ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ БЮДЖЕТНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «СИБИРСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

С.Н. Мамойленко
О.В. Молдованова

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебное пособие
Часть 1. Операционная система Linux

Новосибирск
2012

УДК 004.451
ББК 32.973-018.2

Мамойленко С.Н., Молдованова О.В. Операционные системы: Учебное пособие. Часть 1. Операционная система Linux. 2-е изд., доп. /СибГУТИ.– Новосибирск, 2012. – 128с.

Учебное пособие предназначено для студентов, обучающихся по направлению 230100 «Информатика и вычислительная техника» и изучающих дисциплину «Операционные системы». В нём содержится материал, предназначенный для проведения практических или лабораторных занятий по указанному учебному курсу с целью изучения операционных систем семейства UNIX (на примере Linux).

Кафедра вычислительных систем

Ил. – 8, табл. – 15, список лит. – 10 наим.

Рецензент: доцент Кафедры телекоммуникационных сетей и вычислительных средств ФГОБУ ВПО «СибГУТИ» О.И.Моренкова.

Для студентов, обучающихся по направлению 230100 «Информатика и вычислительная техника».

Утверждено редакционно-издательским советом ФГОБУ ВПО «СибГУТИ» в качестве учебного пособия.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Функционирование любой современной электронной вычислительной машины (ЭВМ) невозможно без специального программного обеспечения, называемого операционной системой (ОС).

Операционная система – это комплекс средств, предназначенный для управления вычислительными ресурсами ЭВМ и организации взаимодействия пользователя (прикладного программного обеспечения) и ЭВМ.

В рамках подготовки специалистов по направлению 230100 «Информатика и вычислительная техника» особое внимание уделяется изучению принципов построения операционных систем. Согласно государственному образовательному стандарту первое знакомство с этими принципами студенты получают в рамках одноимённого курса, отнесённого к базовой части (Б.3) профессионального блока дисциплин. Очевидно, что ограничиваться только получением теоретических навыков нецелесообразно для студентов. Важной является организация практических занятий, помогающих студентам закрепить полученные теоретические знания.

В указанном направлении имеется несколько профилей подготовки, в том числе 230101 – «Вычислительные машины, комплексы, системы и сети» и 230105 – «Программное обеспечение средств вычислительной техники и автоматизированных систем». Именно для студентов, обучающихся по этим профилям подготовки, предназначено данное учебное пособие.

За всю истории развития вычислительной техники человечеством разработано множество различных ОС, различающихся как по архитектуре, так и по набору выполняемых функций. Последнее время все ОС принято подразделять на два семейства – Windows и UNIX.

В учебном пособии представлен материал, предназначенный для организации практических занятий по изучению основных принципов работы операционных систем семейства UNIX. Все примеры разработаны для выполнения под управлением свободно распространяемой версии ОС семейства UNIX, называемой Linux. Несмотря на то, что сегодня под маркой Linux распространяются различные дистрибутивы, примеры могут выполняться в любом из них, так как рассчитаны на изучение ядра ОС, которое одинаково во всех дистрибутивах.

ГЛАВА 1. ИСТОРИЯ СОЗДАНИЯ ОПЕРАЦИОННОЙ СИСТЕМЫ LINUX

«Я стал распространять свою операционную систему прежде всего, чтобы доказать, что всё это не пустая болтовня – я действительно что-то сделал».

Линус Торвальдс

Операционная система (ОС) Linux изначально разработана Линусом Торвальдсом (рисунок 1), студентом Унивеситета Хельсинки (University of Helsinky, Финляндия).

Рисунок 1. Линус Торвальдс. Создатель операционной системы Linux

Линус Торвальдс родился 28 декабря 1969 года в городе Хельсинки (Финляндия). Родители, шведскоговорящие финны Нильс и Анна Торвальдс, были в 60-х годах студентами-радикалами, отец даже был коммунистом, в середине 70-х проведшим год в Москве. Линус был назван в честь Линуса Полинга[1]. В школе преуспевал в физике и математике. Был малообщительным, скромным мальчиком. Его часто дразнили из-за политических взглядов его отца. В 1988 году Линус поступил в Университет Хельсинки, который окончил в 1996 году, получив степень магистра кибернетики.

Значимым событием в жизни Торвальдса было прочтение им книги Эндрю Таненбаума «Операционные системы: разработка и реализация» (Operating Systems: Design and Implementation). В этой книге на примере написанной Таненбаумом ОС Minix представлена структура систем семейства UNIX.

В начале 1991 году Линус Торвальдс приобрёл персональный компьютер, собранный на базе центрального процессора Intel 80386, который на тот момент был только что выпущен компанией Intel. В комплекс поставки входила урезанная версия операционной системы DOS, но Линус был очень заинтересован Minix, которую он первым делом и установил.

Изучив систему Minix, а также прочитав обсуждение этой системы в посвящённой ей телеконференции, Линус обнаружил в ней множество недостатков. Больше всего нареканий вызвала программа эмуляции терминала, которую Линус использовал для подключения к университетскому компьютеру.

[1] Линус Карл Полинг (англ. Linus Carl Pauling; 1901 – 1994) – американский химик, кристаллограф, лауреат двух Нобелевских премий: по химии (1954) и премии мира (1962), а также Международной Ленинской премии «За укрепление мира между народами» (1970).

Несмотря на то, что исходные коды ОС Minix были доступны в полной мере, Линус решил не дорабатывать уже имеющуюся программу, а написать собственную программу эмуляции терминала, которая использовала только возможности аппаратуры и никак не была связана с Minix. Для чего первым делом ему пришлось детально изучить, как функционирует центральный процессор Intel 80386.

Первая тестовая программа, которую написал Линус, имела два независимых процесса, один из которых использовался для выдачи на экран буквы A, а другой – для выдачи буквы B. С практической точки зрения это было абсолютно бессмысленно, но зато позволило ему изучить принципы переключения между процессами. На написание этой программы Линусу потребовался почти месяц, потому что во всём ему приходилось разбираться с нуля, читая горы технической документации.

В дальнейшем один процесс был доработан таким образом, чтобы он читал информацию, поступающую от модема, и выдавал её на экран, а другой – считывал информацию с клавиатуры и передавал её модему.

Со временем эмулятор Линуса обрастал дополнительными функциями: появилась возможность передавать файлы (для чего потребовалось разработать драйвер для доступа к файловой системе ОС Minix), появилась первая версия командной оболочки (переделанная из shell) и т.д. Линус называл её «программой эмуляции терминала типа gnu-emacs». Gnu-emacs начинался как текстовый редактор, но его создатели встроили в него множество разных функций.

Первым упоминанием о том, что Линус задумался о разработке собственной операционной системы, было сообщение, сделанное им 3 июля 1991 года в телеконференции Minix, которое содержало следующее:

> *Привет, сетяне!*
> *Я сейчас делаю один проект (под minix), и мне нужно определение стандартов POSIX. Кто-нибудь знает, где можно взять их последнюю версию, желательно в электронном виде? Ftp-сайты годятся.*

Стандарты POSIX – это подробнейшие правила для каждого из системных вызовов в UNIX. Они разрабатываются специальной организацией, состоящей из представителей компаний, которые хотят договориться об общих стандартах для UNIX.

Это сообщение не вызвало особого интереса, однако было замечено Ари Лемке, преподавателем из Технического университета Хельсинки, который обратился к Линусу с предложением, выложить его операционную систему (когда она будет готова) на FTP-сайт, чтобы все желающие могли ознакомиться с ней.

Первым именем для нынешней операционной системы Linux было Freax (от англ. freak – фанат и на конце x от UNIX). Это имя не понравилось Ари Лемке. Ему больше нравилось рабочее название – Linux, которое Линус использовал в процессе разработки своей операционной системы.

Получив письмо от Ари Лемке, Линус ещё больше воодушевился созданием новой операционной системы, о чём свидетельствует его сообщение в телеконференции:

Привет всем пользователям minix! Я тут пишу (бесплатную) операционную систему (любительскую версию – она не будет такой большой и профессиональной, как gnu) для 386-х и 486-х AT. Я вожусь с этим с апреля, и она, похоже, скоро будет готова. Напишите мне, кому, что нравится/не нравится в minix, поскольку моя ОС на неё похожа (кроме всего прочего, у неё – по практическим соображениям – то же физическое размещение файловой системы).
Пока что я перенёс в неё bash (1.08) и gcc (1.40), и всё вроде работает. Значит, в ближайшие месяцы, у меня получится уже что-то работающее, и мне бы хотелось знать, какие функции нужны большинству. Все заявки принимаются, но выполнение не гарантируется :-)

Линус.

PS. Она свободна от кода minix и включает мультизадачную файловую систему. Она НЕ переносима (используется переключение задач 386 и пр.) и, возможно, никогда не будет поддерживать ничего, кроме AT-винчестеров – потому что у меня больше ничего нет :-(.

17 сентября 1991 года Линус выложил исходный код программы (версии 0.01) для общедоступной загрузки. Система сразу же вызвала большой интерес. Линусу начали приходить сообщения об ошибках. В начале октября 1991 года была выпущена версия 0.02, с исправлением ошибок и добавлением некоторых программ. В ноябре была выпущена версия 0.03. В конце ноября появилась версия 0.10. Такой скачёк нумерации был связан с тем, что Линусу пришлось полностью отказаться от Minix (так как он по ошибке испортил раздел жёсткого диска, на котором она находилась) и начать использовать только собственную операционную систему. Ещё через несколько недель появилась версия 0.11. В начале января 1992 года была выпущена версия 0.12, в которой Линус реализовал механизмы страничной подкачки. После этого популярность свободно распространяемой операционной системы Linux начала расти не по дням, а по часам.

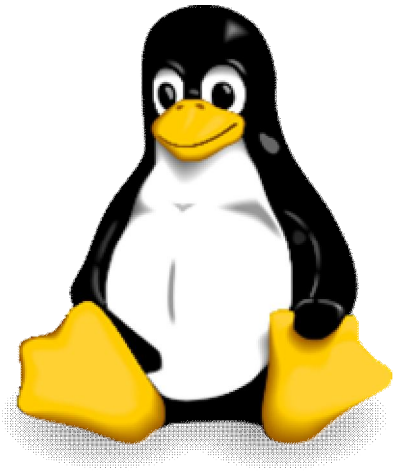

Рисунок 2. Пингвин Такс. Личный талисман Линуса Торвальдса, ставший эмблемой ОС Linux

В течение нескольких лет ядро новой системы дорабатывалось, и в марте 1994 года в аудитории Факультета информатики Университета Хельсинки была представлена версия 1.0.

С тех пор прошло уже больше пятнадцати лет. Сотни, потом тысячи программистов стали интересоваться системой и работать над её улучшением и дополнением. Она распространялась и по сей день распространяется на условиях универсальной общественной лицензии GNU[2] (англ. GNU General Public License, GNU GPL). Появилось немало коммерческих версий этой операционной системы. По сути, сегодня под названием Linux скрывается ядро операционной системы, которое снабжается множеством прикладных программ, создаваемых коллективом программистов со всего мира.

В заключение истории Linux следует сказать, что эмблемой ОС Linux является пингвин Такс (Tux) – личный талисман Линуса Торвальдса (рисунок 2).

Контрольные вопросы

1. Расскажите об истории создания ОС Linux.
2. Каким было первое имя нынешней ОС Linux?
3. В каком году вышла версия 1.0 ОС Linux?
4. На каких условиях распространяется ОС Linux?

[2] Лицензия на свободное программное обеспечение, созданная в рамках проекта GNU в 1988 г. Цель – предоставить пользователю права копировать, модифицировать и распространять (в том числе на коммерческой основе) программы, а также гарантировать, что и пользователи всех производных программ получат вышеперечисленные права.

ГЛАВА 2. ПЕРВОЕ ЗНАКОМСТВО С LINUX

«Вы скорбите о тех временах, когда мужчины были настоящими мужчинами и сами писали драйверы устройств?»

Из объявления Линуса Торвальдса о выпуске Linux версии 0.0.2.

Первое впечатление от работы с любым компьютером связано с тем, каким образом он «общается» с пользователем. Способ общения определяется ***интерфейсом*** используемой операционной системы.

2.1. Интерфейс операционной системы

В Linux, как и в любой другой ОС семейства UNIX, может применяться два типа интерфейса: *текстовый* и *графический*.

В первом случае (рисунок 3) пользователь «общается» с компьютером, вводя команды с клавиатуры, и получает ответ в виде текстовых сообщений на экране монитора. Во втором случае (рисунок 4) пользователь имеет возможность просматривать графическую информацию на экране и управлять работой компьютера при помощи клавиатуры и дополнительных средств ввода, таких как манипулятор «мышь», световое перо, сенсорный экран и т.п.

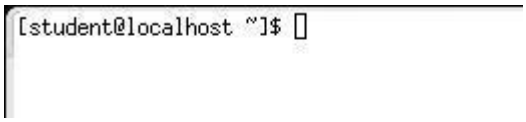

Рисунок 3. Вид текстового интерфейса операционной системы (командная оболочка)

Рисунок 4. Вид графического интерфейса операционной системы (окно входа в систему)

Чаще всего пользователям предоставляется несколько виртуальных терминалов[3], каждый из которых может использовать свой тип интерфейса. Переключение между терминалами производится комбинацией клавиш CTRL+ALT+F?[4], где F? означает одну из функциональных клавиш F1–F12.

[3] Что такое терминал, смотрите в курсе «Организация ЭВМ и систем».

[4] Если текущий терминал использует текстовый интерфейс, то для переключения можно использовать комбинацию клавиш <Alt+F?>.

Обычно (хотя и не всегда) первые шесть терминалов используют текстовый режим, а седьмой терминал – графический.

2.2. Процедура регистрации пользователя в системе

Независимо от используемого типа интерфейса первое, что необходимо сделать прежде, чем работать с операционной системой Linux – это пройти ***процедуру идентификации*** (регистрации), которая преследует две цели: во-первых, система проверяет, тот ли Вы, за кого себя выдаёте, и, во-вторых, она открывает сеанс работы и настраивает для пользователя рабочую среду.

Независимо от типа используемого интерфейса в процессе регистрации, называемой так же «logging in», система запрашивает имя пользователя[5] (*login*) и пароль (*password*).

Имя пользователя чаще всего вводится с клавиатуры. Иногда, если число пользователей компьютера невелико и используется графический интерфейс, то предоставляется возможность выбрать пользователя из списка. Пароль всегда вводится с клавиатуры, причём набираемые символы на экране не отображаются (или отображаются в виде каких-либо других символов, например символа * – «звёздочка»). Ввод имени пользователя и пароля завершается нажатием клавиши Enter[6]. Позиция (или поле), куда будет вводиться информация на экране, обычно указывается при помощи ***курсора*** – мигающей вертикальной или горизонтальной черты.

Пример регистрации в системе с использованием текстового режима. После загрузки операционной системы на экране появится приглашение для ввода имени пользователя (*login*):

```
ASPLinux release 10.0
Kernel 2.6.20 on an i686
Welcome to Computer systems chair.
login: _
```

В этом случае в поле имя (*login*) надо ввести с клавиатуры строку, соответствующую Вашему системному имени (например, *student)* и нажать клавишу Enter. После этого система попросит ввести пароль (*password*):

```
ASPLinux release 10.0
Kernel 2.6.20 on an i686
Welcome to Computer systems chair.
login: student
password:
```

Если имя пользователя и пароль введены неправильно, то будет выведено сообщение об ошибке (англ. login incorrect), и процедура ввода повторится за-

[5] Пользователи создаются администратором системы.

[6] Обратите внимание, что в именах пользователей и паролях учитывается регистр символов. Поэтому под именами Student и student могут работать два совершенно разных пользователя.

ново[7]. В случае правильного ввода на экран будет выдано приглашение к вводу команд (если иное не предусмотрено рабочей средой пользователя), и система будет готова к их исполнению:

```
ASPLinux release 10.0
Kernel 2.6.20 on an i686
Welcome to Computer systems chair.
login: student
password:
[student@rwp1 student]$_
```

При использовании графического интерфейса процедура регистрации производится аналогичным образом, а после её завершения появится графическое рабочее пространство, называемое рабочим столом (рисунок 5).

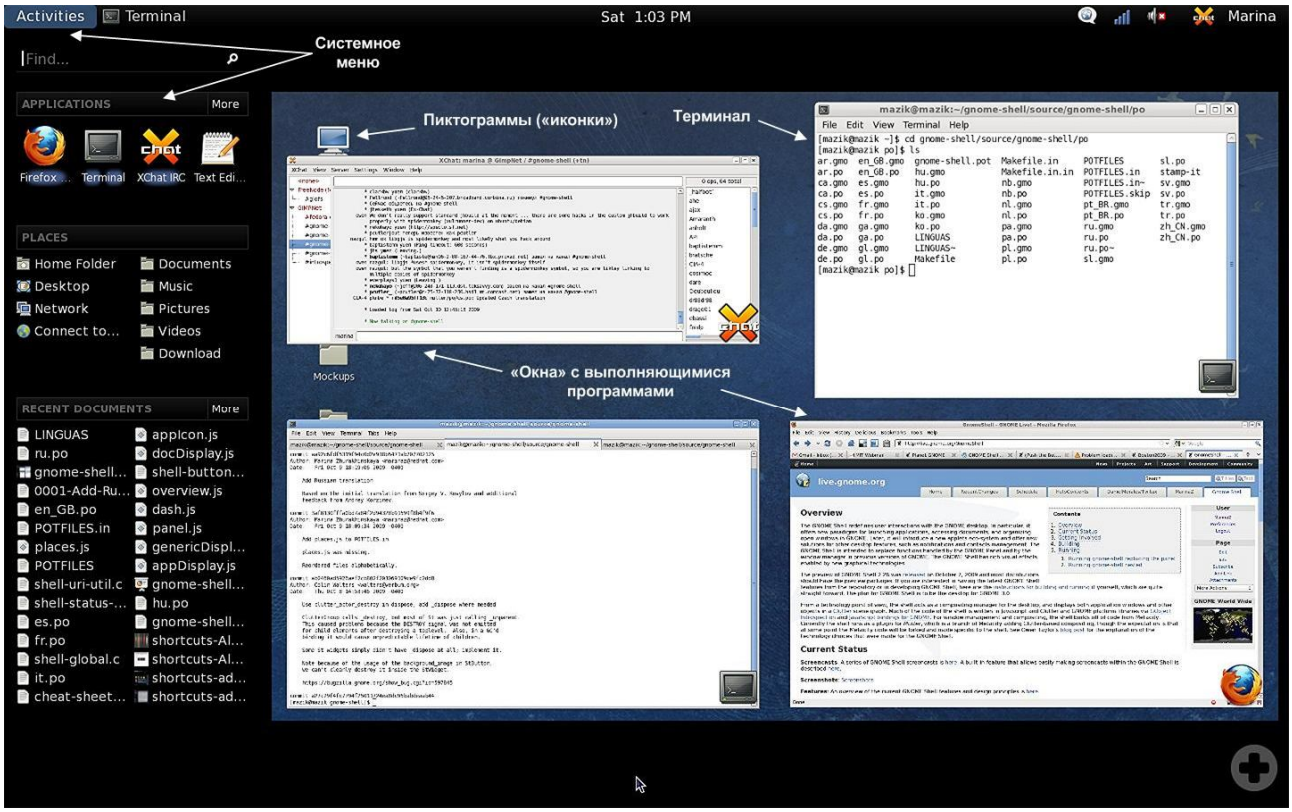

Рисунок 5. Вид рабочего стола после регистрации в системе

2.3. Процедура завершения сеанса работы с системой

Когда пользователь закончил работу с системой, он должен завершить свой сеанс работы. Для этого он должен ввести команду **logout** и нажать клавишу Enter, после чего операционная система завершит текущий сеанс, очистит экран и вернётся в режим регистрации пользователя. В графическом режиме обычно вместо команды *logout* пользователь выбирает один из пунктов системного меню (рисунок 5).

[7] Следует помнить, что в случае нескольких неудачных попыток входа в систему компьютер может быть заблокирован.

Контрольные вопросы

1. Что такое интерфейс ОС? Назовите типы интерфейсов ОС Linux.
2. Какая комбинация клавиш используется для переключения между терминалами?
3. Какие цели преследует процедура идентификации? Расскажите, в чём она состоит.
4. В чём разница процедуры идентификации в текстовом и графическом интерфейсе ОС?
5. Как завершить сеанс работы в ОС?

ГЛАВА 3. СИСТЕМА ХРАНЕНИЯ ИНФОРМАЦИИ

«Мой эмулятор терминала обрастал наворотами. Я регулярно использовал его, чтобы подключиться к университетскому компьютеру и получить почту или поучаствовать в конференции по Minix. Беда была в том, что я хотел скачивать и закачивать файлы. То есть мне нужно было уметь писать на диск. Для этого моей программе эмуляции нужен был драйвер дисковода. А ещё ей был нужен драйвер файловой системы, чтобы она могла вникать в организацию диска и записывать скачиваемые файлы».

Линус Торвальдс.
Из книги «Just for fun».

Основным назначением любой вычислительной машины является обработка информации. Этот процесс требует наличия двух составляющих – саму информацию и правила (алгоритмы, программы), согласно которым эта информация будет обрабатываться. Поэтому, приступая к работе с компьютером, важно чётко понимать, как в нём хранятся данные.

В современных компьютерах вся хранимая информация представляется в виде ***файлов***, каждый из которых имеет своё имя, содержимое и описывается определённым набором атрибутов.

3.1. Имена файлов. Маски файлов

Имя файла – последовательность символов из заданного алфавита, включающего русские и английские буквы, цифры и символы: подчёркивание (_), тире (-) и точка (.). В операционной системе Linux *длина имени* файла ограничена 255 символами. Часто (хотя это и необязательно) файлам дают такие имена, чтобы они отражали их содержание. Например, файл с именем *lab1.c* хранит в себе программу на языке Си.

Если в имени файла присутствует точка, то часть имени, следующая после неё, называется **расширением** или **суффиксом**. Обычно (хотя это и необязательно) суффиксы используются для того, чтобы указать тип содержимого файла. В Linux имя файла может содержать несколько точек, что используется для указания на изменения типов содержимого файла. Например, имя *lab1.c.gz* говорит о том, что файл *lab1.c*, содержащий программу на языке Си, был сжат при помощи программы gzip.

Если **точка является первым** символом в имени файла, то она указывает на специальное назначение этого файла[8]. Например, файл *.bashrc*.

В операционной системе Linux в именах файлов кроме символов из заданного алфавита возможно использование других (специальных) символов, например, пробел, звёздочка (*), процент (%) и т.п. В этом случае[9], чтобы

[8] Обычно такие файла содержат настройки программ пользователя.

[9] При работе с файлами, содержащими в своих именах специальные символы, с использованием графического интерфейса или специальных программ-оболочек этого делать нет необходимости (операционная система сама преобразует имя файла в нужную форму).

указать специальный символ, необходимо перед ним поставить знак «\» **(обратный слеш)** или заключить имя файла в **одинарные или двойные кавычки**[10]. Такой способ также называют **цитированием символа**. Например,
*Имя\ файла\ со\ специальными\ знаками\ *_и_\?.doc* или
*'Имя файла со специальными знаками *_и_?.doc'* или
*"Имя файла со специальными знаками *_и_?.doc"*.

Часто при работе в командной оболочке пользователям необходимо в одном имени указать сразу несколько файлов (группу)[11]. Для этого в имена файлов включаются специальные символы[12] «*» (звёздочка), «?» (вопрос), «[]» (квадратные скобки), которые называются **символами расширения**. Имя, содержащее эти символы, преобразуется командной оболочкой в список имён.

Знак **«*» (звёздочка)** применяется для того, чтобы указать, что в этом месте имени файла может находиться любое число (включая нуль) любых символов. Например, имя *a** определяет все файлы, начинающиеся с буквы «а». Имя **xx*.dat* включает любое имя файла, оканчивающееся символами *.dat*, в имени которого присутствуют буквы «xx». Это могут быть имена *abxx.dat*, *lxx33.dat*, *xxyyzz.dat* или даже *xx.dat*.

Знаком **«?» (вопрос)** указывается, что на его месте в имени файла может находиться один (не больше и не меньше) любой символ. Например, имя *???* определяет все файлы, имена которых состоят только из трёх символов. Сгенерировать список имён, окачивающихся тремя символами, отделёнными от остальной части имени точкой, позволяет выражение *.???*.

«[]» (квадратные скобки) используются для указания множества символов, которые могут находиться в имени файла на том месте, где располагается открывающаяся квадратная скобка. Между скобками указываются необходимые символы или их диапазон (с использованием символа «-» (тире)). Например, имя *f[124]*, соответствует файлам *f1*, *f2*, *f4*. А имя *f[1-4]* – файлам *f1*, *f2*, *f3*, *f4*. Имя *f[abc-f]* – файлам *fa*, *fb*, *fc*, *fd*, *fe*, *ff*. С помощью квадратных скобок и знака «-» можно указывать сразу несколько диапазонов символов. Например, *[123, 5-8, abe, A-E]*. Следует отметить, что если символы «*» и «*?*» указаны внутри квадратных скобок, то они являются обычными символами, а не символами расширения. А символ «-» наоборот имеет смысл диапазона только внутри скобок. Например, имя *-[*?]abc* будет определять группу из всего лишь двух имён: *-*abc* и *-?abc*.

3.2. Система именования файлов. Каталоги. Путь файла

Для структурирования хранимой информации файлы, в свою очередь, объединяются в ***каталоги*** (директории или папки), организованные в виде

[10] Об отличии одинарных кавычек от двойных будет рассказано далее.
[11] Например, для того чтобы удалить несколько ненужных файлов с похожими именами.
[12] Эти символы указываются без обратного слеша. Если перед ними поставить обратный слеш, то они потеряют свою функцию символов расширения, а станут просто символом «звездочка», «вопросик», «квадратные скобки».

древовидной структуры (дерева). **Имя каталога**, как и имя файла, – это последовательность символов из заданного алфавита.

Основным каталогом для файловой системы является корневой каталог, обозначаемый символом **«/» (слеш)**. Все остальные файлы и каталоги располагаются в рамках структуры, порождённой корневым каталогом, независимо от их физического местонахождения[13].

Чтобы указать, в каком каталоге находится файл, нужно определить его **путь** – перечень каталогов, которые необходимо пройти до файла. При перечислении каталоги разделяются символом **«/» (слеш)**. Например, путь *кат1/кат2/кат3/файл* – означает, что *файл* находится в каталоге *кат3*, который находится в каталоге *кат2*, находящемся в каталоге *кат1*.

Указывать путь файла можно либо ***относительно*** (начало пути находится в каком-то каталоге), либо ***абсолютно*** (началом пути является корневой каталог). Например, *кат1/кат2* – это относительный путь, в котором говорится, что надо перейти в *кат1*, который находится в текущем каталоге, а затем – в *кат2*. А */кат1/кат2* – это абсолютный путь, в котором говорится, что надо перейти в *кат1*, который находится в корневом каталоге, а затем – в *кат2*.

Для указания в относительном пути текущего или родительского каталога используются символы «.» (**точка**) и «..» (**две точки**) соответственно. Путь *./файл* означает, что *файл* находится в текущем каталоге. Путь *../кат2/кат3/файл* означает, что файл находится в каталоге *кат3*, который находится в каталоге *кат2*, находящемся в родительском (для текущего) каталоге. Можно перемещаться сразу на несколько уровней по дереву каталогов, задавая имя «..» соответствующее количество раз. Например, путь ../../ означает родительский каталог, расположенный на два уровня вверх.

В операционной системе Linux, как и в любой другой ОС, предусмотрен ряд обязательных каталогов:

- каталог **/bin.** В нём находятся наиболее часто употребляемые программы и утилиты системы, как правило, общего пользования;
- каталог **/dev** содержит специальные файлы устройств, являющиеся интерфейсом доступа к периферийным устройствам. Каталог **/dev** может содержать несколько подкаталогов, группирующих специальные файлы устройств одного типа;
- каталог **/etc.** В этом каталоге находятся системные конфигурационные файлы и многие утилиты администрирования;
- каталог **/lib.** В нём находятся библиотечные файлы языка Си и других языков программирования. Стандартные названия библиотечных файлов имеют вид *lib*.a* (или *lib*.so*). Например, стандартная библиотека Си называется *libc.a*, библиотека системы X Window System имеет имя *libX11.a*. Часть библиотечных файлов также находится в каталоге */usr/lib*;
- каталог **/lost+found.** Каталог «потерянных» файлов. Ошибки файловой системы, возникающие при неправильном выключении компьютера или аппаратурных сбоях, могут привести к появлению так называемых

[13] Подробнее о физической организации смотрите ниже.

«безымянных» файлов. Структура и содержимое файла являются правильными, однако для него отсутствует имя в каком-либо из каталогов. Программы проверки и восстановления файловой системы помещают такие файлы в каталог */lost+found* под системными числовыми именами;

- каталог **/mnt.** Стандартный каталог для временного связывания (монтирования) физических файловых систем с корневой для получения дерева логической файловой системы. Обычно содержимое каталога **/mnt** зависит от назначения системы и полностью определяется администратором;
- каталог **/home**. Содержит домашние каталоги пользователей. Например, для домашнего каталога пользователя *student* он будет называться */home/student*;
- каталог **/usr.** В этом каталоге находятся подкаталоги различных сервисных подсистем. */usr/bin* – исполняемые утилиты, */usr/local* – дополнительные программы, используемые на данном компьютере, */usr/include* – заголовочные файлы, */usr/man* – электронные справочники и т.д.;
- каталог ***/var.*** Этот каталог используется для хранения временных файлов различных сервисных программ (системы печати, почтового сервиса и т.п.);
- каталог **/tmp.** Общедоступный каталог. Используется для хранения временной информации.

Как и в именах файлов, при указании путей можно использовать специальные символы и символы расширения. В дополнение к этому для указания абсолютного пути файла, находящегося внутри домашнего каталога пользователя, можно использовать специальный символ **(~) (тильда)**. Например, путь *~/имя_пользователя* будет равен пути */home/имя_пользователя*. А путь *~/кат1/файл* обозначает *файл*, находящийся в каталоге *кат1*, располагающемся в домашнем каталоге текущего пользователя.

