ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
Государственное образовательное учреждение высшего профессионального образования
«ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

А.В. Замятин, Д.В. Сидоров

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Лабораторный практикум

Издательство
Томского политехнического университета
2009

УДК 681.3.066(075)
Л 12

Замятин А.В., Сидоров Д.В.
Л12 Лабораторный практикум по курсу «Операционные системы»/ А.В. Замятин, Д.В. Сидоров. – Томск: Изд-во. Томского политехнического университета, 2009. – 128 с.

В лабораторном практикуме изложены теоретические основы, задания и другой справочный материал для выполнения лабораторных работ по курсу «Операционные системы». Материалы практикума позволяют получить базовые навыки по работе в операционных системах семейства Unix (Linux), по разработке программных проектов с использованием специализированных утилит, а также по управлению процессами и потоками и средствами их синхронизации.

Практикум подготовлен на кафедре вычислительной техники Томского политехнического университета и предназначен для студентов направления 230100 «Информатика и вычислительная техника».

УДК 681.3.066(075)

Материалы лабораторного практикума по курсу «Операционные системы» рекомендованы научно-методическим семинаром кафедры вычислительной техники АВТФ (протокол №1 от 04.09.2008 г.).

Рекомендовано к печати Редакционно-издательским советом
Томского политехнического университета

Рецензенты:

Заведующий кафедрой прикладной информатики, декан факультета информатики Томского государственного университета, доктор технических наук
С.П. Сущенко

Научный сотрудник Института высокопроизводительных компьютерных и сетевых технологий Санкт-Петербургского государственного университета аэрокосмического приборостроения, кандидат технических наук
А.В. Винель

© Замятин А.В., Сидоров Д.В., 2009
© Томский политехнический университет, 2009
© Оформление. Издательство Томского политехнического университета, 2009

СОДЕРЖАНИЕ

ВВЕДЕНИЕ..5

1 ЛАБОРАТОРНАЯ РАБОТА №1. ПРАКТИЧЕСКОЕ ЗНАКОМСТВО С ОПЕРАЦИОННОЙ СИСТЕМОЙ *UNIX* 6

1.1 ЦЕЛЬ РАБОТЫ 6
1.2 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ 6
1.3 ОСНОВЫ РАБОТЫ В ОПЕРАЦИОННОЙ СИСТЕМЕ UNIX.....................6
1.3.1 Интерфейс командной строки в системах Unix 6
1.3.2 Основы интерактивной работы в оболочке bash.............16
1.3.3 Файловая система...23
1.4 ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ 33
1.5 ТРЕБОВАНИЯ К ОТЧЕТУ...34

2 ЛАБОРАТОРНАЯ РАБОТА №2. ПРАКТИЧЕСКОЕ ЗНАКОМСТВО СО СТАНДАРТНОЙ УТИЛИТОЙ *GNU MAKE* ДЛЯ ПОСТРОЕНИЯ ПРОЕКТОВ В ОС *UNIX*.................................36

2.1 ЦЕЛЬ РАБОТЫ 36
2.2 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ 36
2.3 ОСНОВЫ ИСПОЛЬЗОВАНИЯ УТИЛИТЫ ПОСТРОЕНИЯ ПРОЕКТОВ MAKE 36
2.3.1 Основные правила работы с утилитой Make 37
2.3.2 Пример практического использования утилиты Make 42
2.4 ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ 53
2.5 ТРЕБОВАНИЯ К ОТЧЕТУ...54

3 ЛАБОРАТОРНАЯ РАБОТА №3. ПРАКТИЧЕСКОЕ ЗНАКОМСТВО С ПОТОКАМИ И СИНХРОНИЗАЦИЕЙ ПОТОКОВ В ОС *UNIX* 55

3.1 ЦЕЛЬ РАБОТЫ 55
3.2 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ 55
3.3 УПРАВЛЕНИЕ ПОТОКАМИ 55
3.3.1 Понятие потока...55
3.3.2 Преимущества и недостатки использования потоков 56
3.3.3 Программирование потоков...57
3.3.4 Синхронизация потоков..62
3.3.5 Компиляция многопоточной программы 71
3.3.6 Особенности отладки многопоточной программы.........72
3.3.7 Примеры практической реализации.................................72
3.4 ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ 81
3.5 ТРЕБОВАНИЯ К ОТЧЕТУ...82

4 ЛАБОРАТОРНАЯ РАБОТА №4. ПРАКТИЧЕСКОЕ ЗНАКОМСТВО С ПРОЦЕССАМИ, ПЕРЕДАЧЕЙ ДАННЫХ МЕЖДУ ПРОЦЕССАМИ И ИХ СИНХРОНИЗАЦИЕЙ 83
4.1 ЦЕЛЬ РАБОТЫ 83
4.2 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ 83
4.3 ПРОЦЕССЫ И МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ......................83
4.3.1 Понятие процесса 83
4.3.2 Межпроцессное взаимодействие.......................................84
4.3.3 Основы оперирования процессами в оболочке bash 86
4.3.4 Механизмы межпроцессного взаимодействия в ОС Unix 93
4.3.5 Очереди сообщений..101
4.3.6 Работа с разделяемой памятью 104
4.3.7 Примеры практической реализации.................................105
4.4 ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ 125
4.5 ТРЕБОВАНИЯ К ОТЧЕТУ 125
ЗАКЛЮЧЕНИЕ 126
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ........................127

ВВЕДЕНИЕ

Информационно-вычислительные системы становятся все более дружественными и понятными даже для неспециалистов. Это связано, прежде всего, с тем, что пользователи и их программы взаимодействуют с вычислительной техникой посредством специального (системного) программного обеспечения – через операционную систему (ОС). На сегодняшний день существует множество различных ОС, построенных на закрытой (ОС семейства *Windows*) и открытой архитектурах (ОС семейства *Unix*). При этом, благодаря своему «открытому» характеру, именно последние более подходят для изучения базовых аспектов функционирования ОС.

Unix-подобные ОС являются достаточно популярными уже более трех десятилетий, что для ОС – очень серьезный срок. Несмотря на большое число разновидностей систем *Unix*, все их объединяет ряд основных черт, таких как язык высокого уровня *C*, положенный в основу кода всей системы, наличие стандартов в архитектуре и интерфейсных решений (*POSIX*, *System V*), использование единой, легко обслуживаемой иерархической файловой системы, различных дополнительных средств, включая средства, предназначенные для упрощения сборки программных проектов.

Современные ОС, включая ОС семейства *Unix*, поддерживают *многозадачность* и *многопоточность*. При многопоточности в системе одновременно может работать несколько задач, и каждая из задач может быть разбита на подзадачи, выполняемые параллельно. Для этого в современных ОС предусмотрен ряд стандартных механизмов, позволяющих достаточно эффективно обеспечить синхронизацию процессов и передачу данных между ними, что позволяет решать задачи и подзадачи как независимо, так и с кооперацией между ними.

В рамках предлагаемого лабораторного практикума студентам предоставляется возможность получить практические навыки работы в одной из разновидностей ОС *Unix* – *Linux*, на примере решения практических задач: разработка программного проекта с помощью утилиты для автоматизированной сборки программных проектов *GNU Make*; разработка многопоточного приложения и разработка многопроцессного приложения с использованием стандартных межпроцессных средств синхронизации и передачи данных между.

