользователь может быть членом нескольких групп. Каждой группе также сопоставляется уникальное **имя** (*англ.* ***name***) и номер — **идентификатор** группы (*англ.* ***group identifier, gid***). Пользователь *root* имеет идентификатор $uid = 0$ и первичную группу *root*, у которой $gid = 0$. Точнее, именно пользователь с $uid = 0$ является привилегированным пользователем (администратором системы), а группа с $gid = 0$ — административной[1].

У всякого файла есть **владелец** (*англ.* ***owner***) и **группа владельца** (*англ.* ***owner group***) — атрибуты *uid* и *gid* пользователя и группы, считающихся владельцами данного файла. Владельцы и группа владельца большинства системных файлов и файлов программ — *root* и *root*, владельцем файлов в домашнем каталоге пользователя является данный пользователь и т.д[2].

У всякого файла (и каталога) также имеются атрибуты **прав доступа** (*англ.* ***access permissions***). Данные права могут устанавливаются для владельца (пользователя, *англ.* ***user***), группы (*англ.* ***group*** и всех остальных (*англ.* ***other***).

Права доступа к файлу для всех трех категорий включают в себя три типа разрешаемых операций — чтение (*англ.* ***read***), запись (*англ.* ***write***) и

[1] Логин и имя группы могут отличны от `root`, однако такой вариант применяется редко и может привести к несовместимости с некоторыми приложениями и настройками.

[2] Следует отметить, что для того, чтобы файлы имели владельца, группу и права доступа необходима соответствующая поддержка в ФС носителя, поэтому для установки ОС следует выбирать подходящую ФС. Также на ФС сохраняются именно *uid* и *gid*, то есть числа, а не логины и имена групп, которые для неадминистративных пользователей могут различаются на разных компьютерах, поэтому такие ФС могут вызвать проблемы с интерпретацией прав доступа при использовании на сменных носителях.

исполнение (*англ.* ***execute***), таким образом такая модель доступа известна как rwx, всего используется 8 битов — $rwxrwxrwx$ — первый триплет для владельца, второй — для группы, третий — для всех остальных. Если бит установлен, то действие разрешено, если не установлен — запрещено.

Для файлов право на чтение означает право чтения содержимого файла, право на запись — право на перезапись содержимого файла и добавления данных в конец файла, право на исполнение — возможность исполнения данного файла в качестве команды оболочки[1]. Отсутствие права на исполнение не означает невозможность исполнения данного файла (его можно выполнить с помощью соответствующего интерпретатора или программы-загрузчика двоичных файлов).

Для каталогов право на чтение означает право на чтение списка файлов каталога, право на запись — право на создание и удаление файлов (и подкаталогов) в данном каталоге[2], право на исполнение трактуется как право на **переход** (вход) (*англ.* ***enter***) в данный каталог. Это самое сильное разрешение: при его отсутствии у данного каталога или хотя бы у одного из его родительских каталогов в дереве у пользователя нет права записи в данный каталог даже при наличии разрешения w, нет права чтения содержимого (разрешение r позволяет прочитать список файлов, но не их атрибуты и содержимое, даже если у файлов есть необходимые разрешения). Отсутствие права x фактически означает запрет на операции с данным каталогом и всеми его подкаталогами, независимо от того, какие у них и файлов в них разрешения (за исключение списка файлов без каких-либо атрибутов и содержимого в самом каталоге при наличии бита r). Поэтому отсутствие разрешения для группы и всех остальных пользователей на чтение, запись и исполнение домашнего каталога гарантирует, что никакой другой пользователь, кроме владельца, не получит доступ к файлам и подкаталогам внутри чужого домашнего каталога, независимо от того, какие права стоят для них[3].

Для пользователя *root* права трактуются таким образом: всегда есть

[1] *Unix*-подобные системы апеллируют к данному атрибуту, а не к расширению файла при определении его как исполняемого. Обычно, для того, чтобы выступать в роли команд, исполняемые файлы не имеют расширения. В качестве исполняемых могут выступать двоичные файлы программ (с машинными командами в формате, предусмотренным ABI конкретной ОС), а также сценарии оболочки и других интерпретируемых языков.

[2] То есть речь идет о чтении и записи каталога, как файла, содержащего список файлов в нем.

[3] Разумеется, средствами данной ОС и при отсутствии административного доступа. Подключив носитель информации к другой ОС или загрузив на компьютере другую ОС, игнорирующей права доступа *Unix*, или к которой есть административный доступ, любые ограничения можно обойти.

право на чтение и запись для файлов и каталогов, а также право на переход в каталог (независимо от актуальных разрешений). Файл считается исполняемым, если хотя бы для кого-то (владельца, группы, всех остальных) установлен бит исполняемости.

Систематизировать разрешения можно в виде таблицы:

Разрешение	Значение для файла	Значение для каталога
r (Read)	Чтение содержимого	Чтение списка файлов
w (Write)	Добавление и перезапись содержимого	Создание и удаление файлов
x (eXecute)	Выполнение как команды	Переход в каталог, доступ на операции с содержимым и подкаталогами.

Просмотреть владельца и разрешения файлов и каталогов можно с помощью команды `ls -l` (или `ls -n`: опция `-l` выводит логины и имена групп, а `-n` — *uid* и *gid*). Разрешения выводятся схематически в формате, подобном

```
-rwxr-xr--
```

где первый символ означает тип файла (прочерк — обычный файл, `d` — каталог, и т.д.), далее следует три символа прав владельца, три символа прав для группы, три символа прав для всех остальных. Если символ указан явно (`r`, `w`, `x`), то права установлены, если стоит прочерк — данного разрешения нет.

Существует также восьмеричная нотация прав. Три бита прав могут быть интерпретированы как восьмеричная цифра: "`---`" — 0, "`--x`" — 1, "`-w-`" — 2, ..., "`rwx`" — 7. Три триплета образуют трехзначное восьмеричное число. Таким образом, 755 означает `rwxr-xr-x` — у владельца есть все права, у группы и всех остальных только права на чтение и исполнение, 644 соответствует `rw-r--r--` и т.д.

У всякого процесса также есть владелец и группа — *uid* и *gid* процесса. Дочерние процессы (запускаемые из другой программы, в т.ч. из оболочки) наследуют *uid* и *gid* родительского процесса. Процесс, исполняемый от имени администратора, может сменить свои *uid* и *gid*, то есть "переключиться" на любого другого пользователя. Также пользовательский процесс может сменить *gid* на любую группу, членом которой является пользователь с *uid* процесса. Программа загрузки системы (инициализирующий процесс) запускается от имени администратора (*root*), она же

запускает программу входа в систему[1] с наследованием $gid = uid = 0$, которая, при успешном входе пользователя в систему, переключает свой $uid = 0$ на uid пользователя, а $gid = 0$ на gid основной группы пользователя, после чего запускает оболочку, установленную для данного пользователя. Этот процесс и все его дочерние процессы уже исполняются с ограниченными полномочиями.

Фактически, поскольку не сам пользователь "напрямую", а процессы, запущенные в системе, осуществляют системные вызовы, в т.ч. для доступа к файлам, при проверке прав доступа к файлу сверяются именно uid и gid процесса с uid и gid файла, для выяснения, какой именно триплет прав следует применять. При попытке провести операцию с файлом (или каталогом) права проверяются следующим образом:

- если uid процесса совпадает с uid файла применяются права для владельца;
- если uid процесса не совпадает с uid файла, но gid пользователя совпадает с gid файла, то применяются права для группы;
- если не совпадают ни uid ни gid, то применяются права для всех остальных.

При создании файлов и каталогов их uid и gid устанавливаются в uid и gid создающего процесса, то есть владельцем становится пользователь, создавший файл.

Права по умолчанию для новых файлов и каталогов определяются значением т.н. **маски режима создания пользовательских файлов** (*англ.* ***user file creation mode mask, umask***), показывающей какие права не будут установлены для нового файла или каталога. Данная маска является свойством процесса — наследуется дочерними процессами и может быть изменена процессом. В частности, встроенная команда `umask` выводит и меняет значение $umask$ оболочки. По умолчанию значение $umask$ есть 002, то есть при создании новых файлов и каталогов право w для остальных пользователей не устанавливается. Также для файлов никогда не устанавливаются права x для всех трех категорий.

§2.11. Изменение владельца файлов и прав доступа

Изменение владельца и группы владельца файла осуществляется с помощью команды *chown*

[1]Например, *login* для текстовых виртуальных консолей и выбранный **менеджер экрана** (*англ.* ***display manager***) для графической сессии, обычно являющийся составной частью графического окружения рабочего стола, но позволяющего выбрать графическое окружение при входе.

```
$ chown [-Rhv] [пользователь][:группа] файл...
```

Если группа не указана, то она не изменяется. Опция `-R` позволяет изменить владельца и группу для указанного каталога и всех файлов и каталогов внутри него (рекурсивно), например

```
$ chown root:root -R .
```

изменит на *root* владельца и группу самого текущего каталога и всего его содержимого рекурсивно.

Опция `-v` приводит к выводу всех затронутых изменением файлов. Опция `-h` приводит к изменению владельца самой символической ссылки (если таковая указана в качестве аргумента или встретится при рекурсивном обходе каталога), а не целевого файла.

Изменение прав доступа осуществляется с помощью команды *chmod*

```
$ chmod [-Rv] права файл...
```

Опция `-R` позволяет изменить права для указанного каталога и всех файлов и каталогов внутри него (рекурсивно), опция `-v` приводит к выводу всех затронутых изменением файлов.

Права могут быть заданы двумя способами. Первый способ — явно в восьмеричном виде, тогда права будут установлены в указанное значение. Например,

```
$ chmod 744 file
```

устанавливает права в `rwxr--r--`. Второй способ использование символа триплета — `u` (user), `g` (group), `o` (other), `a` (все три триплета), символа выполняемого действия: “`+`” — добавить права, “`-`” — отозвать права, “`=`” — установить указанные права и символа права “`r`”, “`w`”, “`x`”. Символы триплета могут присутствовать в любой комбинации[1], символы прав также могут присутствовать в любой комбинации. Добавление и отзыв прав выполняет изменение имеющихся прав, а установка — замену, подобно указанию восьмеричного значения. Комбинаций прав может быть несколько, разделенных через запятую. Например,

```
$ chmod a+x file
```

делает файл исполняемым для владельца, группы и всех остальных;

```
$ chmod og-w file
```

отзывает право на запись у группы и всех остальных

```
$ chmod u=rwx,go=rx file
```

устанавливает права в `rwxr-xr-x`, что равносильно вызову

```
$ chmod 755 file
```

Сменить владельца и группу владельца файла может только администра-

[1] Могут и отсутствовать вообще, это соответствует указанию всех трех или “`a`” с одним отличием: права, которые не должны устанавливаться в соответствии с режимом *umask*, не устанавливаются.

тор. Сменить права доступа может только владелец файла и администратор.

§2.12. Дополнительные биты прав доступа

Помимо прав *rwx* имеются еще три бита, отвечающие за права доступа.

Бит *Setuid*, установленный для исполняемых файлов, означает, что будучи выполненным в качестве команды, процесс файла получил *uid* файла, а не пользователя, запустившего его. Использование *Setuid* — единственный способ "сменить" *uid* для неадминистративных пользователей, лимитированный возможностью создания дочерних процессов отдельных программ от имени выделенного пользователя. В том числе привилегии могут быть повышены до административных (если у запускаемого файла $uid = 0$) при исполнении команд от обычного пользователя. Именно так команды *su* и *sudo* получают административные привилегии — их владелец *root* и установлен бит *Setuid*. Установить данный бит можно с помощью значения "`s`" для "`u`" в команде *chmod*, при просмотре в `ls` выводится строчная "`s`" на месте "`x`" в первом триплете, если сам бит "`x`" также установлен, или заглавная "`S`", если "`x`" для владельца не установлен. Для каталогов данный бит большинством ОС игнорируется[1].

Бит *Setgid*, установленный для исполняемых файлов, означает, что будучи выполненным в качестве команды, процесс файла получил *gid* файла, а не группы пользователя, запустившего его. Установить данный бит можно с помощью значения "`s`" для "`g`" в команде *chmod*. При просмотре в `ls` выводится строчная "`s`" на месте "`x`" во втором триплете, если сам бит "`x`" также установлен, или заглавная `S`, если "`x`" для группы не установлен. Для каталогов данный бит интерпретируется следующим образом: новые файлы и подкаталоги данного каталога в качестве группы владельца будут иметь *gid* родительского каталога, а не *gid* создающего процесса.

Бит *Sticky* для исполняемых файлов в старых ОС нес следующий смысл: ОС загружала файл в оперативную память целиком и держала настолько долго, насколько возможно, для ускорения работы программы в условиях дефицита памяти. В настоящее время используются более эффективные механизмы оптимизации памяти, чаще всего памяти достаточно для хранения исполняемых файлов всех программ целиком. Для

[1] Бит может интерпретироваться следующим образом: новые файлы и подкаталоги данного каталога в качестве владельца будут иметь *uid* родительского каталога, а не *uid* создающего процесса.

каталогов этот бит означает, что право на удаление у пользователя будет только для тех файлов (и подкаталогов), владельцем которых пользователь является. Например, это применяется для каталога `/tmp`, где различные пользователи хранят свои временные файлы: защитить файлы от доступа с помощью стандартных прав *rwx* можно, а от удаления "спасет" только снятие бита *w* у самого каталога *tmp*, что в свою очередь запретило бы и создание файлов. *Sticky* бит решает данную проблему. Установить этот бит можно с помощью значения "`t`" для "`o`" в команде *chmod*, при просмотре в `ls` выводится строчная "`t`" на месте "`x`" в третьем триплете, если сам бит *x* также установлен, или заглавная "`T`", если *x* для остальных пользователей не установлен.

В восьмеричной нотации эти три разрешения образуют дополнительный, четвертый (т.е. записываемый слева от остальных) триплет, причем 1 — *Sticky*, 2 — *Setuid*, 4 — *Setgid*, то есть 0777 означает `rwxrwxrwx`, 1777 — `rwxrwxrwt`, 4744 — `rwxr-Sr--`, 7755 — `rwsr-sr-t` и т.д.

Таким образом, расширенная таблица прав приобретает следующий вид.

Разрешение	Значение для файла	Значение для каталога
r (Read)	Чтение содержимого	Чтение списка файлов
w (Write)	Добавление и перезапись содержимого	Создание и удаление файлов
x (eXecute)	Выполнение как команды	Переход в каталог, доступ на операции с содержимым и подкаталогами.
t (sTicky)	Игнорируется	право на удаление только своих файлов
s (Setuid)	Выполнение от *uid* файла	Игнорируется или наследование *uid* каталога
s (s, Setgid)	Выполнение от *gid* файла	Наследование *gid* каталога

§2.13. Текстовые редакторы: vim и другие

Текстовый редактор *vim* (запускается командой `vi` или `vim`, *vim* — улучшенная версия редактора *vi*) является стандартным редактором в *Unix*-подобных системах. Его интерфейс отличается от привычного пред-

ставления о текстовом редакторе: для работы с ним требуется специальное изучение возможностей, детали которых выпадают за рамки данного пособия.

Ключевая особенность данного редактора — он может работать в двух режимах: **нормальном режиме** (*англ.* ***normal mode***) и **режиме вставки** (*англ.* ***insert mode***). Режим вставки близок к режиму работы "обычного" редактора, нормальный режим позволяет производить различные операции над текстом с помощью команд редактора: от перемещения курсора и поиска, от копирования блоков текста до исполнения целых сценариев обработки текста, осуществляемого с помощью комбинаций клавиш. При запуске редактор входит именно в командный режим. Переключение в режим вставки осуществляется с помощью клавиши *i*, возврат в нормальный — по клавише *Esc*.

Кроме нормального режима и режима вставки есть еще режим командной строки. Переход в него производится клавишей "*:*", после чего следует набрать команду и нажать *Enter*. Примеры основных команд: `:w` — сохранение изменений, `:q` — выход (если нет не сохраненных изменений), `:q!` — выход без сохранения, `:help` — вызов справки по редактору.

Многие приложения автоматически запускают *vim*, когда необходимо редактировать текстовый файл. Выбор редактора обычно осуществляется ими исходя из значения переменной окружения `EDITOR`, причем, если ее значение пусто, то выбирается `vi`. Разумеется, редактирование можно проводить другими текстовыми редакторами — из наиболее известных следует отметить *nano*, обладающий легковесным и доступным, но не всегда удобным интерфейсом (имеется подсказка по горячим клавишам внизу экрана) и упомянутый выше *mcedit*.

§2.14. Управление процессами

Создание дочернего процесса — системный вызов, который может быть осуществлен другим процессом, в т.ч. оболочкой ОС. Таким образом, процессы образуют "дерево", в корне которого находится первичный процесс, создаваемый при запуске операционной системы. Это так называемый *init*-процесс (инициализирующий процесс), ранее его роль играла одноименная программа, на современных ОС на базе *Linux* используется программа *systemd*. Процессы ядра *Linux* могут образовывать "параллельное" дерево процессов.

Всякий процесс имеет уникальный идентификатор *pid* (*англ.* ***process identifier*** *— идентификатор процесса*), а также имя. Имя процесса по умолчанию представляет собой имя исполняемого файла, поэтому не яв-

ляется уникальным — один и тот же файл можно запустить несколько раз с созданием разных, независимых процессов, с различным идентификатором, но одинаковым именем. Помимо собственно исполняемого кода в ОС с каждым процессом ассоциированы

- идентификатор пользователя и группы владельца процесса *uid* и *gid*;
- текущий рабочий каталог;
- переменные окружения;
- маска режима создания файлов *umask*;
- открытые файлы, установленные сетевые соединения;
- выделенные страницы памяти

и другие элементы. При создании дочернего процесса ряд элементов (*uid*, *gid*, *umask*, текущий каталог, переменные окружения) наследуются им, но не передаются родительскому при изменении. Например, при смене рабочего каталога в дочернем процессе рабочий каталог родительского процесса не изменится[1].

Запуск всякой внешней команды (программы) из оболочки создает дочерний процесс. По умолчанию процесс остается "присоединенным" к оболочке — работа оболочки останавливается до его завершения. Если завершение процесса по каким-то причинам не происходит в штатном режиме, его можно прервать принудительно с помощью комбинации клавиш *Ctrl+C*, при этом несохраненные данные процесса будут утеряны.

Если после завершения команды указать символ "`&`", то процесс запускается в **фоновом режиме** *англ. (**background mode**)* — работа оболочки продолжается, а процесс работает параллельно. При этом оболочка сообщает *pid* созданного процесса и количество фоновых процессов. Фоновые процессы остаются дочерними по отношению к оболочке и будут принудительного завершены при выходе из оболочки. Переменная `$!` хранит *pid* последнего запущенного фонового процесса (переменная не задана, если таких процессов нет), а переменная `$$` — *pid* самого процесса оболочки.

Просмотреть список процессов можно с помощью команды `ps`. Она выводит идентификатор процесса (первый столбец), имя процесса (последний столбец), а также дополнительную информацию (где запущен процесс, время работы и т.п.), которую можно настроить аргументами. Будучи запущенной без аргументов, программа выводит только процесс данной оболочки и его дочерние процессы. Для просмотра всех процессов системы следует вызвать ее с аргументами `ax`:

[1] Поэтому команда `cd` обязательно должна быть внутренней командой оболочки.

```
$ ps ax
```
или ключом `-ef`:
```
$ ps -ef
```
(во втором случае формат вывода будет несколько отличен). Ключ `-H` позволяет просмотреть дерево процессов.

Процессами можно управлять с помощью **сигналов** (*англ.* ***signal***) — системных сообщений, обрабатываемых самим процессом или операционной системой. Отправка сигналов осуществляется с помощью команды `kill`, принимающей в качестве аргументов список сигналируемых процессов:
```
$ kill [-СИГНАЛ] pid...
```
Сигнал может быть задан по номеру или по имени.

Если сигнал не указан, то посылается сигнал `TERM` (номер 15) — мягкой остановки процесса. Такая остановка позволяет аварийно завершить процесс, хотя и может быть перехвачена самим процессом, поэтому может не привести к успеху[1]. Например,
```
$ kill 12345
```
завершает процесс с $pid = 12345$. Более жесткое завершение, неперехватываемое процессом — сигнал `KILL` (номер 9):
```
$ kill -KILL 12345
```
или
```
$ kill -9 12345
```
приводит к немедленному завершению процесса[2].

Еще одна полезная пара сигналов — `STOP` (19) и `CONT` (18). Первый позволяет приостановить исполнение процесса: процесс прекращает исполняться, но его состояние, данные, открытые файлы и т.д. сохраняются, т.е. процесс останавливается в точке исполнения и перестает тратить процессорное время. Второй сигнал восстанавливает работу приостановленного процесса. Приостановка и продолжение почти всегда происходит незаметно для процесса[3], и бывает полезной для временного освобожде-

[1] Сочетание *Ctrl-C* передает активному процессу сигнал `TERM`.

[2] Строго говоря, даже в этой ситуации процесс может не завершиться, а перейти в состояние "зомби" — процесс сохраняется в системе, но не выполняет никаких действий, хотя и может удерживать открытыми файлы, файловые системы и иные устройства и ресурсы, мешая их открытию, размонтированию и использованию другими процессами, что неприятно. Ошибка в файловых системах (аппаратная или программная) является частой причиной возникновения процессов-зомби. Сам по себе зомби не тратит процессорное время и не несет значительного расхода памяти, но освобождение остальных захваченных ресурсов в этой ситуации может оказаться сложной задачей, решаемой творчески по обстоятельствам (если вообще решаемой) или перезагрузкой.

[3] Однако если процесс удерживал оболочку, то работа оболочки будет продолжена и процесс станет фоновым.

ния процессорного времени “тормозящим” работу процессом для других процессов[1].

С помощью команды `killall` можно послать сигнал всем процессам с указанным именем, с помощью `pgrep` — находить *pid* процессов по фрагменту имени, с помощью `pkill` посылать сигнал процессам, найденным по фрагменту имени. Например,

```
$ killall -STOP procname
```

приостанавливает работу всех процессов с именем *procname*, а

```
$ killall -CONT procname
```

продолжает. Такие команды удобны для автоматизации и массовой обработки процессов, но могут послать сигнал процессам, которые не должны быть затронуты, поэтому в интерактивном режиме ручной анализ с помощью `ps` бывает предпочтительнее. Облегчить анализ огромного вывода `ps ax` позволяет описанный ниже поиск по регулярным выражениям.

Утилита `top` (а также ее варианты `htop` и другие) с текстовым интерфейсом позволяет вывести список процессов, потребляющих больше всего процессорного времени и памяти, а также сигналировать их с помощью сочетаний клавиш (стрелки влево и вправо — выбор столбца сортировки, вверх и вниз — просмотр таблицы процессов, “`k`” — интерактивная отправка сигнала процессу)[2].

В файловой системе *proc* (обычно монтируемой в `/proc`) каждому процессу соответствует каталог с именем *pid* данного процесса, а каталог `self` представляет собой символическую ссылку на *pid* текущего процесса. Внутри этих каталогов можно найти файлы с информацией о процессе, например, имя (`comm`), командную строку (`cmdline`), объем ввода-вывода (`io`), символическую ссылку на исполняемый файл (`exe`), символические ссылки на используемые процессом файлы (в подкаталоге `fd`) и другую системную информацию, ассоциированную с процессом.

Утилита `renice` позволяет менять приоритет процесса. Приоритет (число *nice*) обычно может принимать значения от -20 до 19, где 0 — стандартный приоритет, а меньше значение означает более высокий приоритет. Обычным пользователям нельзя устанавливать отрицательное значение приоритета. Более высокоприоритетные процессы получают больше процессорного времени.

Команда `wait` позволяет дождаться завершения дочернего фонового

[1]Также приостановленный процесс не будет претендовать на восстановление данных из подкачки в физическую память, поэтому может увеличиться и доступный другим процессам объем оперативной памяти.

[2]Существуют и утилиты `iotop`, `jnettop` и др., позволяющие, соответственно, находить процессы, осуществляющие наибольший объем операций ввода-вывода, обмена данными по сети и др.

процесса по его *pid* или всех дочерних процессов данной оболочки.

§2.15. Переменные

Задание значения переменной осуществляется с помощью указания имени и значения, разделенных с помощью знака равенства "=":

```
$ var=value
$ echo var=$var
var=value
```

Если попытаться получить значение переменной, значение которой не задано, оболочка подставит пустую строку, в этом нет ошибки[1]:

```
$ var=value
$ echo var=\'$var\'
var='value'
$ echo vars=\'$vars\'
vars='' Следует обратить внимание на использование символа экранирования \textbackslash перед апострофом, необходимое для того, чтобы апостроф был выведен на экран, а не обрабатывался оболочкой (как символ экранирования).
```

Отметим, что *переменные окружения* и *переменные оболочки* — не тождественные понятия. Действительно, все переменные окружения, которые получает процесс оболочки, становятся переменными оболочки, однако обратное неверно: только те переменные, которые помечены в оболочке как "экспортные", становятся переменными окружения запускаемых из данной оболочки программ. Сделать переменную экспортной можно с помощью команды `export`:

```
$ EDITOR=mcedit
$ export EDITOR
```

или сразу с заданием значения

```
$ export EDITOR=mcedit
```

Можно также запустить программу с однократно измененным значением переменной: например, команда

```
# EDITOR=mcedit visudo
```

запускает программу редактирования настроек прав на запуск `sudo` в редакторе `mcedit` вместо `vim`, используемом по умолчанию, но значение переменной оболочки изменено не будет, изменение коснется только программы `visudo` и только для данного конкретного запуска. Для создания

[1] Следует, однако, учесть, что незаданная переменная и переменная, значением которой является пустая строка — разные вещи. Хотя при прочтении значения они ведут себя одинаково, существуют способы проверить, задана ли переменная. Сделать ранее заданную переменную незаданной можно с помощью команды `unset`.