3.3. Типы файлов

В зависимости от содержимого в операционной системе Linux различают несколько типов файлов, например:

- *обычный файл* (англ. regular file). Содержит информационные данные. Для операционной системы такие файлы представляют собой просто последовательность байтов. Вся интерпретация содержимого файла производится прикладной программой, обрабатывающей файл, для которой содержимое файла представляется в определённом формате. К этим файлам относятся текстовые файлы, бинарные данные, исполняемые программы и т. п.
- *специальный файл устройства* (англ. special device file). Является интерфейсом для взаимодействия с устройствами вычислительной

машины. Различают *символьные* (англ. character) и *блочные* (англ. block) файлы устройств. Символьные файлы устройств используются для небуферизированного обмена данными с устройством, в противоположность этому блочные файлы позволяют производить обмен данными в виде пакетов фиксированной длины – *блоков*.

- *файлы взаимодействия между процессами* – именованный канал FIFO (англ. named pipe) и сокет (англ. socket)[14].
- *ссылка* (англ. link). Содержит указатель на другой файл или каталог. Операционная система Linux позволяет создавать указатели (ссылки) на файлы или каталоги, которые позволяют одним и тем же файлам иметь несколько имён. Указатели бывают двух типов: жёсткие и символьные (символические). ***Жёсткие ссылки***, по сути, являются именем файла или каталога. Пока существует хотя бы одна жёсткая ссылка, существует и сам файл или каталог. При создании файла для него обязательно создаётся одна жёсткая ссылка. ***Символьная ссылка*** является файлом, который содержит лишь путь, указывающий на другой файл или каталог. Если удалить символьную ссылку, то файл, на который она указывает, останется нетронутым. И обратно, если удалить файл, на который указывает символьная ссылка, то она останется, но будет «неразрешённой».

3.4. Атрибуты файлов. Контроль доступа к файлам и каталогам

Кроме имени и содержимого каждый файл и каталог имеет ряд атрибутов, предназначенных для описания его свойств. К атрибутам относятся:

- дата и время создания файла;
- дата и время последней модификации файла;
- пользователь и группа пользователей (используется для организации доступа к файлу или каталогу);
- права доступа к файлу или каталогу;
- тип содержимого файла;
- и т.д.

Атрибуты «Дата и время создания» и «Дата и время модификации» задаются и изменяются операционной системой при создании файла или каталога или их модификации соответственно.

Контроль доступа к файлам и каталогам в операционной системе Linux осуществляется путём указания ***прав на чтение, запись и исполнение (изменение)***. Указанные права определяются для трёх типов пользователей – владельца файла, группы пользователей, всех остальных.

Права владельца определяют, что может делать с файлом тот пользователь, которому этот файл непосредственно принадлежит (кто указан в атрибуте «пользователь»). ***Права группы*** определяют, что могут делать с файлом члены группы, указанной в атрибуте «группа». И, наконец, ***права прочих пользовате-***

[14] Подробнее о способах межпроцессного взаимодействия будет сказано далее.

лей определяют возможные действия тех пользователей, которые не принадлежат к предыдущим двум категориям.

Пользователь, имеющий право чтения, может просматривать содержимое файла или каталога. Пользователь, имеющий право записи, может редактировать файл или изменять содержимое каталога. И, наконец, пользователь, имеющий право выполнения, может запускать программу, содержащуюся в этом файле, или делать каталог текущим (переходить в него).

Контрольные вопросы

1. Что такое файл? Какова максимальная длина имени файла в ОС Linux?
2. Что такое расширение? Возможно ли наличие нескольких расширений?
3. В каких случаях необходимо использовать «\» в имени файла?
4. Какие символы расширения Вы знаете? Для чего они используются?
5. Можно ли указывать символы «*» и «?» внутри «[ ]»? Что они значат в том случае?
6. Что такое каталог? Какой каталог является основным для файловой системы ОС Linux?
7. Что такое относительный и абсолютный путь к файлу?
8. Что означают символы «.» и «..» в пути к файлу?
9. Перечислите обязательные каталоги ОС Linux? Для чего они нужны?
10. Какое значение имеет символ «~»?
11. Какие типы файлов различают в ОС Linux?
12. В чём отличие символьных файлов от блочных?
13. В чём отличие жёстких ссылок от символьных? Если удалить символьную ссылку, то удалится ли файл, на который она указывает?
14. Какими атрибутами обладают файлы?
15. Как осуществляется контроль доступа к файлам и каталогам в ОС Linux?
16. Какими правами могут обладать владельцы файла, члены группы и прочие пользователи?

ГЛАВА 4. КОМАНДНАЯ ОБОЛОЧКА BASH

«В UNIX оболочка – это своего рода мать всех программ».
Линус Торвальдс, из книги «Just for fun».

При использовании текстового интерфейса пользователь взаимодействует с компьютером (а точнее с операционной системой) при помощи команд, которые вводятся им с клавиатуры. Введённые команды интерпретируются специальной программой, называемой ***командной оболочкой***, которая проверяет их правильность и организовывает запуск соответствующих программ. После завершения выполнения одной команды оболочка ждёт следующую и так до тех пор, пока пользователь не выйдет из системы (т.е. не выполнит команду *logout*).

В современных UNIX-системах, к которым относится операционная система Linux, применяются различные оболочки, отличающиеся набором функций. Наиболее популярными являются:

- Bourne shell (sh), названная в честь своего создателя Стивена Борна (Steven Bourne) из AT&T Bell Labs;
- Bourne Again Shell (bash), расширенная версия предыдущей оболочки;
- C shell (csh), разработанная Биллом Джоем (Bill Joy) и реализующая специальный набор команд в стиле, соответствующем языку программирования Си;
- Korn shell (ksh), созданная Дэвидом Корном (David Korn) на базе Bourne shell, но также реализующая некоторые возможности оболочки C shell.

Пользователи могут использовать любую из имеющихся в системе командных оболочек и даже переключаться между ними[15]. Однако в один момент времени пользователь может взаимодействовать только с одной оболочкой. После завершения регистрации в системе[16] запускается та командная оболочка, которая назначена администратором системы.

Оболочка – это одна из многих программ. Программные файлы для всех командных оболочек обычно находятся в каталоге */bin*. Так, например, путь к программному файлу оболочки Bourne Again Shell будет выглядеть так */bin/bash*. Пользователь временно может изменить тип используемой оболочки, запустив ту, которая ему нужна.

4.1. Понятие команды. Ввод и редактирование команд

Командой называется символьная строка, вводимая пользователем для управления операционной системой и завершаемая символом перевода каретки (клавиша Enter). Команды вводятся в командной строке, обычно содержащей

[15] Например, если пользователю потребуются какие-либо функции отсутствующие у текущей оболочки и имеющиеся у другой оболочки.

[16] В случае, если для входа используется текстовый интерфейс.

приглашение, за которым установлен курсор. Форма приглашения может быть различной[17], но по умолчанию она имеет следующий вид:

```
[student@wp1 student]$_
```

Приглашение *[student@wp1 student]$*, содержит информацию о текущем пользователе (*student*), имени рабочей станции (*wp1*) и текущем каталоге (*student*). Далее идёт символ *$*, после которого установлен курсор.

Каждая команда состоит минимум из одного поля – *имени команды*, представляющего собой (полное) имя файла (или внутренней команды), которую требуется выполнить. Полное имя файла, т.е. содержащее путь к нему, указывается редко. Зачастую командная оболочка ищет указанный без пути файл в известных ей каталогах. Подробнее об этом будет сказано ниже.

Кроме имени команда может ещё содержать разделяемые пробелами аргументы. Если аргумент начинается с символа минус (или тире), то он называется ***опцией***. Если – с двух минусов, то – ***длинной опцией***. Количество и значение аргумент зависят от конкретной команды.

Например, команда для вывода содержимого домашнего каталога[18] может иметь вид:

```
ls /home/STUDENTS/student
```

Здесь *ls* – имя команды, а */home/STUDENTS/student* – аргумент, конкретизирующий работу команды, т.е. указывающий имя каталога, содержимое которого надо вывести.

Если в процессе ввода команды была допущена ошибка, то её можно исправить, используя определённые клавиши управления курсором или специальные комбинации клавиш (таблица 1). Также можно воспользоваться историей команд, в которую помещается некоторое число ранее введённых пользователем команд.

Таблица 1. Клавиши редактирования командной строки в оболочке BASH

Клавиша	**Назначение**
СТРЕЛКА ВПРАВО (Ctrl+F)	Перемещение вправо по командной строке на один символ (в пределах уже введённой строки)
СТРЕЛКА ВЛЕВО (Ctrl+B)	Перемещение на один символ влево
ESC+F	Перемещение на одно слово вправо
ESC+B	Перемещение на одно слово влево
HOME (Ctrl+A)	Перемещение в начало строки
END (Ctrl+E)	Перемещение в конец строки
DEL	Удаление символа справа от курсора (текущего символа)

[17] Подробнее о формате командной строки будет рассказано ниже.
[18] Подробнее о команде *ls* будет рассказано ниже.

Таблица 1. Клавиши редактирования командной строки в оболочке BASH

Клавиша	**Назначение**
BACKSPASE	Удаление символа в позиции, предшествующей курсору
Ctrl+U	Удаление левой части строки, включая символ, который находится слева от курсора
ENTER (CTRL+M)	Запуск на выполнение команды, определяемой набранной цепочкой символов
Ctrl+L	Очистить экран и поместить текущую команду в верхней строке экрана
Ctrl+T	Поменять местами два символа: символ, на который показывает курсор, и символ слева от курсора, затем, курсор переместить на один символ вправо
ESC+T	Поменять местами два слова: слово, на которое указывает курсор, и слово слева от первого
Ctrl+K	Вырезать часть строки от текущей позиции курсора до конца строки (вырезанная часть строки сохраняется в буфере, её можно вставить в другое место строки)
ESC+D	Вырезать часть строки от текущей позиции курсора до конца текущего слова (если курсор указывает на пробел между словами, то вырезается всё слово справа от курсора)
Ctrl+W	Вырезать часть строки от текущей позиции курсора до предыдущего пробела
Ctrl+Y	Вставить последний вырезанный текст в позицию курсора
ESC+C	Символ, на который указывает курсор, заменить на тот же символ, но заглавный, а курсор переместить на первый пробел справа от текущего слова
ESC+U	Сделать символы данного слова заглавными, начиная с символа, на который указывает курсор, а курсор установить на пробел справа от слова
ESC+L	Превратить символы, начиная с символа, на который указывает курсор, до конца данного слова в прописные (маленькие) буквы, а курсор установить на пробел справа от слова
TAB	Продление текущего слова с использованием имён файлов, расположенных в каталоге или в путях поиска. Если таких файлов несколько, то повторное нажатие TAB выдаст их список
СТРЕЛКА ВВЕРХ (Ctrl+P)	Переход к предыдущей команде в списке (движение назад по списку)
СТРЕЛКА ВНИЗ (Ctrl+N)	Переход к следующей команде в списке (движение вперёд по списку)

Таблица 1. Клавиши редактирования командной строки в оболочке BASH

Клавиша	Назначение
PGUP	Переход (вызов в командную строку) к самой первой команды, сохранённой в истории команд
Ctrl+R	Поиск в истории команд, начинающихся с указанных после символов

Если команда не помещается в командную строку, то её можно продолжить на следующей строке, для чего текущую строку надо завершить символом «\» (обратный слеш) и нажать Enter. Командная оболочка после нажатия Enter определит, что последним символом в команде был символ «обратный слеш», и будет ожидать продолжения команды на новой строке. Точно так же можно завершить вторую, третью и последующие строки.

4.2. Электронный справочник

Для удобства пользователей в операционной системе Linux имеется электронный справочник **man**, содержащий информацию обо всех командах[19]. Чтобы получить справку по какой-либо команде, необходимо вызвать электронный справочник, указав ему в качестве параметра имя команды. Например, если необходимо получить справку по команде *echo*, то вызвать справочник можно так:

```
[student@wp1 student]$man echo
```

После этого на экране появится текст справки, который можно пролистывать с использованием клавиш «стрелка вниз», «стрелка вверх», PgUp, PgDown. Чтобы выйти из справочника нужно нажать клавишу «q».

Электронный справочник *man* имеет несколько разделов, каждый из которых содержит информацию о командах или функциях, имеющих соответствующее назначение.

Часто в разных разделах может располагаться информация о командах или функциях, имеющих одинаковые имена. Например, *read* – это команда получения информации от пользователя (раздел 1) и функция чтения информации из файла (раздел 2). Чтобы указать, в каком разделе необходимо искать требуемую команду, следует в команде запуска электронного справочника явно указать номер раздела. Например:

```
[student@wp1 student]$man 3 echo
```

[19] Большинство справочной информации приводится на английском языке.

4.3. Команды работы с файлами и каталогами

4.3.1. Определение текущего каталога

Для определения имени текущего каталога можно использовать команду **pwd**. В любой момент только один каталог является текущим, и команды оболочки по умолчанию применяются к файлам или подкаталогам этого каталога. После регистрации в системе текущим будет домашний каталог пользователя. Перед выполнением каких-либо команд необходимо убедиться, что вы находитесь в нужном каталоге. Например:

```
[student@wp1 student]$pwd<Enter>
/home/student
[student@wp1 student]$_
```

4.3.2. Изменение текущего каталога

С помощью команды **cd** Вы можете сделать текущим другой каталог, указанный в качестве параметра командной строки. Например:

```
[student@wp1 student]$pwd<Enter>
/home/student
[student@wp1 student]$cd /home/anotherstudent<Enter>
[student@wp1 anotherstudent]$pwd<Enter>
/home/anotherstudent
[student@wp1 anotherstudent]$_
```

Если выполнить команду *cd* без параметров, то текущим станет домашний каталог пользователя.

В качестве параметра команды *cd* может быть задан либо абсолютный путь (как в предыдущем примере), либо относительный. Например:

```
[student@wp1 student]$pwd<Enter>
/home/student
[student@wp1 student]$cd ../anotherstudent<Enter>
[student@wp1 anotherstudent]$pwd<Enter>
/home/anotherstudent
[student@wp1 anotherstudent]$_
```

4.3.3. Вывод информации о содержимом каталога

Для отображения содержимого каталога (имён расположенных в нём файлов и каталогов) используется команда **ls**. Если не будет указано ни одного параметра, то будет выведено содержимое текущего каталога. Чтобы вывести содержимое другого каталога, необходимо указать соответствующий путь в качестве параметра. Например:

```
[student@wp1 student]$ls<Enter>
Desktop  Mail  lab1.c  lab2.txt  nsmail
[student@wp1 student]$_
```

Кроме этого, команда *ls* обрабатывает ряд опций, определяющих внешний вид списка файлов и информацию, отображаемую для каждого из них (таблица 2). Более подробный перечень используемых опций можно найти в электронном справочнике *man*.

Опции команды *ls* могут быть указаны как по отдельности, так и в виде одной последовательности символов. Это означает, что выражения *ls -l -F* и *ls -lF* дадут одинаковый результат.

Таблица 2. Опции команды *ls*

Опция	Описание
-a	Выводит информацию обо всех файлах. По умолчанию команда *ls* не отображает файлов, имена которых начинаются с точки («.»).
-d	Выводит информацию не только для каталога, но и для его содержимого.
-l	Выводит информацию об атрибутах файлов и каталогов. Если данная опция не указана, выводятся только имена файлов.
-r	Изменяет порядок сортировки на обратный.
-t	Располагает файлы по дате их изменения. Если содержимое файла изменялось недавно, он будет отображаться в начале списка.
-u	Сортирует файлы по времени последнего обращения к ним.
-R	Показывает содержимое указанного каталога и всех его подкаталогов.

4.3.4. Просмотр содержимого файла

Посмотреть содержимое файла в неинтерактивном режиме (т.е. просто выдать его на экран) можно с использованием команды **cat** с указанием требуемого файла в качестве аргумента. Если размер файла большой, то начало окажется за пределами экрана. Чтобы посмотреть его можно использовать комбинацию клавиш Shift+PgUp для прокрутки вверх и Shift+PgDown для прокрутки вниз.

Чтобы **просмотреть файл в интерактивном режиме** (т.е. иметь возможность при просмотре перемещаться по файлу), можно использовать команду **less**. При этом на экран будет выдана помещающаяся на него часть файла, и *less* будет ждать от пользователя нажатия клавиши (в левом нижнем углу будет выведен символ «:» (двоеточие)). Перемещаться по файлу можно с использованием клавиш управления курсором «стрелка вверх», «стрелка вниз», PgUp, PgDown. Для завершения просмотра файла необходимо нажать клавишу «q». Более подробную информацию о возможностях программы *less* можно найти в электронном справочнике *man*.

4.3.5. Создание файлов

Файлы обычно создаются при помощи прикладных программ. Командная оболочка BASH позволяет создать пустой файл при помощи встроенной команды **touch**. Например, создать файл *file1* в текущем каталоге можно так:

```
[student@wp1 student]$ls<Enter>
Desktop  Mail  lab1.c  lab2.txt  nsmail
[student@wp1 student]$touch file1<Enter>
[student@wp1 student]$ls<Enter>
Desktop  Mail  file1  lab1.c  lab2.txt  nsmail
[student@wp1 student]$_
```

Если файл с таким именем уже существует, то команда *touch* не будет выполнять никаких действий.

4.3.6. Создание каталогов

Для создания каталогов используется команда **mkdir**, которой в качестве аргумента указывается имя требуемого каталога. Если имя указано без пути, то каталог будет создан в текущем каталоге. В противном случае каталог будет создан в указанном месте.

В процессе создания каталога система автоматически создаёт в нём две записи, описывающие каталоги с именами «.» и «..». Такие записи содержатся во всех каталогах, и пользователи не вправе их удалить. Каталог, который содержит только «.» и «..», считается пустым.

```
[student@wp1 student]$ls<Enter>
Desktop  Mail  lab1.c  lab2.txt  nsmail
[student@wp1 student]$mkdir dir<Enter>
[student@wp1 student]$ls<Enter>
Desktop  Dir  Mail  file1  lab1.c  lab2.txt  nsmail
[student@wp1 student]$cd dir<Enter>
[student@wp1 dir]$ls -A<Enter>
. ..
[student@wp1 dir]$_
```

4.3.7. Удаление файлов

Команда **rm** с указанным именем файла позволяет удалить его. При использовании команды *rm* необходимо помнить, что удалённые файлы не подлежат восстановлению. Единственный способ вернуть информацию – обратиться к системному администратору и восстановить файл, воспользовавшись последней резервной копией (версия файла, хранящаяся в ней, необязательно окажется самой последней).

```
[student@wp1 student]$ls<Enter>
Desktop  Mail  lab1.c  lab2.txt  nsmail
[student@wp1 student]$rm lab2.txt<Enter>
```

```
[student@wp1 student]$ls<Enter>
Desktop  Dir  Mail  lab1.c  nsmail
[student@wp1 student]$_
```

4.3.8. Удаление каталогов

Для удаления каталога прежде следует удалить всё его содержимое (включая все подкаталоги со всем их содержимым) и только после этого использовать команду **rmdir**: *rmdir имя_каталога*. Либо сразу для удаления каталога и всего его содержимого можно воспользоваться командой *rm* с опцией *-r*, например *rm -r имя_каталога*.

```
[student@wp1 student]$ls<Enter>
Desktop  Dir  Mail  lab1.c  lab2.txt  nsmail
[student@wp1 student]$rmdir dir<Enter>
[student@wp1 student]$ls<Enter>
Desktop  Mail  lab1.c  lab2.txt  nsmail
[student@wp1 student]$_
```

4.3.9. Копирование файлов и каталогов

Для копирования файлов и каталогов используется команда **cp**, которой в качестве первого аргумента указывается, что надо скопировать, а в качестве второго – куда. Например, команда *cp файл1 файл2* создаст копию файла с именем *файл1* и назовёт её *файл2* (оба файла находятся в текущем каталоге). Если в качестве файла назначения указано имя каталога, команда *cp* скопирует исходный файл в этот каталог. Чтобы скопировать каталог вместе с его содержимым необходимо указать опцию *-r*. Например, *cp -r ../cat1/ ../../cat2*.

```
[student@wp1 student]$ls<Enter>
Desktop  Mail  file1  lab1.c  lab2.txt  nsmail
student@wp1 student]$cp file1 file2<Enter>
[student@wp1 student]$ls<Enter>
Desktop  Mail  file1  file2  lab1.c  lab2.txt nsmail
[student@wp1 student]$_
```

4.3.10. Переименование (перемещение) файлов и каталогов

Для переименования файлов и каталогов используется команда **mv**, которой в качестве первого аргумента указывается, что переименовать, а второго – во что. Если в качестве второго аргумента указан файл, находящийся в другом каталоге (или просто другой каталог), то указанный в первом аргументе файл будет перемещён в указанное место с указанным файлом (с тем же именем).

```
[student@wp1 student]$ls<Enter>
Desktop  Mail  file1  lab1.c  lab2.txt  nsmail
[student@wp1 student]$mv file1 file2<Enter>
[student@wp1 student]$ls<Enter>
```

```
Desktop  Mail  file2  lab1.c  lab2.txt nsmail
[student@wp1 student]$_
```

4.3.11. Создание ссылок на файлы и каталоги

Создать ссылку можно с использованием команды **ln** с указанием файла (каталога), на который нужно сделать ссылку, и название самой ссылки. Чтобы создать символьную ссылку, надо указать опцию *-s*. Следует помнить, что жёсткие ссылки на каталоги может создавать только администратор системы. Например:

```
[student@wp1 student]$ls<Enter>
Desktop  Mail  file1  lab1.c  lab2.txt  nsmail
[student@wp1 student]$ln file1 file2<Enter>
[student@wp1 student]$ls<Enter>
Desktop  Mail  file1  file2  lab1.c  lab2.txt nsmail
[student@wp1 student]$ln –s file1 file3<Enter>
[student@wp1 student]$ls –l file3<Enter>
file3 -> file1
[student@wp1 student]$_
```

4.3.12. Определение прав доступа к файлам и каталогам

Определить права доступа к файлу или каталогу можно, используя команду *ls* с ключом *-l*. Например:

```
[student@wp1 student]$ls –l file3<Enter>
-rw-rw-r--  1 student student 0 Янв  9 22:21 file3
[student@wp1 student]$_
```

В первом столбце в символьной форме указаны права на файл. Первый символ определяет тип файла (в данном случае «-» означает обычный файл). Далее указаны **права на файл в виде триад из символов**, где *r* – право на чтение, *w* – право на запись, *x* – право на исполнение. Структура каждой триады одинакова. Сначала идёт право на чтение, затем – на запись, после чего – на исполнение. Если вместо символа указан знак «-», то это означает отсутствие соответствующего права. Триады расположены в порядке: пользователь, группа, остальные.

4.3.13. Изменение прав доступа к файлам и каталогам

Обычно для установки прав доступа к файлам и каталогам используется команда **chmod**, которой в качестве параметров указывают права доступа и требуемый файл или каталог**.**

Права могут указываться двумя способами: символьным и восьмеричным.

В ***символьных выражениях***, как и следует из их названия, для указания прав доступа используются символы *r*, *w*, *x*. Выражение такого типа имеет следующую структуру: (категория пользователя) (действие) (права).

Категория пользователя может принимать следующие значения: *u* – владелец файла, *g* – группа, *o* – прочие, *a* – все.

Действие: «+» – добавление прав к существующим правам доступа (если они уже существуют, то ничего не изменится), «-» – отмена указанных прав (если их нет, то ничего не изменится), «=» – явное указание прав доступа (т.е. будут установлены именно такие права доступа). Например, чтобы изменить права доступа на файл таким образом, чтобы владелец мог читать файл, группа записывать в файл, остальные – исполнять файл, нужно указать команду *chmod* со следующими аргументами: *chmod u+r,g+w,o+x file*.

При использовании ***восьмеричных выражений*** права указываются явным образом в виде четырёхзначного числа, представляющего права для всех категорий пользователей: владельца (старшая цифра), группы (средняя цифра), прочих (младшая цифра) и специальные права файла. В каждой цифре (кроме первой) право на чтение представляется числом 4 (2^2), на запись – 2 (2^1), на выполнение – 1 (2^0). Результирующее право формируется как сумма этих цифр. Таким образом, значение, определяющее права доступа конкретной категории пользователей, лежит в диапазоне от 0 до 7. Например, если требуется, чтобы пользователь имел права на чтение и выполнение, а остальные не имели никаких прав, необходимо выполнить команду: *chmod 0500 file*. Первая цифра в восьмеричной записи указывает специальные права файла: установить эффективного пользователя (4), установить эффективную группу (2) и задать флаг «прикреплённости» файла (1). Подробнее об этих правах будет рассказано в п. 5.3.

4.3.14. Изменение владельцев и групп

Права доступа к файлу или каталогу указываются для владельца файла, для группы, к которой относится владелец файла, и для всех остальных. Для того чтобы изменить владельца файла или его группу, используются команды **chown** и **chgrp**, которым в качестве параметров указывают требуемого пользователя (команда *chown*) или группу (*chgrp*) и требуемый файл или каталог.

Например, *chown student /home/student* объявляет пользователя *student* владельцем указанного каталога. Администратор системы (пользователь с именем *root*) может изменять владельца любого файла. Обычный пользователь может сменить только группу файла или каталога, которым он владеет, и только на ту группу, членом которой он является.

Если команда *chown* вызывается с опцией *-R*, она выполняется рекурсивно, т.е. для всех файлов и подкаталогов указанного каталога.

4.4. Списки команд

Часто бывает необходимо выполнить несколько команд последовательно одну за другой, т.е. *списком*. Для этого команды объединяются символом **«;»**: *команда1* ; *команда2* ; и т.д. Например:

```
[student@wp1 student]$mkdir docs ; cd docs ; ls<Enter>
[student@wp1 docs]$_
```

Списки также применяются в тех случаях, когда выполнение команды должно зависеть от статуса завершения предыдущей команды[20]. При этом используются логические операторы **&&** и ||: *команда1 && команда2*, *команда1* || *команда2*.

В первом случае (**оператор &&**) команды объединены оператором конъюнкции. При этом *команда2* выполняется только в том случае, если статус завершения *команды1* равен 0, что означает «успешно завершена». Например:

```
[student@wp1 student]$mkdir docs && cd docs<Enter>
[student@wp1 docs]$_
```

В данном примере сначала создаётся каталог *docs*, а затем, если он создан (т.е. команда *mkdir* вернёт 0), этот каталог будет сделан текущим, что и произошло.

Во втором случае (**операция ||**) команды связаны между собой операцией дизъюнкции, и *команда2* выполняется только в случае, если *команда1* возвращает значение больше 0, т.е. результат, отличный от «успешно завершена». Например:

```
[student@wp1 student]$mkdir docs || cd docs<Enter>
[student@wp1 docs]$_
```

В отличие от предыдущего примера каталог *docs* будет сделан текущим, если произошла ошибка при его создании. Результат будет такой же, как и в предыдущем примере, так как каталог *docs* уже создан, и его повторное создание вызовет ошибку.

4.5. Каналы ввода-вывода. Перенаправление каналов. Конвейер

Любая выполняющаяся программа имеет как минимум три канала для взаимодействия с пользователем: поток ввода, поток вывода и поток вывода ошибок. По умолчанию, эти потоки связаны терминалом, т.е. позволяют получать информацию о нажимаемых пользователем клавишах и выводить пользователю информацию на экран.

[20] Как определить статус завершения последней команды, будет сказано ниже. В данном случае командная оболочка сама определяет статус завершения каждой команды из списка.

При запуске программы эти потоки могут быть перенаправлены на другое устройство или на файл. Это бывает необходимо, например, для того чтобы информация об ошибках программы выводилась не на экран, а печаталась на принтере. Или для того чтобы программа получала информацию не с клавиатуры, а считывала её из какого-то файла, содержащего обрабатываемую информацию. Конечно, эти действия могла бы сделать и сама программа. Но в этом случае, при необходимости обрабатывать информацию, получаемую и от пользователя, и находящуюся в файле, потребовалось бы иметь две программы: одну, считывающую информацию с терминала, другую – из файла, что оказывается неэффективно.

Перенаправление потоков ввода-вывода производится с помощью операторов **«>», «>>», «<», «<<»**, после которых указываются новые источники или получатели информации. Например, *команда > файл*.

Оператор «>» перенаправляет поток вывода. Например, *команда > файл*, приведёт к созданию (если такой файл не существует) или перезаписи (в противном случае) указанного файла, в который будет помещаться всё, что *команда* выводит в поток вывода.

Оператор «>>» также перенаправляет поток вывода, но при этом, если указанный приёмник уже существует, то производится дозапись в него. Если источник не существует, то он будет создан. Например, *команда >> файл.*

Для перенаправления потока вывода ошибок перед операторами «>» и «>>» ставится цифра 2 (что соответствует 2-му стандартному потоку). Например, *команда 2> файл*.

Поток ввода может быть перенаправлен, используя **оператор «<»**: *команда < файл.* В этом случае любая попытка получения командой данных со стандартного потока ввода приведёт к чтению необходимой информации из указанного файла.

Если требуется передать программе несколько следующих команд, то используется **оператор «<<»**, после которого указывается команда, которая будет являться завершающей: *команда << разделитель*. В этом случае после нажатия клавиши Enter командная оболочка будет ждать от пользователя следующие команды, и помещать их во временный файл до тех пор, пока пользователь не введёт команду-разделитель. После этого *команда* будет запущена на выполнение, и её поток ввода будет связан с созданным временным файлом, который по завершении команды будет удалён. Например,

```
[student@wp1 student]$sort << END<Enter>
>Апельсин
>Мандарин
>Банан
>END
Апельсин
Банан
Мандарин
[student@wp1 student]$_
```

В примере запускается команда *sort*[21], вход которой связан с файлом, содержащим три строки: *Апельсин*, *Мандарин*, *Банан*. Результат работы команды – отсортированные по алфавиту строки файла.

Следует отметить, что при ожидании команд пользователя, помещаемых во временный файл, формат командной строки изменился на «>».

Операторы перенаправления ввода и вывода могут присутствовать одновременно в одной команде. Например, в предыдущем примере отсортированные строки можно было бы поместить в файл. Сделать это можно так:

```
[student@wp1 student]$sort << END > file<Enter>
>Апельсин
>Мандарин
>Банан
>END
[student@wp1 student]$cat file<Enter>
Апельсин
Банан
Мандарин
[student@wp1 student]$_
```

При обработке информации в UNIX-системах выходные данные одной программы могут поступать на вход другой. Это необходимо, например, для того, чтобы вывод одной программы был автоматически обработан другой программой[22]. Такой способ взаимодействия программ называется **конвейером**. Для организации конвейера команды объединяются **оператором** «|», т.е. формируют *конвейер*: *команда1* | *команда2* | и т.д. Обратите внимание, что при организации конвейера используется оператор «|», а не «||», как в случае со списком команд. Например:

```
[student@wp1 student]$cat file | sort –r>file1<Enter>
[student@wp1 student]$cat file1<Enter>
Мандарин
Банан
Апельсин
[student@wp1 student]$_
```

В этом примере поток вывода команды *cat* (в который она помещает содержимое файла *file*) соединяется с потоком ввода команды *sort,* вызванной с опцией *-r* (обратная сортировка). Вывод команды *sort* помещается в файл *file1*, который следующей командой выводится на экран.

