Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

А. В. Бородин, А. В. Бородина

ОПЕРАЦИОННЫЕ СРЕДЫ, СИСТЕМЫ И ОБОЛОЧКИ

Базовый курс

Учебное пособие

Петрозаводск
Издательство ПетрГУ
2011

Печатается по решению редакционно-издательского совета
Петрозаводского государственного университета

Р е ц е н з е н т ы:
канд. физ.-матем. наук Ю. В. Чуйко;
канд. физ.-матем. наук, доцент Р. В. Воронов

Бородин, А. В., Бородина, А. В.
Операционные среды, системы и оболочки. Базовый курс: учебное пособие / А. В. Бородин, А. В. Бородина. – Петрозаводск : Изд-во ПетрГУ, 2011. – 104 с.

Учебное пособие предназначено для студентов математического факультета первого курса специальностей «Прикладная математика», «Информационные системы и технологии», «Бизнес информатика», изучающих курс «Операционные среды, системы и оболочки».

Содержание

Введение

Владение средствами операционной системы и командным интерпретатором является необходимой компетенцией специалиста в области информационных технологий, в первую очередь - специалиста в области разработки программного обеспечения.

Материалы пособия используются в Петрозаводском государственном университете на математическом факультете в следующих курсах:

- «Операционные оболочки» в рамках дисциплины «Информатика» для студентов I курса направления «Прикладная математика и информатика» и специальности «Информационные системы и технологии» (с 2005 г.);
- «Операционные оболочки» в рамках дисциплины «Операционные системы, среды и оболочки» для студентов II курса очного направления «Бизнес-информатика» (с 2007 г.).

Изначально курс базировался на материалах учебного пособия, подготовленного Ю. А. Богоявленским, О. Ю. Богоявленской и Г. С. Сиговцевым [4], представляющего вводный пользовательский курс по работе в ОС Unix System V. Затем А. В. Бородиным и А. В. Бородиной был разработан дистанционный курс в системе WebCT «Командный интепретатор Bourne Shell». Сейчас курс является необходимым связующим звеном между различными дисциплинами, в частности: «Информатика», «Архитектура ЭВМ», «Операционные системы», «Системное программирование».

С переходом на ОС Linux основным предметом изучения в курсе «Операционные оболочки» стал интерпретатор Bourne again shell (bash). Значительное развитие средств командного языка и изменение состава стандартного набора утилит обусловили появление настоящего пособия.

Данное издание представляет вводный курс по работе с командным интерпретатором bash в ОС Linux и предполагает освоение рабочей среды на уровне опытного пользователя.

Для обучения shell-программиста необходима дополнительная, более специализированная, информация, тем не менее основывающаяся на элементах языка shell, описанных в настоящем учебном пособии.

§ 1. Командный интерпретатор

В данном параграфе речь пойдет о выборе и использовании *командного языка* (или, иначе, *языка оболочки операционной системы*) как базового языка, обеспечивающего взаимодействие между пользователем (программистом) и операционной системой. Приведем определения, ставшие уже классическими.

Определение 1.1. *Оболочка операционной системы (от англ. shell оболочка) — это интерпретатор команд операционной системы (ОС), обеспечивающий интерфейс для взаимодействия пользователя с функциями системы.*

Определение 1.2. *Пользовательский интерфейс — это аппаратно-программные средства, обеспечивающие взаимодействие пользователя с элементами системы компьютера.*

Оболочки делятся на два класса в зависимости от типа используемого интерфейса:

CLI — с командным интерфейсом (сокр. от Command Line Interface);

GUI — с графическим интерфейсом (сокр. от Graphical User Interface)[1].

В контексте рассматриваемой темы нас больше интересует командный CLI-интерфейс. За реализацию командного интерфейса в ОС отвечает *командный интерпретатор*, который представляет собой самостоятельный язык программирования с собственными функциональными возможностями и достаточно простым синтаксисом. Например, в ОС MS-DOS и ОС Windows 9x в качестве командного интерпретатора используется `command.com`, в ОС Windows NT — интерпретатор `cmd.exe`. В ОС типа UNIX/Linux используется командный интерпретатор shell, а точнее, его разновидности. Рассмотрим некоторые из них:

- **sh** — Bourne shell[2], самая первая командная оболочка ОС UNIX, распространялась платно;

[1] Об истории развития графических интерфейсов см. в [13].

[2] Разработана Стивеном Борном для компании AT&T.

- **ksh** — оболочка Korn shell[3], совместимая с sh, распространялась платно до 2000 г.;
- **bash** — оболочка Bourne again shell[4], самая популярная на данный момент для ОС Linux, является модернизированной версией оболочки sh, распространяется свободно;
- **zsh** — Z shell[5], современная оболочка для ОС Unix, совместимая с sh, не входит в проект GNU, но распространяется свободно;
- **csh** — оболочка C shell[6], встроенный язык которой имеет синтаксис языка Си.

Прежде чем перейти к обсуждению назначения и функций командного интерпретатора, ответим на два популярных вопроса.

1) Почему именно Linux?

2) Почему именно bash?

Почему именно Linux? ОС Linux является свободнораспространяемым программным обеспечением (ПО), разработанным в рамках проекта GNU (сокр. от «GNU is Not Unix»), и в последние годы приобретает все большую популярность. Проект GNU занимается разработкой и распространением свободного программного обеспечения [19]. Первыми разработками проекта GNU были программы, совместимые с ОС UNIX[7]. Более подробно об идеях свободно-распространяемых программ можно прочитать в книге создателя проекта GNU Ричарда Столлмана [12].

Ввиду популярности и доступности программ проекта GNU, большинство учебных курсов ПетрГУ для программистов ориентированы на использование ОС Linux. Выбирая ОС Linux, мы также выбираем и пакет совместимых программ и утилит GNU: компиляторы, отладчики, редакторы, Web-сервер, сервер баз данных и т. д.

[3]Разработана Дэвидом Корном для лаборатории AT&T.

[4]Разработана Брайаном Фоксом для проекта GNU.

[5]Разработана Паулем Фалстадом, развивается под руководством Петера Стефенсона.

[6]Разработана в Университете Беркли для проекта BSD Unix.

[7]ОС, разработанная компанией AT&T, распространяемая платно.

Почему именно bash? Для обеспечения переносимости и совместимости различных приложений существуют стандарты, в рамках которых разрабатывается программное обеспечение. Стандарт POSIX описывает множество базовых системных сервисов, необходимых для функционирования прикладных программ [3]. Он состоит из нескольких частей. Часть 3 содержит описание *интерфейса системных сервисов на уровне командного языка и служебных программ* и обозначена POSIX.2 в соответствии с нумерацией документов IEEE и ISO/IEC [18]. Стандарт POSIX.2 описывает так называемый **POSIX shell**.

Командный язык bash соответствует стандарту POSIX.2, следовательно, с помощью bash можно создавать POSIX-приложения, которые являются мобильными и легко переносимыми.

Считается, что основой стандартного интерпретатора POSIX shell был ksh, который не являлся свободнораспространяемым продуктом и, следовательно, не мог использоваться в ОС Linux (хотя свободная реализация pdksh оболочки ksh существует).

Среди всех разновидностей командных интерпретаторов shell для ОС Linux, bash выбран в качестве стандартной оболочки и используется по умолчанию [17]. Сам по себе интерпретатор bash — это достаточно мощная оболочка, объединяющая в себе функции ksh и csh, поддерживающая больше возможностей, чем предполагается стандартом POSIX. Однако bash способен эмулировать стандартный POSIX shell, что, собственно, и происходит при запуске оболочки ОС Linux. А именно: файл, который запускает POSIX shell в ОС Linux (чаще всего это `/bin/sh`), является ссылкой на реальный bash (`/bin/bash`). В этом случае bash запускается в режиме эмуляции стандартного POSIX shell, теряя при этом некоторые свои функции.

Внимание! Далее слова shell и bash будем считать синонимами.

1.1. Что такое shell?

Соберем воедино все вышесказанное. Итак, само понятие shell включает в себя несколько интепретаций.

Во-первых, shell — это **макропроцессор** для исполнения команд. Напомним определение [1].

Определение 1.3. *Макропроцессор — это программа, выполняющая преобразование входного текста в выходной путем замены последовательностей символов по правилам макроподстановки.*

Чаще всего в качестве правила макроподстановки используется замена определенной строки (макрокоманды) другой строкой, возможно, с использованием параметров. Правила макроподстановки могут включать также определение процедур и функций, вычислительные алгоритмы и т. д.

Во-вторых, shell — это **командный интерпретатор**, с помощью которого можно вызывать команды оболочки ОС и, тем самым, взаимодействовать с богатым набором утилит GNU и встроенных команд.

В-третьих, shell — это **язык программирования**, который позволяет комбинировать команды посредством типовых конструкций языка программирования и создавать новые команды, реализованные в виде сценариев shell (shell-скрипты).

1.2. Зачем shell программисту?

Для ответа на вопрос давайте разберемся в сути отношений между командной оболочкой shell и разными категориями пользователей.

Для обычного пользователя. Основное назначение интерпретатора командной строки — это обеспечение взаимодействия пользователя и системы. Другими словами shell — основной инструмент для решения пользовательских задач, например, таких как запуск пользовательских приложений; создание, удаление файлов и каталогов; редактирование текстовых документов; обеспечение доступа к внешним носителям и т. д.

Необходимо отметить, что в операционных системах семейства UNIX/Linux язык командной оболочки используется гораздо чаще, чем привыкли пользователи, например, операционной системы Windows, где доминируют приложения с графическим интерфейсом — так сложилось исторически.

Для администратора. Приложения командной строки часто востребованы для решения задач администрирования системы. К таким задачам относится настройка конфигурационных файлов системы,

управление доступом пользователей к ресурсам системы, наблюдение за работой системы, поддержка сети и сетевых сервисов и т. д. (более подробно см. в работе [2]). Справедливости ради отметим, что в ОС Linux для администрирования имеются достаточно популярные утилиты с графическим интерфейсом (например, YaST в openSUSE), избавляющие администратора от взаимодействия с командной оболочкой напрямую.

Для программиста. В функции программиста входит создание средств для решения задач как простого пользователя, так и администратора системы. Поэтому написание программ в той или иной операционной системе требует от программиста знания не только особенностей функционирования системы, но и, в первую очередь, командного языка как основного инструмента. Язык shell входит в базовый инструментарий программиста в ОС Linux и хорош для написания небольших приложений или решения отдельных подзадач.

Согласно [7] shell нецелесообразно использовать, например, в следующих случаях:

- для ресурсоемких задач (поиск, сортировка и т. д.);
- для выполнения математических вычислений с повышенной точностью;
- для создания кроссплатформенных приложений;
- для сложноструктурированных приложений (включающих, например, контроль за типами переменных, прототипы функций и т. д.);
- для обеспечения целостности системы и защиты ее от взлома;
- для задач, выполняющих огромный объем работ с файлами или многомерными массивами;
- для GUI-приложений;
- для прямого доступа к аппаратуре компьютера.

1.3. Режимы работы командного интерпретатора shell

Командный интерпретатор shell может работать в двух режимах: **интерактивном** и **пакетном**.

1.3.1. Интерактивный режим

Интерактивный режим подразумевает диалог shell и пользователя, примерно следующий:

Пример 1.1

```
shell: приглашение
user: команда 1
shell: приглашение
user: команда 2
shell: приглашение
```

Последнее слово (приглашение) всегда остается за shell. Примером работы в интерактивном режиме является процесс авторизации пользователя в системе, т. к. при входе пользователя в систему запускается экземпляр оболочки shell (т. н. login shell). Опишем основные этапы диалога более подробно.

Этап 1: приглашение. После запуска экземпляра командного интерпретатора shell выводит на экран **приглашение командной строки** — это является началом диалога. Приглашение командной строки может выглядеть, например, так:

Пример 1.2

```
# 1: Для домашнего каталога пользователя pupkin
pupkin@kappa:~>
# 2: Для домашнего каталога пользователя pupkin
pupkin@kappa:/home/pupkin>
# 3: Вид приглашения для главного каталога
pupkin@kappa:/>
```

Можно заметить, что строка-приглашение содержит информацию об имени пользователя (`pupkin`), имени сервера (`kappa`) и текущем каталоге, в котором в данный момент находится пользователь. Для случаев 1 и 2 из примера 1.2 текущий каталог один и тот же: `/home/pupkin`, он является **домашним** для пользователя pupkin и имеет специальное обозначение $\sim$ (тильда). В случае 3 текущим является **главный** каталог с именем `/`. Подробнее об иерархии каталогов см. в § 6.

На самом деле, вид приглашения командной строки можно настраивать по собственному желанию (изменив значение переменных окружения PS1 и PS2). Подробнее о переменных окружения см. в § 7.

Этап 2: команда. Пользователь вводит команду, интерпретатор — выполняет. Как происходит запуск команды? Для всех команд, которые не являются встроенными, при выполнении запускается подоболочка shell (точная копия текущей интерактивной оболочки shell), и уже она запускает на исполнение введенную пользователем команду (подробнее см. § 2).

Этап 3: приглашение. После завершения выполнения команды снова выводится приглашение командной строки.

1.3.2. Пакетный режим

Пакетный режим не требует вмешательства пользователя. Интерпретатор считывает и исполняет команды из строки или построчно из файла сценария shell.

1.4. Набор команд начинающего пользователя

Поначалу работа в командной строке вызывает некоторые трудности. Уверенность приходит с освоением частоиспользуемых команд для решения «повседневных» задач. Опишем некоторые из них в таблице.

Частоиспользуемые команды Таблица 1

№	Команда	Действие
1	`man команда`	вызов страницы руководства для команды
2	`passwd`	смена пароля
3	`whoami`	вывод имени текущего пользователя
4	`pwd`	текущий каталог
5	`du -sf ~`	использование дискового пространства файлами в домашнем каталоге
6	`ls, ls -l`	вывод информации о файлах и каталогах текущего каталога. Если в качестве параметра указан файл (каталог), то выводится информация о файле (файлах этого каталога)
7	`ls -d каталог`	вывод информации о каталоге
8	`cd каталог`	переход в каталог. `cd ..` — переход в каталог на уровень выше
9	`touch файл`	создает пустой файл
10	`mkdir каталог`	создает каталог
11	`rmdir каталог`	удаление пустого каталога
12	`rm -r имена`	рекурсивное удаление файлов и каталогов
13	`cp что куда`	копирование файлов и каталогов
14	`mv что куда`	перемещение файлов и каталогов
15	`recode к1..к2 файлы`	перекодировка из к1 в к2. Варианты кодировок: utf8, koi8-r, cp1251, cp866
16	`iconv -f к1 -t к2 файл > к2_файл`	перекодировка из к1 в к2. С перенаправлением `>` выводит результат в файл `к2_файл`, иначе — на экран
17	`pine`	чтение электронной почты
18	`mc`	визуальная оболочка, внешне похожая на FAR
19	`mcedit, emacs, vi`	текстовые редакторы

1.5. Удаленный доступ к учебному серверу kappa

На кафедре ИМО запущен учебный сервер kappa.cs.karelia.ru (далее kappa) с возможностью удаленного доступа. После регистрации в системе студент имеет возможность работать в дисплейных классах ПетрГУ в ОС Linux как локально, так и удаленно. Для обычного входа в систему достаточно ввести логин и пароль в окне, предложенном загрузчиком ОС.

Для **удаленного** взаимодействия с учебным сервером kappa через SSH-протокол (от англ. Secure Shell) необходим ssh-клиент. В качестве ssh-клиента в ОС Windows можно использовать программу `putty.exe`. В ОС Linux есть свой ssh-клиент OpenSSH, который можно запустить командой `ssh`.

Пример 1.3

```
# Запуск ssh-клиента для пользователя user в ОС Linux
user@epsilon:~> ssh user@kappa.cs.karelia.ru
Password:
```

Для **файлового обмена** данными в ОС Windows с сервером kappa можно использовать программу WinSCP. WinSCP — это графический клиент SFTP (от англ. SSH File Transfer Protocol), предназначенный для защищенного обмена файлами между компьютером и серверами. Есть также отдельный WinSCP-модуль для FAR-менеджера. Для файлового обмена в ОС Linux можно использовать утилиты `sftp`, `scp`.

Все файлы пользователь должен хранить в своем домашнем каталоге, владельцем которого он является и права на который он может менять. Полный путь к домашнему каталогу: `/home/имя_пользователя`. Подробнее об иерархической структуре файловой системы см. в § 6.

§ 2. Простая команда shell

Приведем несколько формальных определений, придерживаясь терминологии основного руководства Linux [15].

Определение 2.1. *Пробельный разделитель — это пробел или табуляция.*

Определение 2.2. *Управляющий оператор — это символ перевода строки или один из операторов* || && & ; ;; | ( )

Определение 2.3. *Слово — последовательность символов, не являющаяся оператором, ограниченная пробельными разделителями.*

Определение 2.4. *Простая команда — это последовательность слов, разделенных пробельными разделителями, ограниченная управляющим оператором.*

Предполагается, что управляющий оператор не является словом.