временно измененного окружения используется также команда `env`.

Если переменная изначально (при старте оболочки) являлась переменной окружения или была помечена как экспортная, то изменение ее значения автоматически приведет к передаче измененного значения дочерние процессы, вызов `export` не требуется.

При задании значения переменной, содержащей пробелы и другие служебные символы, его следует указывать в кавычках, так как иначе оболочка развернет их, в частности, пробел будет воспринят как конец значения переменной и начало команды.

```
$ var="a long string value"
```

Если при получении значения переменной в команде после нее оказывается символ, то данный символ будет воспринят как продолжение имени переменной, что может привести к получению неверного результата:

```
$ animal=cat
$ echo plural form: $animals
plural form:
```

В этом случае имя переменной следует брать в фигурные скобки:

```
$ echo plural form: ${animal}s
plural form: cats
```

Существует также команда `declare`, позволяющая задавать значения переменной, сразу объявлять ее экспортной, доступной только для чтения и др.

Прочитать значение переменной с клавиатуры можно с помощью команды `read`:

```
$ read var
value
$ echo $var
value
```

(во второй строке "`value`" введено с клавиатуры).

§2.16. Потоки ввода и вывода, перенаправление

Всякий процесс и всякая команда оболочки (даже внутренняя) имеет три потока данных:

- **стандартный поток ввода** (*англ.* ***stdin, standard input stream***) — поток вводимых данных, по умолчанию осуществляется ввод с клавиатуры;
- **стандартный поток вывода** (*англ.* ***stdout, standard output stream***) — поток выводимых данных, по умолчанию осуществляется вывод на терминал;

- **стандартный поток ошибок** (*англ.* ***stderr, standard error stream***) — поток сообщений об ошибках, по умолчанию осуществляется вывод на терминал, то есть поток вывода и поток ошибок совмещены.

В поток ошибок поступают текстовые сообщения об ошибках, обнаруживаемых и анализируемых самой программой (командой), соответственно программа выбирает, куда выводить данные — в поток вывода или в поток ошибок. Например, команда `ls` выводит содержимое каталога в стандартный поток вывода, а в случае возникновения ошибки — например, неверно заданных аргументов, отсутствия каталога, отсутствия прав доступа к каталогу и т.п. — сообщение об этом будет выведено в поток ошибок.

```
$ ls
a.txt   b.txt   a.csv
$ ls *.txt *.doc
ls: невозможно получить доступ к '*.doc': Нет такого файла или
каталога
a.txt   b.txt
```

В первом вызове `ls` был только вывод информации о содержимого текущего каталога, во втором также появилось сообщение об ошибке отсутствия файла `*.doc`[1].

Стандартные потоки можно перенаправить в файлы, для этого в конце команды следует указать соответствующий комбинацию перенаправления и файл, куда следует записать данные:

- "`<`*файл*" или "`0<`*файл*" перенаправляет стандартный ввод в указанный файл;
- "`>`*файл*" или "`1>`*файл*" перенаправляет стандартный вывод в указанный файл, если файл существует, он перезаписывается;
- "`>>`*файл*" или "`1>>`*файл*" перенаправляет стандартный вывод в указанный файл, если файл существует, информация дописывается в конец файла;
- "`2>`*файл*" перенаправляет поток ошибок в указанный файл, если файл существует, он перезаписывается;

[1] Оболочка не развернула символ "`*`" в виду отсутствия подходящих файлов, а передала "как есть" в команду `ls`, хотя он и не был экранирован. Вызов `ls "*.txt"` привел бы к аналогичному сообщению об ошибке, хотя подходящие под шаблон файлы имеются.

- “2>>*файл*” перенаправляет поток ошибок в указанный файл, если файл существует, информация дописывается в конец файла.

Цифра соответствует номеру потока: 0 — ввода, 1 — вывода, 2 — ошибок, если номер не указан, то используется поток ввода или вывода в зависимости от знака перенаправления, поэтому только для потока ошибок его указание обязательно. Пример.

```
$ ls
a.txt   b.txt   a.csv

$ ls *.txt *.doc >out
ls: невозможно получить доступ к '*.doc': Нет такого файла или
каталога
$ cat out
a.txt   b.txt

$ ls *.txt *.doc 2>err
a.txt   b.txt
$ cat err
ls: невозможно получить доступ к '*.doc': Нет такого файла или
каталога

$ ls *.txt *.doc >out 2>err
$ cat err
ls: невозможно получить доступ к '*.doc': Нет такого файла или
каталога
$ cat out
a.txt   b.txt
```

В первом случае был перенаправлен только поток вывода, во втором — ошибок, в третьем — оба, но в разные файлы. Перенаправление в один файл возможно, но для этого во втором случае нужно указать вместо файла номер первого потока, предварив его символом “&”:

```
$ ls *.txt *.doc >outerr 2>&1
$ cat outerr
ls: невозможно получить доступ к '*.doc': Нет такого файла или
каталога
a.txt   b.txt
```

так как в противном случае возникнет ошибка оболочки из-за невозможности дважды открыть один и тот же файл для записи.

Для перенаправления стандартного потока ввода рассмотрим команду `cat`: она выводит на стандартный поток вывода указанные в качестве параметров файлы, а если файлы не указаны, то выводится содержимого стандартного потока ввода. То есть, если запустить `cat` без аргументов и перенаправлений, то программа будет ждать ввода данных с клавиатуры. Завершить такой ввод можно комбинацией клавиш *Ctrl+D* (отображается как `^D`):

```
$ cat
Entering text here
Entering text here
Next line
Next line
Last line
Last line
^D
```

первая из дублирующихся строк введена пользователем, вторая — выведена программой `cat`. Таким же способом можно создавать текстовые файлы, например:

```
$ cat >file.txt
Entering text here
Next line
Last line
^D
$ cat file.txt
Entering text here
Next line
Last line
$ cat <file.txt
Entering text here
Next line
Last line
```

Последние две команды эквивалентны: в первом случае выводится указанный файл, во втором тот же результат достигнут путем перенаправления потока ввода.

Специальное устройство `/dev/null` может использоваться для перенаправления потоков: будучи использованным в качестве файла ввода, оно представляет собой пустой файл, а в качестве файла вывода — "бездонную бочку", информация, выводимая в которую, будет удаляться (то есть операция вывода в данное устройство игнорируется). Таким образом можно

“подавить” нежелательный вывод команды

```
$ команда >/dev/null
```

сохранит только вывод сообщений об ошибках,

```
$ команда 2>/dev/null
```

оставит только поток вывода, а

```
$ команда >/dev/null 2>&1
```

всегда отработает молчаливо. Это удобно при запуске графических приложений в фоновом процессе, чтобы не “засорять” терминал их выводом, который будет перемешиваться с выводом команд, работающих на переднем плане, и командной строкой оболочки.

Поток вывода можно перенаправить не только в файл, но и непосредственно в командную строку — осуществить **подстановку команды** (*англ.* ***command substitution***). Это можно сделать с помощью классической комбинации обратных апострофов (обратных одинарных кавычек), между которыми указать команду, либо воспользоваться более современным вариантом — указать команду в круглых скобах после символа “$”: `` `команда` `` или `$(команда)`. Например, результат подстановки можно использовать в качестве части команды `echo` для дополнения вывода. Команда `date` выводит текущие время и дату на экран — в стандартный поток вывода — отедьной строкок:

```
$ date
Пт 13 сен 2013 13:13:13 MSK
```

Дополнить эту строку можно следующим образом:

```
$ echo Сегодня `date`
Сегодня Пт 13 сен 2013 13:13:13 MSK
```

или сохранить в переменной:

```
$ data="`date`"
$ echo $date
Пт 13 сен 2013 13:13:13 MSK
$ host="$(hostname)"
$ echo $host
localhost.localdomain
```

Еще один пример использования — команда `mktemp`, которая создает *временный файл*. Строго говоря, создаваемый ею файл не является временным автоматически, его нужно удалять вручную, но, важно, что во-первых, файл создается с уникальным случайным именем (никогда не перезаписывается существующий файл), во-вторых, файл создается в каталоге, предназначенным для временных файлов, например по умолчанию в `/tmp` или в каталоге, заданным в переменной окружения `TMPDIR` (содержимое каталога `/tmp` обычно очищается при перезагрузке системы,

т.к. хранится в оперативной памяти, а не на постоянном носителе; в то же время объем такого каталога невелик, сохраненные в нем данные следует очищать сразу же, как только они станут не нужны). Имя созданного файла выводится на поток вывода. Таким образом, требуется сохранить его для дальнейшего использования — записи и чтения данных и удаления по завершении работы и удаления. Сделать это можно в переменной с помощью

```
$ tfile=`mktemp`
```

или

```
$ tfile=$(mktemp)
```

после чего можно использовать файл, имя которого записано в переменную `tfile`, например для перенаправления ввода-вывода:

```
$ команда >"$tfile"
$ cat "$tfile"
...
$ rm "$tfile"
```

Экранирование переменной в кавычках желательно, так как теоретически имя созданного файла может содержать пробелы (хотя такое обычно не практикуется).

Еще два способа перенаправления потока ввода — создание **местного документа** (*англ.* ***here document***) с помощью символов `<<` и **местной строки** (*англ.* ***here string***) с помощью символов `<<<`.

```
$ cat <<EOF >file.txt
Entering text here
Next line
Last line
EOF
$ cat file.txt
Entering text here
Next line
Last line
```

Здесь `EOF` — выбранная последовательность, которая будет означать конец файла[1]. Все строки, введенные после команды с перенаправлением ввода на местный документ, до указанной после "`<<`" строки, будут считаться содержимым стандартного ввода. Использование местного документа на первый взгляд аналогично использованию стандартного ввода, но между

[1] EOF расшифровывается как *End Of File* — конец файла. Это стандартная аббревиатура, используемая в этом случае чаще всего, но вместо нее можно указать любую строку.

ними есть принципиальная разница: при написании сценариев *содержимое местного документа записывается в самом файле сценария, а не запрашивается интерактивно*, то есть местный документ — не стандартный ввод, а часть команды. Следующий пример — местная строка: здесь стандартным вводом считается строка, указанная после "<<<"[1].

```
$ cat <<< "Here string" >file.txt
$ cat file.txt
Here string
```

Конечно, такой результат можно достичь и с помощью

```
$ echo Here string >file.txt
```

что и является более предпочтительным в данном примере, однако в более сложных случаях использование местной строки может быть оправдано.

Группировка команд позволяет перенаправить поток с нескольких команд в один файл. Группировка осуществляется с помощью фигурных скобок.

```
$ {
> echo Directory temp content:
> ls -1 /tmp
> echo file.txt content:
> cat file.txt
> } > out.txt
```

Заметим, что сделать это можно и в одну строку. Для этого следует использовать символ разделения команд без их прямого выполнения — точку с запятой ";" (указаны только две команды из предыдущего примера для краткости):

```
$ { ls -1 /tmp; cat file.txt ;} > out.txt
```

Отделение открывающей фигурной скобки пробелом и точка с запятой перед закрывающей фигурной скобкой обязательны.

§2.17. Фильтры: утилиты обработки файлов

Многие команды работают по принципу **фильтров** (*англ.* ***filter***) — программ, преобразующих данные входного потока в данные выходного потока по определенному правилу. Программа *cat* является примером

[1]Кавычки вокруг строки нужны из-за наличия в строке пробелов, так же, как, кстати, и в имени файла при перенаправлении потоков из или в него, если в имени файла содержатся пробелы или другие служебные символы оболочки.

фильтра: она выполняет тождественное преобразование, то есть данные входного передаются на выходной в неизменном виде. Многие фильтры, как и *cat*, могут преобразовывать файлы, указанные в качестве аргументов, с передачей результата на поток вывода. Благодаря таким возможностям оболочки, как перенаправление потоков, в том числе описанное ниже перенаправление потока вывода одной программы в поток ввода другой, фильтры являются очень мощным инструментом автоматизации обработки файлов.

Рассмотрим несколько примеров фильтров:

фильтр	описание	пример вызова	пояснение примера
sort	сортировка файла (упорядочивание строк по алфавиту)	`sort -rn`	сортировка в порядке убывания (`-r`) чисел (`-n`: в отличие от алфавитного порядка, 1000 числовое больше, чем 500)
head	вывод начальной части файла	`head -5`	вывод первых 5 строк файла
tail	вывод конечной части файла	`tail -7`	вывод последних 7 строк файла
uniq	удаление повторяющихся строк файла	`uniq -i`	работа без учета регистра: строки, записанные одними символами в разном регистре будут считаться одинаковыми
tr	удаление или конвертация символов	`tr [:upper:] [:lower:]`	перевод верхнего регистра в нижний
cut	получение части строки каждого файла	`cut -d" " -s -f2,4`	вывести только 2 и 4 слово каждой строки (слова разделяются пробелами)

Не являются фильтрами в классическом смысле этого слова, но полезны для просмотра входных потоков, такие программы как *more* и *less*: первая осуществляет постраничный показ входного потока, переход на следующую страницу производится клавишей *Enter*, а вторая представляет собой просмотрщик с возможностью пролистывания документа, поиска и т.д. — именно он используется командной `man` для демонстрации документации. Также утилита *diff* позволяет провести построчное сравнение содержимого двух текстовых файлов, т.е. ей недостаточно одного входного потока, два файла должны быть указаны в качестве аргументов.

§2.18. Конвейеры и подстановка процессов

Существует возможность перенаправить выходной поток одной команды во входной поток другой команды. Это позволяет комбинировать фильтры между собой с обработкой данных "на лету", то есть без создания промежуточных временных файлов. Такая обработка называется **конвейером** (*англ.* ***pipeline***). Конвейер организуется с помощью символа "`|`", который перенаправляет стандартный вывод первой команды на стандартный ввод второй. Например,

```
$ iconv -f cp1251 file.txt | sort | head -10
```

конвертирует файл `file.txt` из кодировки `cp1251` в текущую, сортирует построчно и выводит первые 10 строк результата. Таким же образом можно переправить длинный вывод в интерактивный просмотрщий `less` или постраничный просмотрщик `more`, например

```
$ ps -H ax | less
```

Перенаправление потока ошибок также возможно с помощью "`|&`".

Команда `tee` позволяет "продублировать" стандартный поток ввода — записать его копии в файл и в поток вывода. Например, чтобы и вывести результат сортировки файла на экран и в другой файл можно использовать

```
$ sort file.txt | tee file.sorted.txt
```

а чтобы сохранить полную версию упорядочивания, но просмотреть только первые 10 строк вызвать

```
$ sort file.txt | tee file.sorted.txt | head -10
```

Программа также позволяет обойти проблему перенаправления вывода `sudo`. Дело в том, что вызов

```
$ sudo echo Message >>/root/msg.txt
```

из пользовательской оболочки приведет к ошибке, так как такая оболочка не имеет права записи в каталог `/root` и не может перенаправить вывод `sudo` туда. В данном случае от имени администратора будет исполнена

только команда `echo`. Решение состоит в следующем:

```
$ echo Message | sudo tee /root/msg.txt >/dev/null
```

Программа `tee` будет запущена от администратора и выполнит запись потока в файл, а перенаправление в `/dev/null` позволит избежать дублирования сообщения в терминал.

Подстановка процессов (*англ.* ***process substitution***) — возможность перенаправить вывод процесса в файл-аргумент команды. Например, если команда требует два файла в качестве аргумента, а нужно использовать вывод двух других команд, простого конвейера недостаточно. Тогда можно использовать данный инструмент. Его формат следующий: "`<(`*команда*`)`". Команда будет исполнена, а ее стандартный вывод окажется в специальном временном файле, который будет передан в качестве аргумента в командную строку. Например, чтобы сравнить содержимое двух каталогов (только списки файлов, но не содержимое файлов) можно использовать

```
$ diff <(ls /dir1) <(ls /dir2)
```

Перенаправление вывода вместо файла-параметра в стандартный ввод другой команды также возможно с помощью "`>(`*команда*`)`". В обоих случаях отсутствие пробела между знаками перенаправления ("`>`", "`<`") и скобкой необходимо[1].

Заметим, что при подобных перенаправлениях данных от одного процесса к другому — а данные конструкции создают отельный процесс для каждой вызываемой команды (программы) — используются т.н. каналы. Это файлы особого типа, в которые один процесс записывает данные к конец, а другой — прочитывает с начала. Средства перенаправления оболочки создают временные безымянные (анонимные) каналы автоматически, прозрачно для пользователя[2].

§2.19. Преобразование кодировки символов и символа переноса строк

Текстовые файлы (*англ.* ***text files***) представляют собой файлы особого формата. В отличие от **двоичных** (*англ.* ***binary***) файлов различных форматов, предназначенных для прочтения и обработки с помощью приложений, поддерживающих данный формат, текстовые файлы являются

[1] Команду также можно использовать в качестве файла-перенаправления после "`<`" или "`>`", в этом случае нужен пробел перед вторым знаком перенаправления: "`> >(`*команда*`)`" — перенаправление вывода, "`< <(`*команда*`)`" — перенаправление ввода.

[2] Создание именованных каналов (файлов соответствующего типа, которые также могут использоваться для обмена данными между процессами) возможно с помощью команды `mkfifo`.

человекочитаемыми практически без дополнительной обработки, просто путем просмотра графического отображения содержащихся в нем символов. Однако следует отметить, что

- то, какой графический символ соответствует кодам символов, записанным в файле, определяет **кодировка символов** (*англ.* ***character encoding, code page — кодовая страница***), различные операционные системы, приложения и локализации могут использовать различные кодировки символов, несовместимые между собой;
- различные операционные системы используют различный код для обозначения конца строки текста: в системах семейства *Unix* это символ с шестнадцатеричным кодом `0A` (десятичный код — 10), в ОС семейства *DOS/Windows* — пара символов `0D0A` (десятичные коды 13 и 10), в Mac OS — только символ с кодом `0D`[1].

Таким образом, при передаче с одной операционной системы в другую, текстовый файл может оказаться нечитаемым без дополнительных действий.

Для преобразования символов перевода строк существуют команды `dos2unix` (перевод из формата *DOS/Windows* в формат *Unix*), `unix2dos` (обратный перевод), `mac2unix` (перевод из формата Mac OS в формат *Unix*), `unix2mac` (обратный перевод). Данные программы могут работать как фильтры (если запущены без аргументов) или преобразовывать файлы "на месте", если они указаны в качестве параметров.

Для преобразования кодировок символов можно использовать программы `iconv`, `konvert`, `recode` и др. Современные версии *GNU/Linux* используют кодировку *UTF-8* в качестве основной, она позволяет использовать в одном файле символы практически всех имеющихся языков. Для отображения русских букв графических приложений *Windows* используется кодировка *cp-1251*[2]. Например, для преобразования из русскоязычного документа *Windows* в *Linux*-формат следует вызвать

```
$ dos2unix file.txt
$ recode windows1251..utf8 file.txt
```

или

```
$ cat file.txt | dos2unix | iconv -f cp1251 >file-cnv.txt
```

[1] Современные *OS X*, как и *Unix*, используются символ `0A`.

[2] Существуют различные способы записи имен кодировок, например *восьмибитовый юникод — UTF-8*, *UTF8*, *utf8*; *кириллица Windows — cp1251*, *cp-1251*, *windows-1251* и др, приложения могут понимать только некоторые из них, поэтому всегда следует обращаться к документации.

В последнем случае не указывается кодировка назначения, т.к. `iconv` в этом случае использует текущую кодировку системы. Фильтрация производится в другой файл.

§2.20. Метасимволы оболочки

Метасимволы (*англ.* ***metacharacters***) — служебные символы оболочки. Будучи указанными в команде (и не будучи экранированными) они приводят к модификации команды или ее поведения самой оболочкой. Оболочка перерабатывает команду как строку с получением новой строки — раскрывает метасимволы — разбивает команду и аргументы по пробелам (причем повторяющиеся пробелы считаются одним символом разбиения), перенаправляет ввод и вывод (предварительно исполнив команды подстановки процессов) после чего передает команде список строк-аргументов. При написании команд и сценариев следует иметь в виду этот процесс — *Bash* не разбивает команду на токены для вычисления выражения и не исполняет операторы в смысле классических языков программирования. Оболочка именно обрабатывает командную строку как строку символов, осуществляя подстановку на место метасимволов и их комбинаций результат их раскрытия. Поэтому

```
$ var=value
$ echo $vars
```

выведет "значение" переменной `vars`, а не `values`, в то время как

```
$ ls $var""s
```

и

```
$ ls "$var"s
```

являются попыткой получить информацию о файле `values`, а не файлов `value` и `s`. Команда

```
$ echo 1   2
```

выведет

```
$ 1 2
```

(т.е. оставит только один пробел — в отличие от "`echo "1   2"`") так как повторяющиеся разделители воспримется как один. Однако

```
$ ls $var "" s
```

есть попытка получить информацию о трех файлах — `value`, `s` и файла с пустым именем (существование которого, конечно, не допускается, но аргументы, значения которых есть пустые строки разрешены).

Приведем таблицу метасимволов и их комбинаций с пояснением смысла (который в некоторых случаях зависит от контекста, поэтому эту таблицу не следует считать точным определением раскрытия отдельных ме-

тасимволов). Некоторые символы уже пояснялись ранее, некоторые будут пояснены в дальнейшем.

символ	пояснение
*	подстановка файлов (любая последовательность символов, в т.ч. пустая)
?	подстановка файлов (любой непустой символ)
[ ]	подстановка файлов (любой символ из списка или диапазона)
[^]	подстановка файлов (любой символ, кроме символов списка)
>	перенаправление вывода (перезапись)
>>	перенаправление вывода (добавление)
<	перенаправление ввода (файл)
<<	перенаправление ввода (местный документ)
<<<	перенаправление ввода (местная строка)
` `	подстановка процесса (вывод в командную строку)
$( )	подстановка процесса (вывод в командную строку)
<( )	подстановка процесса (входной файл из стандартного вывода)
>( )	подстановка процесса (выходной файл в стандартный ввод)
|	конвейер (вывод во ввод)
(( ))	арифметические и логические выражения (вычисление)
$(( ))	арифметические и логические выражения (вычисление и подстановка результата)
$	значение переменной
${ }	обработка значения переменной
$' '	получение специальных символов
( )	создание массива
a[ ]	получение элемента массива *a*
{ }	группировка команд
;	последовательность команд (вместо переноса строки)
||	исполнение по логическому ИЛИ
&&	исполнение по логическому И
!	исполнение по логическому НЕ
&	запуск в фоновом процессе

`:`	команда “ничего не делать” (используется там, где команда обязательна синтаксически, но не нужна семантически)
`#`	комментарий (игнорирование текста до конца строки)
`{ , }`	подстановка вариантов (формирование списка аргументов из одного с подстановкой значений из списка)
`{ .. }`	подстановка последовательности чисел (начало, конец, инкремент, считается 1 если не указан)
`~`	домашний каталог текущего пользователя
`~`*логин*	домашний каталог указанного пользователя
`~`*n*	*n*-й с начала каталог в стеке каталогов
`~-`*n*	*n*-й с конца каталог в стеке каталогов
`\`	экранирование следующего символа
`" "`	нестрогое экранирование (почти всех метасимволов символов между двойных кавычек)
`' '`	строгое экранирование (всех символов между одинарных кавычек)

§2.21. Экранирование метасимволов

Экранирование (*англ.* ***escaping***) служебного символа — запрет его раскрытия в качестве служебного, “превращение” в обычный символ. Символ “\” (обратный слеш, символ экранирования, ESC-символ — эскейп-символ) подавляет восприятие следующего за ним служебного символа оболочки в качестве такового, превращая в обычный символ, элемент командной строки. Таким образом возможно экранирование пробелов, шаблонов подстановки, знака “$”, кавычек и апострофов и других метасимволов. ESC-символ экранирует и самого себя, то есть сочетание “\\” означает передачу команде одиночного символа “\”.