1 ЛАБОРАТОРНАЯ РАБОТА №1. ПРАКТИЧЕСКОЕ ЗНАКОМСТВО С ОПЕРАЦИОННОЙ СИСТЕМОЙ *UNIX*

1.1 ЦЕЛЬ РАБОТЫ

Ознакомится с операционной системой *Unix*, получить практические навыки работы в наиболее распространенном командном интерпретаторе *Bash*, изучить принципы организации файловой системы *Unix* и базовых команд управления файлами.

1.2 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Осуществить в локальной сети с использованием программы *PuTTY*, использующей протокол *ssh*, доступ к удаленному компьютеру (необходимые данные для доступа указывает преподаватель) под управлением ОС *Linux* в консольном режиме. Ознакомится с перечнем основных команд, используемых пользователями ОС *Linux* при работе в системе.

1.3 ОСНОВЫ РАБОТЫ В ОПЕРАЦИОННОЙ СИСТЕМЕ UNIX

1.3.1 Интерфейс командной строки в системах Unix

В *Unix*-подобных ОС базовый уровень общения с пользователем заключается во вводе с клавиатуры команд и просмотре выводимой текстовой информации на дисплее (такой способ общения часто называют «интерфейсом командной строки»). Понятия «клавиатура» и «дисплей» в данном случае во многом условны и не обязательно означают реальные устройства компьютера, на котором работает пользователь. Например, «дисплеем» может быть окно графической среды пользователя или интерфейс программы удаленного доступа *telnet* или *ssh*. Но при этом смысл от этого не изменяется: базовый интерфейс пользователя предполагает ввод команд и вывод текстовой информации. Для того, чтобы подчеркнуть «виртуальность» устройств ввода и вывода текста, их вместе называют терминалом.

Unix – многопользовательская ОС, следовательно, каждый компьютер под управлением этой ОС может иметь множество терминалов, что позволяет одновременно работать многим пользователям. Терминалы могут быть непосредственно подключены к компьютеру или существовать где-либо в сети (локальной или глобальной). Сетевые терминалы обычно представляют собой компьютеры с собственной ОС, на которых запущена программа

удаленного доступа, подобная стандартной программе *telnet*, работающей с использованием сетевых протоколов на основе *TCP*. В настоящее время вместо *telnet* обычно используют программы, работающие по защищенному шифрованием протоколу *ssh*.

Интерфейс командной строки – не единственный способ общения с ОС *Unix* (например, существуют графические интерфейсы пользователя: *KDE*, *GNOME*, *Xfse* и т.д., различные файловые оболочки и т.п.), но именно в *Unix* умение работать с командной оболочкой очень важно. Во многом это обусловлено огромным набором базовых команд, их чрезвычайной гибкостью и возможностью совместного использования для автоматизации обработки данных.

Регистрация

Во всех *Unix*-подобных ОС, установленных на конкретном компьютере, имеется некоторая база данных пользователей, имеющих право использования ресурсов этого компьютера. Пользователей, не включенных в этот список, система к работе не допустит. База данных ведется администратором компьютера и содержит для каждого пользователя следующую информацию:

- регистрационное имя;
- зашифрованный пароль;
- идентификатор пользователя (*User ID – UID*);
- список групп, в которые включен пользователь;
- путь к командной оболочке;
- путь к домашнему каталогу;
- другая дополнительная информация.

Регистрационное имя и пароль необходимы для процедуры регистрации пользователя. Идентификатор *UID* востребован внутренними функциями системы и непосредственно используется редко. Механизм групп позволяет объединять пользователей по определенным полномочиям на доступ к файлам и программам. Путь к командной оболочке нужен для ее запуска после процедуры регистрации. И, наконец, домашний каталог – это обычно место в файловой системе, целиком принадлежащее данному пользователю и, конечно, администратору.

Начало работы

Чтобы начать работу с *Unix*, нужно получить доступ к терминалу и зарегистрироваться в системе. В случае удаленной работы подключение терминала можно осуществить запуском программы *ssh* и соединением с тем компьютером, на котором предполагается вести работу. Когда

компьютер готов зарегистрировать пользователя, на экране отображается приглашение к вводу его имени:

login: _

В ответ на это приглашение нужно ввести регистрационное имя, согласованное с администратором или владельцем компьютера (понятно, что на собственном компьютере пользователь сам вправе выбирать имена пользователей). Имя пользователя рекомендуется составлять из строчных латинских букв и цифр. Некоторые имена (например, *root*, *ftp* и т.п.) могут быть зарезервированы для системных целей и не могут быть отданы обычному пользователю. Особый частный случай в большинстве систем – имя *root*, которое принадлежит администратору или владельцу компьютера. Это имя дает практически неограниченные права по управлению системой. Пользователя с правами *root* часто называют привилегированным пользователем.

После ввода имени и нажатия клавиши *Enter* («*Return*», «*Ret*», «*CR*») система выведет на экран запрос на ввод пароля. Например:

login: alex

password: _

Ввод пароля также завершается нажатием клавиши *Enter*. Вводимый пароль не отображается на экране. Если пользователь с указанным именем существует и его пароль введен правильно, система сделает домашний каталог пользователя текущим и запустит командную оболочку, связанную с данным пользователем. Оболочка обычно выполняет некоторый начальный набор команд, который может вывести приветственное сообщение, указать на наличие или отсутствие новой почты, выполнить начальный набор команд (все эти действия зависят от особенностей конкретной ОС). И, наконец, на экране появится приглашение к вводу команды:

$ _

Приглашение не обязательно имеет такой вид, как показано выше, и зависит от конкретной командной оболочки и ее конфигурации.

Командные оболочки ОС Unix

В системах *Unix* используются различные командные оболочки (*command shells*), называемые также командными процессорами или интерпретаторами команд. Среди них наиболее известны и распространены:

- *sh* (*Bourne shell*) – *оболочка Борна* (испытана временем, но не слишком удобна в работе);

- *csh* (*C-shell*) – *оболочка C* (несколько более удобна по сравнению с sh, но несовместима с ней по командному языку);
- *ksh* (*Korn shell*) – *оболочка Корна* (включает мощный командный язык, основанный на языке *sh* и развитые средства интерактивной работы);
- *bash* (*Bourne-Again Shell*) – «*снова*» *оболочка* «*Борна*» (удобна для интерактивной работы, создана на основе *sh* и во многом с ней совместима).

Тип оболочки, как правило, можно определить по последнему символу приглашения: знак доллара («*$*») указывает на *sh*-совместимую оболочку (*sh, bash, ksh*), а знак амперсанда («*&*») соответствует оболочке *csh*. Однако у привилегированного пользователя независимо от используемого командного процессора последним символом приглашения обычно бывает знак решетки («*#*»).

Основными функциями командных оболочек являются:

- организация диалога с пользователем (ввод команд);
- выполнение внутренних команд;
- запуск внешних программ;
- исполнение командных файлов.

Возможности командных файлов в системе *Unix* являются гораздо более полными, чем в системе *MS-DOS*, и командные языки вполне могут быть названы полноценными языками программирования. Командные языки в разных оболочках различаются, а стандартным принято считать командный язык оболочки *bash*.