Простая команда — это основная «единица» при работе в командной строке. Общая структура простой команды предполагает наличие имени команды и последующих параметров, введенных через пробельный разделитель:

```
:~> Имя_команды Параметры
:~> Имя_команды Опции Аргументы
```

Имя команды соответствует программе (*реализации* команды), которая хранится на диске и запускается на выполнение при вводе команды в командную строку. Параметры можно разбить на две группы: *опции* (*опциональные* параметры) и *аргументы* (*неопциональные* параметры).

Опции управляют поведением программы. Они делятся на короткие и длинные. Короткие опции начинаются с символа `-`, длинные — с `--`. Для некоторых коротких опций имеются длинные аналоги. Так, для команды wc опции `-l` соответствует `--lines`. Опция может иметь аргумент. Например, для команды `find -type d` опция `-type` с аргументом `d` указывает на поиск файлов, которые являются директориями. У некоторых команд есть специальная опция `--` для завершения списка опций.

Аргументы команды могут быть обязательными (`имя_файла` для команды `rm`) и необязательными (`имя_файла` для команды `ls`). Чаще всего аргументы задают входные и выходные данные.

2.1. Встроенные и внешние команды

Команды shell делятся на две категории:

1) *встроенные* (внутренние) команды shell (SHELL BUILTINS);

2) *внешние* программы (GNU SHELL-UTILS).

Основное отличие внешней команды от внутренней заключается в *порождении нового подпроцесса* [7].

Определение 2.5. *Ветвление процесса (forking) — это действие, когда либо команда, либо командная оболочка порождает новый подпроцесс, чтобы выполнить какую-либо работу. Порождающий процесс называется* **родительским**, *или* **предком**. *Порожденный —* **дочерним**, *или* **потомком**.

После ветвления родительский и дочерний процессы продолжают выполняться параллельно (независимо по времени).

Встроенные команды shell [14] не порождают новый подпроцесс, а выполняются в текущей оболочке shell. Встроенная команда выполняется быстрее и является более производительной. Реализацию встроенной команды выполняет программа `bash` (исполняемый файл `/bin/bash`). Количество встроенных команд может быть разным в зависимости от версии командного интерпретатора bash. Полный список встроенных команд для bash 4.1 приведен в таблице 2.

Внешние программы [16] порождают дочерний процесс и выполняются в подоболочке shell (subshell). Реализация внешней команды — это отдельный исполняемый файл (программа), который находится в директории `/bin`. Например, внешняя команда `echo`[8] реализуется программой `/bin/echo`.

[8]Команда `echo` — вывод строки на экран.

Список встроенных команд bash 4.1 Таблица 2

:	.	[	alias	bg	bind	break
builtin	caller	cd	command	compgen	complete	compopt
continue	declare	dirs	disown	echo	enable	eval
exec	exit	export	false	fc	fg	getopts
hash	help	history	jobs	kill	let	local
logout	mapfile	popd	printf	pushd	pwd	read
readarray	readonly	return	set	shift	shopt	source
suspend	test	times	trap	true	type	typeset
ulimit	umask	unalias	unset	wait		

Встроенная команда может иметь внешние аналоги с таким же именем (например, команда `echo`). Как правило, интерпретатор shell по умолчанию выбирает встроенный вариант (см. пример 2.3). Однако можно запретить использование внутренней команды (см. пример 2.6) или вызвать внешнюю команду вместо внутренней, указав полный путь к внешней программе, как показано в примере 2.1.

Пример 2.1

```
# Вызов внешней команды echo вместо встроенной
:~> /bin/echo "Я - внешняя!"
```

Ниже приводится набор полезных команд:

- **type имя** — показывает вариант интерпретации команды по умолчанию. Если команда по умолчанию будет проинтерпретирована как встроенная, то выводится `"имя is a shell builtin"`. Если команда по умолчанию интерпретируется как внешняя, то выводится текущая реализация команды (см. примеры 2.2, 2.3, 2.4);
- **type -a** — проверка, дублируется ли встроенная команда внешней;
- **man bash** — список встроенных команд (BASH BUILTIN COMMANDS);
- **help имя** — справка по встроенным командам;
- **which имя** — вывести путь к внешней программе;

- **enable** — вывести список всех разрешенных встроенных команд;
- **enable -n имя** — запретить использование встроенной команды;
- **enable имя** — разрешить использование встроенной команды.

Внимание! Команда `enable` является встроенной и, как правило, не дублируется внешней. Поэтому не рекомендуется применять команду `enabel -n` к самой себе. Приведем несколько примеров использования команд, описанных выше.

Пример 2.2

```
# Для встроенной команды, не имеющей внешнего аналога
:~> type -a type
type is a shell builtin
:~> type type
type is a shell builtin
```

Пример 2.3

```
# Для встроенной команды, имеющей внешний аналог
:~> type -a echo
echo is a shell builtin
echo is /bin/echo
:~> type echo
echo is a shell builtin
```

Пример 2.4

```
# Для внешней команды, не имеющей встроенного аналога
:~> type -a grep
grep is /usr/bin/grep
grep is /bin/grep
grep is /usr/bin/X11/grep
:~> type grep
grep is /usr/bin/grep
```

В примере 2.4 команда `grep`[9] является внешней и имеет три реализации. По умолчанию командный интерпретатор будет использовать реализацию `/usr/bin/grep`.

Вместо команды `type` для вывода пути к внешней программе можно использовать команду `which` (сравните примеры 2.3, 2.4 и 2.5).

Пример 2.5

```
# Путь к реализации внешней команды
:~> which echo
/bin/echo
:~> which grep
/usr/bin/grep
```

Пример 2.6

```
# Запрет использования встроенной команды echo
:~> enable | grep echo
enable echo
:~> enable -n echo
:~> enable | grep echo
:~>
```

Пример 2.7

```
# Разрешение использования встроенной команды echo
:~> enable echo
:~> enable | grep echo
enable echo
```

Пояснение. Для удобства в примерах 2.6 и 2.7 используется конвейер[10]

```
enable | grep echo
```

[9] Команда `grep` — поиск в тексте по образцу, см. § 8.

[10] Более подробно см. в § 5.

Такая конструкция означает, что в тексте, который выводит на экран команда `enable` (список разрешенных встроенных команд, каждая команда — в отдельной строке), команда `grep` найдет строку, содержащую слово `echo`, и выведет ее на экран. В примере 2.6 — это строка

```
enable echo
```

Если команды `echo` не окажется в списке разрешенных команд, то команда `grep`, выполнив поиск, ничего не выводит, и выдается приглашение командной строки для ввода следующей команды (см. пример 2.6).

2.2. Псевдонимы

Если требуется часто запускать одни и те же команды с большим количеством параметров, то удобно использовать *псевдонимы*.

Определение 2.6. *Псевдоним — это синоним команды, определяемый командой* `alias`.

Команда `alias`, вызываемая без параметров, выводит список всех определенных синонимов. Часть из них определены пользователем, часть являются *предопределенными*. Ниже представлен пример создания и удаления пользовательских псевдонимов. Одна и та же команда может иметь несколько псевдонимов.

Пример 2.8

```
# Создание псевдонима команды ls
:~> alias ls="ls -a"
# Удаление псевдонима
:~> unalias ls
```

Предопределенные синонимы команд инициализируются при запуске командной оболочки, соответствующие команды `alias` для них приписаны в конфигурационных файлах (в каких именно — зависит от операционной системы). Наиболее часто используемые предопределенные псевдонимы:

- **..="cd .."** — переход из текущего каталога на уровень выше;
- **...="cd ../.."** — переход на два каталога выше;
- **dir="ls -l"** — вывод содержимого текущего каталога в виде расширенного списка;
- **ll="ls -l"** — аналогично dir;
- **la="ls -la"** — аналогично dir с выводом скрытых файлов (имена которых начинаются с символа '.');
- **o="less"** — постраничный просмотр текстовых файлов;
- **rd="rmdir"** — удаление пустого каталога.

Внимание! Вместо двойных кавычек "заменяемую команду-строку можно заключать в одинарные '.

2.3. Код возврата

Известно, что при завершении любая программа возвращает в операционную систему *код возврата*, или *статус выхода* (от англ. exit status). Команда shell, введенная в консоли, также возвращает код возврата текущей оболочке shell.

Определение 2.7. *Код возврата — это число в диапазоне 0 — 255, которое определяет успешность выполнения запущенной программы (команды).*

Завершение **с кодом ноль** (`0`) является **успешным**, **ненулевое** завершение означает **неудачу**. Также определены следующие специальные значения кода возврата [15]:

- **127** — команда не найдена;
- **126** — команда найдена, но не может быть выполнена;
- **128 + N** — прерывание выполнения команды по сигналу с номером `N`. Например, сигналу `SIGTSTP`, посылаемому с терминала (`Ctrl + Z`) для приостановки процесса выполнения команды, будет соответствовать `N = 20`;

- **2** — некорректное использование встроенной команды shell.

Управлять значением кода возврата в скрипте shell можно с помощью встроенной функции `exit`. Код возврата последней команды хранится в специальной переменной `$?`, значение которой можно вывести на экран (см. пример 2.9) или использовать в сценариях shell.

Пример 2.9

```
# Вывод кода возврата последней команды
:~> echo $?
```

§ 3. Состав командной строки

В предыдущих параграфах мы выяснили, что же такое командный интерпретатор shell и как выглядит инструкция языка командной оболочки shell. Пока ограничимся рассмотрением только **простых** команд языка shell. Речь об объединении простых команд в списки и о структуре сложной команды shell пойдет в § 5.

Как shell выполняет разбор командной строки? Прежде чем ответить на этот вопрос, необходимо понять, **что** может встретить интерпретатор shell в командной строке при ее разборе.

Цель данного параграфа — описать, какие еще **средства языка** можно использовать в командной строке shell, кроме простых команд. Перечислим основные, наиболее часто используемые:

- метасимволы,
- подстановка команд,
- перенаправление ввода-вывода,
- подстановки в именах файлов (шаблоны),
- регулярные выражения.

Причем шаблоны и регулярные выражения присутствуют в командной строке не сами по себе, а являются аргументами какой-нибудь команды shell.

3.1. Метасимволы

В каждом языке, как правило, имеются символы, которые в разных условиях ведут себя по-разному. Именно поэтому они называтся *мета*символами.

Внимание! Набор таких символов для каждого языка свой. Приведем стандартное определение.

Определение 3.1. *Метасимвол — это символ, принимающий в зависимости от контекста несколько различных значений.*

Для нас интересны следующие группы метасимволов:

- метасимволы языка shell;
- метасимволы в шаблонах имен файлов;
- метасимволы языка регулярных выражений.

Интерпретатор bash способен распознавать все эти группы метасимволов, однако делает он это **на разных стадиях** разбора командной строки (более подробно об этом см. в § 4). Различные варианты толкования метасимволов могут приводить к ошибкам, когда пользователь подразумевает одно значение, а shell назначает другое. Рассмотрим подробнее две первые группы. Метасимволы регулярных выражений имеет смысл изучить непосредственно перед их использованием в § 8.

3.1.1. Метасимволы языка shell

Точный список метасимволов bash можно посмотреть в руководстве [15] или выполнив команду `man bash`. Метасимволом bash считается *пробел*, *табуляция* и символы:

```
| & ; ( ) < >
```

При разбиении командной строки на слова эти символы считаются границами слов. Простейшим примером метасимвола shell является пробел. Обнаруживая пробел, интерпретатор считает текущее слово законченным. Например, создав файл с именем `my file` (именно так, с пробелом) с помощью команды `touch my file`, мы обнаружим два файла: `my` и `file` вместо одного предполагавшегося.

Пример 3.1

```
# Имя файла с пробелом (не экранируется)
:~> touch my file
:~> ls -l
-rw-r--r-- 1 user    group    0 Июн 30  2011 file
-rw-r--r-- 1 user    group    0 Июн 30  2011 my
```

3.1.2. Метасимволы в шаблонах имен файлов

Метасимволы шаблонов используют для задания группы файлов в виде параметра какой-либо команды. Таким образом, одну команду можно применить сразу к нескольким файлам, имена которых удовлетворяют шаблону. Метасимволами шаблонов являются:

- `*` — любая (в том числе и пустая) последовательность символов, не являющаяся '.'. Символ '.' в начале имени файла должен указываться явно;
- `?` — любой одиночный символ;
- `[...]` — любой из заключенных в скобки символов;
- `[!...]` — любой символ, не совпадающий с символами, заключенными в скобки;
- `[^...]` — аналогично предыдущему;
- `[:класс:]` — класс символов POSIX (см. таблицу 3).

Для унификации описания наборов символов стандартом POSIX зарезервированы так называемые классы символов.

Классы символов POSIX Таблица 3

№	Класс	Назначение
1	`[:alpha:]`	Все буквы английского алфавита в обоих регистрах
2	`[:alnum:]`	Все буквы английского алфавита в обоих регистрах и арабские цифры
3	`[:digit:]`	Все арабские цифры
4	`[:cntrl:]`	Все управляющие символы
5	`[:blank:]`	Все горизонтальные пробельные разделители
6	`[:xdigit:]`	Все шестнадцатеричные цифры
7	`[:print:]`	Все печатные символы, включая пробел
8	`[:punct:]`	Все знаки пунктуации
9	`[:space:]`	Все пробельные разделители
10	`[:lower:]`	Все буквы английского алфавита в нижнем регистре
11	`[:upper:]`	Все буквы английского алфавита в верхнем регистре

Если интерпретатор встречает один из метасимволов шаблонов, он производит **подстановку** значения, соответствующего символу. Поэтому чаще метасимволы в шаблонах называют **подстановками в именах файлов**, что соответствует действиям интерпретатора (более подробно о разборе подстановок см. в § 4).

3.2. Средства экранирования

Поскольку в разных случаях может потребоваться разная интерпретация (специальное значение либо обычное значение) метасимвола, то необходим инструмент, управляющий выбором нужного значения.

Определение 3.2. *Экранирование (quoting) — отмена специального значения метасимвола.*

Экранирование заставляет интерпретатор трактовать метасимвол как обычный символ. Можно выделить три варианта экранирования символов:

1) эскейп-экранирование;
2) использование кавычек;
3) экранирование в стиле ANSI-C.

3.2.1. Эскейп-экранирование

Эскейп-экранирование — это использование символа обратной косой черты (\, backslash) для экранирования метасимвола, следующего за \. Символ \ сообщает интерпретатору, что следующий за ним метасимвол должен восприниматься как обычный.

Сам символ обратной косой черты называется *эскейп-символом* (от англ. escape character), а обратная косая черта вместе со следующим за ней символом — *эскейп-последовательностью*. Как и в примере 3.1, выполним команду `touch`, только экранируем символ пробела.

Пример 3.2

```
# Имя файла с пробелом (экранируется)
:~> touch my\ file
:~> ls -l
-rw-r--r-- 1 user    group    0 Июн 30  2011 file
-rw-r--r-- 1 user    group    0 Июн 30  2011 my
-rw-r--r-- 1 user    group    0 Июн 30  2011 my file
```

Пример 3.3

```
# Вывод символа ;
:~> echo ;

:~> echo \;
;
```

Эскейп-экранирование — достаточно трудоемкий способ, если нужно экранировать сразу несколько символов.

Внимание! Для некоторых команд (например, `echo` и `sed`) символ \ выполняет обратное действие — переключает обычное значение символа на специальное. Например, `\n` для команды `echo` с опцией `-e` означает перевод строки.

3.2.2. Экранирование кавычками

Экранировать символы можно с использованием одинарных или двойных кавычек.

Одинарные кавычки отменяют специальные значения всех заключенных между ними метасимволов. Однако между ними не должно быть внутренней одинарной кавычки, иначе она будет восприниматься как закрывающая.

Двойные кавычки экранируют все метасимволы кроме \, $ и обратной кавычки `. Символ обратной косой черты внутри двойных кавычек сохраняет особое значение только в том случае, если предшествует символам $, `, ", \ или символу перевода строки.

Пример 3.4

```
# Экранирование двойными кавычками
# 1: \ - теряет специальное значение, $ - сохраняет
:~> echo "a \ b $c"
a \ b
# 2: \ - сохраняет специальное значение, $ - теряет
:~> echo "b \$c"
b $c
```

Пояснение. Символ $ используется с позиционными параметрами и определенными пользователем переменными как простая подстановка значения переменной (см. § 7). В примере 3.4 в случае 1 символ $ означает начало имени переменной. Интерпретатор обращается к переменной `$c` и подставляет в строку ее значение (пустое).

Пример 3.5

```
# 1: $ - теряет специальное значение
:~> echo 'b $c'
b $c
# 2: $ и ' - теряют специальное значение
:~> echo "b '$' c"
b '$' c
```

Если не экранировать знак $ внутри двойных кавычек, то он сохраняет специальное значение (означает простую подстановку значения переменной `c`). Сравните случаи 2 в примерах 3.4 и 3.5; в обоих случаях мы пытаемся экранировать символ $.