Почти все символы, заключенные в двойные кавычки, считаются экранированными, кроме символа “$”, символа обратного апострофа “`”, а также символа “\”, если он указан перед символом “$”, символом “`”, символом “\” или символом двойной кавычки. Это означает, что в двойных кавычках происходит только раскрытие переменных, вызов команды для получения ее стандартного вывода в раскрываемую команду и сохраняется возможность экранировать эти символы, а также собственно саму кавычку.

```
$ echo $PATH
/bin:/usr/bin:/sbin:/usr/sbin:/opt/bin
```

```
$ echo "$PATH"
/bin:/usr/bin:/sbin:/usr/sbin:/opt/bin
$ echo "\$PATH"
$PATH
$ echo "\""
"
$ echo "\'"
\'
$ echo "\\"
\
```

Заметим, что попытка набрать команду

```
$ echo "\"
```

приведет к тому, что оболочка обнаружит непарную кавычку и предложит завершить ввод команды:

```
>
```

Если ввести

```
> 123"
```

получим

```
"
123
```

поскольку символ перевода строки стал частью команды. ***Если незавершенная команда возникла по ошибке и исполнять ее не нужно, можно нажать сочетание Ctrl+C для прерывания.***

Все метасимволы, заключенные в одинарные кавычки (апострофы), считаются экранированными, кроме самого символа апострофа “’”.

```
$ echo '\$PATH'
\$PATH
$ echo '\$PATH'
\$PATH
$ echo '\"'
\"
$ echo '\\'
\\
$ echo '\''
> 123'
\
123
```

(в последней команде второй апостроф не был экранирован).

Допустимы комбинации различных способов экранирования:

```
$ echo "'"'\$PATH'\'
```

```
'\$PATH'
```

то есть ***символы экранирования не добавляют пробелов, не влияют на границы аргументов.*** Оболочка "не плодит сущностей", а четко преобразует командную строку в соответствии с правилами, результатом преобразования является упорядоченный набор строк, передаваемых в команду в качестве аргументов, точками разрыва являются только неэкранированные пробелы.

Заметим, что перенос строки также может быть "экранирован" с помощью ESC-символа, что позволит разбить команду на несколько строк. Но в этом случае, в отличие от экранирования кавычками и апострофами, символ не будет включен в команду.

```
$ echo 1\
> 2
12
```

После раскрытия значений переменных происходит дальнейшее раскрытие шаблонов подстановки. Например,

```
$ all="*"
```

записывает в переменную *all* символ "*", но при вызове

```
$ echo $all
```

будет выведен список всех файлов текущего каталога, а не символ "*". Чтобы избежать этого, нужно экранировать получение значения переменных:

```
$ echo "$all"
*
```

Заметим, что фигурные скобки не являются шаблоном подстановки:

```
$ all="{1,2,3}"
$ echo $all
{1,2,3}
```

В то же время символы подстановки не раскрываются при присвоении значений переменной, поэтому записи

```
$ all="*"
```

и

```
$ all=*
```

эквивалентны.

Символ дефиса (минуса) — "-" — не является метасимволом и обрабатывается не оболочкой, а самими командами. Поэтому, если в имени файла встречается метасимвол, то экранирование является решением проблемы, но если имя файла начинается с символа "-", то он может быть воспринят как верная или неверная опция команды. Например, удаление файла с именем `-f` затруднено: вызов

```
$ rm -f
```
является ошибочным (имеется опция, но не указано имя файла),
```
$ rm \-f
```
имеет тот же эффект. Поэтому для достижения указанного результата следует использовать альтернативный подход: избежать аргумента начинающегося с минуса
```
$ rm ./-f
```
или опцию, разграничивающую опции и параметры "`--`"
```
$ rm -- -f
```
(если таковая поддерживается командой).

§2.22. Поиск по регулярным выражениям

Регулярные выражения (*англ.* ***regular expression***) — способ задания шаблонов для поиска **образцов** (*англ.* ***pattern***) в строках, то есть поиска строк, содержащих вхождения подстрок, попадающий под заданный шаблон поиска. Регулярные выражения похожи на символы-джокеры для задания шаблонов поиска файлов с подходящими именами, но отличаются от них применяемыми символами и их комбинациями. В различных языках программирования и утилитах могут использоваться различные (иногда несовместимые) типы регулярных выражений, но в большинстве из них имеется общая база. Стандартным средством поиска по регулярным выражениям в *GNU* является утилита `grep`, могут использоваться также расширенные версии `egrep` (допускающая, например, поиск с разрывом строк), `pcregrep` и другие.

Принцип работы этих команд — находить файлы, содержащие строки, включающие указанный образец, сами строки в этих файлах, или анализировать стандартный поток ввода, если файлы не указаны. В простейшем виде `grep` является утилитой поиска текстовых файлов, содержащих указанный текст, или выделения только тех строк потока ввода, в которых содержится указанный текст.

Например, для поиска процессов, чье имя содержит слово `bash`, можно набрать:
```
$ ps ax | grep bash
```
Тогда на экран будут выведены только те строки вывода `ps`, которые содержат `bash` — так можно найти *pid* нужных процессов, не прибегая к анализу длинного списка всех процессов. С помощью ключей `-A` и `-B` можно указать сколько строк вывести после и до найденной строки соответственно, например для анализа дерева процессов можно использовать
```
$ ps ax -H | grep mc -A 5 -B 10
```

чтобы увидеть дочерние и родительские процессы процесса, имя которого содержит `mc`. Заметим, что команда `pgrep` позволяет найти *pid* процессов автоматически, также используя регулярные выражения, а `pkill` — послать сигнал найденным процессам аналогичным образом.

Для поиска файлов, содержащих образец, следует указать

```
$ grep образец файл...
```

Список файлов можно задать явно или с использованием шаблонов оболочки.

Регулярные выражения — не просто строки, а шаблоны: в них используются символы подстановки и их комбинации, но символы подстановки оболочки и регулярных выражений различны. Приведем некоторые примеры метасимволов регулярных выражений.

символ	пояснение	пример	подходящие строки примера
`.`	любой непустой символ	`a.b`	“`aab`”, “`a1b`” и др.
`*`	повторение предыдущего символа любое количество раз, в т.ч. нулевое	`a.*b`	“`ab`”, “`aab`”, “`a123b`” и т.д.
`+`	повторение предыдущего символа любое ненулевое количество раз	`a.+b`	“`aab`”, “`a123b`” и т.д.
`[]`	один из указанных символов	“`a[123]b`”	“`a1b`”, “`a2b`”, “`a3b`”
`[-]`	диапазон символов	“`1[a-d]2`”	“`1a2`”, “`1b2`”, “`1c2`”, “`1d2`”
`[^]`	символ, не совпадающий с заданным	“`a[^12]b`	все, кроме “`a1b`”, “`a2b`”
`[[:digit:]]`	цифра	“`a[[:digit:]]b`”	“`a0b`”, “`a2b`”, ..., “`a9b`”

`[[:lower:]]`	строчная буква	“`a[[:lower:]]`”	“`aa`”, “`ab`”, . . . , “`az`”, а также все возможные строчные буквы из текущего языка, предваренные “`a`”
`[[:upper:]]`	заглавная буква	“`[[:upper:]]+`”	любое непустое слово, записанное заглавными буквами без пробелов
`^`	начало строки	“`^#`”	все строки, начинающиеся с “`#`”
`$`	конец строки	“`;$`”	все строки, заканчивающиеся на “`;`”

Дополнительно можно отметить “`{n}`” — повторение предыдущего символа ровно n раз, “`{n,}`” — повторение предыдущего символа по крайней мере n раз, “`{,n}`” — повторение предыдущего символа не более, чем n раз, “`{m,n}`” — повторение предыдущего символа не менее, чем n, но не более, чем m раз и другие выражения.

Некоторые из метасимволов регулярных выражений совпадают с метасимволами оболочки, поэтому строка-образец должна быть обязательно экранирована. В свою очередь внутри строки может потребоваться экранирование метасимволов, чтобы они перестали быть символами регулярного выражения, а были восприняты как обычные символы. Сравним

```
$ grep a.*b
```

команда получит все файлы текущего каталога, попадающие под шаблон “`a.*b`”. Имя первого из них будет воспринято `grep` как регулярное выражение для поиска, остальные — как файлы, в которых следует осуществлять поиск. Скорее всего, такое поведение отличается от желаемого. Только в случае отсутствия файлов, попадающих под указанный шаблон, `grep` получит указанную строку в качестве аргумента и этот случай совпадет с правильно экранированным:

```
$ grep "a.*b"
```

осуществляет поиск регулярного выражения “`a.*b`” — то есть строк, содержащих буквы “`a`” и “`b`” (в указанном порядке), разделенных любой последовательностью символов — в потоке ввода.

С другой стороны

```
$ grep "a\.*b"
```

осуществляет поиск строк, содержащих буквы “a” и “b” (в указанном порядке), разделенных любым количеством символов “.” в потоке ввода.

```
$ grep "a.\*b"
```

осуществляет поиск строк, содержащих букву “a” и сочетание “*b” (в указанном порядке), разделенные любым непустым символом, в потоке ввода.

```
$ grep "a\.\*b"
```

осуществляет поиск строк, содержащих подстроку “a.*b”, в потоке ввода.

§2.23. Рекурсивный поиск файлов

Команда `find` позволяет осуществлять поиск файлов во всех подкаталогах заданного каталога, удовлетворяющим заданным критериям. Например, чтобы найти все файлы с расширением `txt` в каталоге `Documents` следует вызвать

```
$ find Documents -name "*.txt" -type f
```

Важно обратить внимание на следующие моменты:

- каталог всегда должен быть указан до всех аргументов, определяющих какие файлы следует искать — условий (тестов) поиска, если каталог не указан, осуществляется поиск в текущем каталоге;
- `-name` указывает шаблон имени файла, которые следует находить: *этот шаблон следует экранировать, так как его должна обработать сама команда* `find`*, а не оболочка*, иначе результат будет не такой, как ожидается;
- `-type` определяет тип файла поиска, в данном случае `-f` — обычный (регулярный) файл (а не каталог, устройство и т.д.);
- существует большой набор условий, задающих критерий отбора файлов — по имени, по типу, по размеру, по дате создания, владельцу, правам доступа и др., которые можно комбинировать с помощью логических операций — `-o` (“ИЛИ”), `-a` (“И”, также применяется по умолчанию, если два условия следуют подряд, как в примере выше);
- различные ключи программы регулируют способы поиска файлов, например глубину рекурсии по дереву каталогов, переход по символическим ссылкам и т.п.;
- ко всем найденным файлам можно сразу применить команду с помощью ключа `-exec` или удалить их с помощью ключа `-delete` (следует применять с особой осторожностью);
- по умолчанию программа выводит имена на стандартный вывод построчно, с полным путем или относительным путем, в зависимости от того, как задан каталог поиска.

При указании `-exec` после данного ключа должна следовать команда для применения к каждому файлу со своими аргументами, завершенная символом точкой с запятой “;”, который следует обязательно экранировать в оболочке. В качестве аргумента команды можно использовать сочетание “`{}`” — оно будет заменено на имя файла. Оба этих символа также должны быть экранированы. Например, можно вызвать

```
$ find -name "*.txt" -exec dos2unix '{}' \;
```

чтобы преобразовать все файлы с расширением `txt` в текущем каталоге и всех его подкаталогах из формата переноса строк *DOS* в *Unix*.

Вопросы для самопроверки.

1. *Чем отличаются внутренние и внешние команды оболочки?*
2. *Что представляют собой переменные окружения?*
3. *В какой переменной хранятся пути поиска внешних команд? Каков формат ее значения?*
4. *Что такое текущий рабочий каталог?*
5. *Что представляют собой относительные и абсолютные пути?*
6. *Каков синтаксис передачи опций и аргументов в команду?*
7. *Как записывается ссылка на родительский и текущий каталог в относительном пути?*
8. *Опишите шаблоны подстановки файлов и их обработку оболочкой.*
9. *Как передаются пробелы и другие спецсимволоы в качестве параметров команды для того, чтобы избежать их обработки оболочкой?*
10. *В чем различие логина и уникального идентификатора пользователя?*
11. *Какие атрибуты прав доступа имеются? Разрешения на какие действия и каким пользователям они предоставляют?*
12. *Как задаются и прочитываются значения переменных в командной строке?*
13. *В чем различие между внутренними и экспортными переменными оболочки?*
14. *Что представляют собой и для чего используются стандартные потоки ввода, вывода и ошибок?*
15. *Как перенаправить стандартный поток в файл?*

16. *Как осуществить перенаправление стандартного потока вывода в командную строку?*
17. *Как осуществляется перенаправление стандартного вывода одной команды в ввод другой?*
18. *Какие есть способы экранирования метасимволов оболочки?*
19. *Чем различаются строгое и нестрогое экранирование метасимволов оболочки?*
20. *Что такое регулярное выражение?*

Глава 3. Средства программирования сценариев оболочки *Bash*

§3.1. Коды возврата и простейшая проверка условий

Помимо стандартного вывода всякая программа имеет **код возврата** (*англ.* ***return code***) или **статус завершения** (*англ.* ***exit status***). Это целое, обычно беззнаковое, число, передаваемое родительскому процессу по завершении процесса. Передачу кодов возврата обеспечивает ОС, они существуют "внутри" процессов, не выводятся куда-либо. Задача передавать информацию посредством кода возврата возлагается на саму программу. При этом принята следующая практика: нулевой код возврата соответствует успешному завершению программы, ненулевой — неудаче, то есть ситуации, когда при работе программы возникла ошибка, причем чем значение выше, тем условно "хуже" эта ошибка. В руководстве *man* можно найти информацию о смыслах конкретных кодов возврата конкретных команд, а если она не указана, то, скорее всего, команда возвращает 0 при успехе, а 1 — при неудаче. В некоторых случаях оболочка настроена таким образом, что ненулевой код возврата предыдущей команды будет указан в приглашении ко вводу следующей.

Просмотреть код возврата предыдущей команды можно в переменной "?", например

```
$ echo $?
```

Следует учесть, что после исполнения любой другой команды, включая команду этого примера, код в `$?` сразу же будет заменен.

```
$ cp prg /bin
cp: невозможно создать обычный файл '/bin/prg':
Отказано в доступе
$ echo $?
```

```
1
$ echo $?
0
```

В первом случае выведено `1`, поскольку `cp` завершилась с ошибкой, во втором — `0`, поскольку предыдущее `echo` сработало без ошибок.

Коды возврата можно использовать для автоматической проверки, успешно ли завершилась предыдущая команда и предпринимать ли дальнейшие действия. Для этого служат специальные символы "`&&`" и "`||`". Первое представляет собой эквивалент логического *И*, второе — логического *ИЛИ*. Операции логического *И* и логического *ИЛИ* устроены таким образом, что вычисление второго операнда производится только, если по первому нельзя однозначно определить результат. Так для логического *И ложь* в первом операнде означает, что и результат — *ложь*. Для логического *ИЛИ истина* в первом операнде означает, что и результат — *истина*. В остальных случаях результат зависит от второго операнда и его следует вычислить.

В оболочке считается, что истина — успех — соответствует коду возврата 0*, а ложь — ненулевому коду возврата.* Поэтому

```
$ cp prg /bin && prg
```

попытается скопировать программу `prg` из текущего каталога в каталог `/bin` и исполнить получившуюся новую команду только в случае успеха.

```
$ cp prg /bin || echo ERROR
```

выведет сообщение `ERROR` в случае провала. Возможна и комбинация проверок условий:

```
$ cp prg /bin && prg || echo ERROR
```

а если при этом подавить сообщение об ошибке команды `cp`

```
$ cp prg /bin 2>/dev/null && prg || echo ERROR
```

то в случае ошибки будет выведено только слово `ERROR`.

Специальный символ `!` позволяет "инвертировать" код возврата — ненулевой заменяется нулем, а нулевой — единицей, что соответствует логическому отрицанию последующей команды. Поэтому

```
$ ! grep -q sample file.txt && echo OK
```

выведет `OK` если образец `sample` в файле `file.txt` отсутствует (опция `-q` приводит к тому, что `grep` работает молчаливо — не выводит найденные строки в поток вывода, а лишь передает код возврата в зависимости от того, найдена хотя бы одна требуемая строка или нет).

§3.2. Тесты и условия

Как уже было описано выше, проверка условий в оболочке заключается в анализе кодов возврата, причем нулевой код возврата соответствует

успеху (истине), а ненулевой — провалу (лжи). В качестве условий могут выступать любые команды, генерирующие различный код возврата в зависимости от заданных параметров и состояния компьютера, ОС или файловой системы. Выше были приведены примеры простейшей проверки условий с помощью конструкций "&&", "||" и "!". Следующие параграфы будут посвящены более сложным и гибким конструкциям. Данный параграф посвящен командам, специально предназначенным для проверки ряда условий — сравнения строк и чисел, наличия файлов и каталогов и т.п.

Таких команд три

- команда `[` — стандартная в *POSIX*, условия выглядят как записанные в квадратные скобки, например для сравнения строк можно написать

```
$ [ "$var" = value ]
```

- команда `[[` — встроенная в *Bash*, имеет расширенные возможности, менее прихотлива к синтаксису, условия выглядят как записанные в квадратные скобки, например для сравнения строк можно написать

```
$ [[ $var = value ]]
```

- команда `test` — аналог команды `[`, но не требующий закрывающую квадратную скобку, например для сравнения строк можно написать

```
$ test "$var" = value
```

Следует обратить внимание на

- наличие пробелов после имени команды (`[`, `[[` и `test`) — без них условие станет частью имени команды, а не аргументов
- наличие пробелов вокруг знака равенства и перед закрывающимися фигурными скобками — они нужны для корректного разделения аргументов, передаваемых в команду, без пробелов вокруг знака равенства он будет воспринят как часть строки;
- наличие экранирования переменной в командах `[` и `test`: без него пробелы в значении переменной будут восприняты как разделители аргументов, а в ситуации, когда переменная пуста, левый операнд сравнения будет отсутствовать — оба случая приведут к ошибке; команда `[[`, будучи встроенной в *Bash* с собственным синтаксисом, "понимает" переменные как отдельные параметры;
- явно указываемые (как строки, а не как переменные) значения сравнения, содержащие пробелы и метасимволы, следует экранировать даже для команды `[[`;

- получить информацию о командах `[` и `test` можно в справочном руководстве по `man test`, по `[[` — с помощью `help [[`;
- оболочка *Bash* использует встроенные версии команд `[` и `test`, которые могут несколько отличаться от стандартных, справка по ним также доступна с помощью `help`.

Пример использования условия:

```
$ [[ $a == $b ]] && echo YES || echo NO
```

выводит сообщение `YES`, если значения переменных `a` и `b` равны, и `NO` в противном случае

Некоторые тестовые операции приведены в таблице (список не полный).

Конструкция	**Пояснение**	**Примечание**
	Сравнения строк	
`-n` *строка*	длина строки не равна 0	
`-z` *строка*	длина строки равна 0	в т.ч. если переменная не задана
s_1 `=` s_2	строки s_1 и s_2 равны	
s_1 `==` s_2	строка s_1 соответствует *Bash*-шаблону s_2	только для команды `[[`
s_1 `~=` s_2	строка s_1 соответствует регулярному выражению s_2	только для команды `[[`
s_1 `!=` s_2	строки s_1 и s_2 не равны	в команде `[[` строка s_2 трактуется как шаблон
s_1 `<` s_2 s_1 `>` s_2	лексикографический порядок строк	только для встроенных команд *Bash*
	Сравнения целых чисел	
n_1 `-eq` n_2	$n_1 = n_2$	`[[ 123 < 23 ]]` — истина (в алфавитном порядке слово 123 следует раньше 23), `[[ 123 -lt 23 ]]` — ложь (число $123 < 23$).
n_1 `-ne` n_2	$n_1 \neq n_2$	
n_1 `-lt` n_2	$n_1 < n_2$	
n_1 `-le` n_2	$n_1 \leqslant n_2$	
n_1 `-gt` n_2	$n_1 > n_2$	
n_1 `-ge` n_2	$n_1 \geqslant n_2$	
	Файловые операции	

`-e` *файл*	файл существует	файл любого типа, в т.ч. каталог
`-f` *файл*	файл существует и является обычным файлом	
`-d` *файл*	файл существует и является каталогом	
`-h` *файл*	файл существует и является символической ссылкой	
`-x` *файл*	файл существует и является исполняемым	установлено разрешение x для текущего пользователя (в т.ч. у каталога)
файл$_1$ `-nt` *файл*$_2$	*файл*$_1$ новее, чем *файл*$_2$	сравнение по дате последнего изменения
файл$_1$ `-ot` *файл*$_2$	*файл*$_1$ старше, чем *файл*$_2$	

Внутри теста можно использовать логические операции "`-a`" (И), "`-o`" (ИЛИ) "`!`" (НЕ) и использовать скобки для определения порядка применения операций (с обязательным экранированием), а также круглые скобки для естественного определения порядка этих операций. Команда `[[` также поддерживает `&&` и `||`. С помощью конструкций *Bash* `!`, `&&` и `||` можно комбинировать результаты разных команд, в т.ч. тестов, если они указаны не в качестве аргумента, а вне команды (до или между команд):

```
$ [ "$a" -lt "$b" ] && [ "$b" -lt "$c" ]
$ [ "$a" -lt "$b" -a "$b" -lt "$c" ]
$ [[ $a -lt $b && $b -lt $c ]]
```

выполняют одну и ту же проверку — число `$b` лежит строго между `$a` и `$c` — разными способами: логикой оболочки и двумя запусками теста, логикой программы `test`, логикой команды `[[`. Следует отметить, что первый вариант может работать медленнее остальных, так как требует создания двух дочерних процессов для запуска внешней команды).

§3.3. Понятие оператора оболочки

Оператор (*англ.* ***statement***) оболочки может рассматриваться как команда особого рода, со своим синтаксисом, потоками ввода, вывода и ошибок, однако имеет и некоторые отличия:

- операторы записываются не в одну строку, а в несколько;
- операторы управляют выбором и порядком исполнения других команд — точнее групп команд, каждая из которых записывается в отдельной строчке, среди них могут быть и другие операторы (вложенные операторы); группа команд, управляемый оператором, называется **телом** (*англ.* ***body***) оператора;
- **условный оператор** (*англ.* ***condition statement***) и **оператор выбора** (*англ.* ***selection statement***) могут имеют несколько тел, оператор выбирает исполнение одного из них или неисполнение ни одного из них в зависимости от условия, **оператор цикла** (*англ.* ***loop statement***) имеет одно тело, которое может быть исполнено несколько раз, в зависимости от условия;
- поток ввода оператора передается командам тела, которые прочитывают его по мере исполнения[1], потоки вывода оператора — конкатенация потоков вывода команд тела; код возврата оператора — код возврата последней команды.

§3.4. Условный оператор

Синтаксис условного оператора имеет три формы. Первая форма — неполная

```
if команда
then команда_1
     команда_2
     ...
     команда_n
fi
```

Вторая форма — полная

```
if команда
then команда_a1
     команда_a2
     ...
     команда_an
else команда_b1
     команда_b2
     ...
     команда_bm
fi
```

[1] Если одна из них прочитает его до конца, последующие получат пустой поток.

Здесь ключевые слова `then` и `fi` (в неполном варианте), `then` и `else` (в полном варианте) ограничивают *then*-тело цикла, `else` и `fi` (в полном варианте) ограничивают *else*-тело цикла. Перенос строки перед `then`, `else` и `fi` обязателен (вместо него, разумеется, можно указать "`;`"). Перенос строки после `then` и `else` возможен, но необязателен. Наличие команды в каждом теле обязательно. Если требуется создать тело без команд (например, использовать только тело `else`, а тело `then` оставить пустым), то в качестве команды можно указать "`:`" — команду, которая ничего не делает.

Команда, указываемая после `if`, определяет условие по своему коду возврата. В случае успеха команды (код возврата 0, истина) исполняется *then*-тело цикла, *else*-тело (при наличии) игнорируется. Если код возврата команды отличен от нуля (провал, ложь), то исполняется *else*-тело цикла (при наличии), *then*-тело игнорируется. *В качестве команды-условия может использоваться любая команда, встроенная или внешняя, в том числе команда-тест.*

Например,

```
if grep -q cat animals.txt
then echo кот найден
else echo кот не найден
fi
```

выводит сообщение, в зависимости от того, найдена ли строка `cat` в файле `animals.txt`. Опция `-q` подавляет стандартный вывод самой команды `grep`, предоставляя возможность сообщить о результатах поиска только самому сценарию.

```
if [[ $file == *.txt ]]
then text=1
    format=txt
fi
```

Третья форма оператора — конструкция *if*/*then*/*elif*/*else* — позволяет избежать использования вложенных операторов `if` при проверке множества условий

```
if команда1
then команда1a
     команда2a
     ...
elif команда2
then команда2a
     команда2b
     ...
```

```
elif команда_3
then команда_2a
     команда_2b
     ...
...
elif команда_n
then команда_na
     команда_nb
     ...
else команда_a
     команда_b
     ...
fi
```

Здесь команды *команда*$_1$, *команда*$_2$, ..., *команда*$_n$ — условия, `elif` можно воспринимать как сокращение для `else if`, но такая конструкция требует только одного `fi`. Будет выполнено тело, соответствующее первому (сверху) найденному истинному условию, последующие условия проверяться (исполняться) не будут, а если такого условия нет, то будет исполнено *else*-тело. Например,

```
if [[ $file == *.txt ]]
then
     text=1
     format=txt
elif [[ $file == *.csv ]]
     text=1
     format=csv
elif [[ $file == *.odt || $file == *.ods ]]
     text=0
     format=office
else
     text=0
     format=unknown
fi
```

§3.5. Цикл по списку

Цикл `for` позволяет “перебрать” в значении переменной все элементы указанного списка и исполнить указанный набор команд (тело цикла) для каждого из значений:

```
for variable in value1 value2 ...
do команда1
   команда2
   ...
   командаn
done
```

Здесь ключевые слова `in`, `do` и `done` разграничивают, соответственно, переменную и список, начало и конец тела цикла. Перенос строки перед `do` обязателен (вместо него можно указать "`;`"). Перенос строки после `do` возможен, но необязателен. Например,

```
$ for i in 1 2 3 "4 5" 6
> do echo $i
> done
1
2
3
4 5
6
$
```

Обратите внимание на то, что элементы списка — как и аргументы — разделяются неэкранированными пробелами. Таким способом можно, например, пройти циклом по всем файлам:

```
for file in *
do echo file=\'"$file"\'
done
```

С помощью стандартной команды `seq` или средствами *Bash* можно организовать проход последовательности чисел с заданным шагом. Например, конструкция "`{1..10}`" формирует последовательность от 1 до 10 с шагом 1, "`{1..10..2}`" — с шагом 2 и т.д.:

```
for i in {10..1..-1}
do echo i=\'"$i"\'
done
```

Тот же результат можно достичь с помощью внешней команды *seq*, печатающей выводящей последовательность чисел на стандартный вывод:

```
for i in `seq 10 -1 1`
do echo i=\'"$i"\'
done
```

В *Bash* также имеется поддержка C-подобных циклов: конструкция

```
for (( i = 10; i >= 1; i- ))
do echo i=\'"$i"\'
```

```
done
```
даст тот же результат.