[21] Подробнее о команде *sort* смотрите в электронном справочнике *man*.

[22] Например, для фильтрации лишних данных или оформления их определённым образом и т.п.

4.6. Изменение пароля пользователя

При входе в систему каждый пользователь, чтобы идентифицировать себя, должен ввести своё имя (или выбрать его из списка) и пароль. Пользователи задаются (создаются) администратором системы, который и назначает каждому из них первоначальный пароль. Пользователи имеют возможность самостоятельно изменить свой пароль, чтобы гарантировать, что никто кроме них не сможет войти в систему под их именем.

Изменить пароль можно, используя либо специальные команды, либо графические оболочки, которые, по сути, просто спрашивают у пользователя необходимую информацию и вызывают соответствующие команды.

Если информация о пользователях хранится на этом же компьютере[23], то изменить пароль можно при помощи команды **passwd**. Если компьютер подключён к сети, в которой используется сетевая система паролей (англ. Network Information Service, NIS), то смена пароля производится с помощью команды **yppasswd**[24].

При изменении пароля система сначала запросит текущий пароль пользователя (чтобы кто-то другой не смог его сменить), затем – новый пароль и его подтверждение (т.е. новый пароль ещё раз).

После ввода нового пароля система проверит, подходит ли он под имеющиеся требования безопасности и, если всё в порядке, то изменит его. Если же новый пароль не удовлетворяет этим требованиям, или новый пароль не совпадает с подтверждением, то система сообщит об ошибке и попросит ввести новый пароль заново. Например:

```
[student@wp1 student]$yppasswd<Enter>
Old password:
New password:
Re-type password:
Your password has been changed to ...
[student@wp1 student]$_
```

При нескольких неудачных попытках смены пароля процедура будет завершена, и пароль останется без изменений.

Обычно администратор системы кроме самого пароля определяет требования о частоте его смены и о том, что этот пароль может содержать. Например, «*пароль должен меняться не реже, чем раз в месяц и содержать минимум 6 символов, включающих маленькие и большие латинские буквы и цифры*».

[23] Место хранения информации о пользователях определяется администратором системы.

[24] Первые две буквы (yp) показывают, что используется система NIS, которая ранее называлась Yellow Pages.

4.7. Получение информации о пользователях системы

Каждый пользователь операционной системы Linux описывается как минимум следующими характеристиками:

- системное имя пользователя;
- пароль;
- идентификационный номер пользователя UID;
- идентификационный номер основной группы пользователя GID;
- информация о пользователе (обычно это ФИО пользователя);
- домашний каталог;
- основная командная оболочка.

Информация о пользователях хранится либо в специальном файле с именем **/etc/passwd**, либо на сервере сети (если используется сетевая информационная служба).

В файле */etc/passwd* содержится несколько строк, каждая из которых состоит из 7 полей, содержащих сведения об одном пользователе и разделённых символом «:» (двоеточие). Формат строки следующий: *имя:пароль:UID:GID:информация о пользователе:home_dir:shell*. Например, строка «*student:*:500:500:Иванов Иван Иванович:/home/student:/bin/bash*» содержит информацию о пользователе с именем *student*, паролем[25] *, его UID равен *500*, GID равен *500*, ФИО пользователя – *Иванов Иван Иванович*, домашний каталог – */home/student*, командная оболочка, запускаемая при входе в систему, – */bin/bash*.

В случае использования сетевой службы паролей (NIS) информация о пользователях системы хранится на соответствующем сервере, и получить её можно с использованием команды *ypcat* с аргументом *passwd*. В результате получится список пользователей в таком же формате, как и файл */etc/passwd*.

Информацию о текущем пользователе можно получить, используя команду **id**[26]. Например:

```
[student@wp1 student]$id<Enter>
uid=500(student) gid=500(students)
группы=500(students),550(ftp_students)
[student@wp1 student]$_
```

Чтобы получить информацию о том, какие пользователи зарегистрированы в системе в данный момент времени, можно использовать команды **whoami** и **who**. Первая команда возвращает имя текущего пользователя[27]. Её вызов аналогичен вызову команды *id* с параметром *-un*. Вторая – список зарегистрированных в данный момент в системе пользователей. Например, вызов:

[25] Для повышения безопасности системы пароли могут храниться в другом файле (/etc/shadow) в зашифрованном виде. Доступ к такому файлу обычно разрешён далеко не каждому пользователю.

[26] Информацию о параметрах команды смотрите в электронном справочнике *man*.

[27] Эта команда часто применяется в совместно используемых скриптах, о чём будет сказано далее.

```
[student@wp1 student]$who<Enter>
student1   pts/1  Jan 18 19:24 (192.168.1.38)
student2   pts/0  Jan 18 20:25 (192.168.1.25)
[student@wp1 student]$_
```

показывает, что в системе зарегистрировано два пользователя: *student1* и *student2*. Первый зашёл в систему 18 января в 19:24 с использованием текстового терминала pts/1 (сетевой вход с компьютера с адресом 192.168.1.38). Второй – 18 января в 20:25, с использованием текстового терминала pts/0 (сетевой вход с компьютера с адресом 192.168.1.25).

Чтобы определить, какой пользователь, когда был зарегистрирован в системе, используется команда **last**, которая выдаёт содержимое журнала регистрации. Например, вызов:

```
[student@wp1 student]$last<Enter>
student1 pts/1 1.2.3.4 Wed Jan 18 19:24 still logged in
student1 pts/1 1.2.3.4 Wed Jan 17 19:24 - 20:00 (00:36)
[student@wp1 student]$_
```

показывает, что пользователь *student1* был зарегистрирован в системе 18 января в 19:24 и до сих пор работает, а также был зарегистрирован 17 января в 19:24 и вышел из системы в 20:00, т.е. провёл в ней всего 36 минут.

4.8. Физическое размещение файлов на носителях информации

В Linux всё доступное пользователям файловое пространство объединено в единое дерево каталогов. В большинстве случаев единое дерево таким, каким его видит пользователь системы, составлено из нескольких отдельных файловых систем, которые могут иметь различную внутреннюю структуру, а файлы, принадлежащие этим системам, могут находиться на различных устройствах (например, дискетах, flash-дисках и т.п.).

Чтобы подключить новую файловую систему, а также посмотреть, какие файловые системы уже подключены, используется команда **mount**.

Если её вызвать без аргументов, то на экран будет выдан список уже подключённых файловых систем. В этом списке каждая строка описывает одну подключённую файловую систему. Первым полем строки является устройство, на котором хранится подключённая файловая система (например, /dev/hda4 – 4-й раздел жёсткого диска, подключённого к первому контроллеру IDE в режиме master[28]). Второе поле содержит метку общей файловой системы, которая является точкой входа. Далее идёт тип подключённой файловой системы и её параметры. Например:

[28] Подробнее об устройстве жёсткого диска и о способах его разделения на разделы смотрите в курсе «Организация ЭВМ и систем».

```
[student@wp1 student]$mount<Enter>
/dev/hda4 on / type ext2 (rw) proc on /proc type proc (rw)
/dev/hda2 on /boot type ext2 (rw)
192.168.1.1:/home on /home type nfs (rw,addr=192.168.1.1)
[student@wp1 student]$
```

Этот пример показывает, что корневая файловая система находится на устройстве, описываемом файлом */dev/hda4* (*/dev/hda4 on / type ext2*) и имеет файловую систему типа *ext2*. Каталог */boot* (вторая строка) ссылается на файловую систему 2-го раздела устройства, подключённого к первому контроллеру первым устройством, и также имеет файловую систему *ext2*. Каталог */home* ссылается на сетевую файловую систему (тип *nfs*), находящуюся на компьютере с адресом *192.168.1.1* и имеющую локальное имя */home*.

Чтобы подключить новую файловую систему, необходимо вызвать команду *mount*[29], указав ей в качестве первого аргумента имя устройства, где она хранится, а в качестве второго – точку в общей файловой системе, к которой её надо подключить. Например:

```
[student@wp1 student]$mount /dev/fd0 /mnt/fl<Enter>
[student@wp1 student]$
```

Чтобы отключить файловую систему, необходимо использовать команду **umount**, которой в качестве аргумента следует указать соответствующую точку в общей файловой системе. Например:

```
[student@wp1 student]$umount /mnt/fl<Enter>
[student@wp1 student]$
```

4.9. Среда окружения

При запуске любой программы (в том числе и командной оболочки) в системе создаётся *среда окружения*, которая кроме прочего включает в себя набор переменных, описывающих текущий сеанс работы пользователя с операционной системой (таблица 3). Все переменные среды окружения доступны всем процессам пользователя, начиная с текущего.

Список всех установленных переменных можно получить, используя команду **env**. Например,

```
[student@wp1 student]$env | head -5<Enter>
HOSTNAME=wp1.csc.neic.nsk.su
TERM=linux
SHELL=/bin/bash
HISTSIZE=1000
```

[29] Подробнее о команде *mount* смотрите в электронном справочнике *man*.

```
USER=student
[student@wp1 student]$_
```

В примере вывод команды *env* направлен на вход команды *head,* которая выводит заданное опцией число строк (в данном случае 5), начиная с начала файла. В результате выдано пять переменных: HOSTNAME, TERM, SHELL, HISTSIZE, USER с соответствующими значениями.

Установка новых и **изменение значения** существующих переменных среды окружения осуществляется путем *экспортирования* (помещения в среду): **export** *переменная=значение*.

Чтобы **удалить переменную**, используется команда **unset**: *unset имя*.

Таблица 3. Некоторые стандартные переменные среды окружения

Имя	**Значение**
UID	Содержит числовой идентификатор текущего пользователя. Инициализируется при запуске оболочки.
HOME	Домашний каталог текущего пользователя.
PATH	Список каталогов, разделённых двоеточием, в которых командная оболочка выполняет поиск файла, в случае если в команде не задан его путь.
PS1	Формат строки-приглашения (первая строка).
PS2	Формат строки-приглашения (вторая строка).
PWD	Текущий каталог.
TERM	Тип используемого терминала.
HOSTNAME	Сетевое имя компьютера.
SECONDS	Число секунд, прошедших с момента запуска оболочки.

4.10. Переменные. Массивы

Кроме переменных среды окружения командная оболочка позволяет во время своего выполнения хранить данные в виде собственных переменных и даже массивов. Значения этих переменных используются только самой оболочкой и, в отличие от переменных среды окружения, недоступны запускаемым из неё программам.

Значение переменной присваивается следующим образом: *переменная=значение* (т.е. без процедуры экспортирования). Например, X=1, или X=a, или X=”f” и т.п.

Значение массивов могут задаваться двумя способами:

1. имя[индекс]=значение;

2. имя=(значение1 значение2 ... значениеN).

В первом случае в качестве индекса могут быть использованы как числовые значения, так и символьные лексемы. Например: FRUIT[0]=apple или FRUIT[first]=apple.

Обратите внимание, **что до и после знака «=» нет пробелов!!!** Если поставить пробелы, например, так x = 1, то командная оболочка будет считать, что введена команда x, и она имеет два параметра: = и 1.

Если в командной оболочке создать переменную с тем же именем, что и переменная среды окружения, то в командной оболочке будет использоваться значение этой переменной, а в запускаемых программах – старое значение переменной среды окружения. Получить список всех переменных можно с использованием команды **declare**.

Удалить значение переменной или массива можно также с использованием команды *unset*.

4.11. Подстановка переменных, команд и арифметических выражений

Командная оболочка BASH позволяет формировать команды с использованием значений переменных, результатов работы других команд и т.п. Такое формирование называется **подстановкой.** Т.е. в команду «подставляется» что-либо (переменная, вывод другой команды и т.п.). Для подстановки используется либо символ $, либо выражение, заключённое в обратные апострофы (`выражение`).

Если в тексте команды встречается символ **«$»,** то следующий за ним текст до пробела или конца команды интерпретируется как имя переменной, значение которой подставляется в текст команды. Например:

```
[student@wp1 student]$FRUIT=Апельсин<Enter>
[student@wp1 student]$echo "Фрукт "$FRUIT<Enter>
Фрукт Апельсин
[student@wp1 student]$_
```

В примере создаётся одна переменная командной оболочки (*FRUIT*), которой присваивается значение «Апельсин». Затем выполняется команда *echo*, которая должна выдать на экран текст, указанный ей в качестве параметра. В данном случае параметром является строка *"Фрукт "$FRUIT*, в которой присутствует символ «$». Поэтому прежде чем выполнить команду *echo*, командная оболочка «подставит» в её аргумент значение переменной *FRUIT*, сформировав тем самым текст *"Фрукт Апельсин"*.

Подстановку переменных можно использовать и для формирования новых команд. Например:

```
[student@wp1 student]$DIR=Docs<Enter>
[student@wp1 student]$COMM="ls -A"<Enter>
[student@wp1 student]$COMM=$COMM" "$DIR<Enter>
[student@wp1 student]$$COM<Enter>
.  ..
[student@wp1 student]$_
```

Обратите внимание, что при присваивании переменной значения её имя указывается без знака доллара. Т.е. если Вы напишите *$FRUIT=apple*, командная оболочка выдаст ошибку*: -bash Апельсин=apple:command not found*. Так как прежде чем выполнить команду, командная оболочка подставит в неё значение переменной *FRUIT*, а затем только попытается её выполнить. И если Вы напишите *echo FRUIT*, то на экран будет выведено слово *FRUIT*, а не значение переменной с таким именем.

Для **подстановки элементов массива** используется запись вида *${имя[индекс]}*. Например:

```
[student@wp1 student]$m[0]=Docs<Enter>
[student@wp1 student]$m[1]= "ls -A"
[student@wp1 student]$COMM=${m[1]}" "${m[0]}<Enter>
[student@wp1 student]$$COM<Enter>
.  ..
[student@wp1 student]$_
```

Если в качестве индекса массива указать символ «*» или «@», то результатом будут все элементы массива. Такую операцию можно выполнять только в том случае, когда все элементы массива проинициализированы.

Запись вида **${имя}** можно использовать и для подстановки переменной, например, в том случае, если имя переменной содержит символ подчёркивания. Например, *${FRUIT_APPLE}*.

Подстановку можно использовать также и в случае, если требуется в команде использовать то, что некоторая программа помещает в поток вывода. В этом случае программа заключается в **обратные апострофы.** Прежде чем выполнить команду, командная оболочка выполнит программу, заключённую в обратные апострофы, затем всё, что она поместит в поток вывода, будет подставлено в командную строку, и только затем команда будет выполнена.

Например:

```
[student@wp1 student]$DATE=`date`<Enter>
[student@wp1 student]$echo $DATE
Ср. нояб. 30 13:32:23 NOVT 2011
[student@wp1 student]$_
```

Переменной *DATE* присваивается текст, который должна была вывести на экран команда *date*, т.е. текущую системную дату. Затем на экран выводится значение переменной *DATE*.

Командная оболочка позволяет выполнять арифметические операции. Для этого выражение, которое необходимо **интерпретировать как арифметическое**, заключается в двойные круглые скобки, и перед ним ставится знак доллара. Например, в результате выполнения команды:

```
[student@wp1 student]$foo=$(((5+3*2)-4)/2))<Enter>
[student@wp1 student]$echo $foo
```

```
3
[student@wp1 student]$_
```

переменной *foo* будет присвоено значение, равное 3.

Условная подстановка переменных является ещё одним механизмом подстановки значений (таблица 4).

Таблица 4. Виды условной подстановки переменных

Форма подстановки	Описание
${перем:-значение}	Если переменная не определена или равна NULL, вместо неё будет использовано значение. Содержимое переменной при этом не изменяется.
${перем:+значение}	Если переменная установлена, то вместо неё подставляется значение. Содержимое переменной при этом не изменяется.
${переменная:=значение}	Если переменная не определена или равна NULL, то ей присваивается значение, которое сразу и подставляется.
${перем:?сообщение}	Если переменная не определена или равна NULL, в стандартный поток ошибок выводится сообщение.
${перем:смещение}	Выделяет из переменной часть строки, начиная со смещения и заканчивая концом строки.
${перем:смещение:длина}	Выделяет из переменной часть строки указанной длины, начиная со смещения.
${#переменная}	Подставляет длину содержащейся в переменной строки в символах.
${перем/что/на_что} или ${перем//что/на_что}	Подставляет значение *«перем»* с заменой указанной строки *«что»* на указанную строку *«на_что»*. Значение самой переменной не изменяется. Если перед полем *«что»* указан один слеш, то произойдёт замена только одного вхождения подстроки *«что»*. Если указано два слеша, то произойдёт замена всех вхождений подстроки *«что»*.

Например, используя условную подстановку переменных, можно проверить правильность задания некоторой команды:

```
[student@wp1 student]$${COMMA:?Нет команды}<Enter>
-bash: COMMA: Нет команды
[student@wp1 student]$_
```

Командная оболочка проверит, есть ли переменная с именем *COMMA* и присвоено ли ей какое-либо значение. Если переменная есть, и у неё есть значение, то оно будет интерпретировано как команда (так как в команде кроме подстановки ничего больше нет). Если же переменной нет, или ей не присвоено никакого значения, то вместо интерпретации команды на экран будет выдано сообщение «Нет команды».

Заменить символ «пробел» в строке, хранящейся в переменной *DB*, на символ «_» можно так: *DB=${DB// /_}*.

4.12. Формат приглашения командной оболочки

Как уже было сказано выше, командная оболочка предлагает пользователю вводить команды, используя определённое приглашение. Формат этого приглашения задаётся с помощью *переменных окружения:* PS1, PS2, PS3, PS4. Переменная **PS1** определяет обычную командную строку. Переменная **PS2** – командную строку, выдаваемую при продолжении команды на следующей строке (2-я и последующие строки, если перед нажатием клавиши Enter был введён символ «\»).

В строку приглашения могут входить любые допустимые символы, команды терминала и специальные символьные последовательности, которые используются для получения информации о системе (таблица 5).

Таблица 5. Специальные последовательности формата командной строки.

Послед.	**Выводимое значение**
\t	время в формате часы: минуты: секунды
\d	дата в формате день_недели месяц число
\n	перевод строки
\s	имя оболочки, базовое имя $0 (участок, следующий за конечным /)
\w	текущий рабочий каталог
\W	базовое имя $PWD
\u	имя текущего пользователя
\h	сетевое имя компьютера
\#	номер этой команды
\!	номер истории этой команды
\nnn	символ, имеющий код, равный nnn в восьмеричной системе
\$	если uid=0, то \#, иначе \$
\\	обратный слеш
\[	начало терминальной последовательности
\]	конец терминальной последовательности

Чтобы изменить формат приглашения командной оболочки, достаточно присвоить необходимое значение соответствующей переменной. Например, выделить текущего пользователя зелёным цветом можно, присвоив переменной PS1 следующее значение:

```
PS1=[\[\033[32m\]\u\[\033[0m\]@\h \W]\$
```

4.13. Получение информации от пользователя

При необходимости командная оболочка позволяет сформировать значение некоторой переменной, «спросив» его у пользователя. Для этого используется команда **read**, которой в качестве аргумента передаётся имя требуемой переменной.

```
[student@wp1 student]$read CHOICE
Привет !!!<Enter>
[student@wp1 student]$echo "Вы ввели "${CHOICE}
Привет !!!
[student@wp1 student]$_
```

Чтобы указать, какое приглашение должно быть выведено в строке для ввода, можно использовать параметр *-p*. Например, *read -p "Введите X" X.*

Время ожидания (в секундах) ввода задаётся или при помощи переменной **TMOUT**, или при помощи параметра *-t*. Если переменная *TMOUT* не определена или её значение равно 0, и не указан параметр *-t*, то время ожидания считается бесконечным. **Обратите внимание**, что значение переменной *TMOUT* также влияет на время ожидания командной оболочкой очередной команды!!!

Используя параметр *-s*, можно запретить отображение вводимых символов на экране. Это удобно, например, при вводе паролей.

4.14. Управляющие структуры

Командная оболочка BASH позволяет организовать ветвление программы в зависимости от различных условий.

4.14.1. Условный оператор if-fi

Выражение **if** записывается следующим образом[30]:

```
$if выражение1 ; then \
>выражение2 ; \
>elif выражение3 ; then \
>выражение4 ; \
>else \
>выражение5 ; \
>fi<Enter>
```

В приведённой выше команде *if* сначала выполняется *выражение1*. Если код завершения *выражения1* равен 0 (что интерпретируется как его

[30] В дальнейшем по тексту будем использовать сокращенный формат приглашения командной строки, состоящей только из символа $.

истинность), то выполняется *выражение2*, и команда *if* заканчивается. В противном случае выполняется *выражение3*, и проверяется код его завершения. Если *выражение3* возвращает значение, равное 0, то выполняется *выражение4* и команда *if*. Если *выражение3* возвращает ненулевое значение, то выполняется *выражение5*. Наличие операторов *elif* и *else* необязательно. В блоке *if-fi* может содержаться несколько *elif*.

Часто в блоке *if-fi* в качестве выражений, результаты которых проверяются, используется команда **test**, которая имеет две формы записи: *test параметры* или *[параметры]*. После интерпретации параметров (таблица 6) как логического выражения команда *test* возвращает значение 0 – истина либо 1 – ложь.

Таблица 6. Некоторые параметры команды *test*.

Выражение	**Значение**
-d файл	существует ли *файл* и является ли он каталогом?
-e файл	существует ли указанный *файл*?
-f файл	существует ли *файл* и является ли он обычным файлом?
-L файл	существует ли *файл* и символьная ли он ссылка?
-r файл	существует ли *файл* и разрешено ли его чтение?
-s файл	существует ли *файл* и имеет ли он нулевой размер?
-w файл	существует ли *файл* и разрешена ли в него запись?
-x файл	существует ли *файл* и является ли он исполняемым?
-O файл	существует ли *файл* и принадлежит ли он текущему пользователю?
файл1 -nt файл2	был ли *файл1* последний раз модифицирован позже, чем *файл2*?
-z строка	указанная строка имеет нулевую длину?
-n строка	указанная строка имеет ненулевую длину?
стр1 == стр2	указанные строки совпадают?
! выражение	указанное выражение *false*?(содержит не нуль)
выр1 -a выр2	логическое И двух выражений
выр1 -o выр2	логическое ИЛИ двух выражений
выр1 -eq выр2	*выр1* равно *выр2*?
выр1 -ne выр2	*выр1* неравно *выр2*?
выр1 -lt выр2	*выр1* меньше (в арифметическом смысле) *выр2*?
выр1 -le выр2	*выр1* меньше либо равно *выр2*?
выр1 -gt выр2	*выр1* больше *выр2*?
выр1 -ge выр2	*выр1* больше либо равно *выр2*?

Ниже приведены примеры использования команды *test* в составе выражения *if-fi*:

```
$if [ -d $HOME/bin ] ; then \
>PATH="$PATH:$HOME/bin" ; fi
```

```
$if [ -z "$DTHOME" ] && [-d /home/student/dt ] ;then \
>DTHOME=/home/student/dt ; \
>fi
$if [ -z "$DTHOME" -a -d /home/student/dt ] ; then \
>DTHOME=/home/student/dt ; \
>fi
```

В первом примере проверяется, существует ли каталог *$HOME/bin*, и, если он существует, то он добавляется к переменной *PATH*.

Во втором и третьем случае проверяется, установлено ли значение переменной *DTHOME*, и существует ли каталог */home/student/dt*. Если переменная не определена или имеет пустое значение, и указанный каталог существует, то переменной *DTHOME* задаётся новое значение. Во втором примере проверка указанного условия происходит с использованием списка из двух команд *test*, выполняемых по условию «И». В третьем примере используется один вызов команды *test*, в параметрах которой указаны все условия (условие «И» реализуется с помощью опции *-a*).

Следует обратить внимание, что при проверке значения переменной *DTHOME* она взята в кавычки. Это сделано так потому, что, если переменная всё-таки не определена или определена, но не имеет значения, то вместо неё в командную строку ничего не будет подставлено. Если кавычки не использовать, то после подстановки потеряется второй параметр команды *test*, и нечего будет проверять (т.е. строка *test -z $DTHOME* будет преобразована к виду *test -z*).

4.14.2. Оператор множественного выбора case-esac

Блок **case-esac** аналогичен оператору *if-fi* со множеством *elfi* и предназначен для проверки одной переменной на несколько возможных значений. Блок *case-esac* записывается следующим образом:

```
$case значение in \
>шаблон1) \
>список команд1 ;;
>шаблон2) \
>список команд2 ;;
>esac<Enter>
```

В данном случае значение – это строка символов, сравниваемая с шаблоном до тех пор, пока она не совпадёт с ним. Список команд, следующий за шаблоном, которому удовлетворяет значение, запускается на выполнение. За списком следует команда «;;», которая завершает работу блока *case-esac*.

Если значение не удовлетворяет ни одному из шаблонов, выражение *case*

завершается. Если необходимо выполнить какие-то действия по умолчанию, следует включить в выражение шаблон «*», которому удовлетворяет любое значение.

В выражении *case-esac* должен присутствовать, по крайней мере, один шаблон. Максимальное число шаблонов не ограничено.

Шаблон формируется по правилам, аналогичным именам файлов и каталогов (с учётом символов расширения), а также используется оператор дизъюнкции «|» (операция ИЛИ). Ниже приведён пример использования блока *case-esac*.

```
$case "$TERM" in \
>*term) \
>TERM = xterm \
>;; \
>network | dialup | unknown | vt[0-9]*) \
>TERM=vt100 \
>;; \
>esac
```

4.15. Циклические конструкции

Циклические конструкции применяются в том случае, когда надо многократно повторить одни и те же действия.

4.15.1. Цикл for

Цикл **for** предназначен для выполнения определённых действий над несколькими данными. Формат записи цикла следующий:

```
$for имя_переменной in список_значений ; \
>do \
>команда1 ; команда2 ...\
>done
```

В цикле **for** переменной с указанным именем последовательно присваиваются все значения из *списка_значений*, и для каждого из этих значений выполняется *список_команд*. Значения в *списке_значений* перечисляются через пробел. Например, следующая команда выдаст на экран десять строк: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10:

```
$for i in 1 2 3 4 5 6 7 8 9 10 ; \
>do \
>echo $i ; \
>done
```

4.15.2. Цикл while

Если необходимо выполнять какие-либо действия до тех пор, пока некоторое выражение истинно, то следует использовать цикл **while**, который записывается следующим образом:

```
$while выражение ; \
>do \
>список_команд ; \
>done
```

На каждой итерации этого цикла выполняется *выражение* и до тех пор, пока оно возвращает значение 0, выполняется *список_команд*. Если в качестве выражения указать команду */bin/true*, то цикл будет бесконечным и завершить его можно только, используя оператор **break.** Пропустить какую-либо часть цикла и перейти на следующую его итерацию можно, используя в *списке_команд* команду **continue**.

Например, вывод последовательности цифр от 1 до 9 можно организовать следующим образом:

```
$x=1
$while [ $x -lt 10 ] ; \
>do \
>echo $x ; \
>x=$(( $x+1 )) ; \
>done
```

4.15.3. Цикл until

Если необходимо выполнять какие-либо действия до тех пор, пока ложно некоторое выражение, то следует использовать цикл **until**, записываемый следующим образом:

```
$until выражение ; \
>do \
>список_команд ; \
>done
```

На каждой итерации этого цикла выполняется *выражение* и до тех пор, пока оно возвращает значение, отличное от 0, выполняется *список_команд*. Если в качестве выражения указать команду */bin/false*, то цикл будет бесконечным и завершить его можно только, используя оператор **break.**

Например, вывод последовательности цифр от 1 до 9 можно организовать следующим образом:

```
$x = 1
$until [ $x -ge 10 ] ; \
>do \
>echo $x ; \
>x=$(( $x+1 )) ; \
>done
```

4.15.4. Цикл select

Если имеется набор значений, и требуется по желанию пользователя выполнить определённые операции над некоторыми из них, то используется цикл **select**, который позволяет создать нумерованное меню значений, пункты которого могут быть выбраны пользователем. Данный цикл записывается следующим образом:

```
$select имя_переменной in список_значений ; \
>do \
>список_команд ; \
>done
```

При запуске цикла *select* в стандартный поток ошибок выводится *список_значений*. Перед каждым пунктом отображается номер, который и предлагается выбрать пользователю. Затем выводится приглашение ввода номера. Формат приглашения определяется переменной **PS3**. Выбрав номер пункта, пользователь вводит его с клавиатуры. После этого для указанного элемента *списка_значений* (который помещается в указанную переменную) выполняется *список_команд*. После этого опять выводится меню цикла *select*.

Завершить цикл *select* можно либо после ввода EOF (нажатия комбинации клавиш Ctrl+D), либо при помощи команды *break*. Пример использования цикла *select*:

```
$select file in ./*.c ; \
>do\
>gcc –Wall -ansi $file –o `basename $file .c` ; \
>done
```

В данном примере пользователю предлагается выбрать один из файлов с суффиксом *.c*, расположенных в текущем каталоге. Выбранный файл будет скомпилирован, и результат компиляции помещён в файл с таким же именем, но без суффикса *.c*. Имя файла результата формируется командой *basename*[31], результат которой подставляется в команду компиляции.

[31] Подробнее о команде *basename* смотрите в электронном справочнике *man*.

4.16. Группирование команд. Функции. Скрипты

Командная оболочка BASH позволяет группировать несколько команд, выполняющих определённое действие, в функции или специальные файлы, называемые скриптами.

Функции определяются так:

```
$function имя () { список_команд ; }
```

В дальнейшем, если будет дана команда *имя*, то вместо неё будет выполнен *список_команд*. Например[32]:

```
$ls
$function ls () { /bin/ls –A $@ ; }
$ls
. ..
```

Здесь объявляется функция с именем *ls*, которая содержит всего лишь одну команду */bin/ls* с соответствующими параметрами. До того, как определена эта функция, команда *ls* приводит к запуску файла */bin/ls*, выводящего на экран содержимое каталога.

Скрипт – это обычный текстовый файл[33], содержащий команды оболочке. Такой файл может быть запущен на исполнение следующим образом:

```
$bash имя_файла
```

Другими словами, для выполнения скрипта необходимо запустить командную оболочку, передав ей в качестве параметра имя соответствующего файла.