В следующем примере показано, как интерпретатор реагирует на нечетное число двойных (одинарных) кавычек. После ввода команды `echo` и нажатия клавиши `Enter` выдается повторное приглашение командной строки (символ `>`). Таким образом интерпретатор дает понять, что предыдущая команда не закончена. В данном случае ожидается ввод символа двойных кавычек. Получив то, что хотел, интерпретатор выводит результат. К строке добавляются еще два символа перехода на новую строку, соответствующих нажатию клавиши `Enter`.

Пример 3.6

```
# Нечетное число двойных кавычки - ожидание
:~> echo "a"b"[Enter]
> [Enter]
> "
ab
[пустая строка]
[пустая строка]
```

Если же мы хотим вывести символ двойной кавычки внутри двойных кавычек, то его следует экранировать. Например, так:

Пример 3.7

```
# Вывод двойной кавычки внутри двойных
:~> echo "a\"b"
a"b
```

А вот такое экранирование двойной кавычки, как в примере 3.8, не пройдет. Однако и в этом случае поведение интерпретатора вполне предсказуемо. Итоговая строка будет состоять из двух частей: первая подстрока `a'` заключена в двойные кавычки (специальное значение символа `'` теряется); вторая подстрока `b"[Enter]` заключена в одинарные кавычки (специальное значение символа `"` теряется).

Пример 3.8

```
# Неверное экранирование внутренней двойной кавычки
:~> echo "a'"'b"
> [Enter]
> '
a'b"
[пустая строка]
```

3.2.3. Экранирование в стиле ANSI-C

Экранирование в стиле ANSI-C имеет следующий синтаксис:

```
$'эскейп-последовательность'
```

и позволяет подставлять последовательности, определенные стандартом языка Си.

Например, символы форматирования (\n, \t); символы, заданные восьмеричным кодом (\ooo); символы, заданные шестнадцатеричным кодом (\xHH).

Обратите внимание на экранирование одинарных кавычек внутри строки в примере 3.9.

Пример 3.9

```
# Примеры эскейп-последовательностей
:~> echo $'строка1\nстрока2'
строка1
строка2
:~> echo $'пробел\x20имеет\x20код 20'
пробел имеет код 20
:~> echo $'код цифры \'\x30\' = 48'
код цифры '0' = 48
```

3.3. Подстановка команды

Подстановка команды означает использование *вывода команды* как обычного текста. Как правило, подстановки команд применяются в параметрах других команд либо как значения, присваиваемые переменным в скриптах. Синтаксис описания подстановки команды:

1-й вариант (в обратных кавычках): `` `команда` ``

2-й вариант: `$(команда)`

Также возможно осуществить вложение подстановок путем комбинации двух вариантов, экранируя символы обратной кавычки при необходимости.

Пример 3.10

```
# Вывод размера занимаемого дискового пространства
# 1: В домашнем каталоге
:~> echo "Объем домашнего каталога = `du -sh ~`"
Объем домашнего каталога = 90,0M /home/student
# 2: В текущем каталоге
:~/bin> echo "Объем текущего каталога = `du -sh $(pwd)`"
Объем текущего каталога = 2,7M /home/student/bin
```

Пояснение. В примере 3.10 мы используем команду `du` для определения, сколько занимают места на диске файлы домашнего и текущего каталогов. Поскольку эту информацию мы выводим с помощью

команды `echo`, необходимо `du` оформить как подстановку команды. Иначе имя команды и ее параметры будут интерпретироваться как обычный текст.

В качестве аргумента команды `du` в первом случае указано обозначение домашнего каталога (∼). Во втором случае аргументом `du` является подстановка команды `pwd`, которая выводит имя текущего каталога.

Внимание! Если подстановку команды `` `du -sh $(pwd)` `` применить к каталогу, имя которого содержит пробелы (например, `C 1`), то будет выдана ошибка. Такая неприятность связана с тем, что при выводе результата подстановки `$(pwd)` символ пробела не экранируется (как, впрочем, и другие метасимволы). Исправить ситуацию поможет экранирование с помощью двойных кавычек, они сохранят специальное значение сивола `$`, что необходимо для совершения подстановки. Экранирование с помощью одинарных кавычек в этой ситуации не годится. Можно поступить, например, так:

Пример 3.11

```
# Вывод размера занимаемого дискового пространства
# В текущем каталоге (с пробелами)
:~/C 1> echo "Объем текущего каталога= `du -sh \"$(pwd)\"`"
Объем текущего каталога= 2,7M /home/student/C 1
```

3.4. Средства перенаправления

Каждая команда shell при выполнении интерпретатором связана с набором открытых файлов для ввода (входной поток) и вывода (выходной поток) данных. Каждый из этих файлов идентифицируется определенным числом, которое называют *дескриптор файла*. Файлы стандартных потоков ввода-вывода имеют следующие дескрипторы:

`0` — стандартный поток ввода (stdin), по умолчанию ассоциирован с клавиатурой;

`1` — стандартный поток вывода (stdout), по умолчанию ассоциирован с экраном;

2 — стандартный поток вывода ошибок (stderr), по умолчанию ассоциирован с экраном.

Стандартные потоки ввода-вывода можно **перенаправить**, используя **операторы перенаправления** (переадресации), см. таблицу 3. Перенаправление означает, что при вызове команды можно указать, откуда принимать входные данные и куда направлять выходные данные и сообщения об ошибках.

Список операторов перенаправления Таблица 4

№	Оператор	Описание
1	команда > файл	Перенаправление вывода в файл (с перезаписью). Если файл не существует, то он будет создан
2	команда >> файл	Перенаправление вывода в файл (с дозаписью в конец файла)
3	команда 2> файл	Перенаправление ошибок в файл
4	команда > файл 2>&1	Связывание потоков вывода и ошибок. Перенаправляет вывод и ошибки в файл с дескриптором 1 (с перезаписью)
5	команда >& файл	Аналогично предыдущему оператору. Ошибки и вывод команды перенаправляются в файл
6	команда >> файл 2>&1	Аналогично предыдущему оператору, но с дозаписью в конец файла
7	команда < файл	Перенаправление ввода из файла
8	команда <<[-]разделитель текст разделитель	**Подстановка текста.** Перенаправление ввода для команды, если входной текст содержит символ перевода строки. Символ тире используется для удаления табуляций
9	команда <<< строка	**Подстановка строки.** Перенаправление ввода, если входной текст умещается в строку

Приведем несколько примеров наиболее часто используемых перенаправлений.

Пример 3.12

```
# Перенаправление ввода для исполняемого файла
:~> ./c_program < text.txt
# Перенаправление ввода и вывода
:~> ./c_program < text.txt > result.txt
```

В примере 3.12 мы перенаправляем ввод-вывод для исполняемого файла Си-программы. Исполняемый файл является аналогом команды shell. Файлы text.txt и result.txt — это обычные текстовые файлы.

В примере 3.13 приведен **стандартный алгоритм подавления ошибок.** В данном случае ошибки перенаправляются на символьное устройство `/dev/null` (псевдоустройство), которое часто используется для **подавления вывода**. Иногда `/dev/null` называют системной корзиной [6]. Предполагается, что запись в `/dev/null` не вызывает переполнения и всегда успешна. Чтение из файла `/dev/null` дает символы `EOF` (от англ. End Of File символ конца файла) и также всегда успешно.

Пример 3.13

```
# Подавление ошибок
:~> ./c_program 2> /dev/null
```

§ 4. Алгоритм разбора командной строки

Алгоритм разбора командной строки интерпретатором shell выполняется каждый раз при вводе команды или запуске сценария и поэтому заслуживает отдельной главы. Выделим основные действия интерпретатора в виде следующего набора шагов.

Шаг 1. Чтение входного текста (ввода) из файла, командной строки или терминала.

Шаг 2. Разбор ввода на слова и операторы в соответствии с правилами экранирования. На этом этапе все экранированные метасимволы потеряют свое специальное значение (исключением являются особые случаи, когда экранирование отменяет обычное значение метасимвола и включает специальное.)

Шаг 3. Разбор слов с выделением команд.

Шаг 4. Реализация подстановок, выделение параметров команд. На данном шаге происходит замена значений предполагаемых подстановок на реальный текст, им соответствующий. В частности, на этом этапе интерпретатор выполняет подстановки команд и подстановки в именах файлов.

Шаг 5. Реализация перенаправлений.

Шаг 6. Исполнение команд.

Шаг 7. Ожидание завершения и получение кода возврата.

Внимание! Здесь снова следует обратить особое внимание на экранирование метасимволов.

Как было отмечено ранее в § 3, разные группы метасимволов, распознаваемых интерпретатором shell, разбираются на разных этапах алгоритма. На шаге 2 учитывается набор метасимволов языка shell. Набор метасимволов языка шаблонов имен файлов учитывается на шаге 4 при реализации подстановок. В случае если метасимволы шаблонов были экранированы, подстановки не произойдет. И далее за реализацию шаблона будет отвечать команда, аргументом которой этот шаблон является.

Внимательно проанализируйте примеры, приведенные ниже. Зачастую непонимание того, когда именно нужно экранировать подстановки в именах файлов, приводит к неправильной работе команд и потере программистом времени на исправление ошибки впустую.

Рассмотрим отдельно использование символа * (звездочка).

Пример 4.1

```
# Вывод списка файлов текущего каталога с расширением .c
:~> ls -l *.c
-rw-r--r-- 1 user    group    0 Июн 30  2011 file.c
-rw-r--r-- 1 user    group    0 Июн 30  2011 my.c
```

В данном случае мы используем шаблон-подстановку в именах файлов `*.c`. Символ звездочки не экранирован, поэтому подстановка будет реализована на шаге 4. В данном случае она означает поиск **в текущем каталоге** списка файлов, чьи имена соответствуют шаблону с последующей его заменой.

А теперь рассмотрим еще один пример со звездочкой. В данном случае команда `find` выполняет поиск файлов по имени в домашнем каталоге.

Пример 4.2

```
# Поиск файлов в домашнем каталоге с расширением .c
:~> find ~ -name "*.c"
./file.c
./my.c
```

В примере 4.2 символ звездочки экранирован с помощью двойных кавычек. Это означает, что шаблон `*.c` будет аргументом для функции `find` и никакой подстановки на шаге 4 выполнено не будет. За реализацию этого шаблона отвечает команда `find` на шаге 6 алгоритма.

Внимание! Если убрать в примере 4.2 двойные кавычки, то вместо шаблона `*.c` на шаге 4 будет подставлен список файлов текущей директории (а вовсе не домашнего каталога), как и в примере 4.1. Тогда команда `find` будет выполнять поиск файлов из указанного списка в домашнем каталоге, что вряд ли соответствует замыслу автора и, скорее всего, приведет к ошибке.

4.1. Алгоритм поиска команды для исполнения

Исполнение команд происходит на шаге 6 алгоритма. К этому моменту текст командной строки уже разобран, выделены команды и определены их аргументы. Далее интерпретатор shell будет в цикле выполнять поиск реализаций, которые соответствуют именам команд, и запускать программы на выполнение. Схема поиска реализации команды примерно следующая:

1) Если имя команды не содержит символы / (slash), то shell ищет реализацию по имени команды:
   (a) среди функций. Существует ли функция shell с этим именем?
   (b) среди встроенных команд. Существует ли встроенная команда shell с таким именем?
   (c) среди внешних утилит. Существует ли файл с таким именем в *каталогах поиска* (см. пояснение ниже)?
2) Если поиск не дал результатов, то будет переход к шагу 7 алгоритма с кодом завершения 127.
3) Если поиск был успешен или в имени команды есть символы /, то:
   (a) если файл имеет исполняемый формат, то произойдет запуск файла-программы на исполнение;
   (b) если файл имеет неисполняемый формат и не является директорией, то предполагается, что это сценарий shell. Для выполнения этого сценария будет запущена подоболочка shell.
4) Если команда была запущена в интерактивном режиме (не в фоновом), то нужно ждать завершения и переходить к разбору следующей команды.

Пояснение. *Каталоги поиска* — это каталоги, прописанные в переменной окружения shell с именем PATH. Список всех каталогов поиска можно вывести командой `echo $PATH` (см. § 7).

§ 5. Структура сложной командной строки

Кроме простых команд, интерпретатор shell допускает использование более сложных конструкций, таких как конвейеры, списки команд, операторы группировки команд, управляющие операторы и т. д. Такие средства языка shell обеспечивают обмен данными или организацию выполнения команд в определенной последовательности.

5.1. Конвейер

Иногда требуется, чтобы в качестве аргумента какой-либо команды передавались данные, сгенерированные другой командой. Можно использовать для этой цели файловый обмен и операторы перенаправления, но есть более удобный способ — конвейер. Принцип действия этой конструкции оправдывает название и действительно схож с обычным производственным конвейером.

Определение 5.1. *Конвейер (pipeline) — это последовательность команд, в которой стандартный вывод предшествующей команды перенаправляется на стандартный ввод последующей.*

Общий синтаксис определения конвейера следующий:

```
[time [-p]] [!] команда1 [| команда2 ...]
```

Квадратные скобки обозначают необязательные аргументы. Применение `time` обеспечивает сбор статистики о времени выполнения конвейера (реальное время, пользовательское время, системное время). Опция `-p` управляет форматом вывода статистики.

Каждая команда в конвейере выполняется в подоболочке (как отдельный процесс). **Код возврата** конвейера совпадает с кодом возврата последней выполненной команды в конвейере. Символ ! инвертирует код возврата. Это можно использовать, например, в условных операторах.

Наиболее часто конвейер используется в минимальном варианте, т. е. выполнются только команды, разделенные оператором |, который называется **программным каналом** (pipe).

Как будет работать конвейер в случае возникновения ошибки при выполнении какой-либо команды? Рассмотрим примеры 5.1 и 5.2, для обоих выведем код возврата. Напомним, что успешным является код возврата, равный нулю.

Пример 5.1

```
# Подсчет числа встроенных команд shell
# 1: Без ошибки
:~> enable | wc -l
61
# Код возврата
:~> echo $?
0

# 2: С ошибкой в первой команде
:~> enbl | wc -l
bash: enbl: команда не найдена
0
# Код возврата
:~> echo $?
0

# 3: С ошибкой во второй команде
:~> enable | wcc -l
bash: wcc: команда не найдена
# Код возврата
:~> echo $?
127
```

В примере 5.1 конвейер состоит из двух команд и неверно задано имя команды (сначала первой, затем второй). Ошибку будет инициировать интерпретатор bash. Можно заметить, что ошибка в первой команде никак не влияет на код возврата всей конструкции. При этом вторая команда выполняется все равно, принимая от первой пустой поток входных данных. Ошибка во второй команде меняет код возврата.

В примере 5.2 укажем имя несуществующей директории. Теперь ошибку инициирует сама команда `ls`. При этом вторая команда выполняется и выдает успешный код возврата.

Пример 5.2

```
# Подсчет числа файлов и папок в директории sd
# Директории sd не существует
:~> ls -l sd | wc -l
ls: невозможно получить доступ к sd: Нет такого файла
или каталога
0
# Код возврата
:~> echo $?
0
```

5.2. Списки команд

Конвейеры могут объединяться в более сложные конструкции языка shell (списки) с помощью управляющих операторов.

Определение 5.2. *Список команд — это последовательность конвейеров, разделенных операторами* `;`, `&`, `&&`, `||` *и ограниченных* `;`, `&` *или переводом строки.*

Если команды (или списки команд) разделены оператором `;`

```
команда1 ; команда2 ; ...командаN,
```

то они выполняются **последовательно**. Интерпретатор shell будет ожидать завершения текущей команды и затем запустит на выполнение следующую. Кодом завершения всей конструкции будет код завершения последней команды (`командаN`).

Если команды (или списки команд) разделены оператором `&`

```
команда1 & команда2 & ...командаN,
```

то они будут запущены на выполнение **параллельно (независимо по времени)**. Каждая команда будет выполняться в своей подоболочке в **фоновом** режиме. Когда команда работает в фоновом режиме, то в командную строку можно параллельно вводить другие команды (см. § 6). В этом случае интерпретатор shell не будет ждать завершения выполнения фоновой команды (`команда1`, ..., `командаN-1`) и вернет нулевой код возврата. Код возврата всей конструкции будет совпадать с кодом возврата последней команды (`командаN`).

Операторы **условного выполнения** `&&` и `||` группируют команды (конвейеры команд) по принципу логического И и ИЛИ, однако **имеют одинаковый приоритет**, который ниже, чем у `&` и `;`. Конструкция

`команда1 && команда2` — означает выполнение `команды2` при **успешном** выполнении `команды1`;

`команда1 || команда2` — означает выполнение `команды2` при **неуспешном** выполнении `команды1`.

Код возврата конструкций И и ИЛИ совпадает с кодом возврата последней выполненной команды в списке.

Для условного выполнения часто используют следующую конструкцию:

```
команда1 && команда2 || команда3,
```

где при успешном выполнении `команды1` выполняется `команда2`, иначе — `команда3`.

5.3. Группировка команд

Для изменения приоритета операторов условного выполнения используют группировку команд в скобках () и {}.

Группировка команд в круглых скобках ()

```
( команда1; команда2; ... )
```

выполняется в подоболочке shell.