§3.6. Цикл по условию

Оператор цикла `while` позволяет выполнять группу команд (тело цикла) до тех пор пока условие истинно, то есть *до тех пор, пока указанная команда возвращает нулевое значение*:

```
while команда
do команда1
   команда2
   ...
   командаn
done
```

Здесь ключевые слова `do` и `done` разграничивают, соответственно, начало и конец тела цикла. Перенос строки перед `do` обязателен (вместо него можно указать ";"). Перенос строки после `do` возможен, но необязателен. Например,

```
while pidof ffmpeg >/dev/null
do sleep 10
done
```

проверяет наличие процесса `ffmpeg`[1] и, как только такого процесса в системе не окажется, цикл завершается. Дело в том, что `pgrep` возвращает 0 (истина) только в том случае, если соответствующий процесс найден, при этом вывод самого номер процесса не производится из-за перенаправления в `/dev/null`. Команда `sleep 10` останавливает исполнения сценария на 10 секунд, что позволяет избежать полной загрузки процессора постоянными проверками.

Другой пример

```
ls -1 | while read file
do echo file='$file'
done
```

организовывает цикл по всем файлам текущего каталога. Команда `read` возвращает 0, только если чтение успешно, конец файла не достигнут (в данном случае конец файла — конец стандартного потока ввода — соответствует концу стандартного потока вывода команды `ls`). Таким образом, каждая строка вывода `ls -1` будет последовательно прочитана в переменную `file` с помощью команды `read`. Такой подход корректен, в отличие от

[1] Программа конвертации аудио и видео.

вызова `for file in `ls -1``, который не сработает при наличии в файле пробелов и других специальных символов.

Существует аналогичный оператор цикла `until`, но выполняющийся до тех пор, пока условие ложно. Оба оператора являются **циклами с предусловием** (*англ.* ***precondition loop***), так как условие проверяется до начала исполнения цикла.

§3.7. Оператор выбора

Оператор выбора позволяет исполнить одну из групп команд, в зависимости от того, под какой шаблон попадает указанная строка.

`$ case` *строка* `in` *образец*$_1$`)` *команды*`;;` *образец*$_2$`)` *команды*`;;` `...` *образец*$_n$`)` *команды*`;;` `esac`

Следует обратить внимание на следующие моменты:

- каждый образец поиска должен завершаться символом “`)`”, каждый блок команд — двойной точкой с запятой, оператор завершается ключевым словом `esac`;
- строка сравнения не требует экранирования двойными кавычками — она автоматические обрабатывается как экранированная нестрогим образом;
- в образце могут использоваться символы-джокеры (“`*`”, “`?`”, “`[]`”); в частности “`*)`” будет соответствовать любой строке (всем остальным случаям);
- в одном образце можно указать несколько вариантов строки, разделив их вертикальной чертой “`|`”;
- после образца следует блок команд, завершающийся двойной точкой с запятой; оператор исполняет блок команд, соответствующий первому (слева направо, сверху вниз) подходящему образцу, если такового образца нет — ни один блок не исполняется;
- оператор сам по себе не требует перенос строк, но если в блоке имеется несколько команд, то они требует переноса строк или символа “`;`”.

Несмотря на последнее обстоятельство, рекомендуется все же делать переносы строк в операторе — каждый образец записывать в отдельную строку. Это существенно улучшит восприятие команды человеком.

Например,

```
case $file in
*.txt)
```

```
    text=1
    format=txt
*.csv)
    text=1
    format=csv
*.odt|*.ods)
    text=0
    format=office
*)
    text=0
    format=unknown
ecas
```

(соответствует примеру вышеприведенному примеру для конструкции *if/then/elif/e*
Конструкция *if/then/elif/else* нередко является более громоздкой, чем оператор выбора, но более универсальной, так как позволяет не только сравнивать строки с образцом, но и проводить проверку любых условий.

§3.8. Арифметические и логические выражения

С помощью двойных круглых скобок можно организовать вычисление арифметических выражений и проверку условий. Это — возможности оболочки *Bash*, подобные языку программирования *Cu*, но работающие только с целыми числами (положительными или отрицательными). Внутри двойных круглых скобок может быть записано выражение с использованием:

- явно заданных целых чисел;
- переменных, причем путем задания их имени, без знака `$`;
- арифметических операций “`+`” (сложение или унарный плюс), “`-`” (вычитание или унарный минус для получения противоположного числа), “`*`” (умножение), “`/`” (деление), “`%`” (взятие остатка);
- операций сравнения “`<`” (строго меньше), “`>`” (строго больше), “`<=`” (меньше, либо равно), “`>=`” (больше, либо равно), “`==`” (равенство), “`!=`” (неравенство);
- логических операций “`&&`” (логическое “И”), “`||`” (логическое “ИЛИ”) и “`!`” (логическое “НЕ”, отрицание);
- операции присваивания “`=`” для изменения значения переменной, а также операций арифметического присваивания “`+=`”, “`-=`”, “`*=`”, “`/=`”, “`%=`”, облегчающих запись выражений вида `((n=n+2))` путем замены их на `((n+=2))`;

- операций инкремента “++” и декремента “--”, соответственно увеличивающих и уменьшающих значение переменной на 1 (переменная может быть записана до или после знака операции[1]);
- круглых скобок для указания порядка действий;
- операций битовой арифметики (“&”, “|”, “^”, “~”), оперирующие с отдельными битами двоичного представления чисел (их важно не путать с похожими на них логическими операциями), и др.

Существует два способа использовать данные выражения.

Первый способ: предваряя знаком доллара. Тогда результат данного выражения будет подставлен в командную строку на место выражения. Например

```
$ n=5
$ echo $((n+2))
7
```

Второй способ — без такого предварения. Тогда выражение будет самостоятельной командой, причем если в выражении имеются присваивания, то они будут выполнены, а если имеются сравнения и их логические комбинации, то такая команда обеспечит адекватный код возврата — 0 (истина) или 1 (ложь), что позволяет использовать такие выражения в условиях и циклах.

```
$ n=5
$ ((n+=2))
$ echo $n
7
$ ((m=n*3))
$ echo $m
21
```

(до вызова присваивания переменная `m` могла быть не задана).

```
$ n=1
$ while ((n<10))
> do
> echo -n $n" "
> ((++n))
```

[1]Разница в том, какой результат вернет данная операция: `((a=n++))` запишет в `a` старое значение `n`, а `((a=++n))` — новое, увеличенное на 1, но часто такие выражения используются без присваивания: `((++n))` и `((n++))` одинаково увеличат значение `n` на 1.

```
> done
1 2 3 4 5 6 7 8 9 $
```

Последний пример — один из способов организовать цикл по индексу, хотя и не лучший. Рекомендуемый способ — сокращать количество исполняемых команд и не использовать арифметику без острой необходимости, например

```
for n in {1..9}
> do echo -n $n" ";
> done
```

будет иметь тот же эффект, но потреблять меньше ресурсов.

Bash оперирует только с целочисленными значениями. Для расчетов с плавающей точкой следует использовать внешние команды, например `bc` или `calc`.

§3.9. Файлы сценариев

Файл сценариев представляет собой текстовый файл, в котором записаны команды оболочки или иного интерпретируемого языка. Рассмотрим простейший пример.

```
echo Hello, world!
```

Это минимальная реализация программы, выводящей "`Hello, world`" на печать. В данном случае следует иметь в виду, что *echo* — не особая команда вывода текста на экран. На самом деле *стандартный вывод любой команды, исполненной в командном файле, будет выведен на стандартный вывод данного командного файла*, аналогично — для потока ошибок. То есть исполнение следующего сценария

```
cp -v * /path/to/dest
```

приведет к копированию всех файлов текущего каталога в указанный каталог, с отображением их имен на терминале (ключ "`-v`") и выводом туда же всех возникающих ошибок (например, отсутствия права на чтение исходного файла, права записи в каталог назначения, нехватки места и т.п.). За исключением некоторых специфических для сценариев конструкций, *исполнение командного файла равносильно последовательному набору его строк в командную строку.*

Если у командного файла имеется право на исполнение, то его можно запустить, просто указав его полный или относительный путь в качестве команды, а если сценарий находится в каталоге, перечисленном в PATH, то в качестве команды достаточно указать его имя. Чаще всего файлы сценария имеют расширение .sh или не имеют расширения вовсе, чтобы быть неотличимыми в плане запуска от внутренних и внешних команд, в т.ч. двоичных исполняемых файлов. Если не предпринять дополнительных действий, сценарий будет исполнен той же оболочкой, откуда произведен запуск[1]. Для какой бы оболочки ни был написан сценарий, всегда есть вероятность, что он будет запущен как команда из другой оболочки. Интерпретация сценария другой оболочкой приведет к ошибкам или неправильной работе, так как оболочки имеют различные внутренние команды и операторы. Поэтому *в сценарии необходимо указать интерпретатор, с помощью которого данный файл должен исполняться.* Делается это следующим образом:

```
#!/bin/bash

echo Hello, world!
```

Интерпретатор всегда задается первой строкой, после сочетания "#!" в виде абсолютного пути (поиск по каталогам из переменной PATH не производится), после которого могут следовать аргументы команды.

На случай запуска в системе, где каталог установки bash отличается от /bin, можно использовать

```
#!/bin/env bash

echo Hello, world!
```

так как программа env осуществляет поиск исполняемого файла по каталогам, перечисленным в переменной окружения PATH, однако привязка к расположению исполняемого файла env сохраняется[2].

Символ "#" является символом комментария: кроме некоторых случаев, все символы, следующие после "#" и до конца строки игнорируются. Исключениями являются экранированный символ "#" и использование

[1] Или оболочкой пользователя по-умолчанию, если запуск произведен не командой оболочки, а, например, из графического окружения.

[2] Впрочем, на не-*GNU* системах вероятность обнаружения bash в отличном от /usr/bin каталоге выше, чем для env.

символа “# в языковых конструкциях: в первой строке файла сценария для задания интерпретатора вышеупомянутым образом, для обработки строк внутри соответствующих выражений и т.п. Поскольку правилами хорошего тона программирования предписывается в каждой программе указывать комментарий, содержащий описание, что эта программа делает (и кто ее автор), пример превращается в

```
#!/bin/env bash

# Simple Bash "Hello, world" script

echo Hello, world
```

Сценарий также имеет код возврата, причем если не предпринять дополнительных действий, то он совпадет с кодом возврата последней команды: в примере с выводом строки “Hello, world” это будет код возврата команды *echo*. Конечно, если вывод состоится успешно, *echo* вернет 0, но в случае непредвиденных ошибок при осуществлении самой операции ввода-вывода (отказ оборудования, иные сбои), код может быть другим. Если стоит задача получить код возврата исходя из реальной причины завершения сценария, следует использовать команду *exit*, аргументом которой является возвращаемый код. Вызов команды приводит к немедленному завершению работы сценария с указанным кодом.

```
#!/bin/env bash

# Simple Bash "Hello, world" script

echo Hello, world

exit 0
```

Нулевой код возврата `exit` не равносилен отсутствию команды `exit` вообще: если выход из сценария производится по завершению файла сценария, то кодом возврата сценария становится код возврата последней исполненной команды.

Для вывода сообщения об ошибке следует перенаправить вывод на поток ошибок, например

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

```
# Simple Bash error message script

echo Error message >&2

exit 1
```

Отметим, что для программы типа *Hello-world* успехом является именно успех вывода соответствующего сообщения, поэтому следовало бы проверить код возврата команды *echo*:

```
#!/bin/env bash

# Advanced Bash Hello, world script

if echo Hello, world! 2>/dev/null
then
     exit 0
else
     echo Error printing message >&2
     exit 1
fi
```

В данном случае производится подавление собственного сообщения об ошибке команды `echo` путем перенаправления потока ошибок в `/dev/null` и выводится собственное сообщение об ошибке в поток ошибок сценария. Конечно, если поток ошибок и поток вывода сценария совпадают, вероятнее всего, что при невозможности вывести первое сообщение, сообщение об ошибке также не сможет быть выведено. Однако в общем случае поведение данного сценария не равносильно сценарию, содержащему единственную исполняемую строчку с выводом сообщения. Такие конструкции обычно не используются в ситуациях с выводом сообщения, но важны при выполнении более сложных действий и команд.

Полезный инструмент при отладке сценариев — вывод каждой исполняемой команды сценария при его исполнении. Для этого следует вызвать `bash` с опцией `-x`. Пусть сценарий “Hello, world” сохранен в файле `hw.sh`. Тогда это будет выглядеть следующим образом:

```
$ bash -x hw.sh
+ echo Hello, world
Hello, world
+ exit 0
```

Обратите внимание, что комментарии при этом не отображаются. Вывод команд осуществляется в поток ошибок. Внутри сценария также можно временно подключить и отключить отображение команд, вызвав команды "`set -x`" и "`set +x`" соответственно.

§3.10. Аргументы сценариев

Сценарий, как и любая команда и программа, получает аргументы командной строки в виде массива строк. Доступ к данным аргументам осуществляется по переменной, имя которой есть порядковый номер аргумента: "`$0`", "`$1`", "`$2`", ..."`$9`", "`${10}`", "`${11}`", ... Фигурные скобки важны для многозначных чисел, т.к. иначе вторая и последующие цифры будут восприняты не как часть номера аргумента, а как простые символы. Аргумент "`$0`" — особенный, в нем содержится имя исполняемого скрипта с полным или относительным путем, в зависимости от того, как сценарий был запущен и найден оболочкой[1] Параметр "`$0`" позволяет идентифицировать исполняемый файл сценария изнутри самого сценария, а также выводить удобные сообщения для пользователя (например, при выводе справки о формате команды сценария или при выводе сообщений от имени сценария) — лучше выводить именно значение "`$0`", которое всегда точно соответствует вызванной пользователем команде, даже если файл сценария был переименован, перемещен или одноименные сценарии встречается в нескольких каталогах. Пусть имеется сценарий `typeargs.sh`:

```
#!/bin/bash

# Simple print values of starting arguments

echo "0: \"$0\""
echo "1: \"$1\""
echo "2: \"$2\""
echo "3: \"$3\""
echo "4: \"$4\""
echo "5: \"$5\""
```

Запустим:

[1]Если сценарий находится в каталоге, указанном в переменной `PATH`, и запущен как команда, то в "`$0`" запишется путь, включающий этот каталог (так, как он указан в переменной). Если сценарий запущен как файл с указанием полного или относительного пути, то в "`$0`" запишется путь к файлу сценария так, как он был указан при вызове.

```
$ ./typeargs.sh arg1 arg2 arg3 arg4 arg5
0: "./typeargs.sh"
1: "arg1"
2: "arg2"
3: "arg3"
4: "arg4"
5: "arg5"
```

Если сценарий находится в `/bin` и этот каталог прописан в `PATH`:

```
$ typeargs.sh arg1 arg2 arg3 arg4
0: "/bin/typeargs.sh"
1: "arg1"
2: "arg2"
3: "arg3"
4: "arg4"
5: ""
```

Заметим, что пятый аргумент не был задан.

```
$ ./typeargs.sh "arg 1" arg 2 "" arg3 arg4 arg5
0: "./typeargs.sh"
1: "arg 1"
2: "arg"
3: "2"
4: ""
5: "arg3"
```

В этом примере первый аргумент включает в себя пробел, так как он был экранирован, второй и третий аргумент разделены неэкранированным пробелом, четвертый аргумент пуст но задан, шестой и седьмой аргумент сценарий не рассматривал.

Определить число аргументов можно с помощью "`$#`": данная переменная содержит номер последнего аргумента и равна 0, если аргументов нет ("`$0`" задан всегда).

Переменные "`$@`" и "`$*`" содержат, соответственно, список всех аргументов в виде отдельных элементов и сплошной строки, соответственно[1]. Для того, чтобы это сработало корректно, переменные должны быть заключены в кавычки, в противном случае обе они будут раскрыты в простую строку с неэкранированными пробелами, независимо от того являются они частью аргумента или разделителем аргументов. С помощью

[1] Всех, кроме "`$0`", то есть собственно аргументов, а не самого сценария

“$@” и цикла for можно организовать простой проход по всем аргументам сценария:

```
#!/bin/bash

# Цикл по всем аргументам

echo Одной строкой '("$*")':
for arg in "$*"
do echo "$arg"
done

echo
echo Каждый аргумент отдельно '("$@")':
for arg in "$@"
do echo "$arg"
done

# Без экранирования конструкции неразличимы:

echo
echo Конструкция '$*' без экранирования:
for arg in $*
do echo "$arg"
done

echo
echo Конструкция '$@' без экранирования:
for arg in $@
do echo "$arg"
```

```
$ ./typeargs.sh "arg 1" arg 2 "" arg3 arg4 arg5
Одной строкой ("$*"):
arg 1 arg 2  arg3 arg4 arg5

Каждый аргумент отдельно ("$@"):
arg 1
arg
2
```

```
arg3
arg4
arg5

Конструкция $* без экранирования:
arg
1
arg
2
arg3
arg4
arg5

Конструкция $@ без экранирования:
arg
1
arg
2
arg3
arg4
arg5
```

Команда `shift` осуществляет сдвиг аргументов. При этом аргумент 0 остается неизменным, аргумент 1 удаляется, аргумент 2 превращается в аргумент 1, аргумент 3 — в аргумент 2 и т.д., а счетчик количества аргументов уменьшается на 1. Таким образом, еще один способ прохода по аргументам состоит в следующем:

```
#!/bin/bash

while [[ $# > 0 ]]
do echo $1
shift
done
```

Однако этот способ менее изящен, чем предыдущий. Команда `shift` бывает полезна в случаях, когда один или несколько первых аргументов имеют специальное значение, и их нужно обработать отдельно, в то время как остальные — стандартным образом, но в большинстве случаев рекомендуется ее избегать. Отменить действие команды `shift` невозможно.

§3.11. Подстановка параметров, средства обработки строк

Подстановка параметров (*англ.* ***parameter substitution***) — набор инструментов оболочки, предназначенный для обработки строк символов, хранящихся в переменных, в т.ч. в аргументах сценариев. Подстановку параметров можно рассматривать как особое раскрытие переменных — на место соответствующей комбинации символов будет подставлен результат ее обработки. Например,

```
$ var=value
$ echo ${var:0:3}
val
```

Подстановка `${`*переменная*`:`*m*`:`*n*`}` вырезает подстроку с позиции m длиной n из переменной (следует обратить внимание на отсутствие дополнительного знака доллар перед подставляемой переменной: подстановка работает именно со строками, содержащимися в переменных и аргументах сценария). В то же время в качестве аргументов подстановки можно использовать переменные, раскрывая их стандартными образом[1]. Разумеется, результат подстановки можно использовать в любой команде, подстановка не экранируется двойными, но экранируется одиночными кавычками.

В таблице приведены способы подстановки параметров.

Синтаксис	Описание	Пример	Вывод	Замечания
Обработка строк				
`${`*var*`:`*m*`}`	Подстрока с позиции m до конца строки	`a=0123456` `echo ${a:3}`	`3456`	
`${`*var*`:`*m*`:`*n*`}`	Подстрока с позиции m длины n	`a=0123456` `echo ${a:3:2}`	`34`	
`${`*var*`/`s_1`/`s_2`}`	Замена первого вхождения подстроки s_1 на строку s_2	`a=abcabca` `echo ${a/bc/x}`	`axabca`	s_1 может содержать символы-джокеры; пустая s_2 — удаление, пуст. s_1 — возвр. исх. строку
`${`*var*`//`s_1`/`s_2`}`	Замена всех вхождений подстроки s_1 на строку s_2	`a=abcabca` `echo ${a//bc/x}`	`axaxa`	

[1]Таким образом, осуществить за один акт два преобразования одной строки не получится, необходимо использовать промежуточную переменную, по использовать результат подстановки в качестве параметра другой подстановки можно.

`${var#s}`	Удаление наиболее короткого префикса *s* из строки	`a=a.bc.def.gh` `echo ${a#*.}`	`bc.def.gh`	Наиболее длинный и наиболее короткий префикс и суффикс — при использовании символов-джокеров в *s*, при отсутствии совпадений возвращает исходную строку
`${var##s}`	Удаление наиболее длинного префикса *s* из строки	`a=a.bc.def.gh` `echo ${a##*.*}`	`gh`	
`${var%s}`	Удаление наиболее короткого суффикса *s* из строки	`a=a.bc.def.gh` `echo ${a%.*}`	`a.bc.def`	
`${var%%s}`	Удаление наиболее длинного суффикса *s* из строки	`a=a.bc.def.gh` `echo ${a%%.*}`	`a`	
`${var/#`s_1`/`s_2`}`	Замена префикса s_1 строки `$var` на s_2	`a=a.bc.def.gh` `echo ${a/#*./X}`	`Xgh`	Наиболее длинный — при использовании джокеров в s_1, пустая s_2 — удаление
`${var/%`s_1`/`s_2`}`	Замена суффикса s_1 строки `$var` на s_2	`a=a.bc.def.gh` `echo ${a/%.*/X}`	`aX`	
Раскрытие переменных				
`${var}`	То же что и `$var`	`a=0123456` `echo ${a}b`	`0123456b`	Используется для конкатенации строк и переменных
`${#var}`	Длина строки `$var`	`a=0123456` `echo ${#b}`	`7`	

`${!var}`	Значения переменной, имя которой записано в `$var`	`a=0123456`
		`b=a`
		`echo ${!b}`
			`0123456`	

`${!prefix*}`
`${!prefix@}`
	имена заданных переменных, начинающихся с *prefix*	`a=1`
		`a1=2`
		`ab=3`
		`echo ${!a*}`
			`a a1 ab`	

Проверка на заданность				

`${var-def}`
`${var:-def}`
	если значение `$var` задано, возвращается оно, иначе — *def*	`a=0`
		`b=`
		`unset c`
		`echo a=${a-def}`
		`echo b=${b-def}`
		`echo c=${c-def}`
		`echo a=${a:-def}`
		`echo b=${b:-def}`
		`echo c=${c:-def}`
			`a=0`
			`b=`
			`c=def`
			`a=0`
			`b=def`
			`c=def`
				Символ “:” разделяет случаи незаданных и заданных, но пустых переменных: при его отсутствии заданная переменная, содержащая пустую строку, считается корректной, при наличии — требующей замены (не работает для разделения пустых и отсутствующих аргументов).