Другой вариант запуска скрипта – просто указать его имя в командной оболочке (т.е. сделав из него некий вид программы). Для этого надо в параметрах доступа определить файл как исполняемый, и в первых строках этого файла явно указать оболочку, для которой предназначен этот скрипт, следующим образом:

```
#!оболочка
```

В общем случае символ # в скрипте означает комментарий, требующий игнорировать строку. Однако если он является первым символом файла и за ним следует символ *!*, то это означает «магическую комбинацию», за которой указывается путь к файлу, использующемуся в качестве интерпретатора скрипта (например, */bin/bash*, */bin/perl*, */bin/sh* и т.д). Встретив такую

[32] Значение переменной @ смотрите ниже.
[33] Обычно такие файлы имеют суффикс *.sh.*

комбинацию символов, командная оболочка запустит соответствующий файл и передаст ему его имя в качестве аргумента.

Скрипт, в свою очередь, может содержать все конструкции, описанные ранее (в том числе и функции).

Выполнение скрипта происходит построчно. При этом если конструкция включает в себя несколько команд, то они могут располагаться на нескольких строках, и указывать символ \ в конце неоконченной строки нет необходимости. Например:

```
#!/bin/bash
X=1
read –p “Введите X = ” X
if [ $X –lt 0 ] ; then
   echo “Вы ввели отрицательное число”
else
   echo “Вы ввели положительное число”
fi
```

4.17. Анализ опций, передаваемых группе команд (функции и скрипту)

Любому скрипту, точно так же, как и функции, могут быть переданы аргументы (так, как это делается при запуске любой команды). Передаются аргументы в виде специальных переменных (таблица 7).

Таблица 7. Специальные переменные, используемые в скриптах.

Перем.	Значение
$0	Имя выполняемой команды. Для скрипта – это путь, указанный при его вызове. Для функции – имя оболочки.
$n	Переменные, соответствующие аргументам, заданным при вызове сценария. Здесь *n* – десятичное число, соответствующее номеру аргумента. (Первый аргумент – $1, второй – $2 и т.д.).
$#	Число аргументов, указанных при вызове сценария.
$*	Строка аргументов, заключённая в двойные кавычки.
$@	Все аргументы, каждый заключён в двойные кавычки.
$?	Статус завершения последней выполненной команды.
$$	Номер процесса, соответствующего текущей оболочке.
$!	Номер процесса, соответствующий команде, запущенной в фоновом режиме.

Последовательно просмотреть аргументы командной строки можно, используя следующую конструкцию:

```
#!/bin/bash
while [ -n "$1" ] ; do
  echo "Имеется аргумент - "$1
  shift
done
```

Аргументы просматриваются в цикле *while,* условием выполнения которого является результат команды *test* с параметрами, требующими, чтобы длина строки в аргументе *$1* была отлична от нуля. В строку подставляется значение первого аргумента. Если аргумент не указан, то переменная *$1* имеет пустое значение, соответственно строка будет иметь нулевую длину, и цикл сразу же прекратится. Если аргумент указан, то он выводится в теле цикла, и затем выполняется команда **shift**, которая изменяет переменные *$n*, сдвигая их на одну влево (т.е. *$1=$2*, *$2=$3* и т.д.). Значение первого аргумента при этом теряется. Когда сдвигается последний аргумент, то переменной *$1* присваивается пустое значение (так как следующей за ней переменной не существует). В качестве параметра команды *shift* можно указать, на сколько позиций требуется сдвинуть строку аргументов. Например, *shift 2* приведёт к следующему изменению: *$1=$3*, *$2=$4* и т.п.

Если требуется определить, есть ли в аргументах определённые короткие опции, можно использовать специальную команду **getopts,** которая сама просматривает специальные переменные и проверяет, является ли аргумент короткой опцией.

Напомним, что **опцией** называется аргумент командной строки, начинающийся с символов «-» или «--». Причём, если используется один знак минус, то опция считается односимвольной или **короткой**, если же – два знака минус, то – многосимвольной или **длинной**[34].

Команда *getopts* вызывается следующим образом:

```
getopts строка_опций_поиска переменная
```

В *строке_опций_поиска* указываются односимвольные опции, которые должны быть «распознаны» *getopts*. Например, строка *acdf* говорит о том, что требуется найти какую-либо из опций: *-a*, *-c*, *-d* или *-f*.

Алгоритм работы команды *getopts* следующий.

Последовательно просматриваются специальные переменные *$1*, *$2*, *$3…$#*, начиная с номера, указанного в переменной **OPTIND**. При первоначальном запуске командной оболочки в *OPTIND* находится 1[35]. Другими словами, если впервые вызвать команду *getopts*, то она начнёт просмотр с переменной *$1*.

[34] Длинные опции в скрипте можно «распознавать» только через специальные переменные.

[35] Если потребуется заново провести проверку опций, то такое присвоение должен будет сделать пользователь.

Если очередной аргумент не пустой, и он не содержит опцию (не начинается с символа «-»), то *getopts* завершается с ненулевым кодом возврата.

Иначе считается, что найдена опция, и производится сравнение следующего за тире символа с содержимым *строки_опций_поиска*.

Если найденная опция совпадает с одним из символов в *строке_опций_поиска*, то соответствующий символ записывается в *переменную*.

Если совпадение не обнаружено (т.е. в командной строке указана опция, которая отсутствует в *строке_опций_поиска*), на экран выводится сообщение об ошибке, *переменной* присваивается значение «?».

Если просматриваемый аргумент больше не содержит символов, то номер следующего аргумента записывается в переменную *OPTIND* (чтобы при следующем запуске команды продолжить просмотр аргументов). В противном случае *OPTIND* остаётся без изменения, а запоминается позиция внутри текущего просматриваемого аргумента (чтобы в следующий раз продолжить просматривать этот аргумент, считая его опцией и начиная со следующего символа).

В случае если найдена «распознаваемая» или «нераспознаваемая» опция, код возврата *getopts* равен 0. В случае если просмотрены все аргументы и больше опций нет, или очередной аргумент не является опцией, то код возврата *getopts* отличен от 0.

Чтобы **запретить выдачу сообщения об ошибке** при нахождении опции, которая «не распознаётся», необходимо либо переменной *OPTERR* присвоить 0, либо в *строке_опций* первым символом указать «:» (двоеточие). В последнем случае *getopts* начинает работать в «тихом» (англ. silent) режиме, и в случае ошибки найденную «нераспознаваемую» опцию она помещает в переменную *OPTARG,* а в *переменную* помещает символ «?».

Команда *getopts* позволяет «распознать» **опции**, **для которых требуется дополнительный параметр**. Для этого надо в строке опций после соответствующего символа включить знак «:»[36]. Например, в строке *a:cdf:* указывается, что опции *-a* и *-f* должны дополняться обязательными параметрами,например, *-a файл* или *-b время*.

Если при «распознавании» найдена подобная опция, и за ней следует непустой аргумент, то он помещается в переменную *OPTARG*, а сама опция помещается в *переменную*, и *getopts* завершается.

Если опция найдена, но за ней нет аргумента, то на экран выдаётся сообщение об ошибке, в *переменную* помещается символ «?», и *getopts* завершается. В «тихом» режиме при нахождении такой опции и отсутствии у неё параметра в *переменную* помещается символ «:», а в переменную *OPTARG* помещается сама опция.

Например:

[36] Обратите внимание, что в данном случае символ «:» не является первым в строке, а следует за символом. При этом в начале строки символ «:» может как присутствовать (чтобы перевести *getopts* в «тихий» режим), так и отсутствовать.

```
#!/bin/bash

while getopts ab:cde:f: OPTION ; do
  case $OPTION in
     \?)
        echo "Найдена \"нераспознаваемая\" опция"
        ;;
     :)
       echo "Тихий режим и найдена опция "${OPTARG}
       ;;
     ?)
        echo "Найдена опция "${OPTION}
        if [ -n "${OPTARG}" ] ; then
          echo "Она имеет параметр "${OPTARG}
        fi
  esac
  echo "OPTIND = "${OPTIND}
done
echo "Finally OPTIND = "${OPTIND}
```

Скрипт содержит в себе цикл, в котором последовательно запускается команда *getopts* до тех пор, пока код её возврата равен 0. В качестве *строки_опций_поиска* команда *getopts* получает строку вида «*ab:cde:f:*». Результат будет помещаться в переменную *OPTION*.

Запуск скрипта с параметрами *-ab -c -d -e -f -g* приведёт к следующему:

```
[student@wp student]$./getopts.sh -ab -c -d -e –f \
>-g<Enter>
Найдена опция a
OPTIND = 1
Найдена опция b
Она имеет параметр –c
OPTIND = 3
Найдена опция d
OPTIND = 4
Найдена опция e
Она имеет параметр –f
OPTIND = 6
./getopts.sh: illegal option – g
Найдена "нераспознаваемая" опция
OPTIND = 7
Finally OPTIND = 7
```

При первом запуске (на первой итерации цикла) команда *getopts* начнёт просмотр переменной *$1*, в которой располагается аргумент *-ab*. Так как он

начинается с символа «-», то считается, что найдена опция. После того, как определено, что аргумент содержит опцию, проверяется, входит ли она в строку опций. В данном случае она входит в неё и не требует дополнительного параметра (так как за ней нет символа «:»). Поэтому в переменную *OPTION* помещается символ *a*, и *getopts* завершается. Значение переменной *OPTIND* не изменяется, так как аргумент содержит ещё другие символы (*b*).

На следующей итерации цикла *getopts* продолжает просматривать первый аргумент (так как *OPTIND* = 1). В нём также оказывается опция, но она требует параметр (в *строке_опций_поиска* после символа *b* стоит двоеточие). Поэтому *getopts* проверяет, указан ли следом за этой опцией параметр. В данном случае он указан и равен *-c*, поэтому в *OPTION* помещается *b*, а *OPTARG* приравнивается к *-c*. Переменная *OPTIND* при этом указывает на аргумент с номером 3, так как текущий аргумент больше не содержит символов, а 2-й аргумент относился к найденной опции.

Далее аналогично просматриваются остальные аргумента до аргумента, содержащего опцию *-g*. Она считается нераспознанной. Поэтому на экран выдаётся сообщение об ошибке (так как никаких указаний по его устранению не было), и в переменную *OPTION* помещается символ «?».

Цикл продолжается до тех пор, пока не будут просмотрены все аргументы.

4.18. Вход в систему и первоначальные настройки

Часто при входе в систему, выходе из неё или при простом запуске командной оболочки[37] пользователю требуется выполнить определённый набор действий[38]. Для этого при запуске и при завершении командная оболочка выполняет определённый набор скриптов, располагаемых в домашнем каталоге пользователя (таблица 8).

Таблица 8. Специальные файлы командной оболочки BASH.

Файл	**Назначение**
.bash_history	Содержит историю всех введённых пользователем команд.
.bash_profile	Исполняется оболочкой при входе пользователя в систему.
.bash_logout	Исполняется оболочкой при выходе пользователя из системы.
.bashrc	Выполняется при простом запуске пользователем оболочки.

Обратите внимание, что все эти файлы имеют имена, начинающиеся с *.bash*.

Кроме этого, в правах на файлы *~/.bash** не указана возможность их исполнения. Это потому, что командой, исполняющей эти файлы, всегда является сама командная оболочка.

[37] Такой запуск также называется «интерактивным».

[38] Например, пользователю необходимо выводить информацию, имеется ли в его электронном почтовом ящике письма, или записать в журнал время и дату, когда он вошёл в систему и т.п.

Таким образом, если необходимо, чтобы какая-то команда выполнялась каждый раз при входе в систему, то эту команду надо поместить в файл *.bash_profile*. Например, для вывода текущей даты и времени при входе в систему в файл *.bash_profile* надо добавить команду **date**.

Контрольные вопросы

1. Что такое командная оболочка? Какие типы командных оболочек используются в ОС Linux?
2. Что такое команда? Какая информация обычно содержится в приглашении для ввода команды?
3. Что такое опция и длинная опция?
4. Какие клавиши можно использовать для редактирования командной строки в оболочке BASH?
5. Как можно ввести многострочные команды?
6. Как получить справку о команде в ОС Linux?
7. Перечислите команды для работы с каталогами.
8. Как одной командой показать содержимое указанного каталога и всех его подкаталогов?
9. Как просмотреть содержимое файла в интерактивном режиме?
10. Для чего используется команда *touch*?
11. Как одной командой удалить непустой каталог?
12. Перечислите команды для работы с файлами.
13. Что выполняет команда *ls –l*?
14. Как изменить права доступа к файлам или каталогам?
15. Как организовать последовательное выполнение нескольких команд?
16. В чём отличие использования «;» и «&&» в списках команд?
17. Какие три потока для взаимодействия с пользователем имеет исполняемая программа?
18. Как перенаправить потоки ввода-вывода? В чём отличие «>» от «>>»?
19. Какой способ взаимодействия программ называется конвейером?
20. Назовите команду для смены пароля пользователя.
21. В каком файле хранится информация о пользователе системы? Опишите формат этого файла.
22. Для чего предназначена команда *mount*?
23. Что такое среда окружения? Как получить список всех переменных среды окружения? Как установить новые значения для переменных среды окружения? Как удалить переменную?
24. Как создать переменную командной оболочки? Для чего используется «$» перед именем переменной? Как организовать массив? Что такое условная подстановка?
25. Какие переменные среды окружения задают внешний вид приглашения командной оболочки? Как изменить формат приглашения?
26. Расскажите об управляющих структурах, используемых в командной оболочке BASH.

27. Какие циклические конструкции можно использовать в командной оболочке BASH?
28. Как определяются функции использовать в командной оболочке BASH?
29. Что такое скрипт? Как запустить скрипт на выполнение?
30. Как передать аргументы скрипту? Чем опция отличается от аргумента?
31. Какая команда служит для распознавания опций?
32. Опишите специальные файлы командной оболочки BASH.

ГЛАВА 5. «ПРОЦЕСС» И «НИТЬ»

«В UNIX всё, что что-то делает – процесс».
Линус Торвальдс.
Из книги «Just for fun».

Функционирование любой ЭВМ заключается в выполнении строго определённой последовательности действий, называемой **программой** или **задачей**. «*Запустить программу на выполнение*» означает загрузить саму программу (написанную на понятном для ЭВМ языке) и необходимые для неё данные в оперативную память и настроить процессор (значения его внутренних регистров) так, чтобы он стал выполнять инструкции, начиная с соответствующей ячейки оперативной памяти.

В мультизадачных системах, к которым относится и ОС Linux, в оперативную память может быть загружено сразу несколько программ. При этом, чередуя выполнение на одном процессоре инструкций то одной программы, то другой, ЭВМ создаёт «иллюзию» их одновременного выполнения. Правила, по которым чередуются инструкции разных программ, определяются операционной системой (подсистема планирования процессов или нитей). При этом программы ничего не знают о существовании друг друга[39].

В многопроцессорных (многоядерных) системах каждый процесс может выполняться на собственном процессоре (ядре). При этом каждый процессор, в свою очередь, может использоваться для выполнения нескольких (псевдо)параллельных процессов.

Для того чтобы операционная система могла определить, инструкцию какой программы следует выполнить следующей, должны выполняться два условия: во-первых, после выполнения очередной инструкции одной из программ процессор должен переключиться на операционную систему[40], и, во-вторых, операционная система должна обладать информацией обо всех «выполняющихся» программах (место в оперативной памяти, где располагаются инструкции программы; номер последней выполненной инструкции; значения регистров процессора после выполнения последней инструкции и т.п.). Информация обо всех выполняющихся программах хранится в оперативной памяти вместе с операционной системой.

Совокупность, состоящая из программы, описывающей её информации и дополнительных служебных данных, называется **процессом**. Информация о программе хранится в операционной системе в виде специальной структуры, называемой **дескриптором процесса**. Дескриптор процесса и дополнительная информация о программе вместе называются **контекстом процесса**.

[39] Следует помнить, что защита программ реализуется как на аппаратурном уровне (защищённый режим процессора), так и на уровне операционной системы.

[40] Напомним, что для ЭВМ операционная система является обычной программой, также загруженной в оперативную память.

5.1. Типы процессов

Все запущенные процессы условно (в зависимости от выполняемой ими функции) можно разделить на три типа: системные, процессы-демоны и прикладные процессы.

Системные процессы являются частью ядра и всегда расположены в оперативной памяти. Они часто не имеют соответствующих им программ в виде исполняемых файлов и всегда запускаются особым образом при загрузке ядра системы. Системными процессами являются, например: keventd (диспетчер системных событий), kswapd (диспетчер страничного замещения – своппинга), bdflush (диспетчер буферного кэша) и т.д.

К системным процессам следует отнести процесс **init**, являющийся первым создаваемым процессом в системе и «прародителем» всех остальных процессов.

Процессы-демоны – это неинтерактивные (т.е. не взаимодействующие с пользователями) процессы, которые выполняются в *фоновом режиме*. Обычно они обеспечивают работу различных подсистем Linux, например, системы терминального доступа, системы печати, системы сетевого доступа и сетевых услуг и т.п.

К **прикладным** относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порождённые в рамках пользовательского сеанса работы.

Пользовательские процессы могут выполняться как в *интерактивном*, так и в *фоновом режиме*, но в любом случае время их жизни (и выполнения) ограничено сеансом работы пользователя. При выходе из системы все пользовательские процессы будут уничтожены. **Интерактивные** процессы связаны с определённым терминалом[41] и через него взаимодействуют с пользователем. **Фоновые** процессы выполняются независимо от пользователя и (псевдо)параллельно. Из фонового процесса можно создать процесс-демон, для чего ему следует отключиться от терминала[42].

Запустить процесс на выполнение в фоновом режиме можно одним из следующих способов:

- ❖ указать в конце командной строки символ амперсанд «&»;
- ❖ приостановить выполнение интерактивного процесса (нажать комбинацию клавиш Ctrl+Z[43]), а затем командой **bg** запустить его на фоновое выполнение.

Чтобы посмотреть список всех фоновых задач, выполняющихся в рамках текущего сеанса пользователя, используется команда **jobs**, которая также выведет и текущее состояние каждой задачи.

---

[41] Напомним, что при запуске любой программы автоматически формируются три потока: ввода, вывода и вывода ошибок (в Си их принято обозначать как stdin, stdout, stderr). По умолчанию все эти потоки связаны с терминалом процесса.

[42] Это только одно из действий, которые необходимо выполнить, чтобы процесс стал полноценным «демоном».

[43] Следует помнить, что указанная комбинация клавиш для разных терминалов может быть разной. Здесь рассматриваются терминалы класса linux и xterm.

Чтобы перевести задачу, выполняющуюся в фоновом режиме, на передний план (в интерактивный режим), надо вызвать команду **fg** и передать ей в качестве аргумента номер фоновой задачи (из списка, выдаваемого командой *jobs*).

Любая фоновая задача будет приостановлена, если ей требуется определённый доступ к консоли (например, для ввода информации).

5.2. Состояние процесса

Каждый процесс в операционной системе Linux может находиться в одном из четырёх состояний: работоспособный, спящий (или ожидающий), остановленный и завершившийся.

Работоспособный (англ. runnable) процесс. Если процесс в текущий момент времени выполняет какие-либо действия или стоит в очереди на получение кванта времени на центральном процессоре, он называется работоспособным и обозначается символом R.

Ожидающий (спящий, англ. sleeping) процесс. Это состояние обозначается символом S и возникает после того, как процесс инициирует системную операцию, окончания которой он должен дождаться. К таким операциям относятся ввод-вывод, истечение заданного интервала времени, завершение дочернего процесса и т.д.

Остановленный (англ. stopped) процесс. Остановленный процесс не использует процессор и может быть полностью выгружен из оперативной памяти. Пользователь может остановить процесс, например, чтобы дать возможность другому процессу использовать больший объём оперативной памяти или быстрее завершиться. Обычный пользователь может остановить и продолжить только свой процесс, а суперпользователь – любой процесс в системе. Операционная система автоматически останавливает фоновые процессы в случае, когда они пытаются ввести данные с терминала. Это состояние обозначается символом T.

Завершившийся («зомби», англ. «zombie») процесс. После завершения процесса информация о нём должна быть удалена операционной системой из таблицы процессов. В Linux такая операция возможна только после того, как родительский процесс выполнит системную операцию «ожидания завершения дочернего процесса» и прочитает статус возврата. До этих пор ОС вынуждена хранить в таблице процессов запись о завершившемся процессе, хотя он реально уже не существует и не потребляет ресурсы ЭВМ. Такое состояние процесса обозначается символом Z.

5.3. Атрибуты процесса

Любой процесс в операционной системе описывается **дескриптором**, содержащим ряд параметров, включая:

- уникальный идентификатор процесса (PID);
- идентификатор родительского процесса (PPID);

- связанный с процессом терминал (TTY);
- идентификаторы пользователя (UID, RUID, EUID, FSUID, SUID) и группы (GID, RGID, EGID, FSGID, SGID), от имени которых процесс выполняется и пытается осуществлять действия в рамках операционной системы или файловой системы;
- номер группы процессов (GROUP);
- идентификатор пользовательского сеанса (SESSION);
- и т.д.[44]

Уникальный идентификатор процесса (PID) – это целое число, позволяющее ядру системы различать процессы, т.е. своего рода «имя» процесса. По сути, PID – это номер ячейки в таблице дескрипторов процессов, выполняемых под управлением операционной системы. Когда создаётся новый процесс, ядро операционной системы использует следующую свободную запись в таблице дескрипторов. Если достигнут конец таблицы, то производится поиск свободной ячейки, начиная с начала таблицы. Когда процесс завершает свою работу, ядро освобождает занятый им идентификатор и соответствующий ему дескриптор.

Каждый процесс «запоминает» своего родителя в поле **PPID**. Зная значения поля PID и PPID, можно построить иерархию порождения процессов, начиная с самого первого процесса в системе (он обычно имеет PID = 0). Такая иерархия называется **деревом процессов**. Значение поля PPID играет особую роль при завершении процесса (см. ниже).

Как известно, каждому интерактивному процессу при создании назначаются потоки для: ввода информации, для её вывода и для вывода ошибок (stdin, stdout, stderr)[45]. Обычно все эти потоки указывают на один терминал, который называется «связанным» с процессом, и имя этого терминала (имя файла устройства) помещается в параметр **TTY**.

Все процессы в операционной системе запускаются каким-либо пользователем[46]. Идентификатор пользователя, запустившего процесс, помещается в параметр «**реальный идентификатор пользователя»** (англ. UID или RUID, Real UID). Часто возникает необходимость, чтобы процесс был запущен одним пользователем, а имел возможность получать доступ к ресурсам, принадлежащим другому пользователю. Контроль доступа к ресурсам системы осуществляется, исходя из значения поля «**эффективный идентификатор пользователя**» (англ. EUID, Effective UID). Примером ситуации выполнения действий от имени другого пользователя может служить смена пользовательского пароля с помощью команды *passwd*. Очевидно, что если база паролей (файлы */etc/passwd* или */etc/shadow*) будет доступна на запись кому угодно, тогда нельзя будет говорить ни о какой безопасности системы. Поэтому *passwd* настроена таким образом, что независимо от того, какой пользователь её выполняет, действия будут производиться всегда от имени

[44] На самом деле дескриптор процесса содержит значительно больше полей. Здесь приведены только те, которые используются в данном учебном пособии.

[45] О переопределении потоков ввода-вывода рассказывалось в п. 4.5.

[46] Считается, что загрузку операционной системы выполняет суперпользователь (администратор, root).

суперпользователя (англ. root, UID = 0). При этом ответственность за правильную и допустимую смену пароля несёт *passwd*. «Настроена» означает, что владельцем файла является пользователь *root*, и файл имеет специальное право «*Установить эффективного пользователя*»[47]. При запуске такого файла в поле EUID операционная система поместит идентификатор владельца файла, и все действия будут осуществляться уже от его имени. Очевидно, что значения полей RUID и EUID могут и совпадать (когда пользователь, запустивший процесс, выполняет все действия от своего имени). В ходе выполнения пользовательский процесс имеет возможность изменять значения RUID и EUID (менять их местами или делать одинаковыми[48]). При изменении значения поля EUID его изначальное значение сохраняется в поле «**сохранённый идентификатор пользователя**» (англ. SUID, Saved UID). Кроме эффективного идентификатора в дескрипторе процессов в ОС Linux имеется нестандартное поле **FSUID** (англ. File System UID), которое используется при обращении процесса к файловой системе. При запуске процесса FSUID эквивалентен EUID. Всё сказанное про идентификаторы пользователей справедливо и для основной группы пользователя (поля GID или RGID, EGID, SGID, FSGID).

В операционной системе Linux процессы, выполняющие одну задачу, **объединяются в группы**. Например, если в командной строке задано, что процессы связаны при помощи программного канала, они обычно помещаются в одну группу процессов. Каждая группа процессов обозначается собственным идентификатором. Процесс внутри группы, идентификатор которого совпадает с идентификатором группы процессов, считается **лидером группы процессов**.

В свою очередь, каждая группа процессов принадлежит к **сеансу пользователя**. Сеанс обычно представляет собой набор из одной группы процессов переднего плана, использующей терминал, и одной или более групп фоновых процессов. Каждый сеанс также обозначается идентификатором, который совпадает с PID процесса-лидера. При завершении пользовательского сеанса все принадлежащие ему процессы автоматически завершаются системой. Именно поэтому *процессы-демоны обязательно должны иметь идентификатор сеанса, отличный от сеанса пользователя*, который их запустил.

5.4. Получение информации о состоянии и атрибутах процессов

5.4.1. Из командной строки

Чтобы получить информацию о том, какие процессы запущены в данный момент в системе и в каком состоянии они находятся, можно использовать команду **ps** (англ. process status) с определённым набором опций (таблица 9).

[47] Подробнее о правах на объекты файловой системы говорится в п.4.3.12.

[48] Пользователю доступны только эти поля. Суперпользователь может изменять поля RUID, EUID, FSUID.

Таблица 9. Опции команды *ps*.

Опция	Действие
-a	Выдать все процессы системы, включая лидеров сеансов.
-d	Выдать все процессы системы, исключая лидеров сеансов.
-e	Выдать все процессы системы.
-x	Выдать процессы системы, не имеющие контрольного терминала.
-o	Определяет формат вывода. Формат задаётся как параметр опции в виде символьной строки, поля которой (таблица 10) разделяются символом «,» (запятая).
-u	Выдать процессы, принадлежащие указанному пользователю. Пользователь задаётся в параметре опции в символьном виде.

Таблица 10. Поля форматного вывода команды *ps*.

Формат	Значение
F	Статус процесса.
S	Состояние процесса.
UID	Идентификатор пользователя.
PID	Идентификатор процесса.
PPID	Идентификатор родительского процесса.
PRI	Текущий динамический приоритет процесса.
NI	Значение приоритета процесса.
TTY	Управляющий терминал процесса.
TIME	Суммарное время выполнения процесса процессором.
STIME	Время создания процесса.
COMMAND	Имя команды, соответствующей процессу.
SESS	Идентификатор сеанса процесса.
PGRP	Идентификатор группы процесса.

Например, список запущенных процессов Вашей системы можно получить, используя команду *ps -e*. Информацию о группах и сессиях процесса можно получить так:

```
$ ps -axo pid,pgrp,session,command
```

Можно также посмотреть «дерево» процессов, используя команду **pstree**.

Чтобы увидеть использование ресурсов «в динамике», можно использовать команду **top**. Выход из просмотра осуществляется нажатием клавиши «q».