Группировка команд в фигурных скобках {}

```
{ команда1; команда2; ... }
```

выполняется в текущей оболочке shell.

Внимание! Содержимое скобок **обязательно** отделяют от самих скобок пробелом (см. пример 5.3).

Пример 5.3

```
# Вывод имени и группы пользователя в файл user.info
# Без пробела - ошибка
:~> {id -un; id -gn;} > user.info
bash: syntax error near unexpected token `}'
# С пробелом - правильно
:~> { id -un; id -gn; } > user.info
```

Кроме вышеперечисленных средств группировки команд, в языке shell существуют условные, циклические операторы, функции и т. д. Но применять их в командной строке не очень удобно. Такие конструкции используют в основном в сценариях shell, поэтому они описаны в § 7.

§ 6. Управление файлами и процессами

Для корректной работы в командной строке shell необходимы некоторые базовые знания об архитектуре ОС Linux, организации файловой системы, подсистеме управления процессами и т. д. В рамках данного учебного пособия нет возможности рассмотреть эти темы подробно, поэтому ограничимся небольшим объемом информации, необходимой на практике.

6.1. Двухуровневая архитектура ОС Linux

Двухуровневая модель архитектуры ОС Linux подразумевает разделение функциональных обязанностей между **уровнем ядра системы** и **уровнем приложений**.

Ядро системы предоставляет базовые услуги. Приложения (или задачи) делятся на *системные* и *прикладные* и общаются с ядром системы посредством **системных вызовов**. Для обращения к базовым услугам ядра приложения (процессы) используют **стандартный интерфейс системных вызовов**. Алгоритм обращения примерно следующий:

1) процесс запрашивает услугу ядра;

2) выполняется системный вызов определенной процедуры ядра;

3) ядро выполняет запрос от имени процесса;

4) процесс получает необходимые данные.

Приложения (процессы) представляют *пользовательский уровень*, устройства — *аппаратный уровень*. *Системный уровень* реализован в ядре и включает интерфейс системных вызовов и аппаратный контроль. Важнейшими компонентами ядра являются:

- файловая подсистема;
- подсистема управления процессами;
- подсистема ввода-вывода.

6.2. Управление файлами

Нельзя сказать, что в настоящее время для ОС Linux существует какая-то одна **стандартная** файловая система (ФС). Популярные на сегодняшний день реализации ФС для ОС Linux (ext3, ext4, reiserfs, btrfs и т. д.) используют основные идеи организации ФС s5 для ОС Unix.

6.2.1. Файл и inode

Прежде чем перейти к понятию файловой системы, остановимся на вопросе «Что такое файл?».

Определение 6.1. *Файл — именованная структура данных с последовательным доступом (доступ к элементам в определенном порядке).*

Ключевая идея организации ФС для ОС Linux:

ВСЕ ЯВЛЯЕТСЯ ФАЙЛОМ!

Файлом является в прямом смысле действительно все: каталоги, HDD (жесткий диск), параллельные порты, web-подключения и т. д. Поэтому для приложения доступ к дисковому файлу неотличим от доступа, например, к принтеру.

Основной особенностью физической организации файловой системы (см. подробнее в [5]) в ОС Linux является отделение **имени файла** от его **характеристик**.

Можно сказать, что любой файл имеет **метаданные**, хранящиеся в отдельной структуре, называемой **индексным дескриптором**, или **inode** (от англ. Information NODE). Какие метаданные хранятся в inode? Перечислим некоторые: идентификатор владельца файла, права доступа к файлу, тип файла, число ссылок на данный inode (количество жестких ссылок), размер файла и т. д.

Каждый индексный дескриптор **имеет уникальный номер** в пределах одной файловой системы. Номера индексных дескрипторов хранятся в специальных inode-таблицах. На самом деле, номер inode является **уникальным именем файла**.

Однако обычные файлы или каталоги имеют привычные нам **символьные имена**.

Определение 6.2. *Символьное имя файла — жесткая ссылка (hard link) на индексный дескриптор файла.*

Собственно поэтому имя файла не принадлежит метаданным. Кроме того, должно быть установлено соответствие между полными символьными именами (путь + имя) и их inode, например с помощью иерархии каталогов. При создании файла, например, командой

```
touch f_name
```

выделяется индексный дескриптор со свободным номером и определяется символьное имя.

Символьных имен (жестких ссылок) у одного файла может быть много. Вывести номер индексного дескриптора позволяет команда `ls -il`, см. пример 6.1.

Пример 6.1

```
# Вывод номера inode
:~> touch f_name
:~> ls -il f_name
4664074 -rw-r--r-- 1 user students 0 Июл 13  2011 f_name
```

Команда `ls -il` (как и `ls -l`) отображает в колонке перед именем пользователя `user` количество жестких ссылок (1 в данном примере). Создать жесткую ссылку на файл можно командой

```
ln имя_файла имя_ссылки
```

Пример 6.2

```
# Создание жесткой ссылки
:~> ln f_name f_link
:~> ls -li f_name f_link
4664074 -rw-r--r-- 2 user students 0 Июл 13 12:45 f_link
4664074 -rw-r--r-- 2 user students 0 Июл 13 12:45 f_name
```

В примере 6.2 создается новый файл с именем `f_link` с таким же номером индексного дескриптора, как и у `f_name`. Теперь жестких ссылок на inode с номером 4664074 стало две.

Если теперь отредактировать файл `f_link`, то автоматически изменения будут продублированы в файле `f_name`, см. пример 6.3.

Пример 6.3

```
# Изменение файла
:~> echo "Hello" > f_link
:~> ls -li f_name f_link
4664074 -rw-r--r-- 2 user students 6 Июл 13  2011 f_link
4664074 -rw-r--r-- 2 user students 6 Июл 13  2011 f_name
musen@epsilon:~> cat f_name
Hello
```

Индексный дескриптор (файл) не будет удален до тех пор, пока на него существует хотя бы одна жесткая ссылка.

6.2.2. Типы файлов

В ОС Linux выражение «тип файла» не характеризует содержимое файла (графический, текстовый, мультимедиа и т. д.), а указывает на то, что он является файлом с данными, каталогом или, например, физическим устройством. Файл интерпретируется как поток байтов, интерпретация содержимого оставлена приложениям. Перечислим основные типы файлов:

1) обычный файл (-);
2) каталог (d);
3) символическая ссылка (l);
4) символьный файл (c);
5) файл устройства (b);
6) именованный канал (p);
7) сетевой сокет (s).

Символы, соответствующие типам файлов, отображаются при выводе информации о файле командой `ls -l` (первый символ в строке).

6.2.3. Символические ссылки

Символическая ссылка (soft link) — это файл, содержащий путь к другому файлу и имеющий специальный тип (тип l). Символическая ссылка — это аналог ярлыка в ОС Windows. Символические ссылки создаются командой

```
ln -s имя_файла имя_ссылки
```

В примере 6.4 показано, что при выводе информации о файле командой `ls -l` указывается тип файла (тип l) — первый символ строки, и имя файла, на который указывает данная символическая ссылка.

Пример 6.4

```
# Создание символической ссылки
:~> ln -s f_name s_link
:~> ls -l s_link
lrwxrwxrwx 1 user students 13 Июл 13  2011 s_link -> f_name
```

6.2.4. Права доступа к файлу

ОС Linux поддерживает два способа управления доступом к файлу: традиционный (унаследованный от ОС Unix) и основанный на использовании списков доступа ACL (от англ. Access Control List).

Поскольку использование ACL имеет отношение больше к сетевой безопасности, то в рамках данного учебного пособия на этой теме останавливаться не будем. Подробнее об ACL можно прочитать в работе [9] или руководстве к командам `acl`, `setfacl`, `getfacl`.

Традиционный способ подразумевает деление пользователей на три категории по отношению к конкретному файлу:

1) владелец файла (u);

2) пользователь группы, с которой ассоциирован файл (g);

3) любой другой пользователь (o).

При этом пользователь может предпринять попытку доступа к обычному файлу тремя способами: прочитать его (r), записать в него данные (w), исполнить его (x).

Для вывода сведений о правах доступа можно опять воспользоваться командой `ls` с опцией `-l`.

Пример 6.5

```
# Вывод информации о файлах, включая права доступа
:~> ls -l
-rwxr-xr-x  1 user group 1024 Июл 11 15:43 file.sh
drwx--x--x 18 user group 4096 Мар 23 20:51 public_html
```

Права доступа к файлу `file.sh` в данном примере определяются символами (в строке вывода команды `ls`) со второго по девятый. Первые три символа подстроки представляют права владельца `rwx` (владелец имеет все права), второй набор из трех символов `r-x` — права группы, третий `r-x` — права всех остальных (в данном случае пользователи группы и все остальные имеют только права чтения и выполнения.

Права доступа к каталогам имеют несколько другое значение. **Чтение** каталога позволяет получать список содержимого каталога — файлов и подкаталогов; **запись** в каталог — создавать новые файлы, переименовывать и удалять старые; право **«исполнения каталога»** — переходить в него командой `cd` (или программно, например запускать программы из этого каталога).

Права доступа к файлу контролируются владельцем файла. Помимо владельца, сменить права доступа может только суперпользователь (пользователь с привилегиями root), который имеет полный доступ к всем файлам (ограничения, накладываемые правами доступа, к нему не применяются).

Для смены прав доступа может быть использована команда `chmod`:

```
chmod [ugoa][+-=][rwxst] имя
```

Символьный синтаксис использует ранее введенные мнемонические обозначения для категорий пользователей (u, g, o, a) и прав (r, w, x, s, t), а также знаки арифметических операций для добавления права (+), удаления (-) и установки (=), см. пример 6.6.

Права могут быть заданы в числовой форме, где вместо символов (rwx) используются их числовые эквиваленты: r — 4; w — 2; x — 1. Числовой режим состоит не более чем из четырех **восьмеричных**

цифр (от нуля до семи): первая цифра — для параметров s,t, вторая — для владельца (u), третья — для группы (g), четвертая — для всех остальных (o). Например, числовым эквивалентом для `rwx` будет цифра 7 (если сложить цифры, эквивалентные буквам).

Пример 6.6

```
:~> ls -l
-rwxr-xr-x  1 user group 1024 Июл 11 15:43 file.txt
:~> chmod a+rw file.txt
:~> ls -l
-rwxrwxrwx  1 user group 1024 Июл 11 15:43 file.txt
```

При создании файлам и каталогам по умолчанию назначаются права 0644 и 0755 соответственно (незначащий 0 вначале можно не учитывать). Эти значения получаются наложением `маски` на значение `режима полного доступа` для файлов (0666) и каталогов (0777) по следующей формуле:

`режим_полного_доступа И НЕ маска`

Пояснение. Разница в режимах полного доступа для файлов и каталогов вполне очевидна. Это обусловлено тем, что права доступа для файлов и каталогов отличаются друг от друга, как было сказано выше. Полный доступ к обычному файлу исключает право на выполнение по умолчанию. Действительно, зачем делать исполняемым каждый созданный файл? А вот если файл является, например, сценарием shell, то добавить право на выполнение такому файлу придется вручную командой `chmod`.

Посмотреть текущее значение маски можно командой `umask`.

Пример 6.7

```
:~> umask
0022
```

Упражнение: убедитесь, что при заданной маске 022 права по умолчанию для файлов устанавливаются равными 644, а для каталогов — 755.

6.2.5. Файловая система

Понятие **файловая система** включает в себя три основных значения:

1) Функции ядра системы, с помощью которых реализован логический доступ к файлам и каталогам — **файловая подсистема**. Файловая подсистема обеспечивает интерфейс доступа к данным, контролирует права доступа, перенаправляет запросы, адресованные периферийным устройствам;

2) Служебные структуры данных на носителях, обеспечивающие абстракцию (конкретной аппаратной реализации) файлов и каталогов;

3) Иерархически организованную с помощью каталогов систему файлов (логическую структуру ФС).

В ОС Linux логическая структура ФС регламентируется стандартом FHS (от англ. Filesystem Hierarchy Standard). Согласно этому стандарту все файлы и каталоги находятся внутри корневого каталога с именем / (slash), независимо от того, расположены они на разных физических носителях или нет. Подкаталоги корневого каталога обычно имеют специальное назначение, перечислим некоторые в таблице 5.

Подкаталоги специального назначения Таблица 5

№	**Имя**	**Содержимое**
1	`/bin`	Исполняемые файлы, утилиты shell
2	`/boot`	Загрузочные файлы, файлы ядра
3	`/dev`	Драйверы, файлы устройств
4	`/etc`	Конфигурационные файлы системы
5	`/home`	Домашние директории пользователей
6	`/lib`	Библиотеки для работы ОС и приложений
7	`/mnt`	Точки монтирования (временные)
8	`/root`	Домашняя директория суперпользователя ...

Для некоторых каталогов существуют короткие сокращения. Например, точка (.) обозначает текущий каталог; две точки (..) — каталог на уровень выше. В языке shell есть подстановки тильды, которые часто используются как сокращения. Каждая подстановка тильды соответствует некоторой переменной окружения shell:

~ — домашний каталог (переменная HOME);

~+ — текущий каталог (переменная PWD);

~- — предыдущий каталог (переменная OLDPWD).

6.3. Управление процессами

Рассматриваемая нами ОС Linux является примером многозадачной системы. Это означает, что несколько приложений могут выполняться параллельно, разделяя процессорное время, память, периферийные устройства и другие ресурсы. Для описания конкретного экземпляра программы, выполняемого в системе, используется термин **процесс**.

Процесс — это не только образ файла программы в памяти компьютера, он включает также необходимые для управления структуры данных в ядре операционной системы, полномочия, перечень дескрипторов открытых файлов, переменные окружения и т. д.

Подсистема управления процессами включает планировщик, обеспечивающий многократное переключение процессов (снятие одного процесса с выполнения и передачу права на выполнение другому процессу), поэтому у пользователя создается впечатление, что программы выполняются одновременно.

6.3.1. Иерархия процессов

Процесс как объект ОС создается посредством системного вызова fork(). В системе создается дубликат процесса, выполнившего fork(), который затем может быть замещен новым кодом. Таким образом, все процессы системы образуют иерархию «родитель-потомок».

Процессы имеют уникальные идентификаторы (PID). Стартовый код ядра системы условно имеет PID 0. Он не создается системой управления процессами, поэтому, строго говоря, процессом не является. В дальнейшем будем его называть **прапроцесс**. Прапроцесс с PID 0 порождает процесс init, имеющий PID 1. Процесс init, в свою очередь, является предком всех процессов в системе, выполняющихся в **пользовательском** пространстве.

Пример 6.8 демонстрирует применение команды `pstree` для вывода фрагмента дерева процессов в некоторый момент времени. От-

метим, что в ходе сбора данных о процессах команда pstree «видит» в том числе и процесс, представляющий саму программу pstree.

Из примера 6.8 видно, что процесс init порождает несколько процессов mingetty — эмуляторов терминала, предоставляющих интерфейс для запуска командного интерпретатора. Как правило, в системе создается несколько эмуляторов терминала, доступных через Ctrl+Alt+Fn, где n — номер эмулятора (обычно, в диапазоне 1 — 6).

При вводе пользователем имени и пароля, эмулятор mingetty замещается процессом login, который порождает дочерний процесс и запускает в нем программу bash (/bin/bash) — командный интерпретатор, обеспечивающий диалог пользователя с системой (эти действия уже обсуждались ранее в § 1). В нашем случае с помощью bash мы запустили программу pstree, для которой bash создал еще один дочерний процесс.

Пример 6.8

```
init-+-acpid
     |-auditd---{auditd}
     |-...
     |-cron
     |-cupsd
     |-login---bash---pstree
     |-5*[mingetty]
     |-...
     `-upowerd---{upowerd}
```

В примере 6.8 приведены только процессы, выполняющиеся в так называемом **пользовательском** пространстве. Ряд процессов выполняется **в ядре** системы — это всевозможные планировщики ресурсов, драйверы и т. д.

Полный перечень процессов можно вывести посредством команды `ps -Af`, см. пример 6.9. Столбец PPID отображает идентификатор процесса-родителя. Обратите внимание, что, помимо init, прапроцессом с PID 0 порождается процесс ядра kthreadd с PID 2, являющийся планировщиком процессов (строго говоря, **потоков**, однако разница между этими понятиями будет пояснена в последующих курсах). Подробнее о процессах и потоках см. в работе [5].