`${var=def}`
`${var:=def}`
	если значение `$var` задано, возвращается оно, иначе — *def* возвращается и записывается в `$var`	`a=0`
		`unset b`
		`echo a=${a=def}`
		`echo b=${b=def}`
		`echo $a`
		`echo $b`
			`a=0`
			`b=def`
			`0`
			`def`	

`${var+alt}`
`${var:+alt}`
	если значение `$var` задано, используется *alt*, иначе — пустая строка	`a=0`
		`unset b`
		`echo a=${a+alt}`
		`echo b=${b+alt}`
			`a=alt`
			`b=`	

`${var?err}`
`${var:?err}`
	если значение `$var`, задано, используется оно, иначе сообщение *err* выводится в поток ошибок, сценарий завершается со статусом 1	`a=0`
		`unset b`
		`echo a=${a?err}`
		`echo b=${b?alt}`
			`a=0`
			`b: err`	

Еще одно средство обработки строки состоит в возможности дописать в конец значения переменной: если использовать вместо присваивания “+=”, то значение переменной будет дополнено указанной справа строкой, а не заменено им. Например:

```
$ var="value"
$ var+="123"
$ echo "$var"
value 123
```

Заметим, что эта операция строковая и равносильна

```
$ var="$var 123"
```

то есть не производит арифметических вычислений:

```
$ n=2
```

```
$ n+=2
$ echo "$n"
22
```

§3.12. Дочерние процессы самой оболочки и группировка команд

При запуске внешней команды создается процесс, внутренние команды обрабатываются текущим процессом оболочки. При создании конвейеров для каждой внешней команды создается отдельный процесс. Если в правой части конвейера оказывается внутренняя команда, для нее также создается отдельный процесс самой оболочки — "**подоболочка**" (*англ.* ***subshell***). Поскольку дочерний процесс наследует, но не передает родительскому переменные окружения, их изменения внутри дочернего процесса не скажутся на основном процессе. Например,

```
n=0

ls -1 *.txt | while read filename
do  # выполняем работу с $filename ---
    # каждым файлом из списка *.txt
    # ...
    ((++n)) # подсчитываем число обработанных файлов
done

echo $n  # всегда выведется 0 (!)
```

Также не сработает

```
$ var=1
$ echo 2 | read var
$ echo $var
1
```

Подобную ситуацию можно "разрулить" с помощью файлов, построенных из стандартного вывода — **подстановкой процессов** (*англ.* ***process substitution***)

```
n=0

while read filename
```

```
do  # выполняем работу с $filename ---
    # каждым файлом из списка *.txt
    # ...
    ((++n)) # подсчитываем число обработанных файлов
done < <(ls -1 *.txt)

echo $n  # выведется число обработанных файлов (!)
```

Для `read` такой подход также сработает:

```
$ var=1
$ read var < <(echo 2)
$ echo $var
2
```

хотя, конечно, проще было использовать

```
var="`echo 2`"
```

(для любой команды) или

```
$ var=2
```

поскольку новое значение известно на момент написания команды.

Группировка команд с помощью фигурных скобок не создает подоболочку:

```
n=1

{
    ((++n))
    echo $n # в файл выведется 2
} > out.txt

echo $n # выведется 2
```

но может привести к этому в случае конвейера:

```
n=1

echo $((n+1)) | {
    read n
    echo $n # в файл выведется 2
} > out.txt

echo $n # выведется 1
```

§3.13. Массивы

Простейший способ создания массива — явно перечислить элементы. Это делается путем задания значения переменной как массива, с перечислением элементов в круглых скобках. Элементы, как аргументы, разделяются пробелами (соответственно, для того, чтобы значение элемента могло содержать пробел или иной метасимвол, его следует экранировать). Например,

```
$ a=(One two "Hello, world")
```

создает трехэлементный массив. Доступ к элементам массива осуществляется по индексу следующим образом:

```
$ echo ${a[0]}
One
$ echo ${a[1]}
two
$ echo ${a[2]}
Hello, world
```

По умолчанию элементы массива нумеруются с нуля. Также можно задать значение отдельного элемента массива (даже если переменная `a` еще не была объявлена массивом), например

```
$ a[1]=three
$ echo ${a[1]}
three
$ echo ${a[2]}
Hello, world
```

Однако при таком подходе *значение переменной не очищается*, даже если она не была массивом, поэтому всегда следует перед первым использованием объявлять массив присваиванием полного списка аргументов. Задать пустой массив можно с помощью

```
$ a=()
```

или просто удалив значение переменной с помощью

```
$ unset a
```

Заметим, что сценарии не гарантируют того, что переменные в начале не заданы: их значения могут быть унаследованы как значения переменных окружения родительского процесса.

Также обратим внимание, что в *Bash* всякая переменная по сути массив, причем доступ к переменной без индекса равносилен доступу к нулевому элементу:

```
$ a=(1 2 3)
$ echo $a
```

```
1
$ a=5
$ echo ${a[0]}
5
$ echo ${a[1]}
2
```

После присваивания значения переменной изменился только нулевой элемент массива.

Пустой массив можно определить путем декларации переменной как массива

```
$ declare -a переменная
```

То есть переменная может быть объявлена массивом явно (с помощью `declare`), путем задания значения (первый пример) и неявно, если сразу же начать присваивать значения элементов незаданной переменной по индексу.

Для работы с массивом имеются следующие конструкции:

Конструкция	**Пояснение**	**Примечание**
`${var[@]}`	Все элементы массива раздельно (при экранировании)	аналогично `$@` для массива аргументов
`${var[*]}`	Все элементы массива одной строкой	аналогично `$*` для массива аргументов
`${#var[@]}` `${#var[*]}`	Количество элементов массива	для массива аргументов — `$#`, `${#@}`, `${#*}`; в то же время подстановка `${#var[0]}` возвращает длину строки-элемента с индексом 0
`${!var[@]}`	Все индексы массива раздельно (при экранировании)	Разница имеет значение для ассоциативных массивов (см. ниже)
`${!var[*]}`	Все индексы массива одной строкой	

Заметим следующие обстоятельства.

- Элементы массива могут использоваться в качестве переменных в

подстановке параметров, например `${a[1]//str/newstr}`.

- Если переменная "`a`" является массивом, то доступ к `$a` не позволит получить все элементы массива, в частности для копирования массива следует использовать `b=("${a[@]}")`.
- Операции с массивом аргументов сценария схожи с операциями над массивами: при передаче массива в качестве аргумента команды его также можно развернуть с помощью `"${a[@]}"`.
- В качестве индекса массива можно использовать переменную, например

```
$ i=1
$ ${a[$i]//str/newstr}
```

  и т.п.
- Массив может содержать пропуски, то есть некоторые индексы могут отсутствовать, например можно задать

```
unset a
a[1]=One
a[5]=Five
```

  В этом случае счетчик элементов массива `${#a[@]}` будет возвращать 2, элементы "`a[2]`", "`a[3]`" и "`a[4]`" остаются незаданными переменными. Также можно удалить элемент из массива (в т.ч. из середины массива) с помощью, например, `unset a[$k]`. Можно также задать массив с явным указанием индексов элементов, например:

```
$ a=([1]=One [5]=Five)
```

  Поэтому *в общем случае использование переменной-счетчика индекса для прохода всех элементов массива с 0 до количества элементов не является корректным путем обхода массива.* Следует использовать циклы по элементам и по индексам.

Добавить элемент к массиву можно с помощью "`+=`", например

```
$ a=()
...
$ a+=( el0 )
$ a+=( el1 )
```

Элементы добавляются в конец массива.

Массивы — один из способов исполнения команды, чьи аргументы сохранены в переменных (например, аргументах сценария, в файле, или

иным способом полученных от пользователя)[1]. Например,

```
$ a=()
$ a[0]=find
$ a[1]=-name
$ a[2]=*.txt
$ "${a[@]}"
```

равносильно вызову

```
$ "find" "-name" "*.txt"
```

Заметим, что использование актуальных (экранированных) кавычек в задании значения `a[2]` не требуется, так как вызов `"${a[@]}"` исключает раскрытие метасимволов в элементах массива и обеспечивает корректное разделение аргументов: нулевой элемент массива — команда, первый — первый аргумент и т.д.

Кроме индексных массивов, *Bash* поддерживает **ассоциативные массивы** (*англ.* ***associative array***) — массивы, в которых доступ к элементам осуществляется не по номерам, а по индексам, представляющим собой произвольные строки символов. Объявлять ассоциативный массив следует с помощью

```
$ declare -A var
```

После этого можно задавать элементы отдельно

```
$ var[index a]="A value"
```

или присвоить массив целиком, с обязательным указанием индексов:

```
$ var=([index a]="A value" [ixs]=5 [idx]=val2)
```

и т.п. Для ассоциативных массивов доступ к элементам, к индексам, к их количеству и т.д. осуществляется с помощью тех же операций, что и для индексных массивов.

§3.14. Функции

Функции (*англ.* ***function***) — именованные блоки команд, которые можно вызывать по их имени. В некотором смысле функции в *Bash* сами становятся (внутренними) командами — сценариями со своими аргументами, потоками ввода-вывода и кодом возврата. Задается функция следу-

[1] Еще один способ состоит в использовании команды `eval`, выполняющей свои аргументы как команду оболочки. Это позволяет избежать использования массивов, однако такой способ менее безопасен, так как добавляет дополнительный шаг раскрытия метасимволов: если в переменной, полученной от пользователя, встретится управляющая последовательность, она будет раскрыта, что обычно нежелательно. Это может привести не только к неверному результату, но и взлому системы защиты, если она зависит от сценария, использующего данную команду. Третий способ — использование `xargs` — команды, формирующей команду из строк стандартного потока ввода, то есть преобразующей строки потока ввода в аргументы команды.

ющим образом:

```
$ function name { команда_1
команда_2
...
команда_n
}
```

Фигурные скобки ограничивают тело функции. Перенос строки перед закрывающей фигурной скобкой обязателен. Также функцию можно определить следующей конструкцией:

```
$ name () { команда_1
команда_2
...
команда_n
}
```

Круглые скобки можно указать в определении и с использованием ключевого слова `function`.

Функция — способ многократного использования кода. Ее можно вызвать несколько раз (но только после определения). Вызывается функция как отдельная команда, по ее имени. В функции можно использовать аргументы (стандартным образом — `$0`, `$1`, ...), а также задать целочисленный код возврата с помощью команды `return`, завершающей работу функции, аналогично `exit` (команда `exit`, встретившаяся в функции, прервет весь сценарий, а не только функцию, если `return` отсутствует, возвращается код последней команды).

Вернуть значение из функции можно не только с помощью кода возврата (который должен быть целочисленным), но и путем печати строки в стандартный поток вывода — при вызове функции его можно перенаправить, например, в переменную.

```
max2 () {
    if [[ $# != 2 ]]
    then return 1
    fi

    (( $1 < $2 )) && echo $2 || echo $1
    return 0
}
```

Функция печатает наибольший из двух аргументов. Если число аргументов отлично от 2, функция сообщит о неудаче ненулевым кодом возврата.

```
$ a=`max2 2 1` && echo OK || echo ERROR
OK
$ echo $a
2
$ a=`max2 2 3` && echo OK || echo ERROR
OK
$ echo $a
3
$ a=`max2 2` && echo OK || echo ERROR
ERROR
```

В функциях можно использовать переменные, но в *Bash* все переменные по умолчанию являются глобальными, то есть функция может изменять переменные сценария, а переменные, созданные внутри функции, будут доступны сценарию. Однако переменные, чьи значения присвоены с помощью команды `local`, будут локальными в данной функции, например

```
local var=value
...
var=newvalue # изменение локальной переменной
echo $var # обращение к локальной переменной
```

Изменение локальной переменной не отразится на переменных вне функции. При использовании переменных для "внутренних" нужд функций, рекомендуется объявлять их локальными, во избежании нежелательного изменения значений других переменных при вызове функции.

§3.15. Общие советы по написанию сценариев

При написании сценариев следует учитывать как общие принципы написания хороших программ и хорошего программного кода, так и специфические.

- Название сценария должно соответствовать его содержанию, то есть выполняемому действию.
- Код программы должен быть хорошо структурирован и хорошо читаем. Это облегчит понимание кода и устранение ошибок в ходе написания программы, в процессе последующей ее доработки и сопровождения, а также при написании или доработке программы различными людьми. В частности, при написании условных операторов, операторов цикла и функций следует выделять команды тела оператора отсупами (стандартно — 4 пробела), разделять логиче-

ски завершенные участки кода (этапы работы программы) пустыми строками, разбивать длинный код на функции, снабжать текстовыми комментариями основные этапы, функции и нетривиальные действия.

- Сценарий должен быть достаточно массовым и универсальным: обрабатывать все возможные ситуации, в том числе неблагоприятные, и адекватно на них реагировать, выводя необходимые сообщения об ошибках в соответствующий поток, содержащие информацию об ошибке и подсказке о способе ее исправления. К неблагоприятным ситуациям относятся проблемы со стороны пользователя (неправильный формат вызова, неверные опции и т.п.), ситуации в файловой системе (отсутствие доступа, ошибки чтения и записи) и другие проблемы. При ошибке следует выходить из сценария с ненулевым кодом возврата, при успехе — с нулевым.
- Рекомендуется следовать соглашению *POSIX* об аргументах и опциях (команда может иметь короткие и длинные опции, начинающиеся соответственно с одиночного или двойного дефиса, а также опции со значениями; пользователь имеет возможность указывать опции в любом порядке, чередовать опции и аргументы, не являющимися опциями; команда должна поддерживать двойной дефис как разделитель опций и параметров и т.д.).
- Сценарий должен иметь опцию для вывода справки (`--help`), содержащую формат вызова команды, поддерживаемые опции, информацию об авторстве программы и т.д., рекомендуется поддерживать и опцию (`--version`), выводящую информацию о версии и авторстве программы, а также опцию (`--usage`), выводящую краткую (одной строкой) информацию о формате вызова (она также может служить стандартной подсказкой при ошибке в параметрах и опциях).
- Следует учитывать, что имена файлов могут содержать пробелы и другие метасимволы, а также начинаться с дефиса, что требует обязательного экранирования соответствующих переменных и аргументов, а также использования разделителя имен файлов и опций.
- Для большей переносимости сценариев рекомендуется учесть, что внешние команды могут находиться в каталогах, отличных от `/bin`, а то и вообще в каталогах, не перечисленных в `PATH`. Также в системе может находиться несколько различных версий одной и той же команды. Поэтому вместо прямого вызова внешней команды следует задать ее в переменной в начале кода сценария, а затем вызывать по данной переменной. Это позволит легко скорректировать сценарий

при переносе на другую систему. Например, вместо

```
cp -- "$source" "$dest"
```

следует указать вначале

```
CP=/usr/bin/cp
```

и затем вызывать

```
$CP -- "$source" "$dest"
```

При изменении ситуации в ОС достаточно будет исправить команды в начале сценария, а не в каждой строке, где они вызываются.

Использование абсолютных путей также позволит избежать проблем при наличии псевдонимов, одноименных стандартным командам.

Отметим, что использование заглавных букв для имен переменных окружения, а также для имен переменных, содержащих вызываемые команды, а строчных букв — для рабочих переменных сценария, является типичным подходом, которому следует придерживаться.

- Следует учитывать, что сценарии оболочки исполняются сравнительно медленно (по сравнению с программами, написанными на таких компилируемых языках, как *Cu*, и даже по сравнению с многими другими интерпретируемыми языками). Сценарии предназначены для быстрого решения небольших задач, ориентированных в основном на массовую обработку и синтаксический анализ файлов. Не следует использовать сценарии для больших объемов вычислений и обработки больших объемов данных — для этого разумнее выбирать соответствующие языки программирования. По возможности следует отдавать предпочтение программам и иным готовым решениям, например использовать *grep*, *sed* и *awk* вместо построчного анализа файлов средствами оболочки. Для оптимизации может быть полезно сокращение создания дочерних процессов там, где в этом нет необходимости (например, использовать файлы-аргументы и подстановку процессов вместо конвейеров).
- Следует избегать традиционного для вычислительных задач и языков программирования подхода — использования арифметики, массивов, индексов и т.п., отдавая предпочтения анализу потоков данных (вывода команд, файлов и др.) и строк с помощью перенаправлений, конвейеров и фильтров во внешние команды или циклы построчного чтения. По возможности эти потоки и массив аргументов следует обрабатывать “на лету”, без сохранения промежуточных результатов в файлы и, особенно, в строковые переменные и в массивы.

Бессмысленно сохранять вывод программы в переменную лишь для того, чтобы следом вывести ее с помощью `echo` или обработать в цикле.

- При использовании значений переменных в сценарии их следует предварительно инициализировать (присваиванием) либо выполнить команду `unset`: сценарий наследует переменные окружения родительского процесса, поэтому не гарантируется, что все переменные изначально пусты (не заданы)[1]. Особенно это касается массивов: не следует начинать с присваивания в элементы массива без очистки самого массива.

- Без острой необходимости следует избегать интерактивных сценариев, то есть сценариев, запрашивающих от пользователя ответов на вопросы и данные для обработки (имена файлов, выполняемые действия, способы обработки и т.п.) — если ответы на вопросы можно получить с помощью аргументов командной строки, следует получить их именно таким образом, тем самым достигнув большей гибкости при автоматизации исполнения сценария и удобства при последовательных вызовов с помощью истории команд (как для вызова того же сценария с несколько отличными параметрами, так и для исправления неправильного ввода). В случае, когда интерактивные запросы со стороны сценария, например, на подтверждение тех или иных действий, желательны, рекомендуется предоставить возможность отключить их соответствующей опцией, автоматически предполагая ответы “ДА” на все запросы.

- Размер командной строки — имя команды со всеми ее аргументами — имеет ограничение, причем устанавливаемое не оболочкой, а операционной системой. Поэтому следует избегать построения огромных команд, например путем передачи большого списка файлов в качестве списка аргументов[2]. Это еще одна причина, почему конструкция вида

```
find -name "*.txt" | while read file
do cat "$file"
done
```

[1] В то же время использование отдельных переменных в качестве средства управления сценарием (подобно тому как, например, переменная `EDITOR` влияет на такие команды, как `visudo`)— нормальная, хотя и нехарактерная для сценариев, практика. Главное, чтобы эти переменные и их смысл был четко документирован.

[2] Это касается и команды `xargs`.

предпочтительнее, чем

`cat \`find -name "*.txt"\`` (Помимо того, что второй вариант не будет работать, если в именах файлов или путях содержатся пробелы, без специального действия по отключению их интерпретации как символа-разделителя).

- Заметим, что еще одним тонким моментом является риск наличия в имени файлов даже таких спецсимволов, как символ перевода строки. Поэтому, в общем случае вышеуказанная конструкция может не сработать. Наиболее универсальным подходом является

```
find -name "*.txt" -print0 | while read -r -d $'\0' file
do cat "$file"
done
```

Здесь опция `-print0` сообщает `find` использовать символ с кодом 0, а не символ перевода строки в качестве разделится строк — единственный символ, запрещенный к использованию в именах файлах (кроме разделителя путей "/"). Опция `-d $'\0'` сообщает `read` считать именно данный символ концом ввода, а `-r` необходим для исключения обработки каких-либо иных символов входного потока как служебных[1].

Из практических советов приведем следующие.

- Поскольку команды интерпретируемого языка можно исполнять непосредственно в командной строке, перед включением в файл сценария сложной (неочевидной) команды можно протестировать ее поведение и построить правильный синтаксис интерактивным образом.
- При решении сложной задачи рекомендуется начать с минимального работающего сценария и затем уже добавлять в него необходимые возможности и расширения поэтапно, тестируя и добиваясь работоспособности на каждом этапе.
- Повторяющийся с точностью до значений отдельных параметров код следует оформлять в виде функций.
- Тестировать сценарий и разобраться с его поведением можно добавляя тестовый вывод (с помощью команды `echo`) значений переменных и проверки факта прохождения по проблемному участку кода,

[1]Строки, завершаемые символом с кодом 0, называются **нуль-терминированными** (*англ.* ***null-terminated strings***). Опцию для обработки нуль-терминированных строк имеет и команда `xargs`.

телу цикла, условия, функции и т.п. Данный тестовый вывод следует удалить перед публикацией (началом использования) сценария.

Вопросы для самопроверки.

1. *Что такое код возврата процесса?*
2. *Как определить код возврата команды?*
3. *Какие коды возврата интерпретируются как успех (истина), а какие как неудача (ложь)?*
4. *Каким образом осуществляется проверка условий в сценариях оболочки?*
5. *Каким образом осуществляется вычисление значения арифметического выражения?*
6. *Приведите синтаксис условного оператора.*
7. *Опишите синтаксис операторов цикла.*
8. *Как задается интерпретатор сценария оболочки?*
9. *Как осуществляется доступ к аргументам сценария оболочки?*
10. *Как задается код возврата сценария оболочки?*
11. *Что такое дочерний процесс оболочки и какие проблемы для переменных сценария возможны при их возникновении?*
12. *Что такое подстановка параметров (какие имеются средства обработки строк символов)?*
13. *Как осуществляется работа с массивами?*
14. *Как задаются и вызываются функции?*
15. *Почему важно инициализировать значения переменных в сценариях?*
16. *Каким образом можно получить справку по внутренним и внешним командам оболочки?*

Приложение А. Философия *Unix*

В литературе и в фольклоре нередко можно встретить словосочетание “философия *Unix*”, хотя при создании ОС *Unix* никакой философии, легшей в ее основу, не было официально оглашено и опубликовано. Поэтому у разных авторов можно встретить разные списки принципов под этим заголовком. Выделим несколько ключевых.

- *Сделай что-то одно, но сделай это хорошо* — предполагает создание небольших программ, решающих какую-то одну задачу и взаимодействующих между собой по требованию пользователя (например, с помощью конвейеров).
- *Всё есть файл* — устройства и другие элементы ОС и работы ядра отображаются как файлы виртуальной ФС.
- *Молчание и минимум вопросов* — программа должна работать автоматически, исходя из данных аргументов, а не диалогово (например, *grep* и *find* сразу ищут файлы, а не задают вопросы какие файлы, где и как искать). Если программе “нечего сказать”, то программа ничего не сообщает (например, при успешном выполнении действия).
- *Предсказуемость, согласованность и наименьшая неожиданность* — поведение программ должно максимально соответствовать ожиданиям пользователей. Это касается как отдельных программ, так и системы в целом (например, следование единому соглашению о формате аргументов и опций). Также, если, например, команда `ls *.tmp` выводит некоторый список файлов, то команда `rm *.tmp` должна удалить те же самые файлы.
- *Программы должны взаимодействовать между собой.* В частности, вывод любой команды может использоваться в качестве ввода другой команды. Следует отдавать предпочтение текстовому формату для хранения, ввода и вывода данных, так как он универсален.
- *Система не должна защищать пользователя от глупостей*, так как заодно можно запретить делать умные вещи. Конечно, из-за этого пользователю следует быть внимательным и осторожным при работе, особенно из административного аккаунта[1].

[1]Классическая шутка на этот счет — вызов `rm -rf /` из административного аккаунта, хотя и вызов `rm -rf ~` от имени рядового пользователя принесет много неприятностей. В любом случае настоятельно рекомендуется регулярно создавать аварийные копии данных и настройки ОС.

Приложение В. Об установке ОС

§В.1. О дистибутивах *Linux*

Одним из наиболее известных дистрибутивов, ориентированных на корпоративное использование, является *Red Hat Enterprise Linux*. Данный дистрибутив является платным (несмотря на то, что в нем используется бесплатное ПО, фирма-производитель берет плату за техническую поддержку) и ориентирован на сервера и предприятия, а не на персональное использование. Существует несколько бесплатных клонов *Red Hat* (например *Scientific Linux*, *CentOS*) содержащих точно то же ПО, но предоставляемых бесплатно и без поддержки.

Самой компанией *Red Hat* выпускается свободный и бесплатный дистрибутив *Fedora*, ориентированный на персональное использование. В отличие от *Red Hat* в *Fedora* используются более новое, содержащее больше возможностей, но менее проверенное на стабильность и качество ПО. В этом смысле данный дистрибутив является площадкой для отбора качественных пакетов в *Red Hat*, однако стандартная сборка *Fedora* не является тестовой (тестовые версии ПО также поставляются в *Fedora* отдельным образом). Поскольку политика дистрибутива *Fedora* состоит в использовании исключительно свободного ПО, не связанного с патентными проблемами, некоторые возможности могут отсутствовать в нем (например, проигрывание и создание некоторых форматов видео и аудио), но могут быть установлены из сторонних источников[1].

Одним из наиболее популярных дистрибутивов *Linux* являются *Ubuntu*, позиционируемый как ориентированный на неквалифицированного пользователя, и *Debian*, пригодный как для персонального использования, так и для серверов.

Также следует отметить такие дистрибутивы как *SUSE Linux Enterprise Desktop*, *OpenSuse*, *Arch Linux*.

Особое место занимают *Slackware* и *Gentoo*, ориентированные на самостоятельный подбор конфигурации и ПО, глубокое изучение и знание *GNU/Linux* изнутри. Первый является исторически первым дистрибутивом среди выпускаемых в настоящее время, второй также предполагает самостоятельную компиляцию ядра и практически всех приложений из исходного кода с адаптацией и оптимизацией для целевого компьютера с расчетом на большую эффективность получаемого машинного кода.

[1] Такая же ситуация с *Red Hat* и его клонами, но установка ПО в *Red Hat* из сторонних источников является нарушением лицензионного соглашения и поддержка установленной ОС в этой ситуации не гарантируется.

ОС семейства *BSD* (*FreeBSD*, *NetBSD*, *OpenBSD*, *ClosedBSD* и др.) являются отдельными *Unix*-подобными ОС, имеющие собственное ядро и не основанные на *GNU* напрямую, хотя и поддерживают большое количество приложений *GNU* и других, изначально ориентированных на *Linux*.

§B.2. О *X-сервере* и графических окружениях рабочего стола

Графический интерфейс в *Unix*-подобных системах реализуется посредством ***X-сервера*** (*англ.* ***X-server****, также известный как* ***Xorg****,* ***X11*** *и др.*), обеспечивающего взаимодействие с оборудованием (экраном, клавиатурой, мышью и др.), отображение курсора мыши, границ окон и др. Поверх *X-сервера* работает **менеджер окон** (*англ.* ***windows manager***), ответственный за заголовки и рамки окон, их перемещение, горячие клавиши и др., как правило предоставляемых графическими окружениями рабочего стола.