5.4.2. При выполнении процессов

Для получения значений идентификаторов процесс может использовать следующие системные вызовы:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid  (void)
pid_t getppid (void)
uid_t getuid  (void);
uid_t geteuid (void);
gid_t getgid  (void);
gid_t getegid (void);
pid_t getpgrp (void);
pid_t getsid (pid_t pid);
```

Назначение функций очевидно из названия. Например, функция *getpid* вернёт идентификатор процесса (PID). Все функции, кроме *getsid*, не принимают никаких параметров и возвращают значение соответствующих полей из дескриптора текущего (вызвавшего функцию) процесса. Функция *getsid* позволяет узнать идентификатор сеанса для любого процесса в системе (если это позволено соответствующему пользователю), идентификатор которого передаётся в качестве параметра. Если передать вызову *getsid* значение 0, то он вернёт идентификатор сеанса вызывающего процесса (т.е. это эквивалентно записи *getsid(getpid());*).

Процесс может создать новую группу или присоединиться к существующей, а также определить новый сеанс при помощи системных вызовов:

```
#include <sys/types.h>
#include <unistd.h>

pid_t setpgid(pid_t pid, pid_t pgid);
pid_t setsid(void);
```

Вызов *setpgid* устанавливает идентификатор группы процесса с идентификатором, равным *pid*, в *pgid*. В случае ошибки возвращается –1.

Изменить значения полей RUID, EUID, FSUID, RGID, EGID, FSGID можно при помощи системных вызовов:

```
#include <sys/types.h>
#include <unistd.h>

int setuid   (uid_t uid);
int seteuid  (uid_t euid);
```

```
int setreuid (uid_t ruid, uid_t euid);
int setfsuid (uid_t fsuid);
int setgid   (gid_t gid);
int setegid  (gid_t egid);
int setfsgid (gid_t fsgid);
int setregid (gid_t rgid, gid_t egid);
```

Назначение функций очевидно из названия. Пояснений требуют лишь функции *setreuid* и *setregid*. Обе эти функции предназначены для изменения полей RUID, EUID и RGID, EGID соответственно в ситуации, когда операционная система не поддерживает дополнительных полей SUID, SGID. Эти функции включены в библиотеку для реализации совместимости с такими операционными системами как 4.3 UNIX BSD, которые не имеют в дескрипторах процессов указанных полей. За подробной информацией об использовании этих функции следует обратиться к электронной справочной системе *man*.

Очевидно, что процесс может получить информацию только о своих атрибутах, и при этом процесс всегда находится в состоянии Running (R).

Использование функции установки значений полей дескриптора процесса рассмотрим на примере создания процесса-демона (листинг 1).

Листинг 1. Создание процесса-демона

```
1. #include <unistd.h>
2.
3. int main(void){
4.   close (0);
5.   close (1);
6.   close (2);
7.   setsid(0);
8.   setpgrp(0,0);
9.
10.  /* тело процесса-демона */
11.  sleep(10);
12.  return (0);
13.}
```

В первой строке подключается необходимый заголовочный файл. В данном случае нам нужны только функции по работе с атрибутами процесса, файловыми дескрипторами и функция *sleep*, которая переводит процесс в состояние ожидания. Далее в строках 4-6 процесс отключается от терминала (вспомним, что «демон» не имеет «присоединённого» терминала). После чего создаёт новый сеанс и группу (строки 7-8). Затем процесс может выполняться независимо от того, находится ли запустивший его пользователь в системе или уже завершил свой сеанс. В нашем случае «демон» просто «спит» 10 секунд (строка 11) и затем завершается (строка 12).

5.5 Потоки одного процесса (нити)

Точно так же, как многозадачная операционная система может выполнять несколько программ одновременно, один процесс может обрабатывать несколько потоков команд, которые называются **нитями**. Каждая нить представляет собой независимо выполняющуюся функцию со своим счётчиком команд, регистровым контекстом и стеком. Понятия процесса и нити очень тесно связаны и поэтому трудноотличимы. Нити даже часто называют *легковесными процессами*[49].

Основные **отличия процесса от нити** заключаются в том, что каждому процессу соответствует своя независимая от других область памяти, таблица открытых файлов, текущая директория и прочая информация уровня ядра. Нити одного процесса наоборот выполняются в общем адресном пространстве, им доступны общие переменные, дескрипторы открытых файлов, терминал и т.д. Другими словами, в системе выполняется несколько независимых процессов, каждый из которых состоит как минимум из одной нити.

5.6. Порождение процессов и запуск программ

5.6.1. Порождение процессов

Новый процесс порождается с помощью системного вызова **fork**:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Порождённый или *дочерний процесс* является точной копией родительского процесса за исключением следующих атрибутов:

- идентификатор процесса (PID);
- идентификатор родительского процесса (PPID). У дочернего процесса он приравнивается к PID родительского процесса (того, кто выполнил вызов *fork*);
- дочерний процесс не имеет очереди сигналов, ожидающих обработки.

После завершения вызова *fork*, оба процесса начинают выполнять одну и ту же инструкцию – следующую за вызовом *fork*.

Чтобы процесс мог определить, кем он является (дочерним или родительским), используется возвращаемое значение вызова *fork*. В первом случае, когда процесс является дочерним, *fork* вернёт 0, во втором – PID вновь созданного процесса. Если *fork* возвращает –1, то это свидетельствует об

[49] В операционной системе Windows NT вводится термин «волокно» (англ. fiber) для обозначения потока, планированием которого занимается не операционная система, а сам процесс.

ошибке (естественно, в этом случае возврат происходит только в процесс, выполнивший системный вызов, и новый процесс не создаётся).

Пример программы, порождающей новый процесс, приведён в листинге 2.

Листинг 2. Порождение нового процесса

```
1. #include <stdio.h>
2. #include <unistd.h>
3.
4. int main(void)
5. {
6.   int pid;
7.   pid = fork();
8.   if (pid == -1)
9.   {
10.    perror("fork"); exit(1);
11.  }
12.  if (pid == 0)
13.  {
14.    /*Эта часть кода выполняется дочерним процессом*/
15.    printf("Потомок\n");
16.    sleep(1);
17.  }
18.  else
19.  {
20.    /*Эта часть кода выполняется родительским
21.      процессом*/
22.    printf("Родитель. Создан потомок %u\n", pid);
23.    sleep(1);
24.  }
25.  return (0);
26.}
```

В строках 1-2 подключаются необходимые заголовочные файлы. В данном случае нам требуются функции ввода-вывода (*stdio.h*) и функции по работе с процессами (*unistd.h*). Порождение процесса происходит в строке 7, после чего проверяется возвращаемое значение функции *fork*. Если новый процесс создан, то условие в строке 8 не выполнится, и каждый процесс перейдёт на строку 12. Далее, проверив значение переменной *pid*, дочерний процесс будет выполнять строки 13-17 (так как в его случае *pid* = 0), а родительский – строки 19-24. Затем оба процесса завершатся с кодом 0 (строка 25).

Особое внимание следует обратить на строки 16 и 22, в которых родительский и дочерний процессы добровольно переходят в «спящее» состояние на 1 секунду. Для чего это сделано? Ответ прост – для того чтобы оба процесса смогли вывести информацию в поток вывода, и она была отображена на экране терминала. Если этого не сделать, то может возникнуть

ситуация, когда один из процессов (в силу алгоритмов планирования) завершит свою работу быстрее, чем другой начнёт выводить информацию в поток вывода. В этом случае завершившийся процесс автоматически закроет все открытые файлы, которыми в нашем случае являются потоки ввода-вывода. В силу того, что файловые дескрипторы у родственных процессов одинаковые, ещё не завершившийся процесс начнёт выдавать информацию в поток, у которого соответствующий файл уже закрыт. И на экран никакой информации выдано не будет.

5.6.2. Запуск новых программ

Очевидно, что простое порождение нового процесса, который бы исполнял точно такой же код, как и родительский процесс, смысла особого не имеет. Важно реализовать возможность выполнять в новом процессе другую программу. Для того чтобы загрузить исполняемый код в адресное пространство[50] процесса и настроить его на выполнение новой программы, используются функции семейства **exec**:

```
#include <unistd.h>

int execl  (const char *path, const char *arg0, ...,
            const char *argn, char * /*NULL*/);
int execv  (const char *path, char *const argv[]);
int execle (const char *path,char *const arg0[], ...,
            const char *argn, char * /*NULL*/, char
            const envp[]);
int execve (const char *path, char *const argv[],
            char *const envp[]);
int execlp (const char *file, const char *arg0, ...,
            const char *argn, char * /*NULL*/) ;
int execvp (const char *file, char *const argv[]);
```

На рисунке 6 показано, как связаны между собой приведённые выше функции.

[50] Обратите внимание, что вызовы *fork* и *exec* никак не связаны между собой. Заменить исполняемую программу можно и в текущем процессе, не порождая новый процесс.

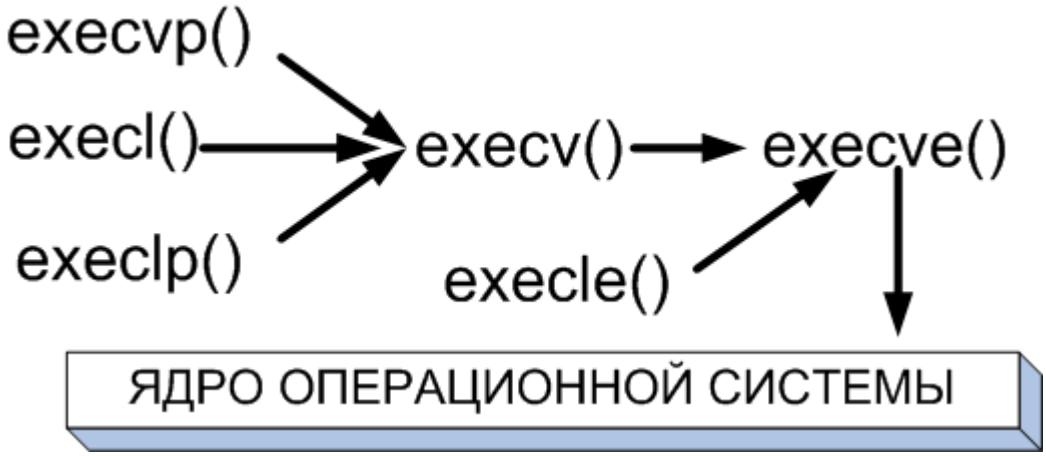

Рисунок 6. Связь функций семейства *exec*

Всё множество системных вызовов *exec* выполняет одну и ту же функцию: они преобразуют вызывающий процесс, загружая новую программу в его пространство памяти, и настраивают дескриптор процесса так, чтобы загруженная программа начала выполняться с начала. Если вызов *exec* завершился успешно, то вызывающая программа полностью замещается новой программой. Важно отметить, что вызов *exec* не создаёт новый процесс, который выполняется одновременно с вызывающим, а вместо этого новая программа загружается на место старой. Поэтому, в отличие от вызова *fork*, успешный вызов *exec* не возвращает никакого значения.

Базовой функцией в указанном семействе является вызов *execve()*, который принимает три параметра:

- полный путь к исполняемому файлу, который необходимо загрузить в память и начать выполнять;
- список аргументов, которые передаются исполняемому файлу;
- список переменных среды окружения нового процесса.

Списки аргументов и переменных среды окружения передаются как массивы указателей на строки, в которых последний элемент содержит NULL (нуль). Такой формат используется для того, чтобы передавать только массивы и не указывать количество элементов в них.

Остальные функции являются надстройками над *execve*() и позволяют программисту либо указывать параметры в другом виде, либо какие-либо параметры не указывать вообще. Функция *execv()* позволяет не указывать переменные среды окружения нового процесса, а использовать копию переменных окружения текущего процесса. Аналогично поступают и функции *execvp() execl()*, *execlp()*. Функции *execle*(), *execlp*() и *execl*() позволяют указать аргументы, передаваемые новому процессу не как массив, а как перечень параметров функции. Функции *execvp()* и *execlp()* позволяют указывать не полный путь к исполняемому файлу, а только его имя. Полный путь будет определён функцией, исходя из значения переменной среды окружение PATH.

Резюмируя, можно сказать, что имена функций определяют перечень и типы параметров, которые они принимают. Буква «v» в имени функции означает, что аргументы, передаваемые новому процессу, задаются как массив

указателей на строки с завершающим элементом, содержащим NULL. Буква «l» – эти аргументы перечисляются как параметры функции. Буква «p» означает, что первым параметром может быть задано только имя исполняемого файла, и полный путь будет сформирован самой функцией.

Пример использования вызова *execl* приведён в листинге 3.

Листинг 3. Загрузка новой программы в текущий процесс

```
1. #include <unistd.h>
2.
3. int main(void)
4. {
5.   printf ("Запуск программы ls\n");
6.   execl ("/bin/ls", "ls", "-l", (char *)0);
7.   perror ("Ошибка вызова execl");
8.   return (1);
9. }
```

Обратите внимание, что в приведённом примере за вызовом *execl* следует безусловный вызов функции *perror*. Это отражает то, что успешный вызов *execl* прекращает выполнение текущей программы, а в случае возникновения ошибки выполнение текущей программы продолжается. В приведённом примере «старая» программа всегда завершается с кодом возврата 1 (строка 8), что сигнализирует о возникшей ошибке.

Для того чтобы совместить действия по порождению нового процесса и по запуску программы, можно использовать вызов *system*:

```
#include <unistd.h>

int system (char * path);
```

Пример использования вызова *system* приведён в листинге 4. В данном примере программа выводит на экран фразу *«Запуск программы ls»* (строка 4), затем запускает интерпретатор *sh* с параметрами «*–c ls*» (строка 5). При этом вызов *system* выполняет следующие действия: порождает новый процесс (*fork*), загружает туда файл */bin/sh*, настраивает его на выполнение команды *ls*, запускает его и дожидается завершения. Текущий процесс при этом никак не изменяется.

Листинг 4. Запуск программы с использованием *system*

```
1. #include <unistd.h>
2. #include <stdio.h>
3.
4. int main(void){
5.   printf ("Запуск программы ls\n");
6.   system ("ls");
7.   return (0);
```

Листинг 4. Запуск программы с использованием *system*

```
8. }
```

5.6.3. Создание потока (нити)

По сути, нить – это некоторая функция программы, которая должна выполняться параллельно. «Создать нить» значит сообщить операционной системе, какую из функций программы требуется выполнять отдельно и как при этом производить планирование использования ресурсов.

По стандарту POSIX для создания нити используется функция *pthread_create*, заголовок которой следующий:

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void* (*start)(void *),
                   void *arg)
```

Первый параметр *thread* этой функции – это указатель на переменную типа *pthread_t*, в которую будет записан идентификатор созданной нити. По сути, это аналог PID.

Второй параметр *attr* задаёт набор некоторых свойств создаваемой нити, описывающей правила планирования исполнения нити, её окружение (стек и т.п.) и поведение нити при завершении (удалять ли информацию сразу или ожидать вызова *pthread_join*). Если требуется запустить нить с параметрами «по умолчанию», то в качестве этого параметра следует передать *NULL*.

Параметр *start* – это указатель на функцию, возвращающую значение типа *void** и принимающую в качестве аргумента значение типа *void**. Именно эту функцию и начинает выполнять вновь созданная нить, при этом в качестве аргумента этой функции передаётся четвёртый аргумент вызова *pthread_create*.

Функция *pthread_create* возвращает нулевое значение в случае успеха и ненулевой код ошибки в случае неудачи.

Пример использования функции *pthread_create* приведён в листинге 5.

Листинг 5. Запуск двух нитей

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.
4.  void * my_thread (void * arg){
5.    char ch;
6.    ch = *((char *)arg);
7.    while (1) printf ("%c", ch);
8.    return (NULL);
9.  }
10.
11. int main(void)
```

Листинг 5. Запуск двух нитей

```
12. {
13.   pthread_t thread_id1, thread_id2;
14.   char ch1 = 'A', ch2 = 'B', ch3 = 'C';
15.
16.   pthread_create (&pthread_id1, NULL, my_thread,
17.                   (void *)&ch1);
18.   pthread_create (&pthread_id2, NULL, my_thread,
19.                   (void *)&ch2);
20.   my_thread ((void *)&ch3);
21.   return (0);
22. }
```

В строках 1-2 подключаются необходимые заголовочные файлы. В данном случае нам требуются функции ввода-вывода (*stdio.h*) и функции по работе с нитями (*pthread.h*)[51]. Далее в строках 4-9 описывается функция, которая будет нами дальше использоваться как тело отдельной нити. Создание нитей и запуск их на выполнение происходит в строках 16-19. Основная нить также выполняет функцию *my_thread* (строка 20). В итоге на экран терминала будет выведена последовательность из символов A, B и C.

Следует обратить внимание, что каждая из нитей использует собственную локальную переменную *ch*. Забегая вперёд скажем, что приведённый пример не совсем корректен, так как функция *printf* из стандартной библиотеки *glibc* не является «безопасной для нитей». Подробнее об этом будет сказано в следующем разделе.

5.7. Завершение процесса

Существует несколько способов завершения программы: возврат из функции *main* (оператор **return***)* и вызов функций **exit**. Системный вызов *exit* выглядит следующим образом:

```
#include <unistd.h>

void exit (int status);
```

Аргумент *status*, передаваемый функции *exit*, возвращается родительскому процессу и представляет собой код возврата программы. При этом считается, что программа возвращает 0 в случае «успеха» и другую величину в случае неудачи. Значение кода неудачи может иметь дополнительную трактовку, определяемую самой программой. Наличие кода возврата позволяет организовать условный запуск программ.

[51] Следует помнить, что для компиляции приведённого примера следует указать ключ *–lpthread*, который определяет, что используется дополнительная библиотека *pthread* по работе с нитями.

При завершении программы **важную роль играет значение поля PPID** дескриптора процесса. Если родительский процесс ещё не завершен, то операционная система не имеет права полностью удалить информацию о процессе, а вынуждена хранить статус завершения до тех пор, пока его не прочитает процесс-родитель. Как было сказано выше, такая ситуация приводит к появлению процессов-зомби.

Помимо передачи кода возврата, функция *exit* производит ряд действий, в частности выводит буферизированные данные и закрывает потоки ввода-вывода. Альтернативой ей является функция *_exit*, которая не производит вызовов библиотеки ввода-вывода, а сразу вызывает завершение процесса. Пример использования функции *_exit* приведён в листинге 6. В приведённом примере функция *printf* ничего не выдаст на экран. Это связано с тем, что вывод на терминал в каноническом режиме по умолчанию буферизируется[52]. Фактический вывод символов на экран осуществляется только после помещения в буфер символа конца строки или выполнения вызова *fflush*.

Листинг 6. Завершение процесса с помощью *_exit*

```
1. #include <stdio.h>
2. #include <unistd.h>
3.
4. int main (void){
5.   printf ("Строка, которая не будет выведена.");
6.   _exit(0);
7. }
```

Функция *exit* может быть вызвана в любом месте программы. В некоторых ситуациях неожиданное завершение процесса может привести к необратимым последствиям. Например, если процесс занимается обработкой некоторой информации, начинает процедуру записи и завершается, не закончив её, то обрабатываемые данные могу быть испорчены. Чтобы избежать подобной ситуации процесс может зарегистрировать *обработчики выхода* (англ. exit handlers) – функции, которые вызываются функцией *exit* после выполнения стандартных действий, но до окончательного завершения процесса (вызова функции *_exit*). Обработчики выхода регистрируются с помощью функции **atexit**. Функцией *atexit* может быть зарегистрировано до 32 обработчиков. На рисунке 7 проиллюстрированы возможные варианты завершения программы.

```
#include <unistd.h>

int atexit(void (*func)(void));
```

Обработчики выхода вызываются по принципу LIFO (последний зарегистрированный обработчик будет вызван первым) только при

[52] Подробнее о терминалах, режимах их работы рассказывается в курсе «Организация ЭВМ и систем».

использовании вызова *exit*. Если процесс завершается с использованием вызова *_exit*, то обработчики выхода не вызываются.

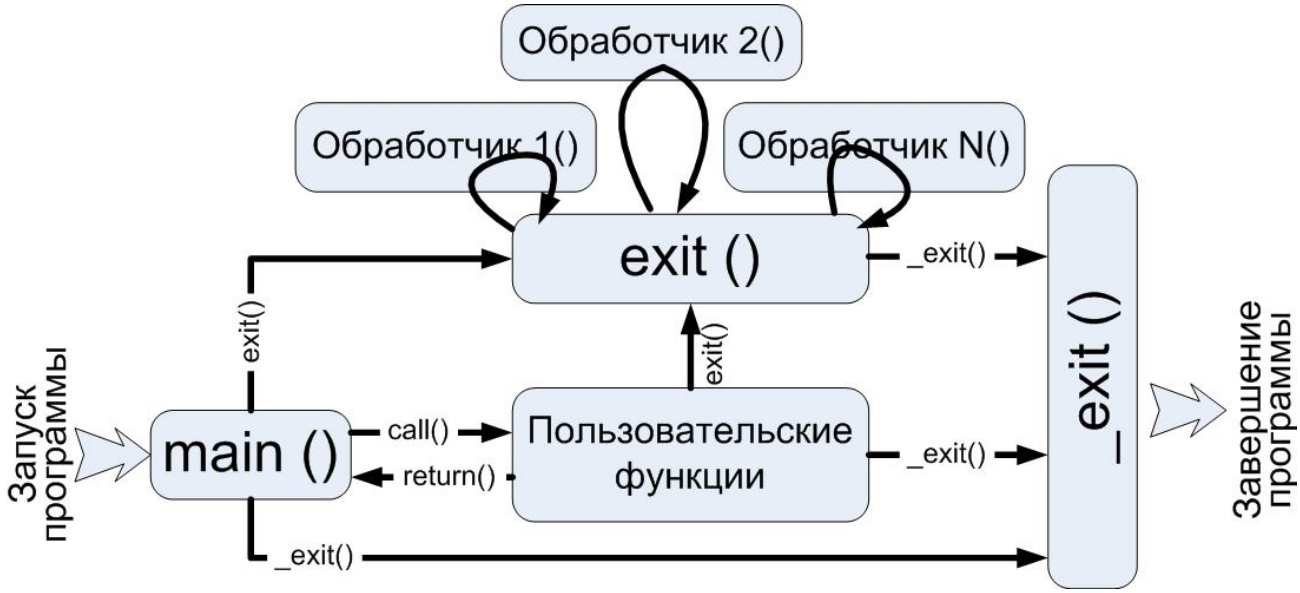

Рисунок 7. Механизмы завершения программ

Пример программы, регистрирующей обработчики выхода, представлен в листинге 7.

Листинг 7. Определение обработчиков завершения процесса

```
#include <stdio.h>
#include <stdlib.h>

void myexit1(void){
  printf ("Это завершение1.\n");
}

void myexit2(void){
  printf ("Это завершение2.\n");
}

int main (void){
    printf ("Программа запущена.\n");
    atexit (myexit1);
    atexit (myexit2);
    printf ("Программа отработала.\n");
    exit (10);
}
```

5.8. Завершение потока (нити)

Нить завершается, когда происходит возврат из соответствующей ей функции или с использованием вызова **pthread_exit**:

```
#include <pthread.h>

int pthread_exit(void *value_ptr)
```

Этот вызов завершает выполняемую нить, возвращая в качестве результата её выполнения *value_ptr*.

Следует помнить, что завершение процесса всегда сопровождается завершением всех его нитей. Т.е. если нить выполнит вызов *exit* (а не *pthread_exit*), то будут завершены все нити и сам процесс.

Нить, так же как и процесс, может быть завершена принудительно. Для отправки требования о завершении нити используется вызов *pthread_cancel* или *pthread_kill*:

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
int pthread_kill(pthread_t thread, int sig);
```

5.9. Обработка ошибок

Обычно в случае возникновения ошибки системные вызовы возвращают –1 и устанавливают значение переменной **errno**, указывающее причину возникновения ошибки. Так, например, существует более десятка причин завершения вызова *open* с ошибкой, и все они могут быть определены с помощью переменной *errno*. Заголовочный файл *errno.h* содержит коды ошибок, значения которых может принимать переменная *errno*, с краткими комментариями.

Библиотечные функции, как правило, не устанавливают значение переменной *errno*, а код возврата различен для разных функций. Для уточнения возвращаемого значения библиотечной функции необходимо обратиться к электронному справочнику *man*.

Поскольку базовым способом получения функций ядра являются системные вызовы, рассмотрим более подробно обработку ошибок в этом случае. Переменная *errno* определена следующим образом:

```
external int errno;
```

Следует обратить внимание, что значение *errno* не обнуляется следующим нормально завершившимся системным вызовом. Таким образом, значение *errno* имеет смысл только после системного вызова, который завершился с ошибкой.

Стандарт ANSI C определяет две функции, помогающие сообщить причину ошибочной ситуации: **strerror** и **perror**.

Функция *strerror* имеет вид:

```
char *strerror(int errnum);
```

Она принимает в качестве аргумента *errnum* номер ошибки и возвращает указатель на строку, содержащую сообщение о причине ошибочной ситуации.

Функция *perror* объявлена следующим образом:

```
void perror(const char *s);
```

Она выводит в стандартный поток сообщений об ошибках информацию об ошибочной ситуации, основываясь на значении переменной *errno*. Строка *s*, передаваемая функции, предваряет это сообщение и может служить дополнительной информацией, например, содержать название функции или программы, в которой произошла ошибка. Следующий пример (листинг 8) иллюстрирует использование этих двух функций.

Листинг 8. Использование переменной *errno* и функций *strerror* и *perror*

```
1. #include <errno.h>
2. #include <stdio.h>
3. int main (int argc, char *argv[]){
4.   fprintf(stderr, "ENOMEM: %s\n", strerror(ENOMEM));
5.   errno = ENOEXEC;
6.   perror(argv[0]);
7. }
```

В Приложении 1 приведены наиболее общие ошибки системных вызовов, включая сообщения, которые обычно выводят функции *strerror* и *perror*, а также их краткое описание.

Контрольные вопросы

1. Что означает «запустить программу на выполнение»?
2. Что такое процесс? Дескриптор процесса?
3. Перечислите основные типы процессов. Объясните, чем они отличаются.
4. В каких состояних может находиться процесс? Какой процесс называется «зомби»?
5. Перечислите основные атрибуты процесса. Расскажите об их значении.
6. Что такое группа процессов? Что такое сеанс пользователя?
7. Какие команды используют для работы с процессами? Расскажите о системных вызовах для получения значения идентификаторов.
8. Что такое поток (нить)? В чём отличия процесса от нити?

9. Для чего используется системный вызов *fork*?
10. Как запустить программу на выполнение во вновь порождённом процессе?
11. В чём отличия функций семейства *exec*?
12. Что делает системный вызов *system*?
13. Расскажите о функциях по работе с нитями.
14. Как происходит завершение процесса? Какие системные вызовы можно для этого использовать?
15. Как происходит завершение нити? Какие системные вызовы можно для этого использовать?
16. Расскажите о функциях, используемых для обработки ошибок.

ГЛАВА 6. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ И НИТЕЙ

Как было сказано ранее, идея многопроцессных (многопрограммных) операционных систем заключается в том, что на одной ЭВМ (псевдо)одновременно исполняется несколько программ (процессов)[53]. При этом считается, что процессы: 1) независимы и 2) ничего не знают о существовании друг друга[54]. Каждому процессу выделяется свой набор ресурсов, которыми он монопольно владеет, и никто другой (кроме операционной системы) не имеет прямого доступа к ним.

Каждый процесс, в свою очередь, может выполнять несколько нитей (потоков инструкций). При этом все нити одного процесса также выполняются независимо друг от друга и обладают некоторым набором «собственных» ресурсов, но без труда могут получить доступ к общей памяти процесса.

Очевидно, что нити изначально предназначены для взаимодействия друг с другом. А могут ли возникнуть ситуации, когда одному процессу необходимо «сообщить» другому (или нескольким другим) процессу(-ам) какую-либо информацию? Ответ очевиден – да. С такими взаимодействиями мы уже сталкивались, когда изучали конвейерный способ запуска программ. Кроме этого, процессу, например, может потребоваться:

- сообщить о своей готовности обработать данные;
- «поделиться» с другим процессом объемом выполняемой работы или взять часть работы другого процесса на себя;
- обеспечить строгую последовательность работы нескольких процессов (например, чтобы начисление процентов на счете в банке производилось только после его пополнения);
- и т.п.

К средствам межпроцессного взаимодействия (англ. Interprocess Communications, IPC), реализованных в операционной системе Linux, относятся:

- ожидание завершения родственного процесса;
- сигналы;
- каналы (именованные и неименованные);
- очереди сообщений;
- семафоры;
- разделяемая память;
- сокеты.

Последний способ IPC позволяет реализовать взаимодействие между процессами, выполняющимися на разных ЭВМ, соединённых каналами передачи данных.

---

[53] Очевидно, что в случае многопроцессорной (многоядерной) ЭВМ процессы на самом деле выполняются параллельно.

[54] Вспомним, что каждый процесс «знает» лишь номер своего родительского процесса (поле PPID дескриптора).

Все средства IPC условно можно разделить по типу процессов, участвующих во взаимодействии:

- «любой процесс». Ресурсы IPC доступные всем процессам (согласно правам доступа). Такие ресурсы создаются и управляются самими процессами. К этому типу IPC относятся: именованные каналы, очереди сообщений, семафоры, разделяемая память;
- «родственные процессы». Такие ресурсы создаются и управляются операционной системой. К этому типу ресурсов относятся: сигналы, неименованные каналы, сокеты.

Кроме того, процессы могут получать информацию о том, каким образом они были запущены и в какой среде. С этого типа взаимодействия и начнём.

6.1 Получение информации о запуске

6.1.1. Взаимодействие с командной оболочкой (строкой запуска)

В программе, написанной на языке Си, первой выполняемой функцией является функция **main**. Традиционно функция *main* определяется следующим образом[55]:

```
int main(int argc, char *argv[]);
```

Из заголовка функции видно, что она имеет два параметра и возвращает целое значение. Первым параметром (**argc**) в функцию передаётся число аргументов, указанных в командной строке при запуске программы, включая её имя. Указатели на каждый из аргументов командной строки передаются в массиве **argv[].** Таким образом, *argv*[0] указывает на строку, содержащую имя программы, *argv*[1] – на первый аргумент и т.д. до *argv* [*argc* – 1]. Элемент *argv* [*argc*] всегда содержит NULL.

Возвращаемое значение функции *main* является статусом завершения программы (процесса).

Приведём пример программы, которая выводит значения всех аргументов, переданных функции *main* (листинг 9).

Листинг 9. Вывод информации о командной строке

```
1. #include <stdio.h>
2. int main(int argc, char *argv[]){
3.   int i;
4.   printf ("Имя программы - %s\n", argv[0]);
5.   printf ("Число параметров - %d\n", argc-1);
6.   for (i = 1; i < argc; i++)
7.     printf("argv[%d] = %s\n", i, argv[i]);
8.   return (0);
```

[55] Некоторые компиляторы с языка Си допускают и другие определения функции *main*. Например, Borland TurboC допускает определение типа возвращаемого значения как *void*. В gcc можно разрешить игнорирование аргументов функции *main*.

Листинг 9. Вывод информации о командной строке

```
9. }
```

Чтобы определить, были ли указаны в командной строке опции, аргументы командной строки могут быть обработаны при помощи следующих функций:

```
#include <unistd.h>

extern char *optarg;
extern int optind, opterr, optopt;

int getopt(int argc, char *argv[], char *optstring);

#include <getopt.h>

int getopt_long(int argc, char *argv[],
                char *optstring,
                struct option *longopts,
                int *longindex);

int getopt_long_only(int argc, char *argv[],
                     char *optstring,
                     struct option *longopts,
                     int *longindex);
```

Напомним, что *опцией* называется аргумент командной строки, начинающийся с символов «-»(тире) или «--» (двойное тире). Причём, если используется один знак тире, то опция считается односимвольной или ***короткой***, если же два знака тире, то многосимвольной или ***длинной.***

Функция **getopt** просматривает аргументы командной строки по очереди и определяет, есть ли в них заданные короткие опции. Поиск только длинных опций осуществляется функцией **getopt_long_only**. Функция **getopt_long** осуществляет поиск как коротких, так и длинных опций.

В качестве аргументов функциям передаются:

- *argc* – количество аргументов командной строки (размер массива *argv*);
- *argv* – массив указателей на строки-аргументы командной строки;
- *optstring* – указатель на строку, содержащую символы коротких опций;
- *longopts* – массив структур *option*, описывающих длинные опции. Последний элемент массива должен быть заполнен нулями;
- *longindex* – указатель на целую переменную, в которую, в случае обнаружения длинной опции, будет помещён номер соответствующего элемента массива *longopts*.

Обратите внимание, что функции *getopt*, *getopt_long* и *getopt_log_only* на самом деле ничего не знают о командной строке, а всего лишь просматривают массив строк на предмет нахождения в нём заданных элементов.

Структура **option** определена в заголовочном файле *getopt.h* следующим образом:

```
struct option {
  char *name;
  int has_arg;
  int *flag;
  int val;
};
```

Значения полей структуры *option* приведены в таблице 11.

Таблица 11. Значения полей структуры *option.*

Поле	**Значение**
name	Является именем длинной опции.
has_arg	Может принимать значения: *no_argument* (или 0), если опция не имеет аргумента; *required_argument* (или 1), если опция требует обязательного указания аргумента; *optional_argument* (или 2), если опция может иметь необязательный аргумент.
val	Значение, которое должно быть помещено по адресу, указанному в аргументе *flag*, в случае если будет обнаружена данная длинная опция.
flag	Указывает адрес, куда требуется поместить значение *val*. Если *flag* равен NULL, то *val* возвращается как значение функции *getopt_long* (или *getopt_long_only*).