Команды Unix и запуск программ

Общий синтаксис команд в *Unix*-подобных ОС выглядит следующим образом:

```
имя_команды [ключи ...] [параметры ...]
```

Первый элемент обозначает конкретную команду, аргументы (ключи и параметры) могут сообщать дополнительную информацию. Ключи обычно начинаются со знака «минус» («-»). Например, команда

```
ls -l -a /home
```

состоит из:

- имени команды «*ls*», выводящей список файлов в заданном каталоге;
- ключа (модификатора) «*-l*», указывающего, что нужно вывести подробный листинг;

- ключа «*-a*», указывающего, что нужно выводить все файлы, включая служебные («дот файлы»);
- параметра «*/home*», задающего путь к каталогу.

В командах ОС *Unix*, их ключах и параметрах регистр букв (строчные или заглавные) различается. Для большей части команд характерна запись строчными буквами. Ключи во многих случаях могут объединяться в одну группу. Например, команда:

```
ls -la /home
```

полностью эквивалентна рассмотренной выше.

Команды разделяются на *внутренние*, которые выполняются командным процессором, и *внешние*. Внутренних команд обычно немного, а их состав и синтаксис могут зависеть от используемой командной оболочки. При использовании *bash* полный список и краткий синтаксис внутренних команд можно получить, набрав после приглашения команду «*help*».

Внешние команды представляют собой запуск программ, независимых от оболочки. Для запуска программы простым указанием ее имени необходимо, чтобы путь к этой программе был указан в переменной среды *PATH* (аналог одноименной переменной среды в *MS-DOS*). Если программа не найдена в каталогах, перечисленных в *PATH*, перед именем программы должен быть явно указан путь, даже если программа находится в текущем каталоге (хотя в современных *Unix* системах это уже не требуется). Например, запуск программы *hello* из текущего каталога может выглядеть так:

```
$ ./hello
```

Здесь и далее в аналогичных примерах вводимую пользователем информацию будем выделять другим шрифтом, что позволит отличать ее от выводимых системой символов. В этом примере знак доллара в начале строки представляет собой приглашение к вводу, формируемое системой, а остаток строки – информация, введенная пользователем.

Пути в переменной среды *PATH* отделяются друг от друга знаками двоеточия («*:*») без окаймляющих пробелов. Если вывести на экран листинги всех каталогов, входящих в *PATH*, можно таким образом получить полный список внешних команд системы, с которой осуществляется работа. Следует отметить, что значение любой переменной среды можно получить, указав в требуемом контексте ее имя с предшествующим знаком доллара. Например, в команде

```
$ echo $PATH
```

выражение *$PATH* будет заменено командным интерпретатором на содержимое переменной *PATH*. Учитывая, что действие команды *echo* заключается в выводе в стандартный поток вывода своего аргумента, то на экран попадет именно содержимое переменной среды *PATH*.

Изменение пароля

Первая команда, которую следует выполнить при первом сеансе работы в системе – команда изменения собственного пароля: *passwd*. Эта команда вызывается без параметров. После ее запуска на экране появится приглашение ввести старый пароль (если пароля не было, этот шаг может быть пропущен). После правильного ввода старого пароля будет предложено ввести новый пароль, а затем ввести его еще раз для исключения случайной ошибки. Пароли при вводе отображаться не будут. Ниже представлен примерный протокол работы команды *passwd*:

```
$ passwd
Changing password for alex
Old password:
New password:
Re-type new password:
$ _
```

Для выбора паролей существуют определенные правила. Основное – пароль не должен быть угадываемым. Не надо писать свое имя в обратном порядке, не следует составлять пароль из одинаковых или ряда соседних на клавиатуре букв и т.п. Пароль должен содержать минимум шесть-семь символов и включать необычные сочетания букв, цифр, дефисов и подчеркиваний. В современных системах предпочтительно использование длинных паролей из 12-ти и более символов. Многие реализации команды *passwd* пытаются определить пригодность нового пароля и выводят предупреждающие сообщения, если пароль неудачен. Типичным случаем, когда *passwd* может проявить «недовольство» – ввод пароля только из строчных букв.

Не следует думать, что вышеуказанные правила чрезмерны. Даже личный домашний компьютер при входе в сеть *Internet* через модемное соединение может стать видимым другим людям, которые могут попытаться поменять пароль или узнать пароль соединения с провайдером. Поэтому простой пароль может существенно упростить задачу злоумышленников и стоить «нерадивому» пользователю очень дорого.

Получение справочной информации

Системы *Unix*, как правило, поставляются с огромным количеством справочной информации в электронном виде. Справочная информация разбита на разделы по тематике. Нумерация разделов в разных системах может быть разной. Пожалуй, самая часто используемая информация содержится в разделе 1, где рассматриваются команды и прикладные программы, доступные рядовым пользователям системы. В пределах раздела справочные материалы организованы по так называемым «страницам» (*manual page*). Каждая такая страница содержит документацию по конкретной команде, функции, интерфейсу, протоколу и т.п. и в реальности может быть многостраничным документом. Для получения справочной информации можно использовать команду

```
man [раздел] ключ
```

Для получения справки по использованию команды или программы аргумент «ключ» должен быть именем соответствующей команды или программы. Параметр «раздел» может представлять собой цифру (или букву) номера раздела справочных руководств, в котором находится нужная страница документации. Отметим, что номер раздела указывать необязательно, т.к. при его отсутствии будет найден первый подходящий раздел, где встречена нужная тема. Чтобы получить справку об использовании самой команды *man*, проще всего ввести:

```
$ man man
```

Существует и другие полезные команды для работы со справочными руководствами. В частности, команда:

```
apropos ключ
```

позволяет найти и вывести перечень тех страниц руководств, которые содержат в строке краткого пояснения заданное ключевое слово *ключ*. Справочная информация *man* доступна только для внешних команд. Для получения подсказки по внутренним командам оболочки необходимо использовать команду *help*, например:

```
$ help cd
```

Простейшие команды для работы с файловой системой

Команда изменения текущего каталога:

```
cd [имя_каталога]
```

Если команда *cd* вызвана без аргументов, текущим каталогом станет домашний каталог пользователя. Чтобы вывести на экран полное

имя текущего каталога, нужно использовать команду *pwd* без аргументов. Команда

```
ls [имя_каталога]
```

позволяет получить листинг указанного каталога. Если *имя_каталога* не указано, то будет выведен листинг текущего каталога. У команды *ls* есть несколько полезных ключей:

-l – вывести полную информацию о каждом файле;

-a – вывести листинг всех файлов, включая такие, имена которых начинаются с символа точки.

Команды *mkdir* и *rmdir* позволяют соответственно создать или удалить указанный каталог:

```
mkdir имя_каталога
rmdir имя_каталога
```

Команда просмотра файлов *less* позволяет просматривать файлы произвольного размера и перемещаться по их содержимому с помощью клавиш управления курсором (для выхода используется клавиша «*q*»):

```
less имя_файла
```

Команда копирования файлов:

```
cp источник приемник
```

Команда перемещения или переименования файлов:

```
mv источник приемник
```

Команда удаления файлов:

```
rm имя_файла
```

С командами *cp* и *rm* может использоваться ключ «*-r*», позволяющий копировать, перемещать или удалять каталоги со всем их содержимым рекурсивно.