Графическое **окружение (среда) рабочего стола** (*англ.* ***desktop environment, DE***) — программное обеспечение, ответственное за отображение главного меню, панелей, рабочих столов, запуск приложений и даже работу с оборудованием (автомонтирование сменных носителей, завершение работы системы и др.). Совместно с графическим окружением нередко поставляется набор программ для работы (например браузеры, просмотрщики изображений и видео, офисные пакеты и др.), однако чаще всего ничто не мешает запускать приложения от одного окружения на другом. Практически всякое графическое окружение идет со своим эмулятором терминала, хотя существуют и сторонние эмуляторы, например, работающий чисто поверх *X-сервера xterm* и др.

От выбора графического окружения сильно зависит графический пользовательский интерфейс. Следует выделить несколько графических окружений:

- *Gnome* — среда, разрабатываемая в рамках *GNU*, и используемая в качестве среды по умолчанию во многих дистрибутивах. Начиная с версии 3 был сильно изменен пользовательский интерфейс в сторону простоты и ориентированности на современные реалии.
- *MATE* — независимое продолжение развития среды *Gnome* версии 2 с классическим интерфейсом менее требовательное к ресурсам.
- *KDE* — очень богатая возможностями, но ресурсоемкая среда.
- *Trinity* — независимое продолжение развития среды *KDE* версии 3 с более классическим интерфейсом.

- *Xfce* — экономичное окружение рабочего стола.
- *LXDE* — одно из наименее требовательных к ресурсом окружений рабочего стола.
- Можно также отметить *ROX Desktop*, *Cinnamon*, *LXQt* и др.

Графические окружения рабочего стола и графические приложения в большинстве случаев работают с использованием графических библиотек, ответственных за отображение элементов окна — меню, кнопок, полей ввода и др. Наиболее известные из них — *GTK* (от *GNU*) и *Qt*. *Gnome*, *MATE*, *Xfce* и др. работают на *GTK*, *KDE*, *Trinity*, *LXQt* — на *Qt*.

§B.3. Об установке на компьютер

Установку ОС следует проводить в соответствии с инструкцией к соответствующему дистрибутиву. Точнее даже, к соответствующей версии, так как этот процесс может меняться. Любая информация быстро устаревает и не может быть изложена в пособии.

Установка на компьютер, как правило, предполагает создание загрузочного носителя (компакт-диска, USB-флеш-накопителя, внешнего HDD или иного) в виде загрузочного *Live* ("живого") образа — установленной ОС, позволяющей работать в ней и установить на компьютер после загрузки, или чисто установочного образа, позволяющего только установить ОС.

При установке на компьютер в качестве управляющей ОС следует проследить, чтобы при создании дисковых разделов не были уничтожены существующие разделы другой ОС или данных, поэтому следует заранее подготовить свободное (неразмеченное) место на носителе или отдельный свободный носитель. Неразмеченное пространство можно подготовить из *Live*-образа (с помощью *GParted* с GUI или средствами CLI — *fdisk* и специфическими для конкретных ОС), но при наличии другой ОС рекомендуется сделать ее средствами во избежании проблем с загрузкой. Разбиение на разделы устанавливаемой ОС лучше производить средствами установщика.

Установщик обычно находит другие установленные ОС и конфигурирует загрузчик на вывод меню выбора ОС на стадии загрузки компьютера (режим *dual boot*, "двойной загрузки"). Следует отметить, что в процессе обновления *Linux* в систему могут быть установлено несколько версий ядра, выбор конкретного ядра также осуществляется при загрузке: если новое ядро по каким-то причинам не работает, можно загрузиться в старое. Также обычно устанавливается вариант загрузки в режим "спасения"

— *rescue* — загружающий систему в минимальное окружение командной строки на случай устранения проблем администратором в ситуации невозможности нормальной загрузки ОС.

§B.4. Об использовании виртуальной машины

В случае, если ОС не предполагается использовать в качестве основной, а также на начальной стадии знакомства с ней, бывает неудобно и рискованно устанавливать ее непосредственно на компьютер. Первое связано с тем, что для переключения ОС необходимо тратить много времени на перезагрузку компьютера, второе — из-за рисков повредить данные и другую ОС неумелыми действиями.

Выходом из ситуации является использование системных **виртуальных машин** (*англ.* ***virtual machine***) — программно созданных виртуальных компьютеров, в которых можно полноценно запустить другую ОС как процесс текущей ОС, например, запустить ОС на базе *GNU/Linux* в окне ОС *Windows*, или наоборот. Запущенная на виртуальной машине ОС называется **гостевой** (*англ.* ***guest***) ОС, а ОС, в которой работает виртуальная машина называется ОС **хоста** (*англ.* ***host***)[1].

Для создания виртуальных машин следует использовать соответствующее ПО, например *VirtualBox*, *QEMU* (свободные), *VMWare Workstation* (проприетарное) и другие. При этом следует учесть следующие четыре момента.

- Для установки гостевой ОС следует создать виртуальный накопитель — лучше всего его создать виде файла ОС хоста, динамически расширяющегося по мере добавления данных. Таким образом, можно, не заботясь о преждевременном перерасходе свободного места на компьютере-хосте, создать большой виртуальный накопитель для гостевой ОС.
- В гостевую ОС обязательно необходимо установить дополнения гостевой ОС, предоставляемые программой, обеспечивающей работу виртуальной машины. Это драйверы виртуального оборудования, взаимодействия гостевой ОС и ОС хоста и др. Для их установки может потребоваться установка средств разработки на *Cu* и других языках (в связи с тем, что драйвера — модули ядра — должны собираться для каждого ядра индивидуально).

[1] *Host* — хозяин, принимающий гостей, организатор мероприятия. Однако такой перевод в русском языке не используется, возможно потому, что “хозяин” может иметь и другие значения: *master* — хозяин по отношению к слуге, *owner* — собственник, и др.

- Для обмена данными между гостевой ОС и ОС хоста следует использовать общие папки, настраиваемыми в соответствии с документацией к ПО виртуальной машины.
- Виртуальные машины поддерживают **снимки** (*англ.* ***snapshot***) состояния с возможностью откатить изменения к более раннему состоянию (например, к последней сохраненной удачной конфигурации после неудачного эксперимента) и переключения между снимками — преимущество виртуальных машин, недоступное на "реальных".

Следует также учесть, что все ОС на виртуальной машине будут работать несколько медленнее, чем непосредственно на оборудовании, и вообще требовать несколько больше аппаратных ресурсов.

§B.5. О разбиении носителя на разделы

При установке рекомендуется создать отдельный раздел для загрузчика (`/boot`) для того, чтобы иметь возможность более гибкого выбора ФС корневого раздела, структуры разбиения или шифрования, так как загрузкик может не поддерживать некоторые из них.

Разумно создать отдельный раздел для домашних каталогов `/home`, так как это позволит избежать проблем при переустановке или смене дистрибутива или версии дистрибутива, а также позволяет легко организовать шифрование содержимого домашних каталогов.

Желательно создание раздела подкачки, `swap`. Этот раздел предназначен не для хранения фалйлов, (хотя *swap* условно называется файловой системой раздела), а для хранения данных из оперативной памяти при нехватки физической памяти. Даже если памяти в компьютере достаточно, раздел *swap* — наиболее удобный способ организовать возможность перехода компьютера в режим сна (*hibernate*), поэтому рекомендуется иметь раздел *swap*, как минимум вдвое превышающий размер оперативной памяти.

Если установщик позволяет, рекомендуется также использовать LVM — систему логических (виртуальных) разделов, размещаемых на одном разделе. Поскольку заранее предсказать, сколько места понадобится под тот или иной раздел сложно, бывает полезно отвести под них минимальный объем, а затем расширять каждый из них отдельно по мере необходимости. Сделать это с реальными разделами сложно, в то время как LVM позволяет провести подобные операции без особых затрат.

§B.6. Об установке и удалении программ

Программы для большинства дистрибутивов предоставляются в виде установочных **пакетов** (*англ.* ***package***), формат которых для разных дистрибутивов различен: *rpm* для *Red Hat*, *Fedora*, *Suse* и др., *deb* для *Debian*, *Ubuntu* и др.

Работа с пакетами осуществляется с помощью программы управления пакетами, который также отличается для разных форматов пакетов (например, *rpm* и *deb* соответственно). Часто использования менеджера пакетов для установки одиночного пакета для установки программы недостаточно, так как требуется установка **зависимых пакетов** (*англ.* ***dependency***) — программ, без которой работа данной программы невозможна (например, разделяемых библиотек).

Практически каждый дистрибутив имеет собственные **репозитории** (*англ.* ***repository***) — коллекции ПО, доступные в сети Интернет. Также существует возможность подключения сторонних репозиториев. Для работы с репозиториями используется **менеджер пакетов** (*англ.* ***package manager***), различный для различных дистрибутивов (*dnf* для *Fedora*, *dpkg* для *Debian*, *aptitude* для *Ubintu* и др.). Менеджер пакетов позволяет автоматически найти необходимые пакеты в репозиториях, скачать и установить их.

Существуют также различные графические приложения для установки и удаления программ. Изучение их и менеджеров пакетов возможно по документации к ним и соответствующему дистрибутиву.

Приложение С. Дополнительные сведения об оболочке *Bash* и других командах

§C.1. Подключение другого файла сценария

При написании больших сценариев бывает полезно их разбиение на части. Также бывает полезно использование одинаковых функций, псевдонимов и т.п. значений в нескольких сценариях. Тогда их полезно выделить в один общий (разделяемый между сценариями) файл.

Для подключения определений, сделанных в другом файле сценария, нельзя просто исполнить данный сценарий как команду (если он исполняемый) и, тем более, вызвав через интерпретатор: в таком случае для его исполнения создается отдельный процесс оболочки и все переменные,

функции, псевдонимы и другие определения не будут переданы в родительский процесс, то есть в исходный сценарий.

Для осуществления желаемого следует использовать команду “точка” (“.”) или ее *Bash*-аналог, команду `source`. Она приводит к исполнению команд указанного файла в текущем процессе. Например,

```
. script.sh
```

Код возврата команды соответствует коду возврата указанного файла. Команда вернет ошибку, если файл не удалось прочитать. Наличие права исполнения файла значения не имеет.

§C.2. Псевдонимы

Псевдонимы (*англ.* ***alias***) — команды, раскрываемые оболочкой *Bash* в другую команду (возможно с аргументами)[1]. Могут использоваться для сокращения длинных команд или последовательностей, например можно задать псевдоним `ll` для вызова “`ls -l`”, псевдоним `ma` — для вызова

“`mount /dev/sda /mnt/sda`”

и т.п.; применения “более безопасных” режимов работы команд типа `rm` по умолчанию, например задание псевдонима `rm` для вызова “`rm -i`”, приведет к тому, что *rm* будет запрашивать удаление каждого файла, если не указан ключ `-f` и др.

Псевдонимы задаются с помощью команды `alias`, с ее же помощью можно просмотреть значения всех или заданного псевдонима. Удаление псевдонима — с помощью команды `unalias`. Например,

```
$ alias rm="rm -i"
$ alias rm
alias rm='rm -i'
$ unalias rm
```

Псевдонимы удаляются при выходе из оболочки. Задать “постоянные” псевдонимы можно с помощью настроечных файлов оболочки.

§C.3. Стек каталогов

Команда `pushd` позволяет сменить текущий каталог, аналогично команде `cd`, но при этом каталог, из которого произошел переход сохраняется в **стеке каталогов** (*англ.* ***directory stack***). Команда `popd` позволяет

[1] Строго говоря, в любую последовательность символов

вернуться в последний каталог, из которого был осуществлен переход[1]. Команда `dirs` позволяет просмотреть текущее состояние стека каталогов. Например

```
$ pushd /tmp
/tmp ~
```
(работаем каталоге `/tmp`)
```
$ popd
~
```
(вернулись в исходный (домашний) каталог)
```
$ pushd /tmp
/tmp ~
```
(работаем каталоге `/tmp`)
```
$ pushd /usr/bin
```
(работаем каталоге `/usr/bin`)
```
/usr/bin /tmp ~
$ dirs
/usr/bin /tmp ~
$ popd
/tmp ~
```
(вернулись в каталог `/tmp`)
```
$ pushd /lib
/lib /tmp ~
```
(работаем каталоге `/lib`)
```
$ popd
/tmp ~
```
(вернулись в каталог `/tmp`)
```
$ popd
~
```
(вернулись в исходный каталог)

С помощью метасимвола "`~`" возможно получение каталогов из стека по номеру. Например, если стек имеет вид

```
$ dirs -v
0 /usr/bin
1 /opt/usr/bin
2 /tmp
```

[1]Стек — список элементов, в который элементы можно добавлять и удалять, причем удаляется тот элемент, который был добавлен последним (среди оставшихся), то есть удаление происходит в обратном добавлению порядке. Используется также аббревиатура *LIFO — Last In, First Out* — последним пришел, первым вышел. Стек — противоположность очереди, при которой первым "выходит" первый пришедший элемент.

```
3 /var
4 ~
```

то `~2` и `~+2` будет ссылаться на каталог `/tmp`, то есть если номер положительный, то нумерация идет с вершины стека (с последнего добавленного каталога), а `~-2` будет ссылаться на `/var`, то есть отрицательные номера отсчитываются с первого добавленного в стек каталога.

§C.4. Форматированный вывод

Команда `printf` позволяет выводить значения данных с учетом формата, в т.ч. выводить символы по коду, регулировать количество полей для чисел с плавающей точкой и т.д. в стиле языка *Cu*:

```
$ printf формат данные...
```

выводит *данные* в соответствии со строкой формата. Например

```
$ Pi=3.14159265358979
$ N=10005
$ msg="Hello, world"
$ printf "Pi = %1.2f\n" $Pi
Pi = 3.14
$ printf "Pi = %1.5f\n" $Pi
Pi = 3.14159
$ printf "N =\t%d\n" $N
N =    10005
$ printf "Pi =\t%1.5f\n" $Pi
Pi =    3.14159
$ printf "\"%s\"\n" $msg
"Hello,"
"world"
$ printf "\"%s\"" $msg
"Hello,""world"$ printf "\"%s\"\n" "$msg"
"Hello, world"
```

§C.5. Задание специальных символов

Комбинация `$'\`*код*`'` позволяет задавать специальные символы, например `$'\t'` — символ табуляции, `$'\n'` — символ перевода строки, `$'\123'` — символ с восьмеричным кодом 123, `$'\x20'` — символ с шестнадцатеричным кодом 20 и т.д. Данные последовательности символов будут раскрыты в соответствие с правилами раскрытия метасимвола `$` и переданы в команду, хотя их корректная обработка самой командой не всегда

гарантируется.

Следует отметить, что файловая система *Linux* (*ext4*, *btrfs*) запрещают использовать символ с кодом 0 и символ "/" в качестве символа имени файла, но отображение многих символов может быть служебным или специфическим, поэтому создавать файлы, имена которых содержат символы, которые нельзя ввести с клавиатуры, следует с осторожностью.

Задание служебных символов нужно, скорее, для обработки содержимого текстовых файлов или форматирования вывода, хотя последнее лучше делать с помощью `printf`, а не `echo`.

§C.6. Настройки оболочки

Конфигурация некоторых опций *Bash* осуществляется заданием значений служебных переменных. Например, переменная *IFS* содержит список разделителей полей, т.е. символов, разделяющих аргументы. По умолчанию это символы пробела, табуляции и переноса строки. Переменная *PS1* позволяет настроить форму приглашения командной строки. В документации можно найти информацию о том, какие комбинации символов в значении этой переменной соответствуют каким отображаемым элементам (имени пользователя, хоста, текущего каталога, даты и др.)

Команда `shopt` также позволяет просматривать и задавать опции оболочки. Например,

```
$ shopt -s dotglob
```

устанавливает опцию `dotglob`, при которой файлы, имена которых начинаются с точки, будут попадать под шаблон, начинающийся с `*` и `?`. Ключ `-u` снимает (выключает) опцию, т.е. для данной опции возвращает поведение по умолчанию

Постоянная конфигурация и персонализация оболочки хранится в файлах настроек — **профилях** (*англ.* ***profile***). Эти файлы представляют собой обычные сценарии оболочки: в них можно исполнять команды, задавать значения переменных (например, `PATH`), псевдонимов, функций и т.д. Они исполняются так, как будто подгружены с помощью команды ". ". Существуют общесистемные профили, хранимые в `/etc`, и индивидуальные профили, хранимые в домашнем каталоге пользователя `~`.

В зависимости от того, как запущена оболочка, применяются разные файлы профилей: оболочка может представлять собой

- **оболочку входа** (*англ.* ***login shell***) — например, при входе в систему программой *login*, удаленно, или с помощью `su`, `sudo` с ключом `-i` или `-l`, явный запуск оболочки с опцией `--login` и др.;
- **интерактивную оболочку** (*англ.* ***interactive shell***), т.е. работа в

которой ведется путем ввода команд — например, при запуске `bash` без опций или запуске с опцией `-i`;

- не являться ни той, ни той, т.е. работать в **пакетном режиме** (*англ.* ***batch shell***) — исполнять команды, находящиеся в файле сценария.

Оболочка входа сначала запускает `/etc/profile`, если этот файл существует, затем проверяет наличие файлов `~/.bash_profile`, `~/.bash_login` и `~/.profile` и запускает первый существующий и только его. Интерактивная оболочка запускает `/etc/bashrc` и `~/.bashrc`, если они существуют. Пакетная оболочка дополнительных файлов не подгружает. При этом оболочка наследует конфигурацию от родительких оболочек, например интерактивная и пакетная оболочки наследуют конфигурацию от оболочки входа. Пакетная также наследует конфигурацию интерактивой оболочки, если запущена из нее[1].

Следует отметить, что *POSIX Shell* использует только файлы профилей `/etc/profile` и `~/.profile`, поэтому конфигурация при ее запуске может оказаться отличной, т.к. *Bash*, при наличии `~/.bash_profile` проигнорирует `~/.profile`.

В файле `~/.bash_history` сохраняется история команд *Bash*.

§C.7. Работа с архивами

Следует отличать понятия "**архивация**" и "**сжатие**" ("**упаковка**", "**компрессия**") файлов (*англ.* ***archiving*** *и* ***compressing*** *соответственно*). Первое представляет собой возможность помещения нескольких файлов и каталогов (с сохранением структуры, атрибутов и т.д.) в один файл, второе — преобразование данных с целью сокращения занимаемого их объема, обычно, в контексте архивации, осуществляемое без потерь. Хотя большинство графических приложений для работы с архивами совмещают обе эти функции, на уровне командной строки эти задачи могут решаться разными программами.

Архивация осуществляется такими программами программами, как *tar* (*GNU*), *pax* (*POSIX*), *cpio* и др. Сжатие может быть осуществлено с помощью программ *gzip* (быстрое, но менее эффективное по уровню сжатия), *bzip2*, *xz* (более эффективное сжатие) и т.д., а распаковка, соответственно, с помощью команд *gunzip*, *bunzip2*, *unxz* и др. При вызове без аргументов программы компрессии и распаковки работают как фильтры, при задании файла в качестве параметра данный файл замещается

[1] В частности, это означает, что сценарии, запущенные демоном `cron`, не наследуют конфигурацию интерактивной оболочки, а только конфигурацию оболочки входа, так как пакетная оболочка запускается из оболочки входа напрямую.

архивом.

Например, для создания xz-архива каталога можно использовать

```
tar -c dir | xz >dir.tar.xz
```

для распаковки

```
unxz <dir.tar.xz | tar -x
```

В обоих случаях исходные файлы и архивы остаются нетронутыми, но файлы назначения могут быть перезаписаны или изменены при совпадении имен.

Использование архиваторов типа *RAR*, *Zip*, *7z* и других также возможно, но для сохранения файлов *Unix*-подобных систем не рекомендуется, так как такие архиваторы могут не сохранять информацию о владельцах, правах доступа (в т.ч. права на исполнение) и т.п.

Особое место среди программ архивации и сжатия данных занимает программа `mksquashfs`, позволяющая создать сжатый образ каталога, который можно стандартным образом подмонтировать как доступную только для чтению файловую систему. При небольшой потере эффективности сжатия такая файловая система будет работать достаточно быстро — с потерей времени только на распаковку, но не на поиск данных в архиве.

§C.8. Программы sed и awk

Программа *sed* представляет собой *неинтерактивный текстовый редактор*. По сути это интерпретатор языка программирования, ориентированного на автоматическую обработку (преобразование) текстовых файлов.

Команды языка *sed* состоят из блока выбора обрабатываемых строк (при отсутствии данного блока обрабатываются все строки файла), команды и ее аргументов. Команды разделяются точкой с запятой или переносом строки. Команды могут задаваться в отдельном файле или могут быть переданы в качестве первого аргумента командной строки. При этом, если не указано других аргументов, программа *sed* будет работать как фильтр, также можно задать имя обрабатываемого файла с выводом результата редактирования на стандартный вывод, или его замены на новый файл с помощью опции "`-i`" (обработка файла "на месте", а не в режиме фильтра).

Одна из наиболее "популярных" команд — команда подстановки (замены) "`s`". У нее два аргумента — исходный шаблон поиска (регулярное

выражение) и строка замены. Аргументы ограничиваются символом слеша "/". Если после последнего слеша указать букву "g" замена будет глобальной, иначе меняется только первое вхождение в каждой строке. Рассмотрим несколько примеров этой и других команд

команда	описание
`sed 's/Win/Linux/g'`	заменяет слово `Win` на `Linux` глобально во всем документе
`sed '5,7d'`	удаляет строки с 5 по 7
`sed '10,15s/Win/Linux/'`	заменяет первое вхождение слова `Win` на `Linux` в строках с 10 по 15
`sed '/OS/s/Win/Linux/g'`	заменяет все вхождения слова `Win` на `Linux` в строках, в которых содержится слово `OS`
`sed 's/[\t ]*/ /g`	заменить все последовательности символов пробела и табуляции любой длины на одиночный символ пробела
`sed '/^$/d'`	удалить все пустые строки
`sed '/^$/d;s/^/# /'`	удалить все пустые строки, а затем вставить в начало строки символ "#" и пробел.

Программа *awk* представляет собой обработчик текстовых файлов, содержащих информацию, записанную в виде таблиц, с *Cu*-подобным языком. По умолчанию столбцы таблицы разделяются пробелами. В языке имеются переменные, автоматический проход по всем строкам файла с доступам к ячейкам в столбцах по переменным `$1`, `$2` и т.д., а также блоки `BEGIN` и `END`, обрабатываемые, соответственно, до и после прохода таблицы. Пример

```
$ awk '{ print $2" -> "$3 + $5; }'
```

выводит содержимое 2 столбца, строку " -> " ("стрелка") и сумму 3 и 5 столбцов (для каждый строки).

```
$ awk '{sum+=$2;} BEGIN { sum=0; } END { print sum; }'
```

подсчитывает и выводит сумму значений во втором столбце.

Приложение D. Средства работы с Интернет

§D.1. Модель клиент-сервер

Сервер (*англ.* ***server*** *— тот, кто подает, сервирует*) — программа, выполняющая действия и дающая ответ на запросы **клиента** (*англ.* ***client***). Клиент, в свою очередь, — программа, отправляющая запросы серверу[1]. Клиент-сервер — модель взаимодействия двух процессов, которые могут быть запущены как на одном компьютере, так и на разных компьютерах сети[2]. Примеры серверов и клиентов — веб-браузер и веб-сервер, почтовый клиент и сервер почты и т.д.

Процесс сервера должен быть постоянно запущен для того, чтобы иметь возможность ответить на поступающий запрос. Чтобы стать сервером, процесс инициирует **прослушивание порта** (*англ.* ***port listening***), после чего остается запущенным, например, в фоновом режиме. По поступлении запроса сервер получает соответствующие данные от клиента и может обработать их. Порт необходим для идентификации процесса сервера на компьютере. На одном компьютере может быть запущено несколько серверов, но они должны прослушивать разные порты. Порт представляет собой целое число от 0 до 65535.

Сетевой протокол (*англ.* ***network protocol***) — набор правил и соглашений, с помощью которых клиент и сервер обмениваются информацией. Примеры сетевых протоколов — *HTTP* (протокол передачи гипертектовых документов), *FTP* (протокол передачи файлов), *HTTPS* (протокол передачи гипертектовых документов в зашифрованном виде) и другие. Для многих протоколов имеются стандартные порты (80, 22, 443 для вышеуказанных соответственно).

§D.2. *IP*-адреса и доменные имена

Протокол интернета (*англ.* ***Internet Protocol, IP***) — протокол адресации компьютеров (**хостов**, *англ.* ***host***) в сети с динамически меняющейся топологией. Каждому компьютеру присваивается уникальный

[1] Данные термины также переносят на компьютеры в сети, на которых установлены соответствующие программы, также под сервером (серверным компьютером) понимают специализированные для запуска серверов компьютеры.