Пример 6.9

```
UID    PID  PPID  C STIME TTY  TIME CMD
root     1     0  0 08:41 ?    00:00:01 init [3]
root     2     0  0 08:41 ?    00:00:00 [kthreadd]
root     3     2  0 08:41 ?    00:00:00 [ksoftirqd/0]
...
root   598     2  0 08:41 ?    00:00:00 [hd-audio0]
root   958     1  0 08:42 ?    00:00:00 /sbin/acpid
root  1161     2  0 08:42 ?    00:00:00 [kauditd]
...
root  2174     1  0 08:43 ?    00:00:00 /usr/sbin/cron
root  2289     1  0 08:43 ?    00:00:00 login -- user
root  2291     1  0 08:43 tty3 00:00:00 /sbin/mingetty tty3
root  2292     1  0 08:43 tty4 00:00:00 /sbin/mingetty tty4
root  2293     1  0 08:43 tty5 00:00:00 /sbin/mingetty tty5
root  2294     1  0 08:43 tty6 00:00:00 /sbin/mingetty tty6
user  6004  2289  0 11:48 tty1 00:00:00 -bash
root  6165     1  0 11:49 tty2 00:00:00 /sbin/mingetty tty2
root  6199  6047  0 11:53 tty1 00:00:00 ps -Af
```

6.3.2. Сигналы

В реальных приложениях процессы не могут быть абсолютно изолированы друг от друга: необходимы средства, обеспечивающие возможность взаимодействия, обмена данными между процессами. Например, один из процессов может поставлять данные из некоторого источника (например, клиент электронной почты получает данные от пользователя), а другой процесс — обрабатывать (реализация протокола отправки электронной почты). По этой причине в большинстве многозадачных систем предусмотрены средства межпроцессного взаимодействия (IPC). Одним из средств IPC является конвейер, см. § 5.

Простейшим средством IPC являются сигналы. Посланный процессу сигнал прерывает выполнение процесса. Получив сигнал, процесс может его обработать — выполнить определенную последовательность действий, однако никаких дополнительных данных посредством

сигнала передать нельзя. Если процесс не обрабатывает сигнал самостоятельно, то выполняется стандартный обработчик, который, как правило, приводит к принудительному завершению процесса.

Сигналы идентифицируются номерами — целыми числами. В ОС Linux имена сигналов — это числовые константы, которые определены в заголовочном файле signal.h стандартной библиотеки языка Си (SIGINT, SIGKILL и др.). Номера сигналов могут варьироваться в зависимости от ОС. Список имен сигналов и соответствующие им номера можно вывести с помощью команды `kill -l`.

Самый простой способ увидеть действие сигналов на практике — запустить длительную по времени задачу, например поиск по всей файловой системе, и нажать Ctrl+C. В этом случае процессу будет отправлен сигнал SIGINT, стандартная реакция на который подразумевает завершение процесса.

Другой способ отправки сигнала процессу — использовать команду `kill -s СИГНАЛ PID`, см. пример 6.10.

Пример 6.10

```
# Запускаем длительный поиск
:~> find / -name '*' >/dev/null 2>&1
# В соседнем терминале определяем PID процесса
:~> ps -Af | grep find
user    9337  8027  6 14:09 pts/2  00:00:00 find / -name *
user    9345  6683  0 14:09 pts/0  00:00:00 grep find
# Отправляем процессу SIGINT
:~> kill -s SIGINT 9337
```

Процесс, обработав сигнал самостоятельно, может игнорировать стандартное поведение. Сценарий shell, например, может установить собственный обработчик с помощью команды trap, см. пример 6.11.

Пример 6.11

```
# Устанавливаем обработчик
trap 'echo "Этот процесс прерывать нельзя!"' SIGINT
# Запускаем длительный поиск
:~> find / -name '*' >/dev/null 2>&1
```

Теперь при попытке отправить сигнал SIGINT запущенному сценарию (по Ctrl+C или с помощью kill) будет выполнена указанная в trap последовательность команд.

Не всякий сигнал можно обработать. Сигнал SIGKILL приводит к незамедлительному завершению (процесс не имеет возможности выполнить какие-либо действия перед завершением). Исключением являются процессы-зомби, процесс init и процессы, заблокированные ядром в ожидании операции ввода-вывода.

6.3.3. Терминальные задачи

Процессы идентифицируются в пределах всей системы, что не очень удобно с точки зрения сеанса работы пользователя, вынужденного разыскивать PID своих 3–5 процессов среди нескольких десятков или даже сотен процессов системы и других пользователей. Поэтому, кроме понятия **процесс**, существует понятие **терминальной задачи**, которая определена (идентифицируема) в пределах сеанса командного интерпретатора.

Пример 6.12

```
# Запуск программы в фоновом режиме:
# - при запуске на экране печатаются JID и PID
# - интерпретатор выводит подсказку и пользователь
#   может исполнять другие команды
# - фоновая программа по-прежнему может выводить
#   на экран
:~> find . -name '*.c' &
[1] 9423
:~> ./2.c
./1.c

[1]+  Done                    find . -name '*.c'
```

С точки зрения bash, одно приложение может быть запущено в интерактивном (основном) режиме работы, несколько — в фоновом. Для запуска задачи в фоновом режиме следует завершать команду символом

&, см. пример 6.12. При этом интерпретатор выводит приглашение командной строки, не дождавшись завершения задачи, и пользователь может выполнять другие программы в том же терминальном сеансе.

Важно! Следует заметить, что программа в фоновом режиме, если не использованы средства перенаправления стандартных потоков вывода, будет по-прежнему печатать свои сообщения на экране, что может конфликтовать с выводом других программ.

Запущенная в интерактивном режиме программа может быть переведена в фоновый после терминальной остановки командой bg. Терминальная остановка — сигнал SIGTSTP, который можно отправить интерактивной задаче, нажав Ctrl+Z.

Перечень задач для текущего терминала может быть выведен командой `jobs`. При этом каждой задаче ставится в соответствие уникальный идентификатор — JID. Перевести одну из фоновых задач в интерактивный режим можно командой `fg % jid`, где `jid` — идентификатор задачи. Для отправки задаче сигнала можно использовать синтаксис `kill -s СИГНАЛ %jid`.

В примере 6.13 мы посылаем сигнал SIGTERM для завершения задачи (этот сигнал используется утилитой `kill` по умолчанию, если никакой другой сигнал не задан).

Пример 6.13

```
# Запускаем длительный поиск в фоновом режиме
:~> find / -name '*' >/dev/null 2>&1 &
[1] 9659
:~> kill -s SIGTERM %1
:~>
[1]+  Завершено      find / -name '*' > /dev/null 2>&1
:~>
```

§ 7. Сценарии shell

Если требуется ввести длинный список команд в командную строку либо использовать достаточно громоздкие управляющие конструкции shell (ветвление, цикл и т. д.), то удобнее оформить эти команды в отдельном файле — сценарии shell — и запускать его в пакетном режиме.

Выполнять отладку или модифицировать участки алгоритма предпочтительнее в сценарии, нежели в командной строке, особенно если команд много, имеются операторы управления и функции.

Дадим отпределение сценария, которое справедливо, впрочем, и для любого другого интерпретируемого языка программирования.

Определение 7.1. *Сценарий (скрипт) shell — это текстовый файл, содержащий допустимые конструкции языка shell и имеющий право на запуск.*

Внимание! Необходимо, чтобы первой строкой скрипта была инструкция выбора исполняющего интерпретатора см. [7], в нашем случае это

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

Символ `#` обозначает **начало комментария shell**. Если за решеткой следует восклицательный знак `#!`, то такая последовательность символов (ее называют sha-bang) указывает на выбор интерпретатора для выполнения сценария (`/bin/bash`).

В качестве интерпретатора может быть указана не только командная оболочка shell, но и любой другой интерпретатор (например, `/usr/bin/php`) или даже реализация команды (например, `/bin/more` или `/bin/rm`).

Обычно скриптам shell присваивают расширение `.sh`, хотя это и не является обязательным требованием.

Оформление собственных сценариев в едином стиле считается правилом хорошего тона для программиста. А иногда позволяет сэкономить бесценное время на поиск необходимой конструкции и ее модификацию.

В примере 7.1 предлагается один из возможных вариантов оформления shell-сценария.

Пример 7.1

```
#!/bin/bash
############################################################
# Сценарий : dtox - преобразование текстового файла DOS к
# стандарту UNIX
# Автор : Хайнер Стивен
# Версия : 1.1
# Дата : 2002.02.26
############################################################
# Описание
# Заменяет последовательности "CR LF"признаком конца
# строки "LF"
# Замечания
# Не преобразует символ конца файла DOS CTRL-Z (ASCII 26)
```

7.1. Запуск сценария

Чаще всего сценарии shell запускаются на выполнение как консольные приложения из командной строки. По умолчанию при создании обычного файла ему не присваивается право на выполнение. Поэтому если сценарий уже написан, то для дальнейшего запуска необходимо изменить его права, добавив право на выполнение.

Пример 7.2

```
# Добавление права на запуск для владельца
:~> chmod u+x script.sh
```

Теперь сценарий можно запустить на выполнение, предварительно сделав переход (`cd`) в каталог, где находится сценарий (либо указать полный или относительный путь к сценарию). Для запуска скрипта есть два варианта:

в подоболочке shell: `./имя_сценария`

в текущем shell: `. имя_сценария` или `source имя_сценария`

В первом случае должен быть указан **полный путь** к сценарию, т. к. `./` предполагает поиск скрипта в текущем каталоге. Если не указать полный путь, и сценарий не найден в текущем каталоге, то интерпретатор попросту его не найдет. Это объясняется тем, что в целях безопасности текущий каталог (`./`) не включен в список каталогов поиска команды (переменная окружения PATH). Как мы помним из § 4, интерпретатор будет искать имя скрипта именно в каталогах поиска.

Во втором случае скрипт будет запущен командой «точка» (или `source`). Это встроенная команда bash, которая запускает сценарий в окружении текущей оболочки. В этом случае полный путь к сценарию можно не указывать, т. к. команда `source` будет искать сценарий в каталогах поиска, а затем в текущем каталоге.

Каждая строка shell-сценария (кроме, может быть, последней) должна завершаться символом перевода строки.

Выше мы рассмотрели основные составляющие командной строки: простые команды, метасимволы shell, подстановки команд, шаблоны имен файлов, конвейеры, списки команд и т. д.

В данном параграфе мы опишем основные средства языка shell, чаще всего используемые в сценариях:

- пользовательские переменные;
- массивы;
- переменные окружения;
- подстановка переменных;
- позиционные параметры;
- управляющие конструкции (условные, циклические).

7.2. Пользовательские переменные

Словосочетание *пользовательские переменные*, или *переменные интерпретатора shell* [6], обычно используют, когда говорят о собственных переменных пользователя, определенных им самостоятельно.

7.2.1. Как определить переменную?

Следует различать **имя переменной** и **значение переменной**. Говоря о переменной, мы подразумеваем пару **имя-значение**. Присвоить переменной какое-либо значение можно так:

```
1) имя=[значение]

2) read имя
```

Если значение не указано, то говорят, что переменная *имеет пустое значение* (*«не установлена»*). Если значение задано, то переменная *«имеет значение»*. Иначе считается, что переменная *не определена* (*«не объявлена»*).

7.2.2. Как обратиться к значению переменной?

Для обращения к значению переменной используется **простая подстановка значения**:

```
${имя}
```

Иногда фигурные скобки, определяющие имя переменной, можно опустить, если это не приведет к неоднозначности (см. пример 7.3).

Пример 7.3

```
# test.sh: Имя переменной в фигурных скобках
var="string_a"; var1="string_b"
echo $var12
echo ${var}12
echo $var$var1
echo ${var}${var1}
# Запуск скрипта test.sh
:~> ./test.sh
string_a12
string_astring_b
string_astring_b
```

Пояснение. Интерпретатор не нашел переменную с именем `var12`, она не объявлена, поэтому вывод ее значения дает пустую строку. Для многосимвольных имен лучше всеже использовать фигурные скобки.

Пояснение. В примере 7.3 команда `echo` при выводе выполняет конкатенацию (склеивание) значений переменных и строк текста.

7.2.3. Время жизни пользовательских переменных

Пользовательские переменные доступны только в текущей оболочке. Время жизни переменных пользователя ограничено временем работы оболочки интерпретатора, в которой они были определены.

Если запускать скрипт в подоболочке, то переменные, определенные в этом скрипте, будут недоступны после завершения скрипта. И наоборот, доступ к переменным, определенным в скрипте, можно получить, если запустить скрипт в текущем shell (см. пример 7.4).

Пример 7.4

```
# test.sh: Определение переменной my_var
my_var="my_string"
echo $my_var
# Запуск в подоболочке
:~> ./test.sh
my_string
:~> echo $my_var

:~>
# Запуск в текущем shell
:~> . test.sh
my_string
:~> echo $my_var
my_string
```

7.2.4. Типы переменных

Язык shell не имеет четко выраженных типов переменных, однако все их можно условно разделить на группы:

- строки символов;
- целые числа;
- массивы.

По умолчанию все переменные bash являются строковыми. Определяющий фактор при их интерпретации — значение переменной.

Минимально контролировать типы переменных и устанавливать их атрибуты можно с помощью встроенной команды `declare`:

- **declare -i имя[=значение]** — определить переменную как целое число. Применительно к такой переменной можно выполнять некоторые арифметические операции без использования специальных функций (таких, как `expr`);
- **declare -a имя** — определить переменную как массив;
- **declare -r имя** — установить для переменной атрибут «только для чтения», аналогично команде `readonly имя`. При попытке присвоить значение такой переменной будет выдано сообщение об ошибке;
- **declare -f** — вывод определений функций;
- **declare -F** — вывод имен функций.

Функция `declare`, как и функция `set`, без опций выводит все заданные имена и определения.

Разберем некоторые нюансы обращения с разными типами переменных на примерах.

Если необходимо при выводе строки командой `echo` сохранить пробельные разделители, то строку следует заключить в двойные кавычки.

Пример 7.5

```
# Сохранение разделителей
n="a b     c"          # n - строка
echo $n                # a b c
echo "$n"              # a b     c
```

Применение арифметических операций к строке не приведет к ошибке, вместо строки будет подставлено числовое значение, равное нулю.

Пример 7.6

```
# Арифметическая операция над строками
n="ab"                  # n - строка
let "n+=1"              # используется значение n равное 0
echo $n                 # 1
```

Применение функции `declare -i` к переменной, уже имеющей строковое значение, не приведет к изменению значения на нулевое.

Пример 7.7

```
# Попытка сменить тип на числовой
n="ab"                  # n - строка
declare -i n            #
echo $n                 # ab
```

Если же присвоить строковое значение переменной, которая объявлена как целочисленная, то она будет иметь нулевое значение.

Пример 7.8

```
# Присвоение строки числовой переменной
declare -i n            # n - число
n="ab"                  # запись строки
echo $n                 # 0
```

7.2.5. Подстановка переменных

Кроме уже известных нам подстановок команды и подстановок в именах файлов, в языке shell имеются и другие подстановки. Напомним, что смысл любой **подстановки** заключается в замене некой

последовательности символов («подстановки») на значение, ей соответствующее.

Подробнее разберем *подстановки переменных*, или *подстановки параметров*. Самая простая подстановка — это подстановка значения переменной.

`${имя:-значение_по_ум}` — альтернативная подстановка. Если переменная не объявлена или имеет пустое значение, то выводится значение по умолчанию.

`${имя:+альт_значение}` — обратная альтернативная подстановка. Если переменная имеет непустое значение, то выводится альтернативное значение.

`${имя:=значение_по_ум}` — переопределяющая подстановка. Если переменная не определена или имеет пустое значение, то она принимает значение по умолчанию.

`${имя:?сообщение_об_ошибке}` — подстановка ошибки. Если переменная не определена или имеет пустое значение, то выводится сообщение об ошибке.

`${имя:смещение}` — подстановка подстроки. Извлечение подстроки из строки `имя`, начиная с позиции `смещение`.

`${имя:смещение:длина}` — подстановка подстроки. Извлечение подстроки, начиная с позиции `смещение`, заданной длины из строки `имя`.

`${#имя}` — подстановка длины переменной.

`${имя#шаблон}` — удаление подстроки. Удаляет из переменной **наименьшую** подстроку, удовлетворяющую шаблону. Поиск ведется **с начала** строки.

`${имя##шаблон}` — удаление подстроки. Удаляет из переменной **наибольшую** подстроку, удовлетворяющую шаблону. Поиск ведется **с начала** строки.

`${имя%шаблон}` — удаление подстроки. Удаляет из переменной **наименьшую** подстроку, удовлетворяющую шаблону. Поиск ведется **с конца** строки.

`${имя%%шаблон}` — удаление подстроки. Удаляет из переменной **наибольшую** подстроку, удовлетворяющую шаблону. Поиск ведется **с конца** строки.

Пояснение. В первых четырех подстановках, описанных выше, можно не указывать символ двоеточия (`:`). Это влияет на результат, только если переменная `имя` не установлена (т. е. объявлена, но имеет пустое значение).

Пример 7.9

```
# Переменная var не определена
a=${var:-aaa}; b=${var-bbb}
echo a=$a                      # a=aaa
echo b=$b                      # b=bbb
# Переменная var имеет пустое значение
var=
a=${var:-aaa}; b=${var-bbb}
echo a=$a                      # a=aaa
echo b=$b                      # b=
# Переменная var имеет значение
var=str
a=${var:-aaa}; b=${var-bbb}
echo a=$a                     # a=str
echo b=$b                     # b=str
```

Можно в качестве значения по умолчанию использовать подстановку команды. В примере 7.10 вместо значения переменной `mypwd` подставляется строка, которую выводит команда `pwd`, если переменная не задана или имеет пустое значение.