Для полной информации о перечисленных командах, их аргументах и вариантах их использования можно обратиться к страницам руководства пользователя (команда *man*).

Стандартные потоки ввода-вывода

С каждой программой, запускаемой из командной строки *Unix*, связаны три стандартных потока данных:

- стандартный поток ввода (*stdin*);
- стандартный поток вывода (*stdout*);
- стандартный поток ошибок (*stderr*).

Программы, требующие входных данных, обычно читают информацию из стандартного потока ввода. Например, команда *wc* подсчитывает количество строк, слов и символов во входных данных. Если запустить эту команду без аргументов, то *wc* будет ожидать входных данных с терминала (чтобы закончить ввод данных, нужно нажать комбинацию клавиш *Ctrl-D*):

```
$ wc
two words
<Ctrl-D>
   1    2   10
```

В данном примере программа *wc* прочитала введенный пользователем текст из стандартного потока ввода (куда пользователь ввел текст «*two words*»). По умолчанию, этот поток соединен с терминалом (с клавиатурой) пользователя, но допускается его перенаправление. Чтобы связать данные стандартного входного потока с произвольным файлом, можно использовать операцию перенаправления «<», например:

```
$ wc < /etc/passwd
  28   37  1052
```

В данном случае команда *wc* уже не требует ввода с клавиатуры, т.к. она уже получила входные данные из файла */etc/passwd*. Заметим, что данная команда может иметь практическое применение – первая цифра означает количество строк в файле */etc/passwd*, что соответствует количеству пользователей, зарегистрированных в системе.

Стандартный поток вывода – это поток, куда программы записывают выходные данные. В предыдущем примере команда *wc* выводила результат (три числа) именно в этот поток. Так же работают и большинство других неинтерактивных команд (включая *echo*, *pwd* и *ls*, рассмотренные выше). Подобно стандартному потоку ввода, выходной поток изначально связан с терминалом, и также допускает перенаправление. Для связывания стандартного потока вывода с файлом используется операция «>», например:

```
$ ls > filelist.txt
```

В этом примере команда *ls*, вместо того, чтобы вывести список файлов на экран, записала его в файл с именем «*filelist.txt*». При этом, если файл с таким именем не существовал, он будет создан, в противном случае его старое содержимое будет потеряно. Существует и другая возможность перенаправления вывода, когда новые выходные

данные будут дописаны в конец существующего файла. Для этого используется операция «>>». В следующем примере текущие дата и время будут дописаны в конец файла с именем «*dates.txt*»:

```
$ date >> dates.txt
```

Сообщения об ошибках выводятся в стандартный поток ошибок. Например, пусть выполняется попытка получить список файлов в каталоге без соответствующих прав доступа:

```
$ ls -l /home/ftp/bin/
ls: /home/ftp/bin/: Access denied
```

В данном случае команда *ls* вывела сообщение в поток стандартной ошибки. Чтобы перенаправить его в указанный файл, можно использовать операции «2>» и «2>>» (по аналогии с «>» и «>>», только цифра 2 говорит о том, что нужно перенаправить поток ошибок), например:

```
$ ls -l /home/ftp/bin/ 2> last-error.txt
```

Операции перенаправления ввода-вывода можно комбинировать, например:

```
$ wc < /etc/passwd 2>> errors.txt > result.txt
```

Существует другой полезный способ перенаправления ввода-вывода – конвейеры команд. Операция «/» (знак вертикальной черты) позволяет перенаправить стандартный поток вывода одной команды на стандартный входной поток другой команды:

```
$ ls -l /etc | less
```

В этом примере команда *ls* выводит длинный список файлов в каталоге */etc*, эти данные попадают на вход программы *less*, которая позволяет пролистывать текст с помощью клавиш управления курсором. Так осуществляется «объединение» двух независимых команд в один «конвейер».

Рассмотрим более сложный пример формирования конвейера команд. Пусть нам требуется получить в файле «*bash-users.txt*» отсортированный список пользователей в системе, пользующихся командной оболочкой *bash*. Этого можно было бы добиться использованием нескольких команд, сохраняя промежуточные данные во временных файлах:

```
$ grep 'bash' /etc/passwd > list1.tmp
# Поиск по заданному шаблону «bash» в файле /etc/passwd
$ sort < list1.tmp > list2.tmp
```

```
# Сортировка по алфавиту данных из файла list1.tmp и запись в list2.tmp
$ cut -f1 -d: < list2.tmp > bash-users.txt
#Выделение первых полей строк по разделителю :
# и запись в файл bash-users.txt
$ rm list1.tmp list2.tmp
# Удаление временных файлов
```

Конвейеризация команд позволяет обойтись одной составной командой без использования промежуточных файлов:

```
$ grep 'bash' /etc/passwd | sort | cut -f1 -d: > bash-users.txt
```

Заметим, что команды типа *sort* или *cut* часто называют фильтрами. Фильтры получают данные из стандартного входного потока, преобразовывают их и выводят в стандартный поток вывода.

Завершение работы с Unix

Каждый сеанс работы с ОС *Unix* должен заканчиваться вводом команды *logout*. Также можно использовать комбинацию клавиш *Ctrl+D*, которая позволяет выполнить команду завершения работы с командной оболочкой, после чего система переходит в режим ожидания регистрации следующего пользователя. Если сеанс работы производился с удаленной машины с использованием программы *telnet* или *ssh*, то завершение работы командной оболочки вызывает разрыв *telnet* или *ssh* соединения.

1.3.2 Основы интерактивной работы в оболочке bash

Оболочка (*shell*) или командный интерпретатор в *Unix*-системах обеспечивает два набора функций:

- интерпретация командного языка и исполнение команд, введенных пользователем или подготовленных заранее в текстовом файле;
- интерактивное взаимодействие с пользователем, т.е. предоставление пользователю возможности редактирования и ввода команд.

Ниже рассмотрены особенности работы второй группы из набора функций, т.е. интерактивные возможности командной оболочки *bash* версий 3.x, которая является стандартной для систем *GNU/Linux*, и может быть установлена в других *Unix*-подобных системах.

Оболочка *bash* предоставляет пользователю развитые средства интерактивной работы. В частности, она поддерживает редактирование командной строки, повтор символов, макросы, «карман» (буфер), а

также историю команд (т.е. возможность повторить ранее введенную команду) и настраиваемое автоматическое дополнение.

Следует отметить, что умение пользоваться интерактивными возможностями оболочки значительно повышает эффективность работы в *Unix*-системе (особенно в сочетании с хорошим знанием командного языка). Более того, работа непосредственно в командной оболочке часто оказывается значительно более продуктивной по сравнению с использованием файловых менеджеров, таких как *Norton Commander*, *Far Manager* или *Windows Explorer*. Обратная сторона преимуществ работы в оболочке *Unix* заключается в длительном начальном периоде изучения.

Далее рассмотрим лишь некоторые наиболее используемые приемы интерактивной работы. Для более полного описания возможностей оболочки следует пользоваться руководством по использованию *bash* (команда *man bash*).

Редактирование командной строки

Классические оболочки *Unix* позволяли вводить команды как последовательность символов, завершая ввод нажатием клавиши *Enter*. Современные версии командных оболочек, такие как *bash*, включают развитые средства редактирования, позволяя свободно перемещаться по тексту команд, вводить текст в произвольной позиции строки, удалять, вставлять отдельные фрагменты команды.