Глобальные переменные **optind**, **optarg**, **opterr** и **optopt** содержат соответственно номер аргумента командной строки, который должен быть проверен на наличие опции, указатель на аргумент найденной опции, наличие ошибки, возникшей в ходе выполнения функции поиска опций, и дополнительные параметры, конкретизирующие работу функций. Подробнее об этих переменных будет сказано далее.

Поиск коротких опций функцией *getopt* производится так же, как командой *getopts* (см. выше). Отличия следующие.

1. Если просмотрены все параметры командной строки, то *getopt* возвращает –1.

2. Если в *optstring* после некоторого символа следует два двоеточия, то это означает, что опция имеет **необязательный** аргумент. Это является дополнением GNU.

3. Если *optstring* содержит символ «W», за которым следует точка с запятой («;»), то опция *-W foo* рассматривается как длинная опция *--foo*. Опция -

W зарезервирована POSIX.2 для реализации расширений. Такое поведение является дополнительным для GNU и недоступным в библиотеках GNU libc более ранних по сравнению со второй версий.

По умолчанию *getopt* переставляет элементы содержимого *argv* в процессе поиска так, что, в конечном счёте, все аргументы, не являющиеся опциями, оказываются в конце. Возможны два других режима работы функции.

4. Если первым символом *optstring* является «+» или задана переменная окружения **POSIXLY_CORRECT**, то обработка опций прерывается на первом аргументе, не являющемся опцией.

5. Если первым символом *optstring* является «-»(тире), то каждый элемент *argv*, не являющийся опцией, обрабатывается так, как если бы он был аргументом опции с символом, имеющим код 1 (один). Такой режим работы используется программами, которые требуют опции и другие аргументы командной строки.

6. Специальный аргумент «--» (два тире) служит для обозначения конца опций независимо от режима работы функции *getopt*.

Если *getopt* не распознал символ опции, то он выводит в стандартный поток ошибок соответствующее сообщение, заносит символ в переменную *optopt* и возвращает «?». Вызывающая программа может предотвратить вывод сообщения об ошибке, установив нулевое значение **opterr**.

Если найден параметр, начинающийся с двух символов «-» (тире), то считается, что **найдена длинная опция**. В этом случае производится сравнение найденного аргумента (исключая двойное тире) со значениями *name* массива *longopts*.

Если совпадение найдено, то значение *val* помещается либо по адресу, указанному в *flag*, и *getopt_long* завершается с результатом 0, либо *getopt_long* завершается с результатом *val*. В любом из двух случаев, если параметр *longindex* не равен *NULL*, то по указанному в нём адресу помещается номер элемента массива, с которым совпала найденная длинная опция.

Длинная опция может иметь параметр, указываемый через знак равенства (*--опция=параметр*) или через пробел (*--опция параметр*). Наличие параметра указывается в поле *has_arg*.

Функция *getopt_long_only* работает так же, как *getopt_long*, но обрабатывает только длинные опции. В качестве указателя опции могут служить не только символы «--», но и «-». Если опция, начинающаяся с «-» (не с «--»), не совпадает с длинной опцией, но совпадает с короткой, то она обрабатывается как короткая опция.

Пример программы, анализирующей опции командной строки, представлен в листинге 10.

Листинг 10. Работа с переменными среды окружения

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <getopt.h>
4.
```

Листинг 10. Работа с переменными среды окружения

```
5.  int main (int argc, char **argv){
6.    int c;
7.    int digit_optind = 0;
8.    int this_option_optind;
9.    int option_index = 0;
10.   static struct option long_options[] = {
11.        {"add", 1, 0, 0},
12.        {"append", 0, 0, 0},
13.        {"delete", 1, 0, 0},
14.        {"verbose", 0, 0, 0},
15.        {"create", 1, 0, 'c'},
16.        {"file", 1, 0, 0},
17.        {0, 0, 0, 0}};
18.   while (1){
19.     this_option_optind = optind ? optind : 1;
20.     c = getopt_long (argc, argv, "abc:d:012",
21.                      long_options, &option_index);
22.     if (c == -1) break;
23.     switch (c){
24.       case 0: printf ("параметр %s",
25.                    long_options[option_index].name);
26.               if (optarg) printf (" с аргументом %s",
27.                                   optarg);
28.               printf ("\n");
29.               break;
30.       case '0':
31.       case '1':
32.       case '2': if (digit_optind != 0 &&
33.                     digit_optind != this_option_optind)
34.               printf ("цифры используются дважды.\n");
35.                 digit_optind = this_option_optind;
36.                 printf ("параметр  %c\n", c);
37.                 break;
38.       case 'a': printf ("параметр a\n");
39.                 break;
40.       case 'b': printf ("параметр b\n");
41.                 break;
42.       case 'c': printf ("c со значением '%s'\n",
43.                           optarg);
44.                 break;
45.       case 'd': printf ("d со значением '%s'\n",
46.                           optarg);
47.                 break;
48.       case '?': break;
```

Листинг 10. Работа с переменными среды окружения

```
49.         default: printf ("результат 0%o ??\n", c);
50.       }
51.     }
52.     if (optind < argc){
53.       printf ("элементы ARGV, не параметры: ");
54.       while (optind < argc)
55.         printf ("%s ", argv[optind++]);
56.         printf ("\n");
57.     }
58.     exit (0);
59. }
```

6.1.2. Взаимодействие Си-программы со средой окружения

Массив **envp[]**, передаваемый третьим аргументом в функцию *main*, содержит указатели на переменные среды окружения. Каждый элемент массива является указателем на строку вида: «имя_переменной = значение». Последним элементом массива является указатель NULL.

Стандарт ANSI C определяет только два первых параметра функции *main* – *argc* и *argv*. Стандарт POSIX.1 определяет также параметр *envp*, хотя рекомендует передачу окружения программы производить через глобальную переменную **environ**:

```
extern char ** environ;
```

Формат массива *environ* такой же, как и у *env*. Пример вывода переменных среды окружения с использованием двух массивов приведён в листинге 11.

Листинг 11. Вывод информации о командной строке

```
1. #include <stddef.h>
2. #include <stdio.h>
3. extern char **environ;
4. int main(int argc, char *argv[]){
5.   int i;
6.   printf ("Запущена программа %s. ", argv[0]);
7.   printf ("Число параметров %d\n", argc-1);
8.   for (i = 1; i < argc; i ++)
9.     printf("argv[%d] = %s\n", i, argv[i]);
10.  for (i = 0; i < 8; i ++)
11.    if (environ[i] != NULL)
12.      printf("environ[%d] : %s\n",i, environ[i]);
13.}
```

Для доступа к переменным окружения по их именам используются функции: **getenv** и **putenv**:

```
#include <stdlib.h>

char *getenv(const char *name);
int putenv (const char *string);
```

Функция *getenv* возвращает значение переменной окружения с именем *name*, а *putenv* помещает переменную и её значение в окружение процесса. Параметр *string* функции *putenv* содержит строку вида *var_name = var_value*.

В качестве примера приведём программу, похожую по своей функциональности на предыдущую, выводящую выборочно значения переменных и устанавливающую новые значения по желанию пользователя.

Листинг 12. Работа с переменными среды окружения

```
1.  #include <stddef.h>
2.  #include <stdlib.h>
3.  #include <stdio.h>
4.
5.  int main(int argc, char *argv[]){
6.    char *term;
7.    char buf[200], var[200];
8.
9.    /* Проверим, определена ли переменная TERM */
10.   if ((term = getenv("TERM")) == NULL){
11.     /* Если переменная не определена, получим от *
12.      * пользователя её значение и поместим       *
13.      * переменную в окружение программы          */
14.     printf ("Переменная TERM не определена.\n");
15.     printf ("Введите её значение: ");
16.     gets(buf);
17.     sprintf(var, "TERM=%s",buf);
18.     putenv(var);
19.     printf("%s\n", var);
20.   }
21.   else {
22.    /* Если переменная TERM определена, предоставим  *
23.     * пользователю возможность изменить её значение,*
24.     * после чего поместим её в окружение процесса   */
25.     printf("TERM=%s. Change? [N]", term);
26.     gets(buf);
27.     if (buf[0] == 'Y' || buf[0] == 'y'){
28.       printf("TERM="); gets(buf);
29.       sprintf(var, "TERM=%s", buf); putenv(var);
30.       printf("new %s\n", var);
31.     }
32.   }
33. }
```

В строке 11 проверяется, определена ли переменная *TERM*. Если переменная *TERM* не определена, функция *getenv* возвращает NULL.

В строках 17 и 29 формируются аргументы для функции *putenv*, содержащие новые значения переменной *TERM*. Далее, в строках 18 и 29 соответсвенно, устанавливается новое значение для переменной *TERM*.

Введённое значение переменной будет действительно только для данного процесса и порождённых им процессов: если после завершения программы вывести значение *TERM*, то видно, что оно не изменилось.

6.2. Взаимодействие «родственных» процессов и нитей

6.2.1. Ожидание завершения дочерних процессов

При выполнении нескольких процессов в операционной системе, и особенно «родственных» процессов, часто возникает необходимость выполнить какое-либо действие всеми процессами одновременно. Примером может служить событие завершения программы. Например, родительский процесс должен перед завершением удалить все временные файлы, создаваемые в ходе работы программы. Очевидно, что если он завершится раньше, чем дочерний процесс закончит работать с временными файлами, то логика работы программы может быть существенно нарушена. Поэтому требуется, чтобы родительский процесс «дождался», пока завершится дочерний, и только потом удалил все файлы.

Одним из способов такой синхронизации является обработка родителем **сигнала SIGCHLD**, отправляемого ему при завершении дочернего процесса. Такой подход требует определённых настроек родительского процесса, в частности, определения, чем он будет заниматься, пока будет «ожидать» сигнал (активное и пассивное ожидание). Ещё одним недостатком использования сигнала SIGCHILD является тот факт, что, во-первых, он ничего не сообщает о статусе завершения процесса, а, во-вторых, при наличии нескольких потомков нельзя будет определить, какой именно из них завершил свою работу.

Последнего недостатка лишены **функции семейства wait**: *wait*, *waitid* и *waitpid*:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
int waitid(idtype_t idtype, id_t id,
           siginfo_t *infop, int options);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Первый вызов *wait* позволяет заблокировать выполнение процесса, пока кто-либо из его непосредственных потомков не прекратит существование. По сути, это аналог использования сигнала SIGCHILD и «сна» в процессе ожидания его появления. Отличие только в том, что статус завершённого

процесса возвращается в *stat_loc,* а его идентификатор – как значение функции. Переменную *stat_loc* можно проанализировать с помощью следующих макросов, приведённых в таблице 12.

Таблица 12. Макросы для анализа статуса завершения процесса

Макрос	**Значение**
WIFEXITED(status)	Возвращает значение «истина» (не нуль), если процесс завершился нормально (вернул 0).
WEXITSTATUS(status)	Если WIFEXITED(status) не равно нулю, определяет код возврата завершившегося процесса (аргумент функции exit).
WIFSIGNALLED(status)	Возвращает значение «истина», если процесс завершился по сигналу.
WTERMSIG(status)	Если WIFSIGNALLED(status) не равно нулю, определяет номер сигнала, вызвавшего завершение выполнения процесса.
WCOREDUMP(status)	Если WIFSIGNALLED(status) не равно нулю, макрос возвращает истину в случае создания файла *core*.

Пример использования вызова *wait* приведён в листинге 13.

Листинг 13. Использованием вызова *wait*

```
1.  cpid = fork();
2.  if(cpid != 0){
3.    cpid = wait (&status)
4.    if (WIFEXITED(status))
         printf ("Процесс %d завершился успешно\
                со значением", cpid, WEXITSTATUS(status));
5.  }
```

Системный вызов *waitid* позволяет указать, завершение какого процесса необходимо ожидать, для чего используются аргументы *idtype* и *id* (см. табл. 13).

Таблица 13. Значения аргумента *idtype*.

Значение аргумента	**Описание *idtype***
P_PID	Необходимо ожидать изменений в состоянии дочернего процесса с идентификатором, заданным в *id*.
P_PGID	Требуется ждать изменения в состоянии одного из дочерних процессов, принадлежащих к группе, указанной в *id*.
P_ALL	Означает, что ожидать нужно изменения состояния любого из дочерних процессов. При этом значение параметра *id* игнорируется.

Ожидаемые изменения состояния дочернего процесса задаются в параметре *options* путём объединения логическим ИЛИ следующих макросов:

Таблица 14. Макросы для задания ожидаемых изменений состояния процесса.

Макросы	**Что ожидается**
WEXITED	Завершение процесса.
WTRAPPED	Ловушки (англ. trap) или точки останова (англ. breakpoint) для трассируемых процессов.
WSTOPPED	Остановки процесса из-за получения сигнала.
WCONTINUED	Ожидать продолжения выполнения процесса после его остановки.

В дополнение к перечисленным макросам можно задать дополнительные параметры поведения функции *waitid*:

Таблица 15. Макросы, задающие дополнительные параметры поведения *waitid*.

Макрос	**Поведение функции *waitid***
WNOHANG	Сразу же завершить свою работу, если в системе отсутсвует процесс, изменение состояния которого требуется ожидать.
WNOWAIT	Предписывает получить статусную информацию, но не уничтожать её, оставив дочерний процесс в состоянии ожидания.

6.2.2. Ожидание завершения нити. Синхронизация выполнения нитей

Чтобы узнать код завершения нити (какое значение будет возвращено её оператором *return*) используется функция *pthread_join*, которая имеет следующий заголовок:

```
int pthread_join(pthread_t thread, void** value_ptr);
```

Эта функция аналогично вызову *waitpid* дожидается завершения нити с идентификатором *thread* и записывает возвращаемое ею значение в переменную, на которую указывает *value_ptr*.

Для организации взаимоисключения стандарт POSIX предлагает использовать так называемый **mutex** (от англ. mutual exclusion – взаимное исключение).

В программах *mutex* представляется переменными типа *pthread_mutex_t*. Прежде чем начать использовать объект типа *pthread_mutex_t*, необходимо провести его инициализацию. Это можно сделать при помощи функции:

```
#include <pthread.h>

int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
```

В случае если мы хотим создать *mutex* с атрибутами по умолчанию, мы можем воспользоваться макросом *PTHREAD_MUTEX_INITIALIZER*.

После того как мы проинициализировали *mutex*, мы можем «захватить» его при помощи одной из функций:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Первая из двух функций просто «захватывает» *mutex*, при этом, если на данный момент *mutex* уже «захвачен» другой нитью, эта функция дождётся его освобождения. Вторая же функция попытается «захватить» *mutex*, если он свободен, а если он окажется занят, то немедленно возвратит специальный код ошибки *EBUSY*. То есть *pthread_mutex_trylock* фактически является неблокирующим вариантом вызова *pthread_mutex_lock*.

При этом надо заметить, что нить, которая уже владеет *mutex*, не должна повторно пытаться захватить *mutex*, так как при этом либо будет возвращена ошибка, либо может произойти то, что в англоязычной литературе называется *self-deadlock* (самотупиковая ситуация). В этом случае нить будет ждать освобождения *mutex* до тех пор, пока сама не освободит его, то есть фактически до бесконечности.

Для освобождения захваченного *mutex*'а предназначена функция:

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Стоит ещё раз подчеркнуть, что нить может освобождать только тот *mutex*, который был ранее захвачен ею. Результат работы функции *pthread_mutex_unlock* при попытке освободить захваченный другой нитью или вообще свободный *mutex* не определён. Другими словами вызов этой функции корректен, если данная нить перед этим успешно выполнила либо *pthread_mutex_lock*, либо *pthread_mutex_trylock* и не успела выполнить комплиментарный им *pthread_mutex_unlock*.

Мы обязаны рано или поздно (но в любом случае до повторного использования памяти, в которой располагается объект типа *pthread_mutex_t*) уничтожить *mutex*, освободив связанные с ним ресурсы. Сделать это можно, вызвав функцию:

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

При этом на момент вызова этой операции уничтожаемый *mutex* должен быть свободен, то есть не захвачен ни одной из нитей, в том числе вызывающей данную операцию. В случае если мы инициализировали *mutex* при помощи макроса *PTHREAD_MUTEX_INITIALIZER*, то необходимость в его явном уничтожении отсутствует.

6.2.3. Неименованные (программные) каналы

Вспомните синтаксис организации программных каналов при работе в командной строке *shell*:

```
$cat myfile | sort
```

При этом (стандартный) вывод программы *cat*, которая выводит содержимое файла *myfile*, передаётся на (стандартный) ввод программы *sort*, которая, в свою очередь, отсортирует предложенный материал.

Таким образом, два процесса обменялись данными. При этом использовался программный канал, обеспечивающий однонаправленную передачу данных между двумя задачами.

Для создания канала в программе используется системный вызов **pipe**.

```
#include <unistd.h>

int pipe (int *filedes);
```

Этот вызов возвращает два файловых дескриптора: *filedes*[1] для записи в канал и *filedes*[0] для чтения из канала. Теперь, если один процесс записывает данные в *filedes*[1], то другой сможет получить эти данные из *filedes*[0]. Вопрос только в том, как другой процесс сможет получить сам файловый дескриптор *filedes*[1]? Вспомним наследуемые атрибуты при создании процесса. Дочерний процесс наследует и разделяет все назначенные файловые дескрипторы родительского. То есть доступ к дескрипторам *filedes* канала может получить сам процесс, вызвавший *pipe*, и его дочерние процессы. В этом заключается серьёзный недостаток каналов, поскольку они могут быть использованы для передачи данных только между родственными процессами. Каналы не могут использоваться в качестве средства межпроцессного взаимодействия между независимыми процессами.

Хотя в приведённом примере может показаться, что процессы *cat* и *sort* независимы, на самом деле оба этих процесса создаются процессом *shell* и являются родственными.

6.3. Взаимодействие «неродственных» процессов

6.3.1. Идентификаторы ресурсов в межпроцессном взаимодействии

Средства межпроцессного взаимодействия первого типа требуют наличия средств идентификации совместно используемых объектов. Множество

возможных имён объектов конкретного типа межпроцессного взаимодействия называется **пространством имён** (англ. name space). Имена являются важным компонентом системы межпроцессного взаимодействия (англ. InterProcess Communications, IPC) для всех объектов, поскольку позволяют различным процессам получить доступ к общему объекту.

Имя для объектов IPC называется **ключом** (англ. key) и генерируется функцией **ftok**, исходя из: имени некоторого доступного для всех процессов-участников взаимодействий файла и односимвольного идентификатора:

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(char *filename, char proj);
```

В качестве первого параметра используется имя некоторого файла (полное или относительное), известное всем взаимодействующим процессам. Например, это может быть имя программы-сервера. Важно, чтобы этот файл существовал на момент создания ключа, при этом содержание файла значения не имеет. Также нежелательно использовать файл, который создаётся, изменяется и/или удаляется в процессе взаимодействия процессов.

Одному идентификатору (ключу) может соответсвовать несколько разнотипных общих ресурсов. Например, набор семафоров и разделяемая память.

Исключением из правила является использование «именованного канала», в котором общий файл непосредственно используется для организации взаимодействий.

6.3.2. Именованные каналы. FIFO

Название каналов FIFO происходит от выражения First In First Out (первый вошёл, первый вышел). FIFO очень похожи на программные каналы, поскольку являются однонаправленным средством передачи данных, причём чтение данных происходит в порядке их записи. Однако в отличие от программных каналов, FIFO имеют имена, которые позволяют независимым процессам получить к этим объектам доступ. Поэтому иногда FIFO также называют именованными каналами. FIFO являются средством UNIX System V и не используются в BSD. Впервые FIFO были представлены в System III, однако они до сих пор не документированы и поэтому мало используются. FIFO является отдельным типом файла в файловой системе UNIX (команда *ls -l* покажет символ «p» в первой позиции). Для создания FIFO используется системный вызов:

```
#include <sys/type.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

```
int mknod(char *pathname, mode_t mode, dev_t dev);
```

Где *pathname* – имя файла в файловой системе (имя FIFO), *mode* – флаги владения, прав доступа и т.д., *dev* при создании FIFO игнорируется. FIFO может быть создан и из командной строки *shell*:

```
$ mknod name p
```

После создания FIFO может быть открыт для записи и чтения, причём запись и чтение могут происходить в разных независимых процессах. Каналы FIFO и обычные каналы работают по следующим правилам:

1. При чтении меньшего числа байтов, чем находится в канале или FIFO, возвращается требуемое число байтов, остаток сохраняется для последующих чтений.
2. При чтении большего числа байтов, чем находится в канале или FIFO, возвращается доступное число байтов. Процесс, читающий из канала, должен соответствующим образом обработать ситуацию, когда прочитано меньше, чем заказано.
3. Если канал пуст и ни один процесс не открыл его на запись, при чтении из канала будет получено 0 байтов. Если один или более процессов открыли канал для записи, вызов *read* будет заблокирован до появления данных (если для канала или FIFO не установлен флаг отсутствия блокирования *O_NDELAY*).
4. Запись числа байтов, меньшего ёмкости канала или FIFO, гарантированно атомарно. Это означает, что в случае, когда несколько процессов одновременно записывают в канал, порции данных от этих процессов не перемешиваются.
5. При записи большего числа байтов, чем это позволяет канал или FIFO, вызов *write* блокируется до освобождения требуемого места. При этом атомарность операции не гарантируется. Если процесс пытается записать данные в канал, не открытый ни одним процессом на чтение, процессу генерируется сигнал SIGPIPE, а вызов *write* возвращает 0 с установкой ошибки (*errno=EPIPE*) (если процесс не установил обработку сигнала SIGPIPE, производится обработка по умолчанию, и процесс завершается).

В качестве примера приведём простейшую программу типа клиент-сервер, использующую FIFO для обмена данными. Следуя традиции, клиент посылает серверу сообщение «Здравствуй, Мир!», а сервер выводит это сообщение на терминал.

Листинг 14. Сервер

```
1.  #include <sys/types.h>
2.  #include <fcntl.h>
3.  #include <stdio.h>
```

Листинг 14. Сервер

```
4.  #include <sys/stat.h>
5.  #define FIFO "fifo.1"
6.  #define MAXBUFF 80
7.
8.  int main (void){
9.    int fd, n;
10.   char buff[MAXBUFF]; /*буфер для чтения данных */
11.   /*Создадим специальный файл FIFO */
12.   if (mknod(FIFO, S_IFIFO | 0666, 0) < 0){
13.     printf("Невозможно создать FIFO\n");
14.     exit(1);
15.   }
16.   /*Получим доступ к FIFO*/
17.   if ((fd = open(FIFO, O_RDONLY)) < 0){
18.     printf("Невозможно открыть FIFO\n");
19.     exit(1);
20.   }
21.   /*Прочитаем сообщение ("Здравствуй, Мир!") */
22.   /* и выведем его на экран                  */
23.   while ((n = read(fd, buff, MAXBUFF)) > 0)
24.     if (write(1, buff, n) != n){
25.       printf("Ошибка вывода\n");
26.       exit(1);
27.     }
28.   /* Закроем FIFO, и удалим файл */
29.   close(fd);
30.   if (unlink(FIFO) < 0){
31.     printf("Невозможно удалить FIFO\n"); exit(1);
32.   }
33.   exit(0);
34. }
```

Листинг 15. Клиент

```
1.  #include <sys/types.h>
2.  #include <sys/stat.h>
3.  #include <stdio.h>
4.  #include <fcntl.h>
5.
6.  /*Соглашение об имени FIFO*/
7.  #define FIFO "fifo.1"
8.
9.  int main (void){
10.   int fd, n;
11.   /*Получим доступ к FIFO*/
```

Листинг 15. Клиент

```
12.    if ((fd = open(FIFO, O_WRONLY)) < 0){
13.      printf("Невозможно открыть FIFO\n");
14.      exit(1);
15.    }
16.    /*Передадим сообщение серверу FIFO*/
17.    if (write(fd, "Здравствуй, Мир!\n\0", 18) != 18){
18.      printf("Ошибка записи\n"); exit(1);
19.    }
20.    close(fd);
21.    exit (0);
22. }
```

6.3.3. Сигналы

Аналогом программных прерываний в UNIX-подобных операционных системах (в том числе Linux) служат сигналы.

Сигнал – это способ взаимодействия программ, позволяющий сообщать о наступлении определённых событий, например, о появлении в очереди управляющих символов или возникновении ошибки во время работы программы (например, Segmentation Fault – выход за границы памяти).

Как и аппаратное прерывание, сигналы описываются номерами, которые описаны в заголовочном файле *signal.h*. Кроме цифрового кода, каждый сигнал имеет соответствующее символьное обозначение, например SIGINT.

Большинство типов сигналов предназначены для использования ядром операционной системы, хотя есть несколько сигналов, которые посылаются от процесса к процессу. Полный список доступных сигналов приведён в электронном справочнике *man* (*man 7 signal*). Приведём некоторые из них:

- SIGABRT – сигнал прерывания процесса (англ. process abort signal). Посылается процессу при вызове им функции *abort*. В результате сигнала SIGABRT произойдёт аварийное завершение (англ. abnormal termination) и запись образа памяти (англ. core dump, иногда переводится как «дамп памяти»). Образ памяти процесса сохраняется в файле на диске для изучения с помощью отладчика;
- SIGALRM – сигнал таймера (англ. alarm clock). Посылается процессу ядром при срабатывании таймера. Каждый процесс может устанавливать не менее трёх таймеров. Первый из них измеряет прошедшее реальное время. Этот таймер устанавливается самим процессом при помощи системных вызовов *alarm* или *setitimer* (см. ниже);
- SIGILL – недопустимая команда процессора (англ. illegal instruction). Посылается операционной системой, если процесс пытается выполнить недопустимую машинную команду. Иногда этот сигнал может возникнуть из-за того, что программа каким-либо об-

разом повредила свой код. В результате сигнала SIGILL происходит аварийное завершение программы;

- SIGINT – сигнал прерывания программы (англ. interrupt). Посылается ядром всем процессам, связанным с терминалом, когда пользователь нажимает клавишу прерывания (т.е., другими словами, в потоке ввода появляется управляющий символ, соответствующий клавише прерывания). Примером клавиши прерывания может служить комбинация Ctrl+C. Это также обычный способ остановки выполняющейся программы;
- SIGKILL – сигнал уничтожения процесса (англ. kill). Это довольно специфический сигнал, который посылается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнал процесса. Иногда он также посылается системой (например, при завершении работы системы). Сигнал SIGKILL – один из двух сигналов, которые не могут игнорироваться или перехватываться (то есть обрабатываться при помощи определённой пользователем процедуры);
- SIGPROF – сигнал профилирующего таймера (англ. profiling time expired). Как было уже упомянуто для сигнала SIGALARM, любой процесс может установить не менее трёх таймеров. Второй из этих таймеров может использоваться для измерения времени выполнения процесса в пользовательском и системном режимах. Сигнал SIGPROF генерируется, когда истекает время, установленное в этом таймере, и поэтому может быть использован средством профилирования (планирования работы) программы;
- SIGQUIT – сигнал о выходе (англ. quit). Очень похож на сигнал SIGINT. Этот сигнал посылается ядром, когда пользователь нажимает клавишу выхода в используемом терминале. Значение клавиши выхода по умолчанию соответствует символу ASCII F6 или Ctrl+Q. В отличие от SIGINT, этот сигнал приводит к аварийному завершению и сбросу образа памяти;
- SIGSEGV – обращение к некорректному адресу памяти (англ. invalid memory reference). Сокращение SEGV в названии сигнала означает нарушение границ сегментов памяти (англ. segmentation violation). Сигнал генерируется, если процесс пытается обратиться к неверному адресу памяти. Получение сигнала SIGSEGV приводит к аварийному завершению процесса;
- SIGTERM – программный сигнал завершения (англ. software termination signal). Используется для завершения процесса. Программист может использовать этот сигнал для того, чтобы дать процессу время для «наведения порядка», прежде чем посылать ему сигнал SIGKILL. Команда *kill* по умолчанию посылает именно этот сигнал;
- SIGWINCH – сигнал, генерируемый драйвером терминала при изменении размеров окна;

- SIGUSR1 и SIGUSR2 – пользовательские сигналы (англ. user defined signals 1 and 2). Так же, как и сигнал SIGTERM, эти сигналы никогда не посылаются ядром и могут использоваться для любых целей по выбору пользователя.

При получении сигнала процесс может выполнить одно из трёх действий:

- выполнить действие по умолчанию. Обычно действие по умолчанию заключается в прекращении выполнения процесса. Для некоторых сигналов, например, для сигналов SIGUSR1 и SIGUSR2, действие по умолчанию заключается в игнорировании сигнала. Для других сигналов, например, для сигнала SIGSTOP, действие по умолчанию заключается в остановке процесса;
- игнорировать сигнал и продолжать выполнение. В больших программах неожиданно возникающие сигналы могут привести к проблемам. Например, нет смысла позволять программе останавливаться в результате случайного нажатия на клавишу прерывания, в то время как она производит обновление важной базы данных;
- выполнить определённое пользователем действие. Программист может задать собственный обработчик сигнала. Например, выполнить при выходе из программы операции по «наведению порядка» (такие как удаление рабочих файлов), что бы ни являлось причиной этого выхода.

Чтобы определить действие, которое необходимо выполнить при получении сигнала, используется системный вызов **signal**:

```
#include <signal.h>

typedef void (*sighandler_t) (int);

sighandler_t signal (int signum,
                     sighandler_t handler);
```

Вызов *signal* определяет действие программы при поступлении сигнала с номером *signum*. Действие может быть задано как адрес пользовательской функции (в таком случае в функцию в качестве аргумента передаётся номер полученного сигнала) или как макросы SIG_IGN (для игнорирования сигнала) и SIG_DFL (для использования обработчика по умолчанию).

Если действие определено как пользовательская функция, то при поступлении сигнала программа будет прервана, и процессор начнёт выполнять указанную функцию. После её завершения выполнение программы, получившей сигнал, будет продолжено, и обработчик сигнала будет установлен с помощью SIG_DFL.

Чтобы вызвать сигнал, используется системный вызов **raise**:

```
#include <signal.h>

int raise (int signum);
```

Процессу посылается сигнал, определённый параметром *signum*, и в случае успеха функция *raise* возвращает нулевое значение.

Чтобы приостановить выполнение программы до тех пор, пока не придёт хотя бы один сигнал, используется вызов **pause**:

```
#include <signal.h>

int pause (void);
```

Вызов *pause* приостанавливает выполнение вызывающего процесса до получения любого сигнала. Если сигнал вызывает нормальное завершение процесса или игнорируется процессом, то в результате вызова *pause* будет просто выполнено соответствующее действие (завершение работы или игнорирование сигнала). Если же сигнал перехватывается, то после завершения соответствующего обработчика прерывания вызов *pause* вернёт значение –1 и поместит в переменную *errno* значение *EINTR*.

Пример программы, обрабатывающей сигналы, приведён в листинге 16.

Листинг 16. Обработка сигналов

```
1.  #include <stdio.h>
2.  #include <signal.h>
3.
4.  /* Функция-обработчик сигнала */
5.  void sighandler (int signo){
6.    printf (“Получен сигнал !!! Ура !!!\n”);
7.  }
8.  /* Основная функция программы */
9.  int main (void){
10.   int x = 0;
11.
12.   /* Регистрируем обработчик сигнала */
13.   signal (SGUSR1, sghandler);
14.   do {
15.     printf (“Введите X = ”); scanf (“%d”, &x);
16.     if (x & 0x0A) raise (SIGUSR1);
17.   } while (x != 99);
```

6.3.4. Сообщения

Данная возможность позволяет процессам обмениваться структурированными данными, имеющими следующие атрибуты:

– тип сообщения (позволяет мультиплексировать сообщения в одной очереди);
– длина данных сообщения в байтах (может быть нулевой);
– собственно данные (если длина ненулевая, могут быть структурированными).

Очередь сообщений хранится в виде внутреннего однонаправленного связанного списка в адресном пространстве ядра. Для каждой очереди ядро создаёт заголовок очереди (*msgid_ds*), где содержится информация о правах доступа к очереди (*msg_perm*), её текущем состоянии (*msg_cbytes* – число байтов и *msg_qnum* – число сообщений в очереди), а также указатели на первое (*msg_first*) и последнее (*msg_last*) сообщения, хранящиеся в виде связного списка. Каждый элемент этого списка является отдельным сообщением. Для создания новой очереди сообщений или для доступа к существующей используется системный вызов **msgget**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key_t key, int msgflag);
```

Функция возвращает дескриптор объекта-очереди или –1 в случае ошибки. Подобно файловому дескриптору, этот идентификатор используется процессом для работы с очередью сообщений. В частности, процесс может:
– помещать в очередь сообщения с помощью функции **msgsnd**;
– получать сообщения определённого типа из очереди с помощью функции **msgrcv**;
– управлять сообщениями с помощью функции **msgctl**.