Пример 7.10

```
# Подстановка команды pwd
:~> echo ${mypwd:-`pwd`}
/home/user
```

Приведем пример 7.11, демонстрирующий проверку наличия переменной в сценарии. Обратите особое внимание на использование команды `:` — это пустая команда. В языке shell необходимо, чтобы каждая строка начиналась с команды, поэтому отдельно подстановку ошибки использовать нельзя.

Почему не годится команда `echo`? Если переменная `my_var` не определена или имеет пустое значение, то команды `echo` и `:` работают одинаково: выдают сообщение об ошибке. Однако если значение переменной `my_var` задано, то команда `echo`, выполняя подстановку, выведет это значение на экран, что было бы нежелательно.

Пример 7.11

```
# Проверка наличия переменной my_var
: ${my_var:?"Не определена или имеет пустое значение"}
```

Пояснение. Набор вариантов подстановок переменных не ограничивается приведенными выше. Мы рассмотрели наиболее часто используемые подстановки. Кроме того, существуют подстановки переменных для работы с массивами, они будут описаны далее.

7.3. Переменные окружения

При запуске команды (процесса) на выполнение или новой оболочки shell требуется передать множество параметров, например тип терминала, вид строки приглашения, перечень каталогов для поиска программ, идентификатор родительского процесса и т. д. Все эти параметры в сумме являются **окружением**. Окружение определяет условия, в которых выполняются программы. При запуске подоболочки дочерний процесс наследует окружение. Параметры окружения хранятся в так называемых **переменных окружения** (иногда их называют **переменные среды**).

Некоторые переменные окружения bash Таблица 6

№	Имя	Назначение
1	PATH	Каталоги поиска исполняемых файлов
2	PS1	Первичное приглашение командной строки
3	PS2-PS4	Остальные приглашения командной строки
4	IFS	Внутренний разделитель полей, используется для выделения слов. Стандартное значение <пробел><табуляция><перевод строки>
5	HOME	Домашний каталог пользователя
6	PWD	Текущий каталог пользователя
7	LANG	Настройки локали (например, ru_RU.UTF-8)
8	RANDOM	Случайное целое в диапазоне [0-32767]
9	TMPDIR	Каталог временных файлов

Любой командный язык имеет свой собственный набор переменных окружения. Некоторые из них являются *информационными*, некоторые можно менять, присвоив переменной другое значение.

Для вывода переменных окружения используют команду `set` или команду `env`. Поскольку кроме переменных окружения `set` выводит еще и другие определения, то удобнее использовать конвейер `set| less` с возможностью прокрутки. Приведем список некоторых переменных окружения.

Для вывода значений переменных окружения используется простая подстановка значения и команда `echo`:

```
echo $PS1
```

Присвоить (изменить) значение переменной окружения можно с помощью команды `export`, тогда оно передается дочерним оболочкам.

```
export PATH=$PATH:/путь/к/каталогу
```

Можно обойтись и без использования команды `export`, тогда установленное значение будет доступно только в текущей оболочке. Удалить значение переменной окружения можно с помощью команды `unset`:

```
unset RANDOM
```

Удаление или изменение переменной окружения будут действительны только в текущей оболочке. При удалении переменной `RANDOM` и последующем восстановлении она теряет свои свойства «случайности».

7.4. Позиционные параметры и параметры командной строки

Чтобы передать сценарию shell аргументы (используемые, к примеру, в функциях скрипта), нужно перечислить их в командной строке, после имени самого сценария. Это означает **задать параметры командной строки**.

Переданные аргументы доступны внутри сценария через **позиционные параметры**, или **позиционные переменные**. Имена переменных соответствуют позиции аргумента в командной строке начиная с нуля. Обращаться к значению переменной нужно через символ `$`:

```
${0}, ${1}, ${2}, ...
```

Имя запускаемого сценария соответствует `$0`. Для одноцифровых номеров фигурные скобки можно опустить, если это не приведет к неоднозначности.

Пример 7.12

```
# Запуск сценария с аргументами
./test.sh my_file my_str
```

В качестве разделителя элементов командной строки интерпретатор bash использует чаще всего пробел (из переменной `IFS`).

В примере 7.12 позиционные переменные будут определены так: `$0 = "./test.sh"`, `$1="my_file"`, `$2="my_str"`.

7.4.1. Специальные параметры

В языке shell существует набор специальных параметров — переменных — с помощью которых можно получить доступ, в том числе и к позиционным параметрам. Нам уже известен один такой параметр — статус завершения `$?`. Приведем список специальных параметров, необходимых для работы с позиционными переменными:

`${*}` — список позиционных параметров в виде одной строки (слова). При составлении содержимого переменной `${*}` bash использует

для разделения параметров первый символ из переменной `IFS`. При использовании необходимо заключать в двойные кавычки — `"${*}"`;

`${@}` — список позиционных параметров. Каждый параметр представляет отдельную строку (слово). При использовании необходимо заключать в двойные кавычки — `"${@}"`;

`${#}` — количество позиционных параметров.

Внимание! Символ `$` и фигурные скобки означают подстановку значения параметра (как и ранее).

7.4.2. Как переопределить позиционные параметры

Позиционные параметры чаще всего используют для обращения к аргументам командной строки, хотя shell позволяет переопределить, например, элементы массива как позиционные параметры. Для управления позиционными параметрами часто применяются команды:

- `set` — сброс и установка позиционных параметров[11] (с опцией `--`);
- `shift N` — сдвиг позиционных параметров на `N` позиций. При этом значения первых `N` параметров теряются.

Пример 7.13

```
# Переопределение позиционных параметров
b="${*}"        # сохранение поз. параметров
str="a b c d"   # инициализация массива
set  -- $str    # переопределение параметров
echo "${*}"     # Вывод: a b c d
set --          # сброс поз. параметров
echo "${*}"     # Вывод: пустая срока
set -- $b       # восстановление поз. параметров
echo "${*}"     # Вывод: my_file my_str
```

[11] Команда `set` может выполнять и другие функции.

Если после опции −− команды `set` ничего не указано, то происходит сброс позиционых параметров. Если после −− указаны аргументы, то набор этих аргументов будет определять новые установленные позиционные параметры.

Разберем работу команды `set` на примере 7.13. Мы переопределяем элементы строки `str` как позиционные параметры, предварительно сохранив строку с параметрами в переменной `b`. Пускай при запуске сценарию будут переданы два параметра `my_file my_str` из примера 7.12.

Если вместо строки в примере 7.13 предстоит в качестве позиционных параметров использовать элементы массива, то сначала необходимо преобразовать массив в строку. Удобнее в этом случае использовать подстановку списка значений элементов массива, см. пример 7.14.

Пример 7.14

```
# Использование списка элементов массива
arr=( a b c d )          # инициализация массива
arr_items="${arr[*]}"    # набор элементов массива
set  -- $arr_items       # переопределение поз. параметров
echo "${*}"              # Вывод: a b c d

shift 2                  # сдвиг на 2 параметра
echo "${*}"              # Вывод: c в
```

Приведем еще один полезный пример использования подстановки переменных (см. п. 7.2.5). Подстановки удобно использовать для проверки существования параметра командной строки.

Пример 7.15

```
# Проверка существования позиционного параметра $1
: ${1?"Порядок использования: $0 параметр"}
```

Если второй параметр не задан, то будет выдана строка о порядке использования сценария.

7.5. Управляющие конструкции

Язык shell имеет набор управляющих конструкций, которые можно разделить на категории: 1) организация ветвлений; 2) организация меню; 3) организация циклов; 4) создание функций.

Отметим основные особенности языка shell, касающиеся практически всех управляющих конструкций. Ключевые слова `if, then, elif, select, ...`, как и команды shell, должны либо начинать строку, либо предваряться символом `;` (для разделения команд). Ниже при описании синтаксиса будем использовать краткую форму записи. Однако во всех конструкциях вместо разделителя `;` можно использовать перевод строки.

Управляющие конструкции, как и любой **блок кода** языка shell, используют стандартный поток ввода (stdin), поэтому данные в эти конструкции можно перенаправить. Благодаря этой особенности управляющие конструкции можно использовать в конвейерах (см. пример 7.19 с циклом while).

7.5.1. Оператор ветвления if

Синтаксис

```
if сп1; then сп2;[elif сп3; then сп4;] ... [else спN;] fi
```

Дополнительные ветви `elif` и альтернатива `else` могут отсутствовать.

«Истинность условия». На самом деле, оператор `if` в языке shell немного отличается от привычных операторов `if` в языке Си или Pascal. Поскольку shell — командный язык, то вместо «настоящего» условия после ключевого слова `if` стоит **список команд** (см. § 5). Он может содержать **команды проверки условий**, такие как `test` или `[`.

Сам по себе оператор `if` не умеет выполнять операции сравнения.

Команды из `сп2` выполняются, если код возврата последней команды `сп1` равен нулю. Аналогично, команды из `сп4` выполняются, если код возврата последней команды в `сп3` равен нулю и т. д.

Код возврата всей конструкции равен коду возврата последней выполненной команды.

7.5.2. Проверка условий test

Команда `test` — это встроенная команда bash для проверки условий, выполняющая строковые, числовые, файловые проверки. Аргументом команды `test` является **условное выражение** shell (см. таблицу 7).

Синтаксис

```
1. test выражение
2. [ выражение ]
```

У команды `test` есть альтернатива — команда `[` (именно команда, а не ключевое слово или оператор). Обратите внимание, что после символа `[` стоит **пробел**. И это вполне логично. Согласно правилам shell, `выражение` передается команде `[` в качестве параметра и отделяется от имени команды пробелом. Что касается закрывающей скобки, то по правилам языка ее можно не ставить, однако последние версии интепретатора bash требуют присутствия закрывающей скобки (опять же через пробел, как параметр).

Некоторые условные выражения для `test` Таблица 7

№	Имя	Назначение
1	`-n строка`	Длина строки не равна нулю
2	`-z строка`	Длина строки равна нулю
3	`стр1 = стр2`	Строки совпадают
4	`стр1 != стр2`	Строки не совпадают
5	`числ1 -eq числ2`	Числа равны
6	`числ1 -ne числ2`	Числа не равны
7	`числ1 -gt числ2`	Число 1 больше числа 2
8	`числ1 -ge числ2`	Число 1 больше или равно числу 2
9	`числ1 -lt числ2`	Число 1 меньше числа 2
10	`числ1 -le числ2`	Число 1 меньше или равно числу 2
11	`-e файл (-s файл)`	Файл существует (и не пустой)
12	`-r файл, -w файл, -x файл`	Файл существует и есть права на чтение, запись, выполнение
13	`-f файл, -d файл`	Файл существует и это простой файл, каталог
14	`файл1 -nt файл2`	Файл 1 более новый, чем файл 2

Внимание! Оператор `if` часто применяется в связке с командой `[`, поэтому некоторые ошибочно принимают квадратные скобки

за правило оформления условия в `if` — это не так. «Условие» в `if` — это список команд `сп1`. Квадратные скобки — это аналог команды `test`.

Сложные выражения. Выражение для `test` может быть составным, допустимы следующие конструкции:

1) `( выражение )` — подвыражение;
2) `! выражение` — инверсия;
3) `выр1 -a выр2` — конъюнкция (И);
4) `выр1 -o выр2` — дизъюнкция (ИЛИ).

Код возврата команд `test` и `[` равен либо 0 (успех), либо 1, в зависимости от выполнения условия.

Пример 7.16

```
# Проверка имени пользователя
name=`whoami`                    # подстановка команды

if [ ${name} != pupkin ]
then
      echo Oh, no! It\'s not me!
fi
```

7.5.3. Организация меню case

Синтаксис

```
case слово in шаблон| ... ) сп;; ... esac
```

Условие выполнения. При нахождении `первого` соответствия `слова` одному из `шаблонов` выполняется соответствующий список команд `сп`.

Как правило, в качестве `слова` используется подстановка значения какой-либо переменной, соответствие которой нужно будет найти среди шаблонов и выполнить указанный список команд.

Пример 7.17

```
# Выбор в зависимости от аргумента командной строки
case ${1} in
        ab|ba ) echo 1 ;;
        abc ) echo 2 ;;
        abc ) echo 3 ;;
        * ) echo 4 ;;
esac
# Варианты ввода
${1}=abb        Вывод: 4
${1}=ab         Вывод: 1
${1}=abc        Вывод: 2
```

Шаблоны. В качестве шаблона можно использовать любые подстановки, в том числе подстановки параметров, команд, арифметические подстановки. Шаблон `*` соответствует ветви по умолчанию (если `слово` не совпало ни с одним `шаблоном`).
Код возврата всей конструкции равен коду возврата последней выполненной команды в списке. Если ни один шаблон не подошел, то код возврата равен нулю.

7.5.4. Организация меню select

Синтаксис

```
select имя [ in сп_слов ] ; do сп_ком; break; done
```

Выполнение. Оператор `select` предлагает выбрать один вариант из перечисленных в списке `сп_слов`. Возможные варианты отображаются на экране в виде нумерованного списка. Затем выводится **приглашение** для ввода выбранного варианта (число в списке). Текст приглашения для ввода соответствует значению переменной окружения `PS3`, и его можно изменить. По умолчанию выводится `#?`. Если опустить `break`, то будет бесконечный цикл.
Варианты ввода. Выполнение списка команд `сп_ком` зависит от того, что выберет пользователь в ответ на приглашение `select`. Возможны следующие варианты:

1) *Номер по списку* — переменной `имя` будет присвоено соответствующее значение (слово), и выполнен список команд `сп_ком`.

2) *EOF* (Ctrl+D) — завершение работы `select`.

3) *Перевод строки* — повтор запроса.

4) *Любое другое значение* — переменной `имя` будет присвоено пустое значение (null).

Код возврата всей конструкции равен коду возврата последней выполненной команды в списке. Если ни одна команда не была выполнена, то код возврата равен нулю.

Пример 7.18

```
# Пример организации простого меню
PS3='Выберите букву: '
select choice in a b c
do
if [ -z ${choice} ]
    then
        echo Неизвестный выбор
    else
        echo Ваш выбор - буква ${choice}
    fi
break
# Вариант выполнения
1) a
2) b
3) c
Выберите букву: 2
Ваш выбор - буква b
```

7.5.5. Оператор цикла while, until

Синтаксис

```
while сп1; do сп2; done
until сп1; do сп2; done
```

Условия выполнения. Команды из списка `сп2` в цикле while (until) выполняются, если код возврата последней команды из списка `сп1` равен нулю (не равен нулю).
Код возврата всей конструкции равен коду возврата последней выполненной команды в списке `сп2`. Если ни одна команда из `сп2` не была выполнена, то код возврата равен нулю.

Приведем пример использования перенаправления стандартного входного потока в цикле while. Сценарий в примере 7.19 производит замену расширения `.html` на `.htm`. На вход блоку кода while перенаправляется список html-файлов, найденных командой `find` в текущем каталоге. В цикле используется подстановка переменных с % — удаление самой короткой подстроки (с конца строки), см. п. 7.2.5.

Пример 7.19

```
# Замена html-расширения для файлов текущего каталога
find -name '*.html' | while read a
do
    mv $a ${a%.html}.htm
done
```

7.5.6. Оператор цикла for

Синтаксис

```
for имя [ in сп_слов ]; do сп_ком; done
for (( выр1; выр2; выр3 )); do сп_ком; done
```

Вторая форма записи цикла for — в стиле языка Си.
Выполнение. В `сп_слов` допускаются подстановки в именах файлов. Если список слов не указан, то переменная `имя` будет принимать значения из списка позиционных параметров сценария.
Код возврата всей конструкции равен коду возврата последней выполненной команды в списке `сп_ком`. Если ни одна команда из `сп_ком` не была выполнена, то код возврата равен нулю.

Приведем простой пример использования цикла for для вывода списка файлов текущей директории.

Пример 7.20

```
# Вывод списка нескрытых файлов текущего каталога
for i in *
do
        echo ${i}
done
```

7.5.7. Функции

Под **функцией** языка shell понимается то же, что и в других языках программирования. Функция — это подпрограмма, реализующая некоторый набор операций.

Синтаксис

```
[function] имя () { список команд } [перенаправление]
```

Слово `function` опциональное и может отсутствовать.
Выполнение. Вызов функции осуществляется указанием ее имени в тексте программы. Доступ к входным аргументам функции производится через позиционные переменные ${1}, ${2} и т. д. Аргументы в функцию передаются по значению. Также можно определять локальные переменные с помощью ключевого слова `local`.
Перенаправление ввода. Поскольку функция является обычным блоком кода команд shell, то для этого блока кода можно выполнить перенаправление стандартного потока ввода (stdin). Для этого достаточно после закрывающей фигурной скобки вместо `[перенаправление]` написать, к примеру, `< text.txt`.
Код возврата функции равен коду возврата последней выполненной команды в списке команд. Можно задать код возврата с помощью команды `return [число]`. Если ни одна команда из `сп_ком` не была выполнена, то код возврата равен нулю.