Для многих функций редактирования используются комбинации клавиш с модификаторами *CTRL* и *META*. Модификатор *CTRL* имеется на клавиатуре *IBM*-совместимых компьютеров, а в качестве *META* чаще всего используется клавиша *ALT*. Работоспособность модификатора *META* зависит от настройки терминала, графической среды или программы удаленного доступа. Если с помощью клавиши *ALT* не удается добиться желаемого результата, можно использовать альтернативный способ ввода *META*-комбинаций. Для этого перед символом нужно нажать (и отпустить) клавишу *Esc*. Таким образом, например, комбинацию клавиш *META-d* можно заменить последовательностью нажатий *Esc*, *d*. Для ввода комбинаций наподобие *META-_* (знак подчеркивания) или *META->* (знак "больше") необходимо нажимать и удерживать клавишу *Shift*.

В табл. 1.1-1.3 приведены основные команды для работы в командной строке. Одному действию соответствует, как правило, несколько разных комбинаций клавиш, т.к. их работоспособность может зависеть от типа терминала. Поэтому, если не работает какая-либо из клавиш (например, *Home*), вместо нее может быть использована

альтернативная комбинация (например, *CTRL-a*). Также следует отметить, что многие из комбинаций клавиш имеют аналогичное или похожее назначение и в других программах, распространенных в *Unix*.

Таблица 1.1

Команды перемещения по командной строке

Комбинация клавиш	Описание действия
вправо *CTRL-f*	перемещение на один символ вправо
влево *CTRL-b*	перемещение на один символ влево
META-вправо *META-f*	перемещение на одно слово вправо
META-влево *META-b*	перемещение на одно слово влево
Home CTRL-a	перемещение в начало строки
End CTRL-e	перемещение в конец строки

Таблица 1.2

Удаление и вставка фрагментов команд

Комбинация клавиш	Описание действия
Backspase CTRL-h	удалить символ слева от курсора
Del CTRL-d	удалить символ в позиции курсора
CTRL-u	вырезать часть строки слева от курсора
CTRL-k	вырезать часть строки справа от курсора
META-Backspace CTRL-w	вырезать слово слева от курсора
META-d	вырезать слово справа от курсора
CTRL-y	вставить последний вырезанный текст в позицию курсора
CTRL-/ CTRL-_	отменить последнюю операцию редактирования

Таблица 1.3

Прочие комбинации клавиш

Комбинация клавиш	Описание действия
Enter	выполнить текущую команду (положение курсора не имеет значения)
CTRL-L	очистить экран и поместить текущую команду в верхней строке экрана
CTRL-d	выйти из оболочки *bash*, аналогично вводу команды *logout* (только если командная строка пуста)

Использование истории команд

Оболочка *bash* поддерживает историю команд, т.е. запоминает введенные ранее команды. Это позволяет вернуться к любой ранее введенной команде, а также использовать отдельные фрагменты команд из истории для ускорения ввода новых команд. История сохраняется при выходе из оболочки в файле с именем *.bash_history* в домашнем каталоге пользователя, и загружается вновь при следующем запуске *bash*. Таким образом, история команд не пропадает в перерывах между сеансами работы. Впрочем, существует ограничение на количество запоминаемых команд (например, 1000), и при превышении этого ограничения самые ранние команды будут автоматически удаляться.

Чтобы просмотреть историю команд, можно использовать команду *history*. Если после имени этой команды указан числовой аргумент, то будет выведено соответствующее число последних введенных команд. Например:

```
$ history 5
 4995  mkdir tmp/work
 4996  cd tmp/work
 4997  cp ~/work/log.txt.
 4998  joe log.txt
 4999  history 5
```

Как видно из вывода команды *history*, каждой команде поставлен в соответствии ее порядковый номер в истории. Чтобы выполнить одну из команд истории, можно ввести в командной строке заданный номер, предварив его восклицательным знаком. Например:

```
$ !4996
cd tmp/work
```

Очевидно, что вызов команд с использованием их номера непрактичен. Удобнее использовать похожий синтаксис, указывая вместо номера первые несколько символов команды. В этом случае будет произведен поиск команды с совпадающими первыми символами, начиная с конца истории, т.е. с наиболее недавно вводимых команд. Пример:

```
$ !cd
cd tmp/work
```

Однако такой способ также имеет недостатки при практическом использовании из-за возможности легко ошибиться и выполнить

неверную команду. Вместо этого чаще используют интерактивные операции навигации и поиска в истории. Наиболее употребительные комбинации клавиш, связанные с историей команд, приведены в табл. 1.4.

Таблица 1.4

Некоторые комбинации клавиш для навигации по истории команд

Комбинация клавиш	Описание действия
вверх *CTRL-p*	перейти к предыдущей команде
вниз *CTRL-n*	перейти к следующей команде
META-<	перейти в начало истории команд
META->	перейти в конец истории команд (т.е. к текущей команде)
CTRL-r	осуществить обратный инкрементальный поиск в истории команд (см. описание ниже)
META-.	вставить последнее слово предыдущей команды в текущую позицию курсора
CTRL-o	аналогично *Enter*, но после выполнения команды показать следующую строку истории

Самый простой способ использования истории заключается в переходе на команду, подобную той, что требуется ввести, ее редактировании и нажатии клавиши *Enter*. Если же при этом вместо *Enter* вводить *CTRL-o*, то это позволит повторить ввод серии последовательных команд, сохраненных в истории.

Отдельного внимания заслуживает возможность инкрементального поиска в истории (комбинация клавиш *CTRL-r*). Это, пожалуй, наиболее мощный способ использования истории команд. После нажатия комбинации клавиш *CTRL-r* обычное приглашение к вводу команд исчезает и появляется индикатор режима инкрементального поиска:

```
(reverse-i-search)`': _
```

В этом режиме можно вводить символ за символом любую часть команды из истории, и в процессе ввода постоянно видеть наиболее позднюю из *совпадающих* команд. Например, если происходит поиск команды, содержащей подстроку «*web*», то после нажатия *CTRL-r*, вводим сначала букву «*w*»:

```
(reverse-i-search) 'w': cd tmp/work
```

Увидим, что поиск пока не дал нужного результата, и уточняем поиск, вводя следующую букву, «*e*»:

```
(reverse-i-search) 'we': ./update-web.sh
```

Видно, что найденная команда уже содержит фрагмент «*web*», и для ее нахождения было достаточно ввести лишь два символа. Если же найденная команда оказалась не той, что искали, можно использовать *CTRL-r* для перехода на более ранние команды, также содержащие строку поиска. Продолжая предыдущий пример, повторно нажимаем *CTRL-r*. При этом будет найдена другая, более ранняя команда, например:

```
(reverse-i-search) 'we': cd work/web/homepage/
```

Теперь можно выйти из режима поиска несколькими способами. Чтобы перейти на найденную команду в истории, достаточно нажать *Esc* или комбинацию клавиш *CTRL-j*. Чтобы отменить поиск и вернуться в исходное состояние, можно нажать *CTRL-g*. И наконец, нажатие *Enter* приведет к немедленному исполнению найденной команды.