[2] Строго говоря, взаимодействие происходит посредством **сокета** (*англ.* ***socket***) — API средством для работы клиентов (клиентский сокет) и серверов (серверный сокет), которые в *Unix*-подобных системах отражаются как файлы особого рода.

IP-адрес — 32-битное двоичное число, обычно записываемое как 4 десятичный числа от 0 до 255, разделенных точкой, например, *93.184.216.34*. Всего имеется чуть более 4 миллиардов *IP*-адресов, что в несколько раз меньше количества устройств, подключенных к сети Интернет. Проблема нехватки *IP*-адресов решается с помощью технологии **преобразования сетевых адресов** (*англ.* ***Network Address Translation, NAT***): выделяются локальные адреса (это адреса, начинающиеся с *192.168.*, *10.*, *172.16.*, *127.*), которые могут повторяться в разных локальных сетях, но должны быть различны в локальной сети. Доступ возможен только к компьютеру в той же локальной сети или имеющим глобальный *IP*-адрес. Также следует отметить, что адрес *127.0.0.1* — локальный адрес самого компьютера, с которого производится запрос. *IP*-адреса могут присваиваться статически (самими хостами) или динамически с помощью *DHCP*-сервера (*англ.* ***Dynamic Host Configuration Protocol*** — *протокол динамической настройки узла*), что предпочтительнее, так как позволяет избежать конфликта адресов, сохраняя возможность присваивать узлам постоянные адреса[1].

Доступ по *IP*-адресам сложен, поскольку их тяжело выучить, также они могут меняться. Поэтому вместо них используются говорящие **доменные имена** (*англ.* ***domain name***) — строковые имена хостов, по которым восстанавливаются *IP*-адреса (например, *example.com*). Адреса могут восстанавливаться динамически путем запроса к **системе доменных имен** (*англ.* ***Domain Name System, DNS***), работающей на специальных серверах — регистраторах доменных имен[2]. В файле */etc/hosts* могут быть перечислены сопоставления *IP*-адресов доменным именам, действительные для данной ОС. Это может быть использовано, например, для задания имен локальных компьютеров. Типично там же прописывается доменное имя локального компьютера — *localhost* — как сопоставляемое *127.0.0.1*.

[1]Каждое сетевое устройство имеет уникальный аппаратный ***MAC***-адрес (*англ.* ***Media Access Control*** — *надзор за доступом к среде*), которому сопоставляется *IP* адрес при включении устройства в сеть.

[2]На самом деле не каждый запрос *DNS* доходит до регистратора во избежании его перегрузки, вместо этого происходит поиск доменного имени в кеше самого компьютера и запрос соседних компьютеров, если такового в кеше нет, т.е. запрос к данному доменному имени производился давно или не производился вообще. Поэтому при смене *IP*-адреса может пройти несколько часов или дней прежде чем сервер снова будет доступен по доменному имени.

§D.3. Протокол *FTP*

Протокол передачи файлов (*англ.* ***File Transfer Protocol, FTP***) — протокол, позволяющий загружать с удаленного сервера или загружать на удаленный сервер файлы. Позволяет организовать авторизованный (по логину и паролю) и анонимный (логин и пароль — `anonymous`) доступ. Может работать в активном или пассивном режиме, в первом случае требуется, чтобы *FTP*-клиент мог принять входящее соединение от сервера для управления передачей данных[1]. Графические браузеры обычно поддерживают загрузку файлов по протоколу *FTP*, выгрузка может осуществляться графическими *FTP*-клиентами. *FTP* — простой и быстрый протокол для передачи файлов по сети. Файловый менеджер *Midnight Commander* позволяет открыть список файлов *FTP*-сервера в панели и работать с ними стандартным для менеджера образом.

Примерами FTP-серверов в *GNU/Linux* системах служат *vsftpd*, *pure-ftp* и др.

§D.4. Веб-страницы и протокол *HTTP*

Протокол передачи гипертекста[2] (*англ.* ***HyperText Transfer Protocol, HTTP***) — протокол, предназначенный для передачи гипертекстовых документах в формате *html* — **язык гипертекстовой разметки** (*англ.* ***HyperText Markup Language***) и файлов, необходимых для отображения данных документов — изображений, видео и др. По этому протоколу возможна закачка с сервера файлов любого формата. На сервер в свою очередь могут передаваться как файлы, так и данные форм гипертекста и строка запроса, формируемые и передаваемые браузером. Формат *html* представляет собой текстовый файл, содержащий т.н. тэги — структуры вида

```
<tag par1="val1" par2="val2"...>text</tag>
```

где `tag` — имя тэга, `par1`, `par2`, ... — параметры тэга `val1`, `val2`, ... — их значения, `text` — текст, управляемый тэгом. Например

```
<a href="http://example.com">Go to example.com</a>
```

представляет собой гиперссылку на сайт `http://example.com`, отображаемый текст которой — *Go to example.com*. С помощью тэгов регулируется

[1] Поскольку из-за нехватки *IP* адресов чаще всего используются постоянные локальные и временные или разделяемые внешние *IP* адреса (*NAT*) использование пассивного режима типично.

[2] Текст, содержащий ссылки с немедленным доступом на другие тексты — **гиперссылки** (*англ.* ***hyperlink***).

формат и место расположения текста на странице, а также строятся формы ввода информации, кнопки отправки формы, добавляются изображения и т.д., поэтому *html* является именно языком разметки гипертекста. Помимо *html*-тэгов, стиль отображения гипертекста определяют **каскадные таблицы стилей** (*англ.* ***Cascade Style Sheet, CSS***), как правило, поставляемые в отдельном файле с соответствующим расширением.

В настоящее время содержимое веб-серверов представляет собой не *html*-файлы, а динамические веб-страницы, формируемые по запросу пользователей путем исполнения сценария на сервере (написанного, например, на языке *PHP*). Поведение гипертекста в браузере в свою очередь регулируются сценариями, написанными на *JavaScript* и исполняемыми непосредственно на компьютере пользователя[1]. Эти сценарии позволяют динамически изменять содержимое страницы (например, по действию пользователя — щелчку мыши, вводу текста и т.п.) без ее перезагрузки, и даже подгружать данные с сервера (с помощью т.н. *веб-сокета* — обменом данными между браузером и сервером в режиме реального времени). Поэтому простая загрузка *html*-файлов с веб-сайта может не позволить получить полные данные о содержимом соответствующей веб-страницы.

Поскольку последовательные запросы к *HTTP*-серверу не связываются сервером между собой, клиенту необходимо идентифицировать себя (чтобы, например, сохранить авторизацию на сервере). Для этого сервер передает клиенту специальную информацию, называемую **куки** (*англ.* ***cookie*** — *печенье*), которую клиент должен сохранить и передать серверу при последующем запросе.

Примерами *HTTP*-серверов в *GNU/Linux* системах служат *apache*, *nginx*, *lighthttp* и др. Чаще всего помимо веб-сервера для работы сайта нужен сервер базы данных, например *mariadb*, *postgresql* и др., хотя возможно и использование файловой базы данных (*sqlite* и др.).

§D.5. Защищенные протоколы и сертификаты

Протоколы *HTTP* и *FTP* не являются защищенными — данные передаются по сети без шифрования и могут легко быть перехвачены и подменены третьими лицами. Их защищенные аналоги — протоколы *FTPS* и *HTTPS*. Защищенное соединение обеспечивается также протоколом *SSH* и *SFTP*[2]. Принцип защищенного соединения состоит в шифровании с открытым ключом. Этот метод основан на присвоении каждому узлу пары

[1] Именно наличие таких сценариев объясняет высокую ресурсоемкость современных веб-страниц.

[2] Не связан с *FTP* и *FTPS* несмотря на схожесть названия

открытого (*англ.* ***public key***) и **закрытого ключа** (*англ.* ***secure key***) — функций P и S соответственно, таких что

$$S(P(M)) \equiv P(S(M)) \equiv M,$$

где M — шифруемое сообщение. При этом считается, что зная ключ P невозможно построить ключ S[1].

Пусть есть два узла A и B с парами ключей (P_A, S_A) и (P_B, S_B). Тогда, для того, чтобы отправитель A мог передать сообщение M получателю B так, чтобы никто кроме получателя не мог его прочитать, необходимо передать сообщение $P_B(M)$ — только зная S_B можно получить $M = S_B(P_B(M))$. Для того, чтобы быть уверенным в том, что автором сообщения действительно является A необходимо передать пару (M, M_s) где $M_s = S_A(M)$, тогда зная P_A можно сверить, что $M = P_A(M_s)$ (электронная подпись). Можно и совместить шифрование и подпись, передавая пару $(P_B(M), P_B(S_A(M)))$.

Шифрование с открытым ключом позволяет передать защищенное сообщение и проверить авторство между адресатами, которые не могут использовать закрытый канал связи для передачи секретного слова (шифра), что типично для компьютерных сетей[2]. Единственная проблема — как убедиться, что данный публичный ключ действительно принадлежит данному узлу. Это решается с помощью сертификатов — средств проверки подлинности узлов и их ключей. Если пользователь доверяет сертификату, то он может его принять, тем самым признав подлинность данного ключа — обычно такая практика применяется для локальной сети, протокола SSH и т.п.. В сети Интернет имеются службы сертифицирования, проверяющие подлинность ключей сайтов, в свою очередь в ОС и браузеры устанавливаются сертификаты этих служб, что позволяет защитить посетителя сайта от “ручной” проверки сертификата, объективно невозможной в большинстве случаев[3].

[1] Это возможно благодаря тому, что найти и перемножить два больших (например 768-значных в двоичной системе счисления) простых числа легко, в то время как зная только их произведение найти два исходных простых числа невозможно, так как на вычисление требуется время, превышающее время существования Земли.

[2] На самом деле шифрование с открытым ключом используется для передачи одноразового закрытого ключа, так как первое требует многократно больше ресурсов для шифрования и расшифровки файлов.

[3] Тем не менее следует обращать внимание на сообщения браузера о проблемах с сертификатом и не игнорировать их.

§D.6. URL-ссылка

Унифицированный указатель ресурса (*англ.* ***Uniform Resource Locator, URL***) — универсальная ссылка на ресурс (файл) в сети, определяющая его местонахождение и способ получения. Типичная структура ссылки

схема`://[[`*логин*`[:`*пароль*`]@]`*хост*`[:`*порт*`]][/`*URL-путь*`]`
`[?`*параметры*`][#`*якорь*`]`

где

- схема — указание на протокол обмена данными (`https`, `ftp` и др);
- логин и пароль — логин и пароль удаленного компьютера[1];
- хост — *IP*-адрес или доменное имя компьютера;
- порт — порт сервера (если не указан, используется порт по умолчанию для данного протокола);
- *URL*-путь — путь к запрашиваемому файлу от корневого каталога сервера или иной путь, обрабатываемый сервером
- параметры — строки вида *имя=значение*, разделяемые символом & (могут использоваться, например, для отправки данных форм веб-страницы[2]).
- якорь — запрашиваемое место в файле (например, указатель на конкретный раздел веб-страницы).

Примеры:

http://example.com/doc/file.php?type=wide&font=bold#sect1

ftp://user:password@example.com/path/to/file

ftp://user@192.168.1.1/home/user/file.txt

§D.7. Примеры приложений CLI и TUI

Загрузка файлов (и даже рекурсивная загрузка целых сайтов) по протоколам *HTTP*, *HTTPS*, *FTP* может быть осуществлена командой *wget*.

[1] Не следует путать их с логином и паролем от веб-сайтов, которые не являются входом в удаленный компьютер, а обрабатываются сценарием, исполняемым на веб-сервере.

[2] *GET*-запрос; в этом случае содержимое формы видно в адресной строке. Альтернатива — *POST*-запрос, осуществляется внутри специальных полей запроса к *HTTP*-серверу, а не в *URL*. Последний по понятным причинам больше подходит для передачи паролей от сайта.

Приложение также поддерживает отправку *POST*-запросов, сохранение куки в текстовые файлы и передачу серверу ранее сохраненных куки[1]. Выгрузка файлов на *FTP* сервер может осуществляться с помощью команды *wput*. Также имеется команда *curl* для доступа по различным протоколам. Эти команды позволяют автоматизировать работу с сетевыми ресурсами в сценариях.

Существуют текстовые (TUI) браузеры, например *lynx*. В настоящее время, ввиду отсутствия полноценной поддержки *JavaScript* и веб-сокетов, доступ ко многим сайтам из этих браузеров, к сожалению, невозможен.

Приложение Е. Средства администрирования

§Е.1. Регистрационные данные пользователей и групп

Добавление, удаление и модификация учетных записей пользователей осуществляется с помощью команд `useradd`, `usermod` и `userdel` соответственно, аналогичные действия с группами — `groupadd`, `groupmod` и `groupdel`. В некоторых системах присутствует команда `adduser`, имеющая больше возможностей, чем `useradd`.

С помощью команды `passwd` можно установить и изменить пароль пользователя.

Регистрационные данные пользователей хранятся в файле `/etc/passwd`; пароли обычно хранятся в зашифрованном виде[2], в файле `/etc/shadow`; данные групп хранятся в файле `/etc/groups`. Возможно и "ручное" изменение данных файлов с помощью любого текстового редактора для анализа, добавления, удаления и модификации регистрационных данных пользователей и групп.

Типичный формат файла `passwd` — таблица: в каждой строке содержится информация об одном пользователе, столбцы разделяются символом "`:`". Столбцы следующие: логин, пароль (`x` — если он в файле `shadow`), *uid*, *gid* основной группы, полное имя, домашний каталог, оболочка для

[1]Можно также использовать куки, экспортированные из браузера, например для доступа к сайтам с авторизацией.

[2]Хранятся хеш-функции паролей, т.е. используется т.н. *односторонне шифрование*: по введенному паролю можно легко получить хеш и сверить его, в то время как имея только хеш проблематично восстановить пароль, т.к. на это требуется очень много вычислительных ресурсов.

входа[1]. Например,

```
root:x:0:0:root:/root:/bin/bash
apache:x:48:48:Apache:/usr/share/httpd:/sbin/nologin
user:x:1000:1000:Computer User:/home/user:/bin/bash
```

Первая — запись администратора, вторая — пользователя для запуска веб-сервера `httpd`, третья — обычного пользователя.

Формат файла `group` аналогичен, столбцы следующие: имя группы, пароль (`x` — если он в файле `shadow`), *gid*, список пользователей-членов группы[2], разделенных, запятой. Например,

```
root:x:0:
wheel:x:10:admin
user:x:1000:
vboxusers:x:931:admin,user
```

Здесь: стандартная группа *root*, группа администраторов `sudo` — *wheel*, включающая пользователя *admin*, основная группа пользователя *user*, группа пользователей виртуальных машин *vboxusers*, куда входят пользователи с логинами *admin* и *user*.

§E.2. Настройка *sudo*

Настройка прав доступа пользователей с помощью программы `sudo` хранится в текстовом файле `/etc/sudoers`. Обычно его прямое редактирование не рекомендуется, т.к. ошибка может привести к проблемам к выполнению `sudo` и невозможности ее исправить без административного пароля. Вместо этого рекомендуется использовать программу `visudo`, которая проверит на наличие ошибок после редактирования и, в случае их наличия, предложит вернуться к редактированию. По умолчанию программа использует редактор *vim*, но его можно изменить с помощью переменной `EDITOR`, например

```
# EDITOR=nano visudo
```

или

```
# EDITOR=mcedit visudo
```

Простейший формат строк файла такой:

[1] Если в качестве оболочки указать `/sbin/nologin`, пользователь не будет иметь возможность войти в систему. Возможно только переключение на данного пользователя процессом с правами администратора для “самоограничения” прав из соображений безопасности. Такие пользователи нужны для запусков серверов и других демонов.

[2] Пользователей, для которых эта группа не основная, а дополнительная.

```
пользователь хост=(кем может стать) допустимые команды
```

Например,

```
user ALL=(superuser) /opt/bin/drop
```

означает, что пользователь *user* может выполнять команду `/opt/bin/drop` с любого компьютера, управляемого данным *sudo*, от имени пользователя *superuser*, вводя свой пароль;

```
user ALL=(root) NOPASSWD: /usr/bin/mount, /usr/bin/umount
```

означает, что пользователь *user* может выполнять *mount* и *umount* с любого компьютера, управляемого данным *sudo*, от имени администратора *root* без ввода пароля;

```
root ALL=(ALL) ALL
```

означает, что пользователь *root* может исполнять все команды от имени всех пользователей, со всех компьютеров (ввода своего пароля для пользователя *root* не требуется);

```
%wheel    ALL=(ALL)    ALL
```

разрешает всем пользователям группы *wheel* (входящим в эту группу, имеющим данную группу в качестве дополнительной) все команды от имени всех пользователей, со всех компьютеров со вводом своего пароля — типичное решение для организации *sudo*[1];

```
%wheel    ALL=(ALL)    NOPASSWD: ALL
```

аналогично, но без ввода пароля. Данное решение, хотя и нередко приводится как возможный вариант, все же считается небезопасным. В этом случае при случайном доступе постороннего к терминалу пользователя члена группы *wheel* возможно получение административных полномочий и причинение вреда всей системе, а не только данному пользователю. Также это создает дополнительные риски при удаленном доступе и запуске

[1] В этом случае административные полномочия выдаются и отзываются у пользователя путем включения его в группу *wheel*.

непроверенных приложений, которые могут получить административные полномочия без санкции пользователя. Поэтому рекомендуется защищать команды *sudo* вводом пароля пользователя за исключением точечного выбора отдельных команд, которые желательно исполнять без ввода пароля в целях автоматизации и удобства, но которые не несут больших рисков безопасности.

§E.3. Демоны

Демон (*англ.* ***daemon***) — программа, работающая в фоновом режиме без взаимодействия с пользователем. В качестве демона могут выступать серверы, служебные программы и т.п. Запуск демона может осуществляться из командной строки, если программа поддерживает **демонизацию** (*англ.* ***daemonize***) при запуске или с помощью специального ключа. Также возможен автоматический запуск демона при старте ОС. Многие команды запуска демона имеют букву `d` в конце имени (`httpd`, `sshd`, `vsftpd` и др.)

На ранних версиях *GNU/Linux* запуск осуществлялся с помощью программы *init* и соответствующих *init*-скриптов, запускающих в т.ч. демоны.

```
# /etc/init.d/имя-демона start
```

для запуска

```
# /etc/init.d/имя-демона stop
```

для остановки, а также с помощью команд *service*. С помощью команды *update-rc.d* или прямым созданием ссылок возможно добавление или удаление демонов из автозагрузки.

В настоящее время в дистрибутивах применяется программа *systemd*, управляемой посредством *systemctl*, с помощью которой можно запустить демон

```
# systemctl start имя-демона
```

остановить демон

```
# systemctl stop имя-демона
```

сделать демон запускаемым по умолчанию при загрузке ОС

```
# systemctl enable имя-демона
```

и убрать его из сценария загрузки ОС

```
# systemctl disable имя-демона
```

§E.4. Демон *cron*

Демон *cron*[1] (`crond`) позволяет запускать выполнять команды по расписанию — ежечасно, ежедневно и т.п. Для его настройки следует вызвать команду

```
$ crontab -e
```

из аккаунта того пользователя, от имени которого должны запускаться программы (в т.ч. можно вызвать и от имени администратора). Программа вызовет редактирование таблицы расписания. Разумеется, если не устраивает стандартный редактор, можно изменит переменную `EDITOR`:

```
$ EDITOR=mcedit crontab -e
```

По выходу из редактора программа проверит, допустим ли формат таблицы и, в случае ошибок в ней, сообщит об этом. Если все в порядке — новая таблица будет активизирована.

Таблица *crontab* состоит из 6 столбцов, разделяемых пробелами или табуляторами: минута, час, день, месяц, день недели и команда. В качестве значений времени выполнения могут выступать конкретные значения (например, `5` в поле "день" означает запуск каждого 5 числа), диапазоны (`0-3` в поле час означает выполнять в 0, 1, 2 и 3 час суток), списки (`1,3,5` в поле дня недели — выполнять по понедельникам, средами и пятницам), символ `*`, означающий любое значение, а также вышеуказанные значения, после которых ставится символ слеша "`/`" и число. Последнее позволяет исполнять каждые n единиц времени (`*/5` в поле дни — каждые 5 дней), или указывать конкретные единицы времени выполнения по модулю (`0-59/2` в поле минуты — каждую четную минуту). Символ `#` в начале строки означает комментарий. Также в файле можно стандартным образом задать значения переменных окружения, экспортируемые командам. Примеры:

```
# выполнять 2 числа каждого месяца в 12 часов 00 минут
00 12 2 * * monthly-command

# выполнять по выходным в 14:30
30 14 * * 0,6 day-off-command

# Выполнять в 0:10, 3:10, 6:10, 9:10, ..., 21:10
10 */3 * * * every-three-hours-command

# Выполнять по будним дням в 8-30 и 17-30
30 8,17 * * 1-5 working-day-command
```

[1] В честь древнегреческого бога времени — Хроноса.

```
# Выполнять каждую четную минуту
0-59/2 * * * * even-minutes-command
```

§E.5. Удаленный вход в систему по *SSH*, протокол *SFTP*

SSH (*англ.* ***secure shell*** — *защищенная оболочка*) — средство удаленного входа в оболочку компьютера, пришедшее на смену незащищенным вариантам *telnet* и *rlogin*. Позволяет войти в удаленный компьютер B с компьютера A в качестве пользователя компьютера B и выполнять команды оболочки на компьютере B с компьютера A. Защищенность соединения обеспечивается шифрованием с открытым ключом.

Формат вызова *ssh*-клиента следующий:

```
$ ssh [-X] [пользователь@]хост [команда]
```

например

```
$ ssh user@server.local
```

Если пользователь не указан, то используется логин текущего пользователя на компьютере, откуда выполняется вход. Если команда указана, то она исполняется и сессия завершается, в противном случае происходит вход в оболочку пользователя на удаленном компьютере. Опция `-X` позволяет перенаправлять запросы приложений к *X-серверу* на *X-сервер* компьютера, откуда осуществляется вход, что позволяет запускать графические приложения на удаленном компьютере B с отображением графического окна на локальном компьютере. Данный *X-сервер* должен быть запущен на управляющем компьютере A, *X-сервер* удаленного компьютера не используется и не обязателен[1]. Это позволит запускать графические приложения из удаленного терминала.

При первом входе на данный удаленный компьютер клиент спросит, следует ли доверять указанному открытому ключу удаленного хоста, т.к. протокол *ssh* не предполагает сертификацию узлов[2], при последующих входах данного сообщения быть не должно. Если при последующих входах появляется сообщение о изменении ключа удаленного хоста, следует

[1] Это сработает без дополнительных действий, если `ssh` вызван на графическом эмуляторе терминала. Существуют *ssh*-клиенты для других операционных систем, например *PuTTY* для ОС *Windows*, но для запуска графических приложений дополнительно потребуется установленный *X-сервер* для данной системы, которые также существуют в виде отдельных приложений. Графического окружения *Windows*, *Android* и других ОС, не основанного на *X-сервере*, недостаточно.

[2] В том числе потому, что ориентирован главным образом на работу в локальной сети.

отнестись к нему внимательно: действительно ли ключ изменился (сменился сервер, компьютер и т.д.) или злоумышленник пытается выдать себя за данный удаленный компьютер. Данные доверительных хостов обычно хранятся в файле `~/.ssh/known_hosts`, специфичном для каждого локального пользователя.

По умолчанию при входе программа запросит пароль от удаленного компьютера. Систему можно настроить на вход без ввода пароля, с помощью ключей. Этот способ удобен и считается более безопасным, так как риск утечки ключа ниже, чем пароля. Для этого следует сгенерировать локальную пару секретного и открытого ключа и передать открытый ключ на удаленный компьютер. Пара ключей хранится в файлах `~/.ssh/id_rsa` (секретный) и `~/.ssh/id_rsa.pub` (открытый) и также специфичны для данного локального пользователя. Если данные файлы уже существуют, то регенерировать ключи не нужно. В противном случае следует вызвать

```
$ ssh-keygen
```

Программа интерактивным образом запросит файл сохранения ключей и пароль для доступа к закрытому ключу. Пароль можно оставить пустым для удобства, хотя его использование может повысить безопасность. После этого следует отправить ключ на удаленный хост с помощью

```
$ ssh-copy-id [пользователь@]хост
```

либо (если данный способ недоступен) вручную:

```
$ cat .ssh/id_rsa.pub | [пользователь@]хост
'cat >> .ssh/authorized_keys'
$ ssh [пользователь@]хост "chmod 700 .ssh;
chmod 640 .ssh/authorized_keys"
```

последнее нужно для установки правильных разрешений, без которых ключи не будет приниматься из соображений безопасности. Именно в файле `~/.ssh/authorized_keys` хранятся авторизованные ключи. При этом в командах выше указывать `~/.ssh` не нужно, т.к. “~” будет псевдонимом локального домашнего каталога, в то время как для исполняемой команды текущим каталогом будет домашний каталог удаленного пользователя. После этого вход в удаленный компьютер из-под данного локального аккаунта в выбранный удаленный аккаунт будет осуществляться автоматически, без ввода пароля[1].