7.5.8. Пример цикла перебора позиционных параметров

Приведем пример сценария, который выводит параметры командной строки в обратном порядке. Для вывода параметров командной строки необходимо организовать цикл для перебора позиционных параметров сценария ${1}, ${2}, ... Для обращения в цикле к значению текущего позиционного параметра с номером `i`, казалось бы, нужно выполнить простую подстановку значения, например такую:

```
pos_param=${$i}
```

Однако такая запись приведет к ошибке:

```
${$i}: bad substitution
```

1-й вариант. Можно использовать встроенную команду `eval`, которая часто используется для вложенных подстановок.

```
eval "pos_param=\${$i}"
```

Сначала будет выполнена подстановка значения переменной `i`.
2-й вариант. На случай обращения к позиционным параметрам в цикле есть специальная подстановка:

```
${!i}
```

Приведем полный текст сценария в примере 7.21. Команда `let`[12] в цикле while используется для удобства. Она возвращает код завершения, равный 1 (ложь), если переменная `i=0` и тем самым обеспечивает выход из цикла while.

Пример 7.21

```
# Вывод позиционных параметров в обратном порядке
i=$#                          # количество параметров
while let i                   # let 0 = 1
   do
     pos_param="${!i}"
     echo "Параметр с номером $i = $pos_param"
     i=$[i-1]
   done
```

7.6. Массивы

Массивы определяются командой `declare -a`. Определяя переменную типа массив, можно сразу присвоить значение элементам и даже указать индексы. Причем индексы не обязательно задавать по порядку. Выделим следующие варианты определения массива:

1) `declare -a имя`;
2) `declare -a имя=(эл_1 эл_2 ...)` ;
3) `declare -a имя=([индекс_1]=эл_1 [индекс_2]=эл_2 ...)`.

Если индексы массива не заданы явно, то нумерация будет автоматической, начиная с нуля.

Кроме вышеперечисленных способов, массив можно инициализировать поэлементно:

[12]Применяется для арифметических вычислений над переменными.

```
имя[индекс]=значение
```

или с помощью функции `read`:

```
read -a имя
```

При использовании функции `read` элементы массива вводятся через пробел. Завершением ввода считается **символ перевода строки**. В этом случае элементы массива будут проиндексированы по порядку, начиная с нуля. Для обращения к элементам массива, списку индексов массива, длине массива и т. д. используют специальные **подстановки переменных**, см. таблицу 8.

Подстановки переменных для массивов Таблица 8

№	Подстановка	Назначение
1	`${имя[индекс]}`	Значение элемента массива с заданным индексом
2	`${имя[*]}`	Набор элементов массива
3	`${имя[@]}`	Аналогично предыдущей подстановке
4	`${#имя[*]}`	Количество элементов в массиве
5	`${#имя[@]}`	Аналогично предыдущей подстановке
6	`${!имя[*]}`	Набор индексов массива
7	`${!имя[@]}`	Аналогично предыдущей подстановке
8	`${!имя[индекс]}`	Длина элемента массива с заданным индексом

Приведем несколько примеров с использованием подстановок для массивов.

Пример 7.22

```
# test.sh: Ввод-вывод элементов массива
read -a arr       # Ввод элементов
echo ${arr[*]}    # Вывод элементов массива
echo ${#arr[*]}   # Вывод количества элементов

# Запуск сценария
$~> ./test.sh
10 20 30 40
10 20 30 40
4
```

Пример 7.23

```
# test.sh: Вывод индексов массива
declare -a arr=([1]=one [5]=five)
arr[6]=six
echo ${!arr[*]}   # Вывод списка индексов

# Запуск сценария
:~> ./test.sh
1 5 6
```

В следующем примере 7.24 переменная `index` принимает значение из множества индексов массива `arr`.

Пример 7.24

```
# test.sh: Вывод непустых элементов массива
declare -a arr=([1]=one [3]= [5]=five [7]= [9]=nine)
for index in ${!arr[*]}
do
    if [ -n "${arr[$index]}" ]; then
        echo ${arr[$index]}
    fi

# Запуск сценария
:~> ./test.sh
one
five
nine
```

§ 8. Средства обработки текста

Обработка текстов — одна из основных операций, выполняемых пользователем с помощью компьютера. При работе в командном режиме это особенно очевидно, в то время как графические среды обычно скрывают большую часть операций по обработке текстов. На самом деле, независимо от того, для каких целей пользователь применяет компьютер и какие программные средства использует, выполняется достаточно много операций обработки текста, например:

- при входе пользователя в систему выполняется проверка совпадения введенных имени и пароля с их контрольными образцами, сохраненными в специальных системных файлах;
- настраивая параметры среды рабочего стола и используемых приложений, пользователь неявно вносит изменения в файлы конфигурации;
- в процессе работы пользователя с прикладными программами неявно модифицируются десятки файлов, например обозреватель сети Интернет пишет данные о посещенных страницах в журнал, файловый менеджер читает содержимое каталога и выводит в удобной графической форме и т. д.;
- пользователь может непосредственно обрабатывать тексты, используя богатый арсенал средств от простых утилит до многофункциональных текстовых процессоров.

В этом разделе рассмотрены утилиты обработки текста, определенные стандартом POSIX.2. Раздел начинается с описания специального языка регулярных выражений, которые активно используются в задачах обработки текста. Второй пункт посвящен описанию ключевого набора утилит обработки текста, ранее составлявших пакет GNU textutils (с апреля 2003 г. текстовые утилиты, утилиты обработки файлов и средства оболочки объединены в один общий программный проект coreutils [16]). В последних двух пунктах описаны важные программы для работы с текстовыми файлами grep и sed.

Помимо базовых утилит, многие современные реализации ОС на базе ядра Linux обычно включают мощные языковые средства манипулирования текстовыми данными, такие как awk, perl и некоторые

другие. Эти средства находятся за пределами данного пособия, исчерпывающие сведения о них можно получить в книгах [11].

8.1. Регулярные выражения

Регулярные выражения — формальный язык описания подстрок в тексте, опирающийся на широкое использование метасимволов. Используя синтаксис регулярных выражений, пользователь может определять шаблоны поиска, соответствующие одной строке или набору строк символов. Регулярные выражения являются одним из наиболее значительных приложений математической теории формальных языков и грамматик. Отметим области практического применения аппарата регулярных выражений:

- проверка корректности формата данных, полученных из ненадежного источника (от пользователя), например, адреса электронной почты, входного имени, пароля и т. д.;
- извлечение определенных данных из файлов структурированных текстовых форматов, например, HTML;
- выделение блоков текста из текстовых файлов для последующей обработки;
- поиск и замена определенных подстрок в текстовых файлах.

Исторически сложилось несколько систем метасимволов для языка регулярных выражений: базовый синтаксис, расширенный синтаксис, совместимые с Perl регулярные выражения.

Далее кратко изложены основные элементы языка регулярных выражений, при этом в основном используется базовый синтаксис, ряд замечаний сделан о применении средств расширенного синтаксиса. Читатели, желающие получить более глубокие сведения о регулярных выражениях, могут обратиться к книге [8].

8.1.1. Символы и наборы символов

Большинство используемых в регулярных выражениях символов не имеют специального значения и представляют самих себя. В простейшем случае регулярное выражение не содержит метасимволов и соответствует единственной совпадающей с ним строке.

Пример 8.1

```
# Это выражение определяет единственную строку abcde
abcde
```

На практике, как правило, бывает недостаточно возможности точного поиска подстроки в строках файла. Например, пользователю может быть важно найти не только словосочетание «скорая помощь», но и все формы («скорую помощь», «скорой помощи» и т. д.). Или, наоборот, при поиске слова «стол» не включать в результаты лишние строки, содержащие слово «стол» в качестве подстроки — «престол», «застолье» и другие.

Для подобных задач поиска язык регулярных выражений предоставляет возможность конструирования шаблонов, относительно точно определяющих интересующий пользователя набор строк. Эта возможность основана на применении метасимволов:

```
[ ] . \ ^ $ | ? * + ( ) { }
```

Следует отметить, что некоторые метасимволы в определенных реализациях языка регулярных выражений могут иметь специальное значение только с экраном, а без него определяют сами себя.

Простейшим метасимволом является символ точки, представляющий любой из символов, за исключением перевода строки.

Пример 8.2

```
# Выражение соответствует любой из строк aac, a5c, a!c,
# т. е. вместо точки может быть любой символ
a.c
```

Набор символов в квадратных скобках представляет любой символ из указанного набора. Для описания интервала символов можно использовать дефис.

Пример 8.3

```
# Любая буква английского алфавита или арабская цифра
[a-zA-Z0-9]
```

Если первым символом в квадратных скобках является ^, то выражение представляет любой символ, не встречающийся в наборе.

Пример 8.4

```
# Любой символ, не представляющий цифру
[^0-9]
```

Если правую квадратную скобку необходимо включить в набор, она должна быть помещена первым символом, дефис — первым или последним. В выражениях в квадратных скобках допустимы классы символов POSIX.

Пример 8.5

```
# Любая строка из a2!, z3! и т. д.
[[:alpha:][:digit:]!]
```

В синтаксисе Perl некоторые классы символов можно заменить специальными метасимволами:

`\d` — цифра, эквивалент `[0-9]`;

`\D` — не цифра, эквивалент `[^0-9]`;

`\s` — пробельный разделитель, эквивалент `[\f\n\r\t]`;

`\S` — не пробельный разделитель, эквивалент `[^\f\n\r\t]`;

`\w` — символ английского алфавита, цифра или подчеркивание, эквивалент `[A-Za-z0-9_]`;

`\W` — не символ английского алфавита, цифра или подчеркивание, эквивалент `[^A-Za-z0-9_]`.

8.1.2. Квантификаторы в регулярных выражениях

Квантификатор — синтаксическая конструкция языка регулярных выражений, определяющая, сколько раз может встречаться предшествующее регулярное выражение в искомой строке:

- `{n}` — ровно n раз,
- `{m,n}` — не менее m и не более n раз,
- `{m,}` — не менее m раз,
- `{,n}` — не более n раз,
- `*` — сколько угодно, т. е. 0 или более (эквивалентно {0,}),
- `+` — по крайней мере один раз, т. е. 1 или более (эквивалентно {1,}),
- `?` — не более одного раза (эквивалентно {0,1}).

В базовом синтаксисе перед символами { } + ? в качестве квантификаторов необходимо ставить экран (обратную косую черту).

Пример 8.6

```
# Любая строка от 3 до 9 символов a
a\{3,9\}
# Любая строка из abc, abbc, ... (эквиваленты)
ab\{1,\}c
ab\+c
abb*c
# Любая строка из символов a, внутри которой может
# встретиться символ b
a\+b\?a\+
```

Квантификаторы могут быть применены к регулярным выражениям.

Пример 8.7

```
# Любая строка
.*
# Любое трехзначное целое число (запись не начинается с 0)
[1-9][0-9]\{1,2\}
```

Как правило, в реализациях регулярных выражений по умолчанию применяется **жадная** квантификация, то есть выражению соответствует максимально длинная подходящая строка. Например, выражению `<.*>` в строке `<b><i>Hello</i></b>, world!` сооветствует подстрока `<b><i>Hello</i></b>`, а не подстрока `<b>`. Для отслеживания более коротких подстрок следует либо уточнять выражение (`'<[^>]*>'`), либо определять выражение как **ленивое**, поставив после него вопросительный знак (во многих реализациях ленивые выражения не поддерживаются).

8.1.3. Группировка и обратные связи

Регулярные выражения можно группировать с помощью круглых скобок (в базовом синтаксисе они также должны быть экранированы).

Пример 8.8

```
# Любая строка в фигурных скобка, состоящая из разделенных
# запятой слов из символов английского алфавита, например
# {abc, def} или {ab, ba, ab, ba}
{\([a-z]\+,\)\+[a-z]\+}
# Подробнее рассмотрим структуру сложного выражения
  # 1. Любое слово из символов английского алфавита
  # с запятой в конце
  [a-z]\+,
  # 2. Любое (не менее одного) число повторений
  # слова с запятой в конце
  \([a-z]\+,\)\+
  # 3. Добавляем справа еще одно слово без завершающей
  # запятой
  \([a-z]\+,\)\+[a-z]\+
```

Иногда требуется отследить строку, содержащую несколько идентичных, но заранее не известных подстрок. Для решения этой задачи можно воспользовать средствами определения обратных связей. Каждая строка, соответствующая регулярному выражению внутри группы, сохраняется в памяти, при этом на каждую из этих строк можно

сослаться по ее номеру. Обратные связи исключены в синтаксисе расширенных регулярных выражений.

Пример 8.9

```
# Одна из строк пам-пам-пам или пум-пум-пум,
# но не пам-пум-пам
\(п[ау]м\)-\1-\1
```

8.1.4. Позиционирование искомой строки

В приложениях часто требуется, чтобы искомая подстрока определенным образом располагалась в просматриваемой строке. Якоря позиционирования в языке регулярных выражений позволяют определить правила размещения.

- `^` — привязка к началу строки,
- `$` — привязка к концу строки,
- `\b` — привязка к границам слова,
- `\B` — привязка к внутренности слова.

Пример 8.10

```
# Любая строка, полностью совпадающая с hello
# (например, вариант hello, world не подходит)
^hello$
# Любое вхождение слова is
# (например, вариант this не подходит)
\bis\b
```

8.1.5. Практические примеры регулярных выражений

Приведем несколько полезных с практической точки зрения примеров использования регулярных выражений. Все приведенные ниже примеры предназначены для проверки корректности формата анализируемых данных (соответствия некоторым ограничениям).

Пример 8.11

```
# Выражение для проверки на соответствие выбранного
# имени или пароля следующим ограничениям:
# - используются только строчные буквы, цифры,
#   знак подчеркивания и дефис
# - допустимая длина составляет 3-8 символов
^[a-z0-9_-]{3,8}$
```

Следующие регулярные выражения используют синтаксис Perl.

Пример 8.12

```
# Адрес электронной почты
/^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$/
# Идентификатор веб-ресурса (URL)
/^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})
([\/\w \.-]*)*\/?$/
# Тэг языка HTML
/^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$/
# Число в формате с плавающей точкой
/^[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?$/
```

8.2. Утилиты обработки текста POSIX

Далее приведен обзор текстовых утилит пакета coreutils, на примерах продемонстрированы типовые сценарии использования команд. Все рассматриваемые утилиты обрабатывают файлы, указанные в качестве параметра, или стандартный ввод, если параметры опущены.

8.2.1. Вывод файлов целиком

В этом пункте рассмотрены команды, работающие с файлами как с одним целым, обеспечивающие минимум трансформаций.

Команды `cat` и `tac` позволяют отправить файлы на стандартный вывод в прямом и обратном порядке следования строк соответственно. Обе команды обеспечивают возможность вывода отметок для специальных невидимых символов. Если указано несколько файлов, их содержимое склеивается (конкатенируется).

Пример 8.13

```
# Вывод содержимого всех файлов с расширением .c
:~> cat *.c
# Вывод файла main.c с нумерацией строк
:~> cat -n main.c
# Вывод данных, полученных со стандартного ввода
# с нумерацией непустых строк
:~> cal 2011 | cat -b
```

Средств нумерации выводимых строк команды `cat` может оказаться недостаточно. Команда `nl` также выводит строки файлов на экран, но при этом предоставляет гораздо большие возможности в управлении нумерацией.

Пример 8.14

```
# Вывод файла main.c с нумерацией строк
:~> nl -b a main.c
# Нумерация строк, начинающихся с символа A
# (поддерживается базовый синтаксис регулярных выражений)
:~> nl -b p^A main.c
# Выравнивание номеров строк по правому краю с дополнением
# нулями
:~> nl -n rz main.c
```

В некоторых случаях бывает необходимо вывести на экран дамп (последовательность кодов) двоичного файла.

Пример 8.15

```
# Вывод шестнадцатеричных (-t x) кодов символов побайтово
:~> echo -n Hello | od -t x1
0000000 48 65 6c 6c 6f
0000005
:~>
```

Команда `od` обеспечивает такую возможность, при этом пользователь может выбрать основание системы счисления для печати кодов: 8, 10 или 16 (по умолчанию 8). Команда `od` печатает смещение относительно начала файла и коды элементов в заданном пользователем формате.

Во многих приложениях возникает задача встраивания двоичных данных в текстовый поток в «читабельной» форме с возможностью последующего извлечения и обратного преобразования. Например, в сообщения электронной почты встраиваются данные вложений. Один из методов кодирования двоичных данных — формат Base64, определяемый стандартом RFC 4648. В пакете coreutils предусмотрена утилита `base64`, выполняющая кодирование в соответствии с этим стандартом.

Пример 8.16

```
# Кодирование строки в base64
:~> echo Hello | base64
SGVsbG8K
# Обратное декодирование
:~> echo SGVsbG8K | base64 -d
Hello
```

8.2.2. Форматирование содержимого файлов

Команда `fmt` переформатирует входной текст, разрывая и склеивая строки, пытаясь создать сбалансированный согласно предложенной длине строки вывод. При этом используется сложная процедура оптимизации переноса слов. Более простой подход к переносу длинных строк предоставляет команда `fold`, разбивая длинные строки по границе символа или слова.