Использование автоматического дополнения в командной строке

Автоматическое дополнение (*completion*) позволяет значительно ускорить ввод команд, имен файлов, имен переменных и имен машин в командной строке. Например, пусть в системе установлена программа *bunzip2*, и нет ни одной другой программы или команды, начинающейся буквами «*bun*». В таком случае в *bash* достаточно набрать в начале командной строки эти три буквы и нажать клавишу *Tab*. При этом остальные символы, формирующие имя команды, будут вставлены автоматически. В оболочке *bash* поддерживается несколько типов дополнения и множество комбинаций клавиш для их активизации. Рассмотрим лишь две наиболее полезные возможности выполнять автоматическое дополнение (табл. 1.5).

Таблица 1.5

Возможности автоматического дополнения в командной строке

Комбинация клавиш	Описание действия
Tab	дополнении наиболее подходящий тип дополнения в зависимости от контекста

META-Tab	дополнение на основе фраз из истории команд (поскольку роль модификатора *META* часто исполняет клавиша *ALT*, а комбинация *ALT-Tab* обычно используется графической средой, для вызова этой команды рекомендуется использовать последовательность нажатий *Esc*, *Tab*).

Дополнение с помощью *Tab* может работать по-разному в зависимости от использования контекста. Табл. 1.6 в упрощенном виде показывает правила выбора типа дополнения.

Таблица 1.6

Возможности автоматического дополнения в командной строке

Контекст	Тип дополнения
начало строки	дополнение имени команды (поиск среди имен встроенных команд оболочки и программ в переменной среды *PATH*)
после символа *$*	дополнение имени переменной (поиск среди имен установленных переменных среды)
после символа @	дополнение имени машины (поиск среди имен машин в файле */etc/hosts*)
после символа ~	дополнение имени пользователя (поиск среди имен известных системе пользователей)
после шаблона имени файла	замена шаблона, только если найден лишь один подходящий файл (в данном случае производится не дополнение, а замена шаблона на подходящее имя файла)
в остальных случаях	дополнение имени файла (поиск среди имен файлов и каталогов)

Дополнение с помощью *META-Tab* всегда ищет дополнения в истории команд, выбирая фразы, начинающиеся с символов, стоящих перед текущей позицией курсора. Если однозначного варианта не найдено, независимо от типа дополнения дописывается только часть, совпадающая во всех вариантах, и, в зависимости от конфигурации оболочки, может быть выведен список подходящих дополнений. Если список вариантов не выводится автоматически, его обычно можно вывести повторным нажатием *Tab* или *META-Tab*.

1.3.3 Файловая система

Особенности формирования файлового пространства

Файловое пространство *Unix* систем представляет собой иерархию файлов, которая имеет единый общий корень – так называемый корневой каталог, обозначаемый знаком прямой косой черты «/». Чтобы однозначно идентифицировать любой файл, можно указать путь к этому файлу от корневого или текущего каталога. Все элементы пути отделяются друг от друга символом прямой косой черты. Если первый символ строки – также косая черта, то путь берет начало в корневом каталоге, в противном случае – в текущем. Путь с единственным именем обозначает файл в текущем каталоге. Примеры:

- *docs.ps* – файл с именем *docs.ps* в текущем каталоге;
- */usr/doc/FAQ/README* – файл с именем *README* в каталоге */usr/doc/FAQ;*
- *work/thesis.tex* – файл *thesis.tex* в подкаталоге *work* текущего каталога.

Понятие текущего каталога несколько отличается от такового в системе *MS-DOS* или *Windows*. В *Unix* у каждого процесса собственный текущий каталог. Корневой каталог файлового дерева *Unix* обычно содержит следующие подкаталоги (в разных системах эта структура может отличаться):

- */bin* – минимальный набор исполняемых файлов, необходимый для работоспособности системы;
- */etc* – файлы конфигурации системы;
- */dev* – файлы устройств;
- */home* – домашние каталоги пользователей;
- */lib* – основные системные библиотеки и модули;
- */root* – каталог администратора системы root;
- */proc* – файлы-образы выполняющихся процессов;
- */sbin* – минимальный набор утилит администратора;
- */tmp* – каталог для временных файлов;
- */usr* – основной объем файлов системы: установленные программы, библиотеки, исходные тексты ядра, файлы данных и прочее;
- */var* – каталог для изменяющейся информации (учетных данных, почтовых ящиков, очередей принтера, отформатированных страниц документации, логов и др.).

Следует отметить, что символ косой черты не является частью имен каталогов, а лишь указывает, что данные элементы находятся в

корневом каталоге. В каждом каталоге так же существует два особых «подкаталога» с именами «две точки» и «точка». Первый из них служит указателем на однозначно определенный родительский каталог (вышестоящий), а второй – на данный текущий каталог. Например, путь *«../readme»* указывает на файл «*readme*», который находится в родительском каталоге (на ступень выше), а путь *«./readme.now»* укажет на файл *«readme.now»*, который находится в текущем каталоге.

Большая часть файлового дерева *Unix* обычно сосредоточена в каталоге */usr*. Как правило, там можно найти следующие подкаталоги:

- */usr/bin* – исполняемые файлы;
- */usr/doc* – документация в различных форматах;
- */usr/etc* – файлы конфигурации программного обеспечения, установленного дополнительно;
- */usr/include* – включаемые файлы для программ, например, на языке *C*;
- */usr/info* – документация пользователя в гипертекстовом формате *info*;
- */usr/lib* – разделяемые библиотеки;
- */usr/local* – локальное программное обеспечение, файлы данных и библиотеки (этот каталог в некоторых системах может не использоваться);
- */usr/man* – руководства пользователя (*manual pages*);
- */usr/sbin* – утилиты администратора;
- */usr/share* – данные, совместно используемые различными прикладными программами;
- */usr/src* – исходные тексты различных компонент системы, включая ядро.

Описанная в данном случае структура каталогов относится к ОС *Red Hat Enterprise Linux* 5.2.

Формирование имен файлов

В связи с тем, что зачастую для одного языка существует несколько кодировок (например, для русского языка существуют следующие кодировки *CP866*, *CP1251*, *KOI-8R* и т.д., хотя в последнее время с распространением *UTF8* ситуация постепенно улучшается), то рекомендуется, чтобы имя файла или каталога составлялось из следующих символов:

- прописные и строчные латинские буквы;
- цифры;
- символ подчеркивания;

- символ точки;
- знак минуса (не должен быть первым символом имени);
- знак плюса (использовать не рекомендуется).

В каждой конкретной ОС в именах файлов могут быть допустимы и другие символы, но их использование может привести к некорректности работы некоторых программ и, кроме того, может затруднить перенос файлов между разными ОС. Не рекомендуется использовать названия файлов из локальных таблиц кодировок (например, имена файлов на русском языке), т.к. очень часто для одного языка существует несколько кодировок.

Максимальная длина имени файла варьируется в разных системах (и зависит скорее от используемой файловой системы, чем от самой ОС). Обычно можно использовать достаточно длинные имена файлов (до 255 символов). Максимальный размер файла в файловой системе так же зависит от ее типа. Для современных файловых систем размер файла более 4 Гбайт не является проблемой.