Для работы *SSH* на удаленном компьютере должен быть настроен и запущен *SSH*-сервер — `sshd`.

[1]Если был задан непустой пароль доступа к ключу, то он будет запрашиваться. Избавиться от этого можно, настроив *ssh-agent*, доступ к паролям будет запрашиваться и открываться при входе в систему.

```
# systemctl enable sshd
# systemctl start sshd
```

Перед запуском может потребоваться сгенерировать серверный открытый и закрытый ключи, например

```
# ssh-keygen -q -f /etc/ssh/ssh_host_rsa_key -N '' -t rsa
```

молчаливо создаст RSA ключ для сервера без пароля (это важно, так как иначе запуск сервера будет затруднен) и сохранит в указанном файле в каталоге конфигурации сервера[1].

Важно аккуратно настроить сервер. Обычно его конфигурация хранится в файле `/etc/ssh/sshd_config`, изначально содержащем базовую настройку (значения опций по умолчанию в конфигурационных файлах серверов и системных утилит часто преведены в закомментированном виде). С помощью опции `PasswordAuthentication` можно разрешить или запретить вход по паролю.

```
PasswordAuthentication no
```

Аналогично `PubkeyAuthentication` разрешает и запрещает вход по ключу. С помощью опций `AllowGroups` и `AllowUsers` можно разрешить доступ только отдельным группам и пользователям. Например

```
AllowGroups users
```

разрешает доступ пользователям, входящим в группу `users`. С помощью `Match Group` и `Match User` можно установить особые значения опций для отдельных групп и пользователей

```
Match Group root
    PasswordAuthentication no
```

запрещает вход по паролю для пользователей, входящих в группу `root`.

Протокол *SSH* можно использовать для передачи файлов[2]. Сделать это можно с помощью команды `scp`. Кроме этого, существует протокол

[1] Тип ключа `rsa` в настоящее время используется по умолчанию. Возможно использование и других типов ключей как для сервера, так и для пользователей.

[2] Разумеется, это можно сделать, выполнив `cat` для удаленного файла в качестве исполняемой на удаленной машине команды с перенаправлением вывода команды `ssh` в локальный файл, и, обратно, перенаправив локальный файл на стандартный ввод, а вывод `cat` в файл на удаленной машине. Чтобы перенаправить ввод и вывод удаленной команды следует экранировать символ перенаправления: тогда он будет передан `ssh` в качестве аргумента локальной оболочкой и обработан удаленной оболочкой. Такой подход, конечно, не слишком удобен и оптимален, но возможен практически всегда.

SFTP (*англ.* ***SSH File Transfer Protocol***) — протокол передачи файлов по *SSH*. Многие приложения, в том числе *Midnight Commander* поддерживают данный протокол для открытия списка файлов на удаленном компьютере и работе с ним. Кроме того, существует реализация файловой системы, основанной на *SSH* — *SSHFS*, позволяющей монтировать каталог удаленного компьютера в локальную точку монтирования. Реализация команды `sshfs` выполнена в пользовательском пространстве, поэтому монтирование и размонтирование данной ФС не требует административных полномочий[1].

§E.6. Терминальный мультиплексер

Терминальный мультиплексер (*англ.* ***terminal multiplexer***) — программа, позволяющая в рамках одного терминала (вкладки эмулятора терминала или одной виртуальной консоли ОС) создать несколько виртуальных терминалов. Подобное программное обеспечение позволяет решить несколько задач:

- запустить несколько независимых экземпляров оболочки в одной виртуальной консоли, тем самым освободившись от необходимости входить в систему для каждой консольной сессии и лимита на количество виртуальных консолей;
- присоединиться к запущенной терминальной сессии с удаленного компьютера (например, по *SSH*), осуществлять синхронную работу с нескольких компьютеров;
- обезопасить себя от потери данных при разрыве соединения с удаленным компьютером (при разрыве *SSH* соединения происходит завершение всех процессов, запущенных на удаленном компьютере по *SSH*; при завершении процесса интерфейса терминального мультиплексера запущенная на нем сессия сохраняется — к ней можно подсоединиться после восстановления соединения).

Наиболее известными представителями терминального мультиплексера являются классический *GNU Screen* и более современный *tmux*.

[1] Это реализовано посредством драйвера **FUSE** (*англ.* ***File System in Userspace*** *— файловая система в пользовательском пространстве*). Монтирование таких файловых систем производится пользователем с помощью отдельной программы, в данном случае `sshfs`, размонтирование — с помощью команды `fusermount -u`.

§E.7. Монтирование разделов

Для того, чтобы подмонтировать устройство в каталог необходимо использовать команду `mount`, указав монтируемый раздел `device` (см. ниже) и точку монтирования `mount-point` в качестве аргументов:

```
# mount устройство точка монтирования
```

В большинстве случаев данная команда должна выполняться с административными полномочиями. Размонтировать раздел можно с помощью команды `umount`

```
# umount устройство
```

или

```
# umount точка монтирования
```

Размонтирование возможно только если файлы подмонтированного раздела не используются ни одним процессом. Например,

```
# mount /dev/sdb5 /mnt/partition
# mount /dev/sdc1 /home
# umount /dev/sdb5
# umount /home
```

Помимо доступа к разделу по имени устройства, возможен также доступ по метке или уникальному идентификатору. Например,

```
# mount /dev/disk/by-label/home /home
```

С помощью команды `mount` без аргументов можно просмотреть список всех монтированных файловых систем.

Команда `df` позволяет просмотреть свободное место на подмонтированных файловых системах.

Команда `lsblk` позволяет просмотреть список всех доступных (подмонтированных и нет) разделов (блочных устройств) — их имена, метки, уникальные идентификаторы, размер, файловую систему и т.д.

Еще одна важная команда — `fstrim` — позволяет "очистить" свободное место на SSD-накопителях. Дело в том, что при удалении файлов не производится их физическое удаление с диска, поэтому с точки зрения накопителя свободное в ФС место остается занятым. Увеличение объема занятого места приводит к замедлению записи на SSD, в этом случае рекомендуется вызвать *fstrim* для файловой системы на нем.

Помимо монтирования раздела `mount` позволяет связать два каталога — подключить содержимое одного из них как содержимое другого. Это делается с помощью ключа `--bind` (или `--rbind`, чтобы подмонтировать все точки монтирования исходного каталога тоже):

```
# mount --bind /mnt/data/user /home/user
```

Файл `/etc/fstab` содержит таблицу монтирования разделов при стар-

те ОС. Каждая строчка соответствует одной единице монтирования, столбцы таблицы разделяются табуляцией. Столбцы следующие: устройство (может быть указано явно, как `/dev/sda1`, по уникальному идентификатору, как `UUID=12345`, по метке `LABEL=disk_label` и др[1].), точка монтирования, тип ФС, опции монтирования (для опций по умолчанию должно быть указано `default`), число, означающее делать ли автоматическую аварийную копию с помощью утилиты `dump`[2], и число, означающее проводить ли проверку ФС при старте системы (0 — не проводить, 1 — проводить, 2 — проводить в первую очередь)[3].

Запуск команды `mount` без аргументов позволяет просмотреть все монтированные в данный момент ФС. Также эта информация доступна в файле `/proc/mounts` и `/etc/mtab`, причем последний является символическом ссылкой на монтированные ФС текущего процесса, которые при особых настройках могут отличаться от общесистемных.

§E.8. Работа с разделами и файловыми системами

Применение описанных в этом параграфе средств следует проводить с особой осторожностью, поскольку неверное применение может привести к потере всех данных. Параграф приводится исключительно с целью обзора существующих возможностей, но для конкретного применения их следует внимательно изучать документацию ПО и специализированные обучающие материалы.

Разбиение диска на разделы осуществляется с помощью команды `fdisk` с интерактивным интерфейсом. Если диск не размечен вовсе, необходимо создать таблицу разделов MBR (старый вариант, только для дисков до 2Тб) или GPT. Также с ее помощью можно удалять и создавать разделы.

Для создания файловых систем на разделах следует использовать специфические для ФС команды (например, `mkntfs`, `mkswap`) или универсальных комбинаций "`mkfs.`" и название ФС (`mkfs.ext4`, `mkfs.btrfs`) и др. С помощью программ типа `tune2fs`, `ntfslabel` и др. можно проводить настройку ФС — смену метки и др. Проверка ошибок файловой системы также может осуществляться специфическими (`e2fsck`) или универсальными командами (`fsck.vfat`). Для многих ФС существует возможность

[1]Рекомендуется использовать уникальный идентификатор или метку, а не имя файла-устройства, так как номера и буквы файлов-устройств могут быть изменены при изменении аппаратной конфигурации компьютера или при изменении разбиения носителей на разделы.

[2]Эта утилита в настоящее врмемя используется крайне редко, поэтому ставится 0.

[3]Должно быть 0 для *btrfs*, т.к. эта система в норме не только не требует проверки, но и может быть нарушена при ее принудительном проведении. Для других систем указание 1 вполне оправдано.

изменения размера, например с помощью команды `resize2fs`, во многих случаях увеличение размера ФС возможно без размонтирования раздела. Следует обращаться к документации по программе обработки соответствующей файловой системы. При этом перед увеличением размера ФС обязательно необходимо увеличить размер раздела (с помощью *fdisk* это производится удалением старого и созданием нового раздела), а после уменьшения — наоборот, уменьшить раздел.

Нередки ситуации, когда необходимо восстановить ранее удаленный файл. На самом деле при удалении файлы не удаляются с диска физически, поэтому шансы восстановить их велики. Для этого необходимо как можно быстрее размонтировать раздел (если это корневой раздел, необходимо загрузиться с *Live*-образа, раздел *home* можно размонтировать, если выйти всем пользователям, кроме администратора, или перейти в монопользовательский режим с помощью `init 1` или

```
# systemctl isolate rescue.target
```

для систем с *systemd*[1]. Далее следует обращаться к специфическим для ФС средствам, например, *extundelete* для систем *ext2*, *ext3* и *ext4*. Также можно использовать программы *Testdisk* для поиска и восстановления удаленных файлов с помощью TUI, а также *Photorec*, которая осуществляет поиск файлов известных форматов по всему диску (длительный, но чаще всего успешный процесс). Также текстовый файл бывает можно восстановить путем поиска его содержимого с помощью *grep*, например,

```
# grep -a -B 10 -A 100 'строка файла' /dev/sda1 >file.txt
```

чтобы получить 10 строк до и 100 строк после указанной строки. Программы *Testdisk* и *Photorec* также позволяют находить "потерянные" разделы и файлы на них (например, в случае повреждения таблицы разделов).

Помимо традиционных ошибок ФС могут возникать аппаратные ошибки — поврежденные сектора диска. Найти такие сектора можно с помощью программы `badblocks`. Однако в современных HDD такие сектора автоматически перемещаются на зарезервированное свободное пространство. Появление поврежденных секторов в *badblocks* в режиме только чтения, скорее всего, говорит о логическом повреждении — например, если произошла ошибка или сбой питания при записи в сектор. Данные из такого сектора нельзя прочитать, но сектор можно восстановить, перезаписав данные в нем с помощью той же программы *badblocks* или `dd`.

Программа `smartctl` позволяет просмотреть состояние атрибутов жест-

[1]При работе с *init* система может находиться в нескольких состояниях, уровнях выполнения: 0 — выключение, 1 — однопользовательский режим, 3 — многопользовательский режим, 5 — графический режим, 6 — перезагрузка. Им соответствуют цели `poweroff`, `rescue`, `multi-user`, `graphical`, `reboot` в *systemd*. Для каждого уровня или цели имеется собственная настройка конфигурации запуска, в т.ч. запускаемых демонов.

ких дисков и SSD — S.M.A.R.T. (*Self-Monitoring, Analysis and Reporting Technology*), говорящих о состоянии жесткого диска. Например, время работы, количество запусков, количество перемещенных секторов. Обычно интерес представляет не абсолютное значение, а оценка состояния диска в процентах — 100% — полное здоровье, 0% — диск на грани отказа. Однако появление перемещенных секторов (даже одного) — наиболее тревожный знак о физических проблемах с диском. В случае с SSD также можно отследить ресурс перезаписи, по исчерпании которого диск будет доступен только для чтения.

С помощью команды `dd` можно создать образ дискового раздела или записать образ дискового раздела на раздел. Например

```
# dd if=/dev/sda1 of=./sda1.img
```

создает образ

```
# dd if=sda1.img of=/dev/sdb2
```

записывает образ (возможно и клонирование разделов “на лету”). Также рекомендуется устанавливать размер блока однократного чтения-записи, отличный от размера по умолчанию (одного сектора) для ускорения процесса:

```
# dd if=/dev/sda1 of=./sda1.img bs=1M
```

установит размер блока в 1 мегабайт, что, обычно, дает хорошую скорость. Программа `dd` требует особой осторожности: она перезаписывает данные без предупреждения, поэтому при указании неверного целевого файла или устройства данные на нем будут уничтожены. В то же время создание образов разделов или целых дисков — мощный инструмент клонирования и сохранения дисков, установленных ОС, переноса их на новое устройство или устройство большего размера и т.д. Сохраненные образы можно монтировать для чтения записи, для этого следует подключить их как блочное устройство зацикливания с помощью команды `losetup` и подмонтировать его, например

```
# losetup /dev/loop0 sda1.img
# mount /dev/loop0 /mnt/sda1
```

Обратное действие:

```
# umount /dev/loop0
# losetup -d /dev/loop0
```

В современных системах `mount` автоматически подключает устройства зацикливания, поэтому достаточно вызвать

```
# mount sda1.img /mnt/sda1
```

и соответствующий `umount`.

§E.9. Смена корневого каталога

Команда `chroot` позволяет сделать некоторый каталог корневым для запускаемой из него программы. Это позволяет, например, устанавливать программы из более старых версий системы, из других дистрибутивов, из минимальной конфигурации и т.п. — со своими зависимостями устанавливаемых пакетов. Чаще всего для адекватной работы окружения *chroot* необходимо подмонтировать системные каталоги, делается это с помощью *bind-mount*, например

```
# mount --rbind /proc          /path/to/temp-root/proc
# mount --rbind /dev           /path/to/temp-root/dev
# mount --rbind /run           /path/to/temp-root/run
# mount --rbind /sys           /path/to/temp-root/sys
# mount --rbind /home          /path/to/temp-root/home
# mount --rbind /var/lib/dbus /path/to/temp-root/var/lib/dbus
# mount --rbind /tmp           /path/to/temp-root/tmp

# chroot /path/ro/temp-root
```

с последующим размонтированием после завершения работы. Разумеется в `/path/to/temp-root` должен быть установленный образ системы — хотя бы для того, чтобы запустить оболочку. Сделать это можно средствами менеджера пакетов ОС. Для работы Интернет также может потребоваться вызвать

```
# cp /etc/resolv.conf /path/to/temp-root/etc/resolv.conf
```

для того, чтобы обеспечить работу службы доменных имен.

Следует учесть, что хотя *chroot* и позволяет изолировать процессы от основной ОС, такая изоляция может быть нарушена самими процессами, поэтому данное средство не следует использовать как инструмент безопасности (например, для запуска сомнительных приложений). Также из-за рисков безопасности (в частности, возможного наличия файлов с битом *setuid* и владельцем *root* внутри такого окружения), *chroot* может быть выполнен только от имени администратора[1].

[1] Существует программа `fakeroot`, позволяющая сымитировать наличие административного доступа без реального повышения привилегий пользователя и риска для безопасности ОС. Программа `fakechroot` аналогично позволяет имитировать *chroot* для пользователя.

§E.10. О загрузчике и загрузке ОС

Существует несколько загрузчиков ОС на базе *Linux* — *LiLo*, *Grub*, *extlinux* и другие. В настоящее время стандартным для многих дистрибутивов является *Grub2*. Для того, чтобы сделать диск загрузочным, необходимо установить загрузчик в соответствующий раздел загрузки (*boot* или корневой раздел системы), а также на MBR или раздел EFI в зависимости от способа загрузки компьютера. В *Grub2* это выполняется командой `grub2-install`. Загрузчик должен быть сконфигурирован на загрузку ОС, в частности ядра *Linux* или иных ОС. В *Grub2* найти операционные системы и создать конфигурацию можно с помощью команды *grub2-mkconfig*, вывод которой нужно сохранить в `/boot/grub2/grub.cfg`.

Процесс загрузки *GNU/Linux* включает в себя старт ядра *Linux*, загрузку инициализирующего RAM-образа ФС (*initramfs*) и передачу управления ему. Данный образ содержит минимальный набор программ и драйверов для проверки и монтирования корневой файловой системы и дальнейшей передачи процесса загрузки в *init* или *systemd*. В случае ошибок с корневой файловой системой можно войти в командную оболочку данного образа и провести необходимые процедуры по их исправлению. Данный образ также ответственен за графическую заставку при загрузке. Образ создается в момент установки или обновления ядра и адаптируется под конкретную конфигурацию ОС. При некоторых изменениях, например, смену заставки загрузки, добавлении новых устройств или ФС, необходимых для загрузки и т.п., может потребоваться пересоздание образа, сделать это можно с помощью команды `dracut`.

После выполнения сценария загрузки *systemd* запускает программу `login` для текстового терминала и экранный менеджер для графического. Обе программы нужны для входа в систему. В состав различных окружений рабочего стола входят различные экранные менеджеры, но практически каждый из них позволяет пользователю выбрать окружение рабочего стола при входе.

Приложение F. Основы безопасности в ОС семейства *Unix*

Хотя традиционно *Unix*-подобные системы считаются обладающими высоким уровнем конструктивной безопасности, которой всегда уделяется большое внимание, существуют определенные риски, связанные с их использованием. Источниками проблем с безопасностью могут являться

как само программное обеспечение, так и действия пользователей, поэтому *GNU/Linux* и другие *Unix*-подобные ОС нельзя считать неуязвимыми для вирусов и другого вредоносного ПО. Кроме того, неосторожные действия пользователя в настоящее время являются одной из самых частых причин нарушения безопасности компьютерных систем.

В первом случае речь идет о т.н. **уязвимостях** (*англ.* ***vulnerability***) — конструктивных недоработках или ошибках в коде (ядра, прикладных программ), приводящие к тому, что определенные локальные или удаленные действия позволяют обойти настройки безопасности ОС. Программы, осуществляющие такие действия, называются **эксплойтами** (*англ.* ***expliot***).

Концепция безопасности *Unix* и *Linux* базируется на том принципе, что любой процесс, исполняемый от имени пользователя, имеет доступ ко всем данным этого пользователя, включая файлы на диске, информацию, выводимую на экран, вводимую с клавиатуры, передаваемую по сети и др. Поэтому исполнение программ из непроверенных источников (сторонних репозиториев, программ, поведение которых трудно проверить сообществу, в частности программ с закрытым исходным кодом) может привести к утечке или повреждению этих данных. Даже условно "проверенные" программы могут содержать уязвимости, которые в определенной ситуации приведут к исполнению вредоносного кода (например, при открытии документа со специально подобранным содержимым, поступлении специфического запроса из сети и т.п.). Все это может происходить скрытно от пользователя.

Для того, чтобы снизить вероятность уязвимости необходимы

- отказ от использования версий ПО, про которые известно наличие уязвимостей;
- регулярное обновление ПО до безопасных версий.

Пользователю и администратору ОС следует также руководствоваться рядом правил:

- не устанавливать ПО из непроверенных источников (особенно малоизвестное ПО с закрытым исходным кодом), не допускать запуск подобного ПО без установки;
- внимательно следить за целостностью устанавливаемого и запускаемого ПО во избежание подмены (сверять контрольные суммы[1],

[1]Хеш-функции, вычисляемые от содержимого файла с низкой вероятностью коллизий (совпадения результатов) при случайном или целенаправленном осмысленном злонамеренном изменении содержимого файла. Вычисляются, например, с помощью программ `md5sum`, `shasum`, `sha256sum` и др.

электронную подпись[1], использовать защищенное соединения для загрузки[2] и др.); данные средства обычно предоставляются репозиториями и другими источниками программ — не следует их отключать и рекомендуется избегать источников, не поддерживающих данную защиту;

- использовать шифрование файлов (например, для защиты от доступа к данным домашнего каталога при изъятии носителя);
- использовать пароли *UEFI/BIOS* и загрузчика ОС для исключения загрузки иной ОС или обхода загрузки ОС путем установления нестандартного или несуществующего инициализирующего сценария в аргументе загрузки ядра;
- использовать системы **мандатного контроля доступа** (*англ.* ***mandatory access control***), например *SELinux*, *AppArmor*, *Tomoyo* и др. — во многих дистрибутивах такая система присутствует и, как правило, в каждом используется какая-то одна. Такие системы обеспечивают защиту доступа и изменения файлов в случае умышленного или случайного некорректного действия той или иной программы или пользователя;
- использовать межсетевой экран для защиты от запуска несанкционированного сервера или несанкционированного доступа к серверу по внешней сети[3].

Следует также отметить еще одно важное правило — регулярно делать аварийные копии данных. К сожалению, никто не застрахован от сбоя носителя информации, случайного удаления или порчи вредоносными программами[4]. Экономия на носителях информации рано или поздно обязательно обернется потерей данных или огромными расходами на их восстановление, поэтому никогда не оправдана. Для создания аварийных копий можно использовать специализированные команды *rsnapshot*,

[1]Позволяет проверить целостность и авторство программы. Репозитории обычно поддерживает автоматическую проверку. Вручную осуществляется, например, программой `gpg`

[2]Позволяет проверить достоверность сайта-источника загрузки и исключить перехват и изменение загружаемого файла при передаче.

[3]Позволяет, например, разрешать подключение только к выбранным портам, открытых только выбранными процессами, с помощью только выбранных протоколов, только по выбранным сетевым картам. Межсетевой экран ядра *Linux* — `iptables` — может быть настроен одноименной командой (требуется специальное изучение структуры правил) или из графического интерфейса, например программой `firewall-config`.

[4]Поэтому важно хранить аварийные копии на отключенных от компьютера и сети носителе.

rdiff-backup, создающие снимки состояния заданных ветвей ФС на каждый момент запуска и другие, а также *rsync* для простой синхронизации содержимого двух каталогов.

От потери данных при отказе носителя информации может защитить использование ***RAID***-массива (*англ.* ***Redundant Array of Independent Disks*** — *избыточный массив независимых дисков*), когда, например, вместо одного диска используется два идентичных диска при полном автоматическом дублировании сохраняемой на них информацией. Такие массивы могут создаваться аппаратно или программно с помощью команды *mdadm*. Однако *RAID* — не аварийная копия, он позволяет быстро восстановить данные при отказе носителя, но не защищает от их удаления или порчи пользователем или вредоносным ПО.

Заключение

Данное пособие не претендует на всю полноту описания *Unix*-подобных систем, их настройки и администрирования, всех возможностей оболочки и других команд. Многое осталась за рамками пособия по разным причинам — из-за отсутствия необходимости углубляться в подробности (полное описание опций всех команд, даже таких как `ls`, вывод цветных сообщений сценариев и приглашений оболочки, подробный разбор команд `eval` и `xargs`), из-за привязанности к дистрибутивам (команды для установки, удаления и проверки целостности программ), из-за быстрого устаревания (настройка загрузчика, сценариев запуска, программ и служб системы), из-за выпадения за основную тему курса (настройка серверов, использование утилит и файловой системы ядра для управления компьютером и устройствами) и другим.

Авторы надеются, что пособие станет полезной отправной точкой для дальнейшего изучения работы в *Unix*-подобных системах и их администрирования, основой для дальнейшего изучения литературы, документации и советов сообщества.

Изучение чего-либо невозможно без практики, поэтому читателям рекомендуется совершенствовать свои навыки, исполняя команды и экспериментируя с ними, стремясь к улучшению рабочей среды и комфорту, а также решая насущные задачи и возникающие проблемы по мере их поступления.

Литература

[1] Machtelt Garrels. Introduction to Linux. A Hands on Guide. Он-лайн `https://tldp.org/LDP/intro-linux/html/`

[2] Mendel Cooper. Advanced Bash-Scripting Guide. An in-depth exploration of the art of shell scripting. Он-лайн `https://tldp.org/LDP/abs/html`

[3] Береснев А.Л. Администрирование GNU/Linux с нуля. — СПб: БХВ-Петербург, 2010. — 559 с.

[4] Волох С. В. Ubuntu Linux с нуля. — СПб: БХВ-Петербург, 2019. — 400 с.

[5] Граннеман С. Linux. Карманный справочник. Необходимый код и команды. — М.:ВИЛЬЯМС, 2016. — 464 с.

[6] Колисниченко Д. Н. Linux: От новичка к профессионалу. — СПб: БХВ:Петербург, 2011. — 656 с.

[7] Лебединская Н. А., Лебединский Д. М. Операционная система LINUX. — СПб:СПбГУ, 2010. — 44 с.

[8] Немет, Эви, Снайдер, Гарт, Хейн, Трент, Уэйли, Бэн. Unix и Linux: руководство системного администратора. — М.: ООО “И.Д. Вильямс”, 2016. — 1312 с.

[9] Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.:

[10] Шоттс У. Командная строка Linux. Полное руководство. — СПб:ПИТЕР, 2019. — 543 с.