Перечисленные системные вызовы манипулирования сообщениями имеют следующий вид:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msgid, const void *msgp,
            size_t msgsz, int msgfig);
int msgrcv (int msgid, void *msgp, size t msgsz,
            long msgtyp, int msgfig);
int msgctl (int msgid, int cmd,
            struct msgid_ds * buf);
```

Здесь *msgid* является дескриптором объекта, полученного в результате вызова *msgget*. Параметр *msgp* указывает на буфер, содержащий тип сообщения и его данные, размер которого равен *msgsz* байт. Буфер имеет следующие поля:

```
long msgtype;      /*тип сообщения*/
char msgtext []; /*данные сообщения*/
```

Параметр *msgtyp* указывает на тип сообщения и используется для их выборочного получения. Если *msgtyp* равен 0, функция *msgrcv* получит первое сообщение из очереди. Если величина *msgtyp* больше 0, будет получено первое сообщение указанного типа. Если *msgtyp* меньше 0, функция *msgrcv* получит сообщение с минимальным значением типа, меньшим или равным абсолютному значению *msgtyp*.

Очереди сообщений обладают весьма полезным свойством – в одной очереди можно мультиплексировать сообщения от различных процессов. Для демультиплексирования используется атрибут *msgtype*, на основании которого любой процесс может фильтровать сообщения с помощью функции *msgrcv*, как это было описано выше.

Функция **msgctl** выполняет контрольную операцию, заданную в *cmd*, над очередью сообщений *msgid*. Возможные значения *cmd*:

IPC_STAT	Скопировать информацию из структуры данных очереди сообщений в структуру с адресом *buf* (причём, у пользователя должны быть права на чтение очереди сообщений).
IPC_SET	Записать значения некоторых элементов структуры *msgid_ds*, адрес которой указан в *buf*, в структуру данных из очереди сообщений, обновляя при этом её поле *msg_ctime*. Предполагаемые элементы заданной пользователем структуры *msgid_ds*, адрес которой указан в *buf*, являются следующими: *msg_perm.uid*, *msg_perm.gid*, *msg_perm.mode*, *msg_qbytes*. Эффективный идентификатор пользователя процесса, вызывающего эту функцию, должен принадлежать либо *root*, либо создателю или владельцу очереди сообщений. Только суперпользователь может устанавливать значение *msg_qbytes* большим, чем *MSGMNB*.
IPC_RMID	Немедленно удалить очередь сообщений и структуры её данных, «разбудив» все процессы, ожидающие записи или чтения этой очереди (при этом функция возвращает ошибку, а переменная *errno* приобретает значение *EIDRM*). Эффективный идентификатор пользователя процесса, вызывающего эту функцию, должен принадлежать либо *root*, либо создателю или владельцу очереди сообщений.

Рассмотрим типичную ситуацию взаимодействия процессов, когда серверный процесс обменивается данными с несколькими клиентами. Свойство мультиплексирования позволяет использовать для такого обмена одну очередь сообщений. Для этого сообщениям, направляемым от любого из клиентов серверу, будем присваивать значение типа, скажем, равным 1. Если в теле сообщения клиент каким-либо образом идентифицирует себя (например,

передаёт свой PID), то сервер сможет передать сообщение конкретному клиенту, устанавливая тип сообщения, равным этому идентификатору.

Поскольку функция *msgrcv* позволяет принимать сообщения определённого типа (типов), сервер будет принимать сообщения с типом 1, а клиенты – сообщения с типами, равными идентификаторам их процессов. Схема такого взаимодействия представлена на рисунке 8.

Атрибут *msgtype* также можно использовать для изменения порядка извлечения сообщений из очереди. Стандартный порядок получения сообщений аналогичен принципу FIFO – сообщения получаются в порядке их записи. Однако используя тип, например, для назначения приоритета сообщений, этот порядок легко изменить.

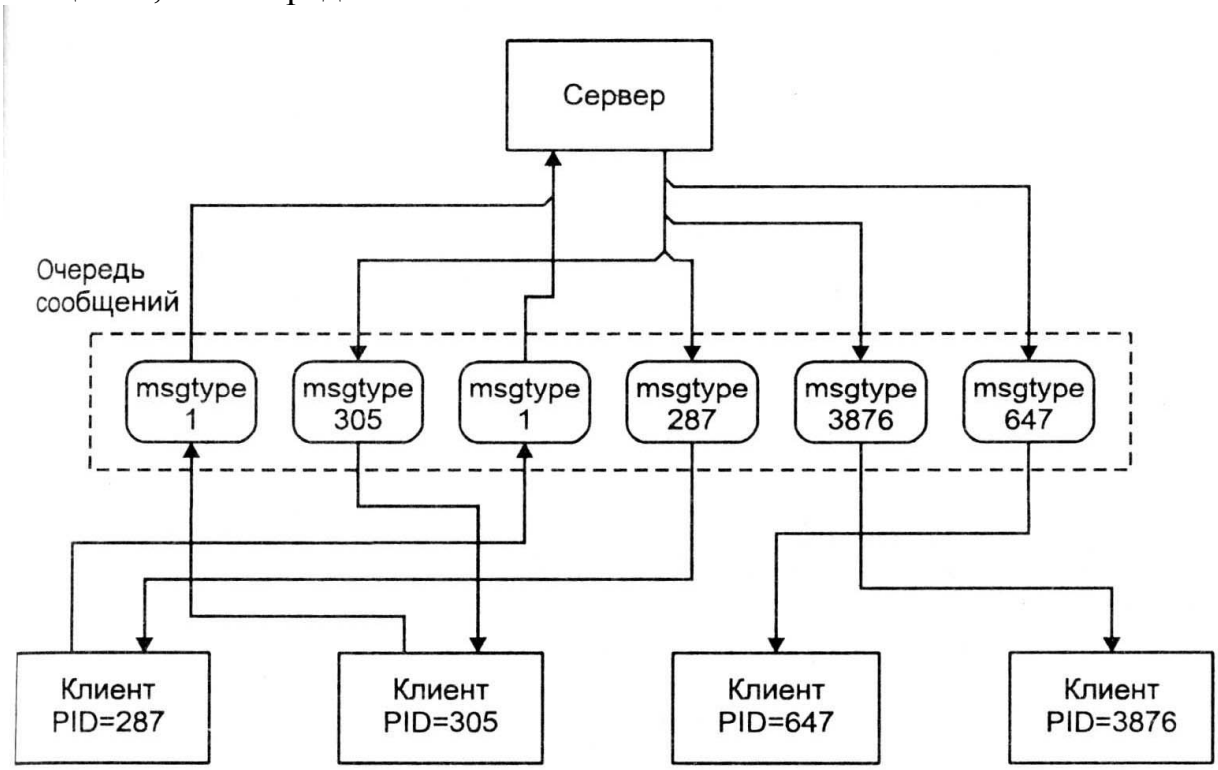

Рисунок 8. Пример работы очереди сообщений

Пример программы «Здравствуй, Мир!», использующей сообщения, приведён в листингах 17–19.

Листинг 17. Файл заголовков mesg.h

```
1.  #define MAXBUFF 80
2.  #define PERM    0666
3.  /*Определим структуру нашего сообщения. */
4.  typedef struct our_msgbuf {
5.    long mtype;
6.    char buff[MAXBUFF] ;
7.  } Message;
```

Листинг 18. Программа, реализующая сервер

```
1.  #include <sys/types.h>
2.  #include <sys/ipc.h>
3.  #include <sys/msg.h>
4.  #include <stdlib.h>
5.  #include <stdio.h>
6.  #include "mesg.h"
7.  int main (void){
8.    Message message;  /* Буфер для передачи сообщения */
9.    key_t   key;
10.   int     msgid, length, n;
11.
12.   if ((key = ftok("server", 'A')) < 0){
13.     printf("Невозможно получить ключ\n"); exit(1);
14.   }
15.   message.mtype=1L; /*Тип принимаемых сообщений*/
16.   /*Создадим очередь сообщений*/
17.   if ((msgid = msgget(key, PERM | IPC_CREAT)) < 0){
18.     printf("Невозможно создать очередь\n"); exit(1);
19.   }
20.   n = msgrcv(msgid, &message, sizeof(message),
21.              message.mtype, 0);
22.   if (n > 0){
23.     if (write(1, message.buff, n) != n){
24.       printf("Ошибка вывода\n"); exit(1);
25.     }
26.   }
27.   else{
28.     printf("Ошибка чтения сообщения\n"); exit(1);
29.   }
30.   /*Удалить очередь поручим клиенту*/
31.   exit(0);
32. }
```

Листинг 19. Программа, реализующая клиент

```
1.  #include <sys/types.h>
2.  #include <sys/ipc.h>
3.  #include <sys/msg.h>
4.  #include <stdlib.h>
5.  #include <stdio.h>
6.  #include "mesg.h"
7.
8.  int main(void){
9.    Message    message;
10.   key_t      key;
```

Листинг 19. Программа, реализующая клиент

```
11.   int msgid, length;
12.   message.mtype = 1L; /*Тип посылаемого сообщения */
13.
14.   /*Получим ключ*/
15.   if ((key = ftok("server", 'A')) < 0){
16.     printf("Невозможно получить ключ\n"); exit(1);
17.   }
18.   /* Получим доступ к очереди сообщений, очередь уже *
       * должна быть создана сервером                    */
19.   if ((msgid = msgget(key, 0)) < 0){
20.     printf("Невозможно получить доступ к очереди\n");
        exit(1);
21.   }
22.   /*Поместим строку в сообщение*/
23.   if ((length = sprintf(message.buff,
                            "Здравствуй, Мир!\n")) < 0){
24.     printf("Ошибка копирования в буфер\n"); exit(1);
25.   }
26.   /*Передадим сообщение*/
27.   if (msgsnd(msgid, (void *)&message, length, 0) !=0){
28.     printf("Ошибка записи сообщения в очередь\n");
29.     exit(1);
30.   } /*Удалим очередь сообщений*/
31.   if (msgctl(msgid, IPC_RMID, 0) < 0){
32.     printf("Ошибка удаления очереди\n"); exit(1);
33.   }
34.   exit(0);
35. }
```

6.3.5. Семафоры

Для синхронизации процессов, а точнее для синхронизации доступа нескольких процессов к разделяемым ресурсам, используются **семафоры**. Являясь одной из форм межпроцессного взаимодействия (IPC), семафоры не предназначены для обмена большими объёмами данных, как в случае FIFO или очередей сообщений. Вместо этого, они выполняют функцию, полностью соответствующую своему названию – разрешать или запрещать процессу использование того или иного разделяемого ресурса.

Применение семафоров поясним на простом примере. Допустим, имеется некий разделяемый ресурс (например, файл). Необходимо блокировать доступ к ресурсу для других процессов, когда некий процесс производит операцию над ресурсом (например, записывает в файл). Для этого свяжем с данным ресурсом некую целочисленную величину – счётчик, доступный для всех процессов. Примем, что значение 1 счётчика означает доступность ресурса, 0 – его недоступность. Тогда перед началом работы с ресурсом процесс должен

проверить значение счётчика. Если оно равно 0, ресурс занят и операция недопустима – процессу остаётся ждать.

Если значение счётчика равно 1, можно работать с ресурсом. Для этого, прежде всего, необходимо заблокировать ресурс, т.е. изменить значение счётчика на 0. После выполнения операции для освобождения ресурса значение счётчика необходимо изменить на 1. В приведённом примере счётчик играет роль семафора.

Для нормальной работы необходимо обеспечить выполнение следующих условий:

1) Значение семафора должно быть доступно различным процессам. Поэтому семафор находится не в адресном пространстве процесса, а в адресном пространстве ядра.
2) Операции проверки и изменения значения семафора должны быть реализованы в виде одной атомарной (т.е. непрерываемой другими процессами) по отношению к другим процессам операции. В противном случае возможна ситуация, когда после проверки значения семафора выполнение процесса будет прервано другим процессом, который в свою очередь проверит семафор и изменит его значение.

Единственным способом гарантировать атомарность критических участков операций является выполнение этих операций в режиме ядра. Таким образом, семафоры являются системным ресурсом, действия над которым производятся через интерфейс системных вызовов.

Семафоры в System V обладают следующими характеристиками:

1) Семафор представляет собой не один счётчик, а группу, состоящую из нескольких счётчиков, объединённых общими признаками (например, дескриптором объекта, правами доступа и т.д.).
2) Каждое из этих чисел может принимать любое неотрицательное значение в пределах, определённых системой (а не только значения 0 и 1).

Для получения доступа к семафору (и для его создания, если он не существует) используется системный вызов **semget**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key_t key, int nsems, int semflag);
```

В случае успешного завершения операции функция возвращает дескриптор объекта, в случае неудачи – –1. Параметр *nsems* задаёт число семафоров в группе. В случае, когда мы не создаём, а лишь получаем доступ к существующему семафору, этот аргумент игнорируется. Параметр *semflag* определяет права доступа к семафору и флажки для его создания (*IPC_CREAT*, *IPC_EXCL*).

После получения дескриптора объекта процесс может производить операции над семафором, подобно тому, как после получения файлового дескриптора процесс может читать и записывать данные в файл. Для этого используется системный вызов **semop**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *semop,
          size_t nops);
```

В качестве второго аргумента функции передаётся указатель на структуру данных, определяющую операции, которые требуется произвести над семафором с дескриптором *semid*. Операций может быть несколько, и их число указывается в последнем аргументе *nops*. Важно, что ядро обеспечивает атомарность выполнения критических участков операций (например, проверка значения – изменение значения) по отношению к другим процессам.

Каждый элемент набора операций *semop* имеет вид:

```
struct sembuf {
short sem_num; /*номер семафора в группе*/
short sem_op;  /*операция*/
short sem_fig; /*флаги операции*/
};
```

UNIX допускает три возможные операции над семафором, определяемые полем *sem_op*:

1) Если величина *sem_op* положительна, то текущее значение семафора увеличивается на эту величину.
2) Если значение *sem_op* равно нулю, процесс ожидает, пока семафор не обнулится.
3) Если величина *sem_op* отрицательна, процесс ожидает, пока значение семафора не станет большим или равным абсолютной величине *sem_op*. Затем абсолютная величина *sem_op* вычитается из значения семафора.

Можно заметить, что первая операция изменяет значение семафора (безусловное выполнение), вторая операция только проверяет его значение (условное выполнение), а третья проверяет, а затем изменяет значение семафора (условное выполнение).

При работе с семафорами взаимодействующие процессы должны «договориться» об их использовании и кооперативно проводить операции над семафорами. Операционная система не накладывает ограничений на использование семафоров. В частности, процессы вольны решать, какое значение семафора является разрешающим, на какую величину изменяется значение семафора и т.п.

Таким образом, при работе с семафорами процессы используют различные комбинации из трёх операций, определённых системой, по-своему трактуя значения семафоров.

В качестве примера рассмотрим два случая использования бинарного семафора (т.е., значения которого могут быть равны только 0 и 1). В первом примере значение 0 является разрешающим, а 1 «запирает» некоторый разделяемый ресурс (файл, разделяемая память и т.п.), ассоциированный с семафором. Определим операции, «запирающие» ресурс и освобождающие его:

```
static struct sembuf sop_lock[2] = {
   0, 0, 0,    /*ожидать обнуления семафора*/
   0, 1, 0     /*затем увеличить значение семафора на
                 1*/
};

static struct sembuf sop_unlock [1] = {
   0, -1, 0      /*обнулить значение семафора*/
};
```

Итак, для «запирания» ресурса процесс производит вызов:

```
semop(semid, &sop_lock[0], 2);
```

обеспечивающий атомарное выполнение двух операций. Ядро обеспечивает атомарное выполнение не всего набора операций в целом, а лишь критических участков. Так, например, в процессе ожидания освобождения ресурса (ожидание нулевого значения семафора) выполнение процесса будет (и должно быть) прервано процессом, который освободит ресурс (т.е. установит значение семафора, равным 1). Ожидание семафора соответствует состоянию «сна» процесса, допускающим выполнение других процессов в системе. В противном случае, процесс, ожидающий ресурс, остался бы заблокированным навсегда:

1) Ожидание доступности ресурса. В случае если ресурс уже занят (значение семафора равно 1), выполнение процесса будет приостановлено до освобождения ресурса (значение семафора равно 0).
2) «Запирание» ресурса. Значение семафора устанавливается равным 1.

Для освобождения ресурса процесс должен произвести вызов:

```
semop(semid, &sop_unlock[0], 1);
```

который уменьшит текущее значение семафора (равное 1) на 1, и оно станет равным 0, что соответствует освобождению ресурса. Если какой-либо из процессов ожидает ресурс (т.е. произвёл вызов операции *sop_lock*), он будет «разбужен» системой и сможет в свою очередь запереть ресурс и работать с ним.

Во втором примере изменим трактовку значений семафора: значению 1 семафора соответствует доступность некоторого ассоциированного с семафором ресурса, а нулевому значению – его недоступность. В этом случае содержание операций несколько изменится:

```
static struct sembuf sop_lock[2] = {
   0, -1, 0,    /*ожидать разрешающего сигнала (1),
                  затем обнулить семафор*/
};

static struct sembuf sop_unlock [1] = {
   0, 1, 0      /*увеличить значение семафора на 1*/
};
```

Процесс запирает ресурс вызовом:

```
semop(semid, &sop_lock[0], 1);
```

а освобождает:

```
semop(semid, &sop_unlock[0], 1);
```

Во втором случае операции получились проще (по крайней мере, их код стал компактнее), однако этот подход имеет потенциальную опасность: при создании семафора, его значения устанавливаются равными 0, и он сразу же «запирает» ресурс. Для преодоления данной ситуации процесс, первым создавший семафор, должен вызвать операцию *sop_unlock*, однако в этом случае процесс инициализации семафора перестанет быть атомарным и может быть прерван другим процессом, который, в свою очередь, изменит значение семафора. В итоге, значение семафора станет равным 2, что повредит нормальной работе с разделяемым ресурсом.

Можно предложить следующее решение данной проблемы:

Листинг 20. Работа с семафорами

```
1.  /* Создаём семафор */
2.  if ((semid = semget(key, nsems,
                  perms | IPC_CREAT | IPC_EXCL)) < 0){
3.    if (errno == EEXIST){
4.     if ((shmid = semget (key, nsems, perms)) < 0)
5.        return (-1);
6.    }
7.    else return (-1);
8.  }/*Если семафор создан нами, проинициализируем его*/
9.  else
```

```
10.   semop(semid,&sop_unlock[0], 1);
```

6.3.6. Разделяемая память

Интенсивный обмен данными между процессами с использованием рассмотренных механизмов межпроцессного взаимодействия (каналы, FIFO, очереди сообщений) может вызвать падение производительности системы. Это, в первую очередь, связано с тем, что данные, передаваемые с помощью этих объектов, копируются из буфера передающего процесса в буфер ядра и затем в буфер принимающего процесса.

Механизм разделяемой памяти позволяет избавиться от накладных расходов передачи данных через ядро, предоставляя двум или более процессам возможность непосредственного получения доступа к одной области памяти для обмена данными.

Безусловно, процессы должны предварительно «договориться» о правилах использования разделяемой памяти. Например, пока один из процессов производит запись данных в разделяемую память, другие процессы должны воздержаться от работы с ней. К счастью, задача кооперативного использования разделяемой памяти, заключающаяся в синхронизации выполнения процессов, легко решается с помощью семафоров.

Примерный сценарий работы с разделяемой памятью выглядит следующим образом:

1) Сервер получает доступ к разделяемой памяти, используя семафор.
2) Сервер производит запись данных в разделяемую память.
3) После завершения записи сервер освобождает разделяемую память с помощью семафора.
4) Клиент получает доступ к разделяемой памяти, «запирая» ресурс с помощью семафора.
5) Клиент производит чтение данных из разделяемой памяти и освобождает её, используя семафор.

Для создания или для доступа к уже существующей разделяемой памяти используется системный вызов **shmget**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflag);
```

Функция возвращает дескриптор разделяемой памяти в случае успеха, и –1 в случае неудачи. Аргумент *size* определяет размер создаваемой области памяти в байтах. Значения аргумента *shmflag* задают права доступа к объекту и специальные флаги *IPC_CREAT* и *IPC_EXCL*. Заметим, что вызов *shmget* лишь создаёт или обеспечивает доступ к разделяемой памяти, но не позволяет

работать с ней. Для работы с разделяемой памятью (чтение и запись) необходимо сначала присоединить (англ. attach) область вызовом **shmat**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat(int shmid, char *shmaddr, int shmflag);
```

Вызов *shmat* возвращает адрес начала области в адресном пространстве процесса размером *size*, заданным предшествующим вызовом *shmget*. В этом адресном пространстве взаимодействующие процессы могут размещать требуемые структуры данных для обмена информацией. Правила получения этого адреса следующие:

1) Если аргумент *shmaddr* нулевой, то система самостоятельно выбирает адрес.
2) Если аргумент *shmaddr* отличен от нуля, значение возвращаемого адреса зависит от наличия флажка *SHM_RND* в аргументе *shmflag*:
   – Если флажок *SHM_RND* не установлен, система присоединяет разделяемую память к указанному *shmaddr* адресу.
   – Если флажок *SHM_RND* установлен, система присоединяет разделяемую память к адресу, полученному округлением в меньшую сторону *shmaddr* до некоторой определённой величины *SHMLBA*.

По умолчанию разделяемая память присоединяется с правами на чтение и запись. Эти права можно изменить, указав флажок *SHM_RDONLY* в аргументе *shmflag*.Таким образом, несколько процессов могут отображать область разделяемой памяти в различные участки собственного виртуального адресного пространства. Окончив работу с разделяемой памятью, процесс отключает (англ. detach) область вызовом **shmdt**:

```
#includde <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(char *shmaddr);
```

При работе с разделяемой памятью необходимо синхронизировать выполнение взаимодействующих процессов: когда один из процессов записывает данные в разделяемую память, остальные процессы ожидают завершения операции. Обычно синхронизация обеспечивается с помощью семафоров, назначение и число которых определяется конкретным использованием разделяемой памяти. Можно привести примерную схему обмена данными между двумя процессами (клиентом и сервером) с использованием разделяемой памяти.

Для синхронизации процессов использована группа из двух семафоров. Первый семафор служит для блокирования доступа к разделяемой памяти, его разрешающий сигнал – 0, а 1 является запрещающим сигналом. Второй семафор служит для сигнализации серверу о том, что клиент начал работу. Необходимость применения второго семафора обусловлена следующими обстоятельствами: начальное состояние семафора, синхронизирующего работу с памятью, является открытым (0), и вызов сервером операции *mem_lock* заблокирует обращение к памяти для клиента. Таким образом, сервер должен вызвать операцию *mem_lock* только после того, как разделяемую память заблокирует клиент. Назначение второго семафора заключается в уведомлении сервера, что клиент начал работу, заблокировал разделяемую память и начал записывать данные в эту область. Теперь, при вызове сервером операции *mem_lock* его выполнение будет приостановлено до освобождения памяти клиентом, который делает это после окончания записи строки «Здравствуй, Мир!».

Листинг 21. Файл заголовков shmem.h

```
1.  #define MAXBUFF 80
2.  #define PERM    0666
3.  /*Структура данных разделяемой памяти*/
4.  typedef struct mem_msg{
5.    int   segment;
6.    char  buff[MAXBUFF];
7.  } Message;
8.  /*0жидание начала выполнения клиента*/
9.  static struct sembuf proc_wait[1] = { 1, -1, 0 };
10. /*Уведомление сервера о том, что клиент начал работу*/
11. static struct sembuf proc_start[1] = { 1, 1, 0 };
12. /*Блокирование разделяемой памяти*/
13. static struct sembuf  mem_lock[2] = { 0, 0, 0,
14.                                      0, 1, 0 };
15. /*Освобождение ресурса*/
16. static struct sembuf mem_unlock[1] = { 0, -1, 0 };
```

Листинг 22. Программа, реализующая сервер

```
1.  #include <sys/types.h>
2.  #include <sys/ipc.h>
3.  #include <sys/sem.h>
4.  #include <sys/shm.h>
5.  #include "shmem.h"
6.
7.  int main(void){
8.    Message * msgptr;
9.    key_t     key;
10.   int       shmid, semid;
```

Листинг 22. Программа, реализующая сервер

```
11.
12.   if ((key = ftok("server", 'A')) < 0){
13.     printf("Невозможно получить ключ\n"); exit(1);
14.   }
15.
16.   /*Создадим область разделяемой памяти*/
17.   if ((shmid = shmget(key, sizeof(Message),
            PERM | IPC_CREAT)) < 0){
18.     printf("Невозможно создать область\n"); exit(1);
19.   }
20.
21.   if ((msgptr = (Message *)shmat(shmid, 0, 0)) < 0){
22.     printf ("Ошибка присоединения\n"); exit(1);
23.   }
24.
25.   /*Создадим группу из двух семафоров */
26.
27.   if ((semid = semget(key, 2, PERM | IPC_CREAT)) < 0){
28.     printf("Невозможно создать семафор\n"); exit(1);
29.   }
30.
31.   /* Ждм, пока клиент начнет работу и *
       * заблокирует разделяемую память    */
32.   if (semop(semid, &proc_wait[0], 1) < 0){
33.     printf("Невозможно выполнить операцию\n");
        exit(1);
34.   }
35.
36.   /* Ждём, пока клиент закончит запись в *
       * разделяемую память и освободит её.  *
       * После этого заблокируем её          */
37.   if (semop(semid, &mem_lock[0], 2) < 0){
38.     printf("Невозможно выполнить операцию\n");
        exit(1);
39.   }
40.
41.   /* Выведем сообщение на терминал */
42.   printf("%s", msgptr->buff);
43.
44.   /* Освободим разделяемую память */
45.   if (semop(semid, &mem_unlock[0], 1) < 0){
46.     printf("Невозможно выполнить операцию\n");
        exit(1);
47.   }
```

Листинг 22. Программа, реализующая сервер

```
48.
49.   /*Отключимся от области*/
50.   if (shmdt(msgptr) < 0){
51.     printf("Ошибка отключения\n"); exit(1);
52.   }
53.
54.   /* Всю остальную работу по удалению объектов *
       * сделает клиент                              */
55.   exit(1);
56. }
```

Листинг 23. Программа, реализующая клиент

```
1.  #include <sys/types.h>
2.  #include <sys/ipc.h>
3.  #include <sys/sem.h>
4.  #include <sys/shm.h>
5.  #include "shmem.h"
6.  int main (void){
7.    Message * msgptr;
8.    key_t     key;
9.    int       shmid, semid;
10.   if ((key = ftok("server", 'A')) < 0){
11.     printf("Невозможно получить ключ\n"); exit(1);
12.   }
13.   if ((shmid = shmget(key, sizeof(Message), 0)) < 0){
14.     printf("Ошибка доступа\n"); exit(1);
15.   }
16.   if ((msgptr = (Message *) shmat(shmid, 0, 0)) < 0){
17.     printf("Ошибка присоединения\n");
18.     exit(1);
19.   }
20.   /* Получим доступ к семафору*/
21.   if ((semid = semget(key, 2, PERM)) < 0){
22.     printf("Ошибка доступа\n"); exit(1);
23.   } /*Заблокируем разделяемую память*/
24.   if (semop(semid, &mem_lock[0], 2) < 0){
25.     printf("Невозможно выполнить операцию\n");
        exit(1);
26.   } /*Уведомим сервер о начале работы*/
27.   if (semop(semid, &proc_start[0], 1) < 0){
28.     printf("Невозможно выполнить операцию\n");
        exit(1);
29.   }
30.   /*Запишем в разделяемую память сообщение*/
```

Листинг 23. Программа, реализующая клиент