Пример 8.17

```
# Печать содержимого каталога /usr/bin в столбец
# шириной не более 40 символов
ls /usr/bin | fmt -w 40
```

Предусмотрена также команда `pr`, разбивающая текст на страницы, при этом к каждой странице добавляется стандартный заголовок с номером.

8.2.3. Печать фрагментов файлов

Громоздкие файлы можно разбить на файлы меньшего размера с помощью `split`. При этом полученные файлы имеют определенный пользователем префикс (по умолчанию `x`), а суффикс для каждого файла подбирается так, чтобы было удобно впоследствии собрать отрезки воедино в правильной последовательности.

Пример 8.18

```
# Получаем содержимое каталога /usr/bin
:~> ls -1 /usr/bin > listing
# Оценим размер листинга
:~> wc -l listing
3181 listing
# Разобьем на фрагмента по 512 байт
:~> split -b 512 listing
# Выведем список полученных файлов
:~> ls x* | fmt -w 40
xaa xab xac xad xae xaf xag xah xai xaj
xak xal xam xan xao xap xaq xar xas xat
xau xav xaw xax xay xaz xba xbb xbc xbd
xbe xbf xbg xbh xbi xbj xbk xbl xbm xbn
xbo xbp xbq xbr xbs xbt xbu xbv xbw xbx
xby xbz xca xcb xcc xcd xce xcf xcg xch
# Склеим отрезки файлов
:~> cat x* > listing.2
# Отсутствие вывода в команде diff говорит о совпадении
:~> diff listing listing.2
:~>
```

Аналогичная команда `csplit` обеспечивает контекстное разбиение текстовых файлов (то есть по заданному шаблону) на возможно неравные отрезки.

Команды `head` и `tail` позволяют напечатать несколько первых или последних строк файла соответственно.

Пример 8.19

```
# Печать имен первых десяти файлов каталога /usr/bin
:~> ls -1 /usr/bin | head -n 10
# Печать нескольких последних системных сообщений
:~> dmesg | tail
```

8.2.4. Суммирование файлов

Команды суммирования производят числовые характеристики содержимого файлов.

Команда `wc` подсчитывает число символов, слов и строк в заданных файлах или в стандартном вводе.

Пример 8.20

```
# Подсчет числа точек монтирования
# (числа строк файла /etc/mtab)
:~> wc -l /etc/mtab
16 /etc/mtab
:~>
# Подсчет числа строк во всех файлах
# исходного кода в каталоге src
:~> cat src/*.c | wc -l
214
:~>
```

Помимо `wc`, предусмотрены команды для вычисления контрольных сумм файлов: `sum`, `cksum` (CRC), `md5sum` (хэш MD5), `sha1sum` (хэш SHA1). Команды могут использоваться для сверки полученных однажды для определенных файлов контрольных сумм с суммами тех же файлов, вычисленных позднее. Это с высокой степенью надежности позволяет убедиться в том, что файлы не были модифицированы (например, при передаче по сети).

Команды суммирования также могут использоваться для вычисления и последующего хранения хэшей паролей, однако на сегодняшний день используемые этими утилитами алгоритмы нельзя считать надежными.

Пример 8.21

```
# Сгенерируем хэш строки mypassword
:~> echo mypassword | md5sum
d84c7934a7a786d26da3d34d5f7c6c86  -
:~>
```

8.2.5. Операции с отсортированными файлами

Команда `sort` позволяет отсортировать строки заданных файлов в том или ином порядке (по умолчанию — лексикографическом).

Пример 8.22

```
# Отсортируем содержимое каталога /usr/bin в обратном
# лексикографическом порядке
:~> ls -1 /usr/bin | sort -r
# Отсортируем подкаталоги текущего каталога по объему
# Опция -n включает числовую сортировку
:~> du --max-depth 1 | sort -n
```

Если в тексте присутствуют строки-дубликаты, то в отсортированной последовательности они будут расположены рядом. Команда `uniq` позволяет удалить последовательные дубликаты.

Пример 8.23

```
# Получим список имен пользователей,
# имеющих активные сеансы на данном компьютере
:~> who | cut -f 1 -d' ' | sort | uniq
```

К командам, выполняющим операции над отсортированными файлами, также относятся следующие:

- `shuf` переставляет строки файла случайным образом,
- `ptx` генерирует перестановочный индекс,
- `comm` выполняет сравнение отсортированных файлов,
- `tsort` выполняет топологическую сортировку.

8.2.6. Операции с полями файлов

В текстовых файлах данные зачастую структурированы, при этом каждая строка представляет запись, состоящую из нескольких полей, выделенных разделителями.

Команда `cut` позволяет выделить заданные поля, используя либо счетчик символов, либо заданный разделитель.

Пример 8.24

```
# Получим список точек монтирования с типом файловой системы
:~> cat /etc/mtab | cut -d ' ' -f2-3
# Получим список пользователей системы в формате
# имя:комментарий
:~> cat /etc/passwd | cut -d ':' -f1,4
```

Команды `paste` и `join` позволяют выполнить обратную процедуру: объединять заданные файлы либо по столбцам, либо по общему полю.

8.2.7. Операции с символами

Команда `tr` позволяет выполнить фильтрацию потока символов, заменив определенные символы на другие, удалив все вхождения определенных символов или только повторные.

При определении наборов символов допустимо использовать классы символов POSIX.

Пример 8.25

```
# Заменим в выводе все строчные буквы на заглавные
:~> ls | tr 'a-z' 'A-Z'
# Удалим повторные пробелы в выводе команды ls
# для того, чтобы иметь возможность выделять столбцы
# командой cut
:~> ls -l ~ | tr -s ' ' | cut -d ' ' -f 6-8
# Удалим перевод каретки в файлах DOS
:~> tr -d '\015'
```

У программистов есть немало вопросов, вызывающих бурные дискуссии, один из них — какой пробельный разделитель (пробел или табуляцию) лучше использовать при форматировании кода. Среди стандартных текстовых утилит предусмотрены программы `expand` и `unexpand`, выполняющие замену каждого символа табуляции на заданное количество пробелов, и наоборот.

8.3. Поиск по образцу командой grep

Команда `grep` обеспечивает поиск строк в файлах на основе шаблонов, которые могут быть описаны посредством регулярных выражений. Синтаксис вызова команды `grep`:

```
grep [опции] шаблон [список файлов]
```

Если файлы не заданы, `grep` обрабатывает стандартный поток ввода, таким образом может использоваться в конвейерах. Утилита возвращает 0, если совпадения найдены, 1 — если не найдены и 2 — в случае ошибки. Если шаблон содержит метасимволы регулярных выражений, следует защитить его от подстановок оболочки экранами.

По умолчанию команда `grep` использует базовый синтаксис регулярных выражений, однако пользователь может переключиться на использование расширенного или Perl-совместимого синтаксиса, задав опцию `-E` или `-P` соответственно.

Рассмотрим наиболее часто используемые сценарии применения `grep` на примерах.

Команда `grep` позволяет указать данные для фильтрации несколькими способами.

Пример 8.26

```
# Поиск строки в одном файле
:~> grep 'this' /usr/share/dict/words
# Поиск строки в нескольких файлах
:~> grep 'main' *.c
# Поиск в стандартном вводе
:~> ps -Af | grep 'bash'
# Рекурсивный поиск во всех файлах и вложенных каталогах
:~> grep -r '#define' src
```

Результат поиска в форме списка найденных строк может оказаться вырванными из контекста и неинформативным. Команда `grep` позволяет вместе с найденной строкой вывести несколько соседних.

Пример 8.27

```
# Выводим каждый результат с 2 предшествующими строками
:~> grep -A 2 'main' main.c
# Выводим каждый результат с 2 последующими строками
:~> grep -B 2 'main' main.c
# Выводим каждый результат с 2 строками до и после
:~> grep -C 2 'main' main.c
```

Следующие примеры демонстрируют применение опций для изменения поведения `grep`.

Пример 8.28

```
# Нечувствительный к регистру поиск
:~> grep -i 2 'MAIN' main.c
# Поиск строк, не удовлетворяющих шаблону
:~> grep -v '^#' myscript.sh
# Вывод только частей строк, удовлетворяющих шаблону
:~> grep -o '<a[^>]\+>' page.html
```

По умолчанию `grep` ищет подстроку в строке, представляющей собой просто набор символов, т. е. более крупные единицы строки не идентифицируются.

Пример 8.29

```
# Поиск полного совпадения со словом
:~> grep -w 'is' /usr/share/dict/words
# Поиск полного совпадения со строкой
:~> grep -x 'is' /usr/share/dict/words
```

Пользователь может быть заинтересован не в результате поиска, а в некоторых характеристиках этого результата.

Пример 8.30

```
# Печать количества найденных строк
:~> grep -c 'is' /usr/share/dict/words
105
# Вывод сведений о местоположении найденного образца
:~> grep -n -b -w 'this' /usr/share/dict/words
339145:3464565:this
```

Для поиска строк в архивах, сжатых `gzip` и `bzip2` предусмотрены соответственно команды `zgrep` и `bzgrep`.

8.4. Потоковый редактор sed

Потоковый редактор `sed` — утилита пакетной (неинтерактивной) обработки текста. Редактор `sed` трансформирует входной поток, поступающий из файла или стандартного ввода, по заданным правилам — программе. Таким образом, `sed` — специализированный интерпретатор, ориентированный на работу с текстом. Поскольку sed обрабатывает в один проход, его применение весьма эффективно, а за счет гибкого языка описания текстовых правок, с помощью сценариев sed можно описать функционал практически всего набора команд GNU textutils. Поскольку отредактированный текст выводится непосредственно в поток стандартного вывода, sed используется в первую очередь в конвейерах. Синтаксис вызова команды `sed`:

```
sed [-n] программа [список файлов]
sed [-n] -f файл программы [список файлов]
```

Первая форма позволяет задавать программу на языке текстовых правок `sed` непосредственно в командной строке, вторая — обеспечивает возможность чтения программы из указанного файла.

Необязательная опция `-n` подавляет вывод строк на стандартный поток вывода, кроме тех, которые выводятся непосредственно командой языка правок «p».

Из полезных опций следует отметить также `-iсуффикс`, позволяющую изменить непосредственно обрабатываемый файл. Если суффикс непуст, создается резервная копия исходного файла.

Программа для `sed` состоит из одной или нескольких строк следующего вида:

```
[адрес[, адрес]] инструкция [список аргументов]
```

Инструкция представляет собой команду редактирования. Наличие, количество и вид аргументов зависят от примененной инструкции, см. таблицу 9.

Основные инструкции `sed` Таблица 9

№	Код	Действие
1	`a`	добавляет одну или более строк к каждой адресованной
2	`c`	заменяет адресованные строки указанным текстом
3	`d`	подавляет вывод адресованных строк
4	`i`	аналогична «a», однако добавляет указанный текст перед адресованными строками
5	`p`	выводит адресованную строку. Если опция -n в вызове sed отсутствует, строка будет выведена дважды
6	`q`	завершает цикл sed немедленно
7	`r файл`	считывает содержимое указанного файла и добавляет его к адресованной строке
8	`w файл`	выводит адресованные строки в указанный файл
9	`n`	выводит текущую строку и считывает следующую
10	`N`	считывает следующую строку, добавляя ее к текущей
11	`s`	выполняет поиск-подстановку на основе регулярного выражения. Инструкция имеет вид: s/шаблон/строка замены/[g][p][w файл]

Адреса задают строки файла, подлежащие обработке, если они опущены — обрабатываются все строки. Адреса могут быть следующего вида:

1) Номер строки непосредственно адресует строку. Специальный случай — номер $ представляет последнюю строку файла.

2) Регулярное выражение адресует все строки, удовлетворяющие ему. Регулярное выражение должно быть ограничено разделителями, при этом на практике чаще всего используются символы «/», но sed допускает любые разделители, кроме «backslash» и символа новой строки.

Если заданы два адреса, sed обрабатывает группу строк начиная с той, которую указывает первый адрес, второй указывает последнюю строку. При этом в обрабатываемом файле может быть несколько групп.

Редактор sed работает с двумя буферами: буфер pattern space по умолчанию содержит только что считанную из входного потока строку, буфер hold space — вспомогательный буфер, в который программно можно переместить строки для временного хранения. Инструкции управления буферами приведены в таблице 10.

Инструкции управления буферами Таблица 10

№	Код	Действие
1	`g`	копирует содержимое HS в PS
2	`G`	добавляет символ новой строки и содержимое HS в PS
3	`h`	копирует содержимое PS в HS
4	`H`	добавляет символ новой строки и содержимое PS в HS
5	`x`	x обменивает содержимое PS и HS

В завершение приведем несколько примеров использования `sed`.

Пример 8.31

```
# Печать первых десяти слов системного словаря
sed -n -e '1,10p' /usr/share/dict/words
# Вывод всех строк файла /etc/services кроме первой
sed -e '1d' /etc/services
# Печать всех строк не короче 65 символов
cat somefile | sed -n '/^.\{65\}/p'
# Вывод только функции main файла на языке Си
sed -n -e '/main[[:space:]]*(/,/^}/p' file.c
```

Список литературы

[1] *Браун П.* Макропроцессоры и мобильность программного обеспечения: (Математическое обеспечение ЭВМ). Пер. с англ. М.: Мир. 1977. 253 с.

[2] *Бурк Р., Хорват Д.* UNIX для системных администраторов: Энцик. пользователя. Киев: ДиаСофт, 1998. 496 с.

[3] *Галатенко В. А.* Программирование в стандарте POSIX. Ч. 1. Интернет-ун-т. информ. технологий. М.: ИНТУИТ.ру, 2004. 560 с.

[4] Командный язык Shell для пользователей: Учеб. метод. комплекс/ Ин-т операц. систем при МИЭТ; Пер. с англ.: Ю. А. Богоявленский, О. Ю. Богоявленская, Г. С. Сиговцев. М.: Науч. книга, 1995. 354с. (UNIX System V. UNIX-технологии)

[5] *Олифер В. Г., Олифер Н. А.* Сетевые операционные системы. СПб.: Питер, 2001. 544 с.

[6] *Тейнсли Т.* Linux и UNIX: программирование в shell: Рук. разработчика: Пер. с англ. К.: BHV, 2001. 464 с.

[7] *Cooper M.* Advanced Bash-Scripting Guide. Version 6.3 [Электронный ресурс]. Б. м., 2011. Режим доступа: http://tldp.org/LDP/abs/html/, свободный.

[8] *Goyvaerts J., Levithan S.* Regular Expressions Cookbook. Sebastopol O'Reilly Media, 2009. 512 pp.

[9] *Gruenbacher A.* POSIX Access Control Lists on Linux [Электронный ресурс]. Б. м., 2003. Режим доступа: http://www.suse.de/ agruen/acl/linux-acls/, свободный.

[10] *Johnson C.* Pro Bash Programming. New York, Apress, 2009. 264 pp.

[11] *Sobell M.* A Practical Guide to Linux(R) Commands, Editors, and Shell Programming. Prentice Hall PTR, 2005. 1008 pp.

[12] *Stallman R. M.* Free Software, Free Society: Selected Essays of Richard M. Stallman. Boston: GNU Press, 2010. 266 pp.

[13] *Tuck M.* The Real History of the GUI [Электронный ресурс]. Б. м., 2001. Режим доступа: http://www.sitepoint.com/real-history-gui/, свободный.

[14] Free Software Foundation. Bash Builtin Commands [Электронный ресурс]. Б. м., 2010. Режим доступа: http://www.gnu.org/software/bash/manual/html_node/Bash-Builtins.html, свободный.

[15] Free Software Foundation. Bash Reference Manual [Электронный ресурс]. Б. м., 2010. Режим доступа: http://www.gnu.org/software/bash/manual/, свободный.

[16] Free Software Foundation. Core GNU utilities manual [Электронный ресурс]. Б. м., 2011. Режим доступа: http://www.gnu.org/software/coreutils/manual/, свободный.

[17] Free Standards Group. Linux Standard Base Specification 1.0.0 [Электронный ресурс]. Режим доступа: http://refspecs.freestandards.org/LSB_1.0.0/gLSB.html, свободный.

[18] IEEE Standards Association [Электронный ресурс]. Режим доступа: http://standards.ieee.org/, свободный.

[19] The GNU Operating System [Электронный ресурс]. Б. м., 2011. Режим доступа: http://www.gnu.org/, свободный.

Учебное издание

Бородин Александр В
Бородина Александра В

ОПЕРАЦИОННЫЕ СРЕДЫ, СИСТЕМЫ И ОБОЛОЧКИ

Базовый курс

Редактор *Т. В. Климюк*
Компьютерная верстка *А. В. Бородиной*

Подписано в печать . .11. Формат 60x84 1/16
Бумага офсетная. Уч.-изд. л. 5. Тираж 120 экз. Изд. №175

Отпечатано в типографии Издательства ПетрГУ
185910, г. Петрозаводск, пр. Ленина, 33