Как отмечалось выше, прописные и строчные буквы в системе *Unix* различаются, т.е. имена «*filename*», «*FILENAME*» и «*FileName*» являются разными. При этом файлы, отличающиеся только регистром букв, могут находиться в одном каталоге.

В отличие от системы *MS-DOS*, знак точки является обычным символом, допустимым в любом месте имени файла, а такого понятия, как расширение имени файла, строго говоря, в системе *Unix* нет. Тем не менее, последние части имен файлов, отделенные от остальной части имени точками, часто указывают на тип файла. В качестве примера, имя файла «*my-photo.tiff.gz*» может означать, что файл представляет собой изображение в формате *TIFF*, сжатое программой сжатия *gzip*.

Точка, являющаяся первым символом имени файла или каталога, имеет особое значение: такие имена по умолчанию не выводятся в листинге содержимого каталогов (хотя к ним можно свободно обращаться), для получения полного списка файлов вместо *ls*, нужно ввести *ls -a*. Другими словами, чтобы сделать файл «скрытым», нужно начать его имя с точки. Этим часто пользуются для именования служебных файлов, на которые не имеет смысла обращать особое внимание.

Просмотр и интерпретация прав доступа к файлам

ОС семейства *Unix* – традиционно многопользовательские системы. Чтобы начать работать, пользователь должен «войти» в систему, введя со свободного терминала свое регистрационное имя и пароль. Человек, зарегистрированный в учетных файлах системы, и,

следовательно, имеющий учетное имя, называется зарегистрированным пользователем системы. Регистрацию новых пользователей обычно выполняет администратор системы. Основными минимальными данными, требуемыми для регистрации пользователя в системе, являются:

- имя пользователя;
- название группы, к которой относится пользователь;
- пароль;

В *Unix* базовые права доступа к файлам включают три составляющие:

- разрешение чтения (обозначается буквой «*r*», от слова *Read*);
- разрешение записи (буква «*w*», от слова *Write*);
- разрешение выполнения (буква «*x*», от слова *eXecute*).

Разрешение на чтение позволяет пользователю читать содержимое файлов, а в случае каталогов – просматривать перечень имен файлов в каталоге (используя, например, команду *ls*).

Разрешение на запись позволяет пользователю писать в файл, т.е. изменять его содержимое. Для каталогов это дает право создавать в каталоге новые файлы и каталоги, или удалять файлы в этом каталоге.

Наконец, разрешение на выполнение позволяет пользователю запускать файлы на исполнение (как программы в машинном коде, так и командные файлы). Если на файле стоит атрибут выполнения, то независимо от его имени он считается программой, которую можно запустить (в отличие от *DOS* или *Windows*, в *Unix* возможность исполнения файла не зависит от «расширения» имени файла, такого как *.exe*). Разрешение на выполнение применительно к каталогам означает возможность перехода в этот каталог (например, командой *cd*). Поэтому для каталогов право выполнения часто называют правом поиска. Отметим, что для каталогов биты чтения и выполнения (*r* и *x*) чаще всего используются в паре, т.е. либо присутствуют оба, либо отсутствуют.

В атрибутах доступа к файлам, перечисленные типы прав доступа могут быть предоставлены для трех классов пользователей:

- владельца (у каждого файла в *Unix* есть один владелец);
- группы (с каждым файлом связана группа пользователей этого файла);
- всех остальных пользователей.

Набор прав доступа для конкретных файлов можно просмотреть с помощью команды *ls -l*. Например:

```
$ ls -l tmp/
```

```
drwxrwxr-x   10 john     users        1024 Aug 30  2002 newdir
-rw-r-----    1 john     users      173727 Jan 13 23:48 archive-0113.zip
```

В этом примере видно, что владельцем файлов является пользователь *john*, а группой владельцев является группа *users*. Набор букв и прочерков в левой части определяет тип файла (первый символ) и права доступа к файлу (остальные девять символов). В приведенном примере первая запись относится к каталогу (первая буква *d*) и демонстрирует права доступа *rwxrwxr-x*. Вторая запись относится к обычному файлу (прочерк на месте первого символа) и показывает права *rw-r-----*. Девять символов прав доступа определяют возможность чтения (*r*), записи (*w*) и выполнения (*x*) для владельца файла (первые три символа), группы владельца (следующие три символа) и всех остальных (последние три символа). Прочерки означают отсутствие соответствующих прав. Следовательно, в приведенном примере:

- *john* и все пользователи группы *users* могут просматривать и изменять содержимое каталога *newdir*, а также переходить в него, а остальные пользователи могут читать и переходить в этот каталог, но не могут создавать или удалять в нем новые файлы;
- *john* может читать и изменять файл *archive-0113.zip*, пользователи группы *users* могут только читать содержимое этого файла, а все остальные не имеют к нему никаких прав доступа.

Кроме символьного представления прав доступа часто используется цифровая форма. В цифровом представлении права доступа составляются из трех восьмеричных цифр, каждая из которых определяет набор из трех битов полномочий *r,w,x*. Чтобы перевести права доступа из символьного представления в числовое, следует:

1) представить набор прав в двоичном виде (например, 110100000 для набора прав *rw-r-----*);

2) перевести полученное двоичное число в восьмеричную систему счисления (например, восьмеричным представлением двоичного числа 110100000 будет 640).

Возможно, более удобным будет способ перевода символьной формы в числовую путем суммирования восьмеричных значений отдельных битов прав доступа:

- 400 – владелец имеет право на чтение;
- 200 – владелец имеет право на запись;
- 100 – владелец имеет право на выполнение;
- 040 – группа имеет право на чтение;
- 020 – группа имеет право на запись;
- 010 – группа имеет право на выполнение;

- 004 – остальные имеют право на чтение;
- 002 – остальные имеют право на запись;
- 001 – остальные имеют право на выполнение.

Можно заметить, что для прав доступа *rw-r-----* получим: 400+200+040 = 640.

Типы файлов

В ОС *Unix* имеются следующие основные типы файлов:

- обычные файлы (*regular files*);
- каталоги (*directories*);
- символьные ссылки (*symbolic links*);
- файлы физических устройств (*device files*);
- именованные каналы (*named pipes*);
- доменные гнезда (*sockets*).

Обычные файлы используются наиболее широко и представляют собой именованные наборы данных с возможностью произвольного доступа.

Каталоги – специальный тип файлов, позволяющий группировать вместе другие файлы и каталоги. Содержимое каталога представляет собой список находящихся в нем файлов.

Файлы устройств в *Unix* являются средством общения прикладных программ с драйверами оборудования компьютера. Для того, чтобы передать данные драйверу какого-либо устройства, прикладная программа должна произвести запись в соответствующий специальный файл. По аналогии, операция чтения из файла устройства означает получение данных от его драйвера. Обычно каталог с файлами имеет имя «*/dev*».

Символьные ссылки подобны «ярлыкам» в Windows. Они позволяют создавать альтернативные имена файлов, причем могут указывать на файлы в других каталогах. При открытии программой символьной ссылки фактически открывается файл, на который она указывает. Символьные ссылки могут указывать как на обычные файлы, так и на каталоги и файлы других типов.

Именование каналы еще называют буфером *FIFO* (*First In First Out*). Через файлы такого типа два независимых процесса могут обмениваться данными: все, что записано в файл одним процессом, может быть прочитано другим.