```
31.   sprintf(msgptr->buff, "Здравствуй, Мир!\n");
32.   /*Освободим разделяемую память*/
33.   if (semop(semid, &mem_unlock[0], 1) < 0){
34.     printf("Невозможно выполнить операцию\n");
        exit(1);
35.   }
36.   /* Ждём, пока сервер в свою очередь не освободит *
       * разделяемую память                             */
37.   if (semop(semid, &mem_lock[0], 2) < 0){
38.     printf("Невозможно выполнить операцию\n");
        exit(1);
39.   }
40.   /*Отключимся от области*/
41.   if (shmdt(msgptr) < 0){
42.     printf("Ошибка отключения\n"); exit(1);
43.   }
44.   /*Удалим созданные объекты IPC*/
45.   if (shmctl(shmid, IPC_RMID, 0) < 0){
46.     printf("Невозможно удалить область\n"); exit(1);
47.   }
48.   if (semctl(semid, 0, IPC_RMID) < 0){
49.     printf("Невозможно удалить семафор\n"); exit(1);
50.   }
51.   exit(0);
52. }
```

Контрольные вопросы

1. Перечислите средства межпроцессного взаимодействия.
2. Как происходит передача аргументов исполняемой программе?
3. Какие функции используются для разпознавания опций? Чем отличается длинная опция от короткой?
4. Расскажите о взаимодействии исполняемой программы со средой окружения.
5. Для чего предназначен сигнал *SIGCHLD*? Почему важно, чтобы родительский процесс «дождался», пока завершится дочерний?
6. Расскажите о синхронизации выполнения нитей. Что такое *mutex*?
7. Расскажите о взаимодействии процессов через программные каналы.
8. Что такое FIFO? Как организовать взаимодействие процессов через именованный канал?
9. Что такое сигнал? Какие типы сигналов Вы знаете? Поясните каждый тип.
10. Для чего используется системный вызов *raise*?
11. Как организовать передачу сообщений между процессами?
12. Что такое семафор? Поясните применение семафоров на примере.

13. Как организовать обмен данными между процессами через разделяемую память? Какие функции для этого используют?

СПИСОК ЛИТЕРАТУРЫ

1. Таненбаум Э. Операционные системы: разработка и реализация. 3-е изд. / Э. Таненбаум, А. Вудхалл. – СПб.: Питер, 2007. – 704 с. – ISBN 978-5-469-01403-4.
2. Стивенс У. UNIX: взаимодействие процессов / У. Стивенс. – СПб.: Питер, 2003. – 576 с. – ISBN 5-318-00534-9.
3. Иванов Н.Н. Программирование в Linux. Самоучитель. / Н.Н. Иванов. – СПб.: БХВ-Петербург, 2007. – 416 с. – ISBN 978-5-9775-0071-5.
4. Немет Э. UNIX: руководство системного администратора / Э. Немет, Г. Снайдер, С. Сибасс, Т.Р. Хейн. – Киев: BHV, 2002. – 928 с. – ISBN 966-552-106-3.
5. Бэкон Д. Операционные системы / Д. Бэкон, Т. Харрис. – Киев: BHV, 2004. – 800 с. – ISBN 966-552-136-5.
6. Краковяк С. Основы организации и функционирования ОС ЭВМ: Пер. с франц. / С. Краковяк. – М.: Мир, 1988. – 480 с. – ISBN 5-03-00481-5.
7. Олифер В.Г. Сетевые операционные системы. 2-е изд. / В.Г. Олифер, Н.А. Олифер. – СПб.: Питер, 2009. – 672 с. – ISBN 978-5-91180-528-9.
8. Таненбаум Э. Современные операционные системы. 3-е изд. / Э. Таненбаум. – СПб.: Питер, 2010. – 1120 с. – ISBN 978-5-49807-306-4.
9. Столингс В. Операционные системы. 4-е изд.: Пер. с англ. / В. Столингс. – М.: Вильямс, 2004. – 848 с. – ISBN 5-8459-0310-6.
10. Таненбаум Э. Распределённые системы. Принципы и парадигмы / Э. Танненбаум, М. ван Стеен. – СПб.: Питер, 2003. – 880 с. – ISBN 5-272-00053-6.

ПРИЛОЖЕНИЕ 1
КОДЫ ОШИБОК

Код ошибки	Сообщение	Описание
E2BIG	Arg list too long	Размер списка аргументов, переданных системному вызову *exec*, плюс размер экспортируемых переменных окружения превышает ARGMAX байт.
EACCESS	Permission denied	Попытка доступа к файлу с недостаточными правами для данного класса (определяемого эффективными UID и GID процесса и соответствующими идентификаторами файла).
EAGAIN	Resource temporarily unavailable	Превышен предел использования некоторого ресурса, например, переполнена таблица процессов, или пользователь превысил ограничение по количеству процессов с одинаковым UID. Причиной также может являться недостаток памяти или превышение соответствующего ограничения.
EALREADY	Operation already in progress	Попытка операции с неблокируемым объектом, уже обслуживающим некоторую операцию.
EBADF	Bad file number	Попытка операции с файловым дескриптором, не адресующим никакой файл; также попытка операции чтения или записи с файловым дескриптором, полученным при открытии файла на запись или чтение, соответственно.
EBADFD	File descriptor in bad state	Файловый дескриптор не адресует открытый файл или попытка операции чтения с файловым дескриптором, полученным при открытии файла только на запись.
EBUSY	Device busy	Попытка монтирования устройства (файловой системы), которое уже примонтировано; попытка размонтировать файловую систему, имеющую открытые файлы; попытка обращения к недоступным ресурсам (семафоры, блокираторы и т.п.).
ECHILD	No child processes	Вызов функции *wait* процессом, не имеющим дочерних процессов или процессов, для которых уже был сделан вызов *wait*.
EDQUOT	Disk quota exceeded	Попытка записи в файл, создание каталога или файла при превышении квоты пользователя на дисковые блоки, попытка создания файла при превышении пользовательской квоты на число *inode*.
EEXIST	File exists	Имя существующего файла использовано в недопустимом контексте, например, сделана попытка создания символической ссылки с именем уже существующего файла.
EFAULT	Bad address	Аппаратная ошибка при попытке использования системой аргумента функции, например, в качестве указателя передан недопустимый адрес.
EFBIG	File too large	Размер файла превысил установленное ограничение RLIMIT_FSIZE или максимально допустимый размер для данной файловой системы.

EINPROGRESS	Operation now in progress	Попытка длительной операции (например, установление сетевого соединения) для неблокируемого объекта.
EINTR	Interrupted system call	Получение асинхронного сигнала, например, сигнала SIGINT или SIGQUIT, во время обработки системного вызова. Если выполнение процесса будет продолжено после обработки сигнала, прерванный системный вызов завершится с этой ошибкой.
EINVAL	Invalid argument	Передача неверного аргумента системному вызову. Например, размонтирование устройства (файловой системы), которое не было примонтировано. Другой пример – передача номера несуществующего сигнала системному вызову *kill*.
EIO	I/O error	Ошибка ввода-вывода физического устройства.
EISDIR	Is a directory	Попытка операции, недопустимой для каталога, например, запись в каталог с помощью вызова *write*.
ELOOP	Number of symbolic links encountered during path name traversal exceeds MAXSYMLINKS	При попытке трансляции имени файла было обнаружено недопустимо большое число символических ссылок, превышающее значение MAXSYMLINKS.
EMFILE	Too many open files	Число открытых файлов для процесса превысило максимальное значение OPENMAX.
ENAMETOOLONG	File name too long	Длина полного имени файла (включая путь) превысила максимальное значение PATHMAX.
ENFILE	File table overflow	Переполнение файловой таблицы.
ENODEV	No such device	Попытка недопустимой операции для устройства. Например, попытка чтения устройства только для записи или операция для несуществующего устройства.
ENOENT	No such file or directory	Файл с указанным именем не существует или отсутствует каталог, указанный в полном имени файла.
ENOEXEC	Exec format error	Попытка запуска на выполнение файла, который имеет права на выполнение, но не является файлом допустимого исполняемого формата.
ENOMEM	Not enough space	При попытке запуска программы (*exec*) или размещения памяти (*brk*) размер запрашиваемой памяти превысил максимально возможный в системе.
ENOMSG	No message of desired type	Попытка получения сообщения определённого типа, которого не существует в очереди.
ENOSPC	No space left on device	Попытка записи в файл или создания нового каталога при отсутствии свободного места на устройстве (в файловой системе).
ENOSR	Out of stream resources	Отсутствие очередей или головных модулей при попытке открытия устройства STREAMS. Это состояние является временным. После освобождения соответствующих ресурсов другими процессами операция может пройти успешно.
ENOSTR	Not a stream device	Попытка применения операции, определённой для устройств типа STREAMS (например, системного вызова *putmsg* или *gefmsg*), для устройства другого типа.
ENOTDIR	Not a directory	В операции, предусматривающей в качестве аргумента имя каталога, было указано имя файла другого ти-

		па (например, в пути для полного имени файла).
ENOTTY	Inappropriate ioctl for device	Попытка выполнения системного вызова *ioctl* для устройства, которое не является символьным.
EPERM	Not owner	Попытка модификации файла способом, разрешённым только владельцу и суперпользователю и запрещённым остальным пользователям. Попытка операции, разрешённой только суперпользователю.
EPIPE	Broken pipe	Попытка записи в канал (*pipe*), для которого не существует процесса, принимающего данные. В этой ситуации процессу обычно отправляется соответствующий сигнал. Ошибка возвращается при игнорировании сигнала.
EROFS	Read-only file system	Попытка модификации файла или каталога для устройства (файловой системы), примонтированного только для чтения.
ESRCH	No such process	Процесс с указанным PID не существует в системе.

ПРИЛОЖЕНИЕ 2
ЗАДАНИЯ НА ЛАБОРАТОРНЫЕ РАБОТЫ

Лабораторная работа № 1. Знакомство с операционной системой Linux. Способы хранения информации

Задание на лабораторную работу

1. Поменять пароль.
2. Используя команду **mount,** описать, как построена файловая система на Вашей машине.
3. Используя команды оболочки, создать в домашнем каталоге дерево каталогов согласно схеме, приведённой ниже: в домашнем каталоге – *cat1*, содержащий каталоги *cat2* и *cat3*. Каталог *cat1/cat2* содержит каталог *cat3*. Каталог *cat1/cat3* содержит каталог *cat4*. Каталог *cat1/cat2* содержит каталог *cat5*. Каталог *cat1/cat2/cat3* содержит *cat6* и *cat7*. Каталог *cat1/cat8* содержит символическую ссылку на каталог *cat1/cat2/cat3/cat6*. Каталог *cat1* содержит каталог *cat8*.
4. Нарисовать граф, соответствующий созданной файловой системе.
5. Удалить каталоги с дублирующимися именами.
6. Удалить неразрешённую ссылку *cat6*.
7. Написать маски файлов для списков приведённых ниже:

   file1, file5, file6, file8
   file, fail, from, fax
   asd, dfg, qwe, dsa, fkl, jss
   adks, aeks, awks, alks
8. Скопировать файлы из домашнего каталога, начинающиеся с символов *.b*, в каталог *cat1/cat8*.
9. Поменять права на скопированные файлы так, чтобы любой пользователь системы мог прочитать их содержимое, используя символьное представление прав доступа. Используя восьмеричное представление прав доступа, изменить права доступа к каталогу *cat5* так, чтобы доступ к нему имели только пользователи группы.
10. Вывести содержимое всех каталогов, начиная с самого верхнего для задания (использовать только одну команду и один раз).
11. Определить тип командной оболочки, используемой Вами.

Контрольные вопросы

1. Как проходит процедура идентификации?
2. Что такое регистрационное имя?
3. Что такое командная оболочка? Как можно определить её тип?
4. Файл */etc/passwd*. Зачем он нужен? Какова его структура?
5. Что такое файл?
6. Что такое файловая система? Как её можно описать? Команда *mount*.
7. Что такое каталог?

8. Что такое путь файла? Абсолютный и относительный путь?
9. Типы файлов, используемых в ОС Linux. Что такое метаданные?
10. Символьные и блочные устройства. Отличия и примеры?
11. Ссылки. Типы ссылок. Команда создания ссылки.
12. Файловая система UNIX. Назначение основных каталогов.
13. Команда определения текущего каталога.
14. Команда изменения текущего каталога.
15. Команда вывода содержимого каталога.
16. Электронный справочник *man*.
17. Генерация имён файлов. Символы «*», «?», [], цитирование.
18. Команда копирования файлов.
19. Команды удаления файлов и каталогов.
20. Команды создания и удаления каталогов.
21. Владельцы, группы и права.
22. Команды изменения прав. Символьные и восьмеричные представления прав доступа.
23. Изменение владельца и группы.

Лабораторная работа № 2. Командная оболочка bash

Задание на лабораторную работу

1. Определить тип используемой командной оболочки.
2. Вывести на экран значения всех переменных среды окружения. Проанализировать полученные результаты и объяснить значения известных вам переменных окружения.
3. Используя процедуру экспортирования, изменить приглашение командной строки так, чтобы в основном приглашении имя машины выводилось красным цветом, а в приглашении для второй строки выводился номер команды и символ «>».
4. Используя системную переменную HOME и список команд, выполнить следующие действия одной командой: перейти в домашний каталог, в случае удачного перехода вывести на экран содержимое файла */etc/passwd*.
5. Используя команды *printf* и *read*, вывести приглашение пользователю ввести команду. Если пользователь нажал Enter без ввода команды, сообщить ему об ошибке, в противном случае выполнить то, что он ввёл.
6. Оформить предыдущий пункт как скрипт и выполнить его.
7. Вывести значения всех переменных среды окружения в файл с именем *~/envs*.
8. Используя системную переменную HOME, список, каналы и перенаправление вывода, выполнить следующие действия одной командой: перейти в домашний каталог, выдать содержимое файла */etc/passwd*, отсортированное по имени пользователя в файл *passwd.orig*. Подсказка: команда сортировки – *sort*.

9. Используя перенаправление ввода с разделителем и перенаправление вывода, добавить в файл *passwd.orig* информацию о себе согласно формату записи файла */etc/passwd* (все поля должны быть обязательно заполнены).
10. Описать содержимое файла *~/.bash_profile* и всех файлов, которые он использует.
11. Написать скрипт, выполняющий следующие действия: вывести меню, содержащее все файлы с расширением *.c* текущего каталога. После выбора пользователем файла, скомпилировать его.
12. Написать скрипт, анализирующий параметры командной строки с использованием специальных переменных. Все параметры должны быть выданы на экран.
13. Написать скрипт, анализирующий параметры командной строки. Параметры должны быть следующие: *-d каталог*, *-f файл*, *-c*, *-r*. При анализе опций должны быть установлены переменные: DIR, FILE, COMPIL, RUN. После анализа опций выполнить следующие действия: если определена переменная DIR и такой каталог существует, то выдать его содержимое. Если определена переменная FILE и такой файл существует, то выдать его содержимое на экран. Если переменная не определена, то в качестве имени файла использовать *.bashrc*. Если определена переменная COMPIL и определена переменная FILE, то откомпилировать указанный файл. Если результат компиляции положительный, то, если определена переменная RUN, исполнить откомпилированный файл.
14. Используя цикл *for*, объединить все файлы с расширением *txt* в текущем каталоге в файл *~/textx.txt*. Для объединения использовать перенаправление потоков ввода-вывода.

Контрольные вопросы

1. Что такое командная оболочка?
2. Что такое команда? Формат команды?
3. Что означает символ «\», введённый в командной строке перед нажатием Enter?
4. Что такое скрипт?
5. Что такое среда окружения? Зачем она нужна?
6. Как задать значение переменной окружения и как вывести его на экран?
7. Переменная оболочки. Отличие от переменной окружения.
8. Как задать формат командной строки? Отличие PS1 от PS2.
9. Списки команд. Логические операции над командами.
10. Подстановка команд, переменных и арифметических выражений.
11. Команда *read*.
12. Зачем нужны файлы *.bash**. Почему они не имеют прав на исполнение?
13. Условный оператор *if-fi*. Команда *test*.
14. Блок *case-esac*.
15. Специальные переменные.
16. Цикл *select*.
17. Циклические конструкции.

18. Функции.
19. Команда *getopts*.
20. Понятие процесса. Типы процессов. Атрибуты процесса. Фоновое выполнение команд. Команды *jobs, fg, bg, ps, pstree*.
21. Каналы ввода-вывода. Перенаправление каналов.

Лабораторная работа № 3. Понятие процесса, группы процессов, сеансов. Фоновое и интерактивное выполнение задач

Задание на лабораторную работу

1. Используя команду *ps*, вывести информацию обо всех процессах системы и ответить на следующие вопросы: 1) Сколько процессов в системе? 2) Сколько процессов принадлежит Вам?
2. Используя форматный вывод команды *ps* нарисовать дерево процессов, принадлежащих всем Вашим сеансам. Определить, сколько сеансов и групп порождено Вами, сколько процессов входит в каждый сеанс и в каждую группу.
3. Используя команду *pstree*, вывести дерево процессов в файл *~/pstrees*. Проанализировать результат работы программы. Сравнить с результатами предыдущего пункта.
4. Используя команду *top*, описать наиболее активные процессы в системе.
5. Выполнить команду *ls -laR / | sort* и остановить её выполнение (Ctrl+Z). Выполнить команду *man ls* в фоновом режиме. Используя команду *jobs*, посмотреть состояние фоновых задач. Перевести задачу 2 в интерактивный режим и объяснить, почему выполнение указанной команды было автоматически приостановлено. Ответить на вопросы: 1) Будут ли указанные процессы (*ls* и *sort*) входить в одну группу или в разные? 2) Какие идентификаторы имеют эти группы?

Контрольные вопросы

1. Отличие процесса от программы или задачи.
2. Типы процессов.
3. Группы процессов? Зачем используются?
4. Сеансы. Зачем они используются?
5. Фоновое выполнение задач. Способы перевода задачи в фоновый режим.
6. Атрибуты процесса.
7. Выделение идентификатора процессу.
8. Что такое приоритет процесса? Зачем он нужен?
9. Что такое идентификатор родительского процесса? Зачем он используется?
10. Что такое терминальная линия? Какие процессы не используют терминальных линий?
11. Состояние процесса.

Лабораторная работа № 4. Программирование в ОС Linux. Основные этапы создания программ. Взаимодействие программ с командной оболочкой. Обработка исключительных ситуаций

Задание на лабораторную работу

Написать программу на языке Си, состоящую из двух файлов: заголовочного файла и файла программы.

В заголовочном файле должны быть подключены необходимые системные заголовочные файлы, и описаны все глобальные переменные и функции, используемые в файле программы.

Программа должна выполнять следующие действия:

1) вывести все переменные окружения на экран двумя различными способами (способ задаётся пользователем в командной строке опциями -1 и -2);
2) обработать все указанные опции командной строки и в случае ошибочных опций выдать сообщение в стандартный поток ошибок о неправильном использовании программы;
3) в командной строке опцией *-f файл* указывается имя файла, который необходимо открыть и вывести его содержимое на экран. Также необходимо обработать ошибки, которые могут возникнуть при открытии, закрытии и выводе файла на экран;
4) при завершении программы (даже в случае возникновения ошибок) на экран должно быть выдано сообщение об авторе программы: ФИО, учётное имя, группа.

Контрольные вопросы

1. Этапы создания программы с использованием языка высокого уровня.
2. Заголовочные файлы. Зачем используются, где находятся?
3. В чём отличие использования символов < > и " " в директиве препроцессора *#include*?
4. Зачем используются производные системные типы (имеющие окончание «t»)?
5. Этапы компиляции программ, написанных на языке Си.
6. Зачем нужны файлы, имеющие суффикс *.o*?
7. Зачем нужна программа-компоновщик *ld*?
8. Способы получения переменных окружения.
9. Обработка параметров командной строки без использования системных функций.
10. Функции *getopt* и *getopt_long*. Принципы их работы.
11. Способы завершения программ. Команды *exit*, *_exit*.
12. Обработчики завершения программ.
13. Функции обработки ошибок в программах.

Лабораторная работа № 5. Порождение, завершение, синхронизация процессов. Группы и сеансы

Задание на лабораторную работу

Написать программу на языке Си, реализующую следующие действия.

1. Вывод информации об идентификаторах процесса, группы, сеанса.
2. Вывод на экран строки-приглашения пользователю для ввода команды.
3. Разбор командной строки, введённой пользователем, и формирование массива *argv*.
4. Порождение процесса, в котором выполняется указанная команда с указанными параметрами. Необходимо предусмотреть возможность поиска указанной программы в путях PATH.
5. Обработка ошибки запуска указанной программы.
6. Вывод информации о статусе завершения порождённого процесса при его завершении.
7. Программа завершается если: пользователь ввёл команду *exit* или нажал Enter без ввода команды.
8. При завершении работы программы на экран должна быть выведена следующая информация: имя автора, группа и логин.

Контрольные вопросы

1. Что такое процесс? В чем его отличие от программы?
2. Атрибуты процесса.
3. Что такое PID, PPID и зачем они нужны? Как их можно получить?
4. Что такое GID, EGID, UID, EUID и зачем они нужны? Как их можно получить?
5. Порождение нового процесса.
6. Семейство вызовов *exec*. Отличия функций.
7. Отличие порождения процесса от выполнения программы.
8. Синхронизация выполнения процессов. Зачем она нужна?

Лабораторная работа № 6. Взаимодействие процессов. Каналы. FIFO

Задание на лабораторную работу

1. Написать программу, выполняющую следующие действия:
   - создание программного канала и порождение дочернего процесса;
   - основной процесс считывает содержимое файла */etc/passwd* и передаёт его в канал;
   - дочерний процесс получает информацию через канал и выводит эту информацию в файл с именем *~/passwds*.
   - передача завершается при поступлении в канал символа с кодом 26.
   - при получении информации дочерний процесс производит замену символов по схеме: b $\rightarrow$ n, i $\rightarrow$ a, n $\rightarrow$ b.
2. Написать программу, использующую именованные каналы.

1) Клиентская и серверная части должны иметь общий заголовочный файл, определяющий требуемые соглашения (имя FIFO, длины строк и т.д.).
2) Серверная часть создаёт именованный канал согласно соглашению и ждёт поступления от клиента строки формата: «FILENAME~имя_файла». Информация, поступившая до указанной строки, игнорируется. После поступления указанной строки сервер создаёт файл, имя которого было передано, и открывает его для записи. Далее, всё, что приходит от клиента, записывается в этот файл. Процедура взаимодействия заканчивается поступлением от клиента символа с кодом 26.
3) Клиент выводит приглашение пользователю для ввода имени файла, после чего формирует строку указанного формата и передаёт её серверу. Клиент передаёт любую информацию серверу, которая завершается символом с кодом 26.

Контрольные вопросы

1. Могут ли процессы получать непосредственный доступ к данным других процессов? Почему?
2. Программные каналы. Зачем нужны? Принцип функционирования.
3. Функция *pipe*.
4. Именованные каналы и их отличие от программных.
5. Функция *mknod*.

Лабораторная работа № 7. Многонитиевые программы

Задание на лабораторную работу

Разработать программу, реализующую модель работы склада, отвечающего за хранение и продажу некоторого товара (одного). Склад содержит N помещений, каждый из которых может хранить определённое количество единиц товара. Поступающий товар помещается в одно из помещений специальным погрузчиком. За товаром прибывает K покупателей, каждому из которых требуется по L_k единиц товара. Площадка перед складом мала, и на ней может в один момент времени находиться либо погрузчик, либо один из покупателей. Если покупателям требуется больше товара, чем имеется на складе, то они ждут новых поступлений, периодически проверяя склад. Время работы склада ограниченно.

1) Основная нить (функция *main*) выполняет следующие действия:
   - формирует начальное заполнение склада (для каждого помещения случайным образом выбирается число из диапазона от 1 до 40);
   - обрабатывает опции командной строки, в которой должно быть указано, сколько клиентов будет обслуживаться складом, и в течение какого времени должен работать склад;
   - порождает заданное количество нитей, каждая из которых реализует алгоритм работы покупателя. Каждому покупателю случайным образом назначается количество требуемых единиц продукции (число из диапазона от 1 до 1000);

– настраивает таймер (*alarm*) таким образом, чтобы он сработал по окончании времени работы склада;
– запускает алгоритм работы погрузчика;
– после срабатывания таймера принудительно завершает все выполняющиеся нити (если таковые имеются);
– завершает работу программы.

2) Алгоритм работы погрузчика:
– пытается попасть на площадку перед складом;
– как только попадёт на площадку, ищет хотя бы один склад, в котором нет продукции, и заполняет его максимально возможным образом;
– покидает площадку;
– «засыпает» на 5 секунд;
– цикл повторяется до срабатывания таймера.

3) Алгоритм работы покупателя:
– пытается попасть на площадку перед складом;
– как только попадёт на площадку, ищет хотя бы один склад, в котором есть продукция, и забирает либо столько, сколько надо, либо всю продукцию;
– покидает площадку;
– «засыпает» на 5 секунд;
– цикл повторяется до тех пор, пока покупателю нужна продукция.

Программа должна выводить на экран информацию о помещениях склада.

Контрольные вопросы

1. Что такое нить? Чем она отличается от процесса?
2. Каким образом можно создать нить и завершить её?
3. Как можно узнать статус завершения нити (возвращаемое значение)?
4. Могут ли нити использовать общие переменные? Приведите пример.
5. Что такое «мьютекс»? Каким образом он используется в лабораторной работе?

Лабораторная работа № 8. Взаимодействие процессов. Идентификация. Сообщения

Задание на лабораторную работу

Написать программу, реализующую механизм взаимодействия типа клиент-сервер, используя систему очередей сообщений. Программы должны иметь общий заголовочный файл, определяющий основные соглашения (идентификатор проекта, имя файла и т.п.) и выполнять следующие действия.

1) Клиент предлагает пользователю ввести символьную строку, которая потом будет отправлена серверу. Далее клиент ожидает ответа от сервера. Работа клиента прекращается в случае поступления от сервера строки, содержащей «bye».

2) Сервер получает строку от клиента и отвечает на неё следующим образом: если клиент прислал строку «hello», сервер отвечает «hi», если – «how_are_you», то – «fine», если – «bye», то – «bye». Работа сервера завершается в случае поступления от клиента строки, содержащей «bye». PID клиента сервер получает в сообщении с идентификатором 2.

Контрольные вопросы

1. Зачем нужна идентификация процессов? Что такое ключ?
2. Функция *ftok*. Какие файлы могут быть использованы в качестве параметров? Что такое идентификатор проекта?
3. Зачем нужны функции класса *get*?
4. Являются ли ключи локальными в рамках одного процесса, или они доступны и другим процессам?
5. Где хранятся ключи?
6. Кто создаёт и кто удаляет ключи?
7. Что такое сообщения? Зачем они нужны?
8. Имеют ли сообщения структуру?
9. Зачем нужен идентификатор сообщения, и какие значения он может принимать?
10. Как хранится очередь сообщений? Можно ли прочитать сообщения от процесса, которого уже нет в системе?
11. Функция *msgget*.
12. Функция *msgrcv*.
13. Функция *msgsnd*.
14. Функция *msgctl*.

Лабораторная работа № 9. Взаимодействие процессов. Семафоры. Разделяемая память

Задание на лабораторную работу

Описать три семафора и следующие операции с ними:

1) Семафор первый – информация о запуске и остановке клиентов. Значение семафора – количество запущенных клиентов. Операции: 1) подключение клиента (увеличение значения семафора на 1); 2) отключение клиента (уменьшение семафора на 1); 3) проверка отсутствия клиентов (проверка на 0 значения семафора).
2) Семафор второй – контроль доступа к памяти. Значения семафора: 0 – доступ разрешён, 1 – доступ заблокирован. Операции: 1) ожидание освобождения и блокирование памяти и 2) разблокирование памяти.
3) Семафор третий – контроль наличия сообщения. Значения семафора: 0 – сообщение отсутствует, 1 – сообщение присутствует. Операции: 1) занесение сообщения (установка значения в 1); 2) считывание

сообщения (ожидание появления сообщения и обнуление семафора); 3) ожидание освобождения сообщения (ожидание обнуления семафора).

Написать программу, состоящую из трёх файлов: заголовочного файла, файла, содержащего программу-сервер, и файла, содержащего программу-клиента. Заголовочный файл описывает общие для сервера и клиента параметры (семафоры и операции над ними, ключ, формат сообщения и т.д.).

Действия сервера:

1) создаёт область разделяемой памяти с размером, достаточным для передачи одного сообщения, создаёт три семафора;
2) подключается к разделяемой памяти и ожидает прихода сообщения;
3) при появлении сообщения блокирует память и обрабатывает его следующим образом: если от клиента пришло «hi», сервер отвечает «hello»; если от клиента пришло «how_are_you?» – «fine»; если от клиента пришло «bye» – «bye». Если строка от клиента не совпадает ни с одним из перечисленных вариантов, то клиенту передаётся фраза «i_don't_understand_you». После передачи сообщения память разблокирывается. Далее опять ожидает появления сообщения. Цикл прекращается после появления строки «bye», после чего сервер ждёт освобождения памяти, освобождения очереди сообщений, завершения работы всех клиентов и удаляет разделяемую память и семафоры.

Действия клиента:

1) Подключается к разделяемой памяти и области семафоров. Устанавливает значение семафора, сигнализирующего о запуске клиентов. Ожидает освобождения очереди сообщений, после чего блокирует память и предлагает пользователю ввести строку, из которой формируется сообщение и через разделяемую память передаётся серверу.
2) После ввода устанавливает семафор появления сообщения, и освобождает память. Ожидает появления сообщения от сервера. После чего блокирует память, считывает сообщение и выводит на экран, освобождает память.
3) Если от сервера пришла строка «bye», цикл передачи сообщений заканчивается. Перед завершением освобождается память, и уменьшается значение семафора, регулирующего запуск клиентов.

Контрольные вопросы

1. Что такое семафор и зачем он используются?
2. Можно ли передавать больший объём информации, используя семафоры?
3. Предусмотрены ли ограничения на использования счётчиков?
4. Где хранятся семафоры?
5. Что такое критическая секция?
6. Функция *semget*.
7. Типы операций над семафорами. Функция *semop*.
8. Разделяемая память. Принцип работы приложения, использующего разделяемую память.
9. Функции *shmget*, *shmdt*.

Сергей Николаевич Мамойленко
Ольга Владимировна Молдованова

Операционные системы

Учебное пособие
Часть 1. Операционная система Linux

Редактор: А.Ю.Поляков
Корректор: В.В.Сиделина

Подписано в печать 23.03.2012г.
Формат бумаги 60 х 84/16, отпечатано на ризографе, шрифт № 10,
изд. л. 8,0 заказ № 21, тираж – 100 экз., СибГУТИ.
630102, г. Новосибирск, ул. Кирова, д. 86