Гнезда – это абстрактные конечные точки сетевого соединения. Записывая данные в этот файл, процесс отсылает их в сеть. При этом

процессы, установившие связь через пару гнезд, могут быть запущены на разных компьютерах, так и на одном.

Монтирование сторонних файловых систем

Как было представлено выше, к файловой системе *Unix* могут быть подключены сторонние файловые системы, например, файловые системы других ОС или файловые системы, расположенные на внешних носителях (флоппи-дисках, *CD-ROM* и др.). Чтобы сторонняя файловая система была доступна ОС *Unix*, необходимо осуществить операцию ее монтирования. Фактически, монтирование – это указание того, куда системе следует адресовываться при обращении к объектам сторонней файловой системы. Для системы это указание называется точкой монтирования.

Теоретически можно указать системе произвольное место точки монтирования, но на практике для монтирования сторонних файловых систем существует каталог */mnt*. В нем необходимо создать подкаталог, который будет служить точкой монтирования. Например, файловую систему *Windows*, физически расположенную на первом логическом разделе того же винчестера (на диске *C*) можно сделать доступной для *Unix*, примонтировав ее с помощью команды *mount* (предварительно нужно создать каталог */mnt/windows*):

mount -t vfat /dev/hda1 /mnt/windows (пример для Linux),

mount -t msdos /dev/ad0s1 /mnt/windows (пример для FreeBSD),

где ключ *-t* и аргумент *vfat (msdos)* означают тип монтируемой сторонней файловой системы (в данном случае *FAT*), */dev/hda1* – первый раздел первого жесткого диска, к которому система обращается с помощью файла устройства, */mnt/windows* – каталог, представляющий собой точку монтирования.

Системный файл регистрации пользователей

В *Unix*-системах регистрация пользователей ведется в файле */etc/passwd*. Содержимое этого файла представляет собой последовательность текстовых строк. Каждая строка соответствует одному зарегистрированному в системе пользователю и содержит семь полей, разделенных символами двоеточия. Эти поля таковы:

- регистрационное имя пользователя;
- зашифрованный пароль;
- значение *UID*;
- значение *GID* основной группы;

- комментарий (может содержать расширенную информацию о пользователе, например, имя, должность, телефоны и т.п.);
- домашний каталог;
- командная оболочка пользователя.

Файл */etc/passwd* должен быть доступен для чтения всем пользователям, т.к. к нему должны обращаться многие программы, запускаемые от имени рядового пользователя (например, чтобы узнать соответствие *UID* регистрационному имени). Но доступность для чтения всех зашифрованных паролей серьезно уменьшает безопасность системы, потому что современные вычислительные мощности позволяют сравнительно быстро подбирать пароли (в особенности, неудачно выбранные некоторыми пользователями). Поэтому часто используется схема теневых паролей (*shadow passwords*), при которой поле пароля в */etc/passwd* игнорируется, а реальный пароль берется из другого файла (например, */etc/shadow*), доступного для чтения только привилегированному пользователю. Файл теневых паролей часто содержит и другую важную информацию: срок, в течение которого допускается использование неизменного пароля, дата последнего изменения пароля и т.п. При использовании теневых паролей второе поле в */etc/passwd* обычно содержит символ звездочки или любой другой произвольный символ. Пустым поле пароля в */etc/passwd* оставлять нельзя, так как в этом случае система может посчитать, что данному пользователю пароль не требуется. Пример строки из файла */etc/passwd* (заметьте, что поле комментария в данном случае отсутствует):

```
john:*:1004:101::/home/john:/bin/sh
```

Дополнительную информацию о формате файла */etc/passwd* можно получить, набрав команду *man* по теме *passwd* в разделе 5 (форматы файлов). Эта команда будет выглядеть так:

```
man 5 passwd
```

Файл регистрации групп пользователей

Информация о группах, известных системе, содержится в файле */etc/group*. Подобно файлу регистрации пользователей, информация в */etc/group* представляет собой набор строк, по одной для каждой зарегистрированной группы пользователей. Каждая строка содержит четыре поля, разделенных двоеточиями:

- регистрационное имя группы;
- пароль группы (обычно это поле пустое, так как группам обычно не назначают пароли);

- значение *GID*, соответствующее данной группе;
- разделенный запятыми список пользователей, входящих в группу (может быть пустым).

Заметим, что пустой список пользователей в записи */etc/group* не означает, что в этой группе нет ни одного пользователя, так как *GID* основной группы пользователя определяется в файле */etc/passwd*.

Определение идентификаторов пользователей и групп

Чтобы определить *UID* пользователя, *GID* и имя его основной группы, а также список прочих групп, в который включен пользователь, можно использовать команду *id*. В случае ее использования без аргументов, команда выведет информацию о текущем пользователе. Если же указать в качестве аргумента имя зарегистрированного пользователя, вывод команды будет соответствовать указанному пользователю.

Частным случаем команды *id* является команда *groups*. Она выдает список имен всех групп, в которые включен текущий или указанный пользователь.

Ввод команды *who* без аргументов позволяет получить список пользователей, работающих в данный момент в системе. Если же набрать *whoami*, система выведет информацию о текущем пользователе.

Как обычно, дополнительную информацию о всех перечисленных командах можно получить с помощью команды *man*, например:

```
$ man who
```

Изменение владельцев файлов

Владельцем файла становится пользователь, создавший этот файл. Группой владельца по умолчанию становится основная группа регистрации пользователя. Для изменения владельцев предназначена стандартная команда *chown* (*change owner*). Однако в современных системах владельца файлов может изменять только привилегированный пользователь (*root*). У обычного пользователя существует возможность изменения только группы владельцев, и то лишь в пределах тех групп, в которые входит сам пользователь. Для изменения группы владельцев удобно использовать команду *chgrp* (*change group*). Например, чтобы сделать группой владельцев каталога *newdir* группу *student*, можно ввести:

```
chgrp student newdir
```

Существует возможность рекурсивного изменения владельцев для всех файлов и подкаталогов заданного каталога. Для этого следует использовать ключ *-R*, например:

```
chgrp -R student newdir
```

Заметим, что вместо одного имени файла или каталога в приведенных командах можно использовать множество имен, разделенных пробелами, или шаблоны имен файлов (это относится и к большей части других команд *Unix*, принимающих имена файлов в качестве последних аргументов).

Изменение прав доступа к файлам

Изменить права доступа к файлу может либо его владелец, либо привилегированный пользователь (*root*). Делается это командой *chmod* (*change mode*). Существует два формата использования этой команды, с использованием символьного и числового представления прав доступа. Использование числового представления позволяет одной командой изменить полный набор прав доступа, например:

```
chmod 770 newdir
```

Данная команда установит права доступа в числовое значение 770, т.е. *rwxrwx---*, что даст полные права владельцу и группе владельца, и никаких прав всем остальным.

Использование символьного представления прав доступа в команде *chmod* может показаться несколько сложнее, но позволяет манипулировать отдельными битами прав доступа. Например, чтобы снять бит записи для группы владельца каталога *newdir*, достаточно ввести:

```
chmod g-w newdir
```

Условный синтаксис этой команды таков:

```
chmod {u,g,o,a}{+,-,=}{r,w,x} файлы ...
```

В качестве аргументов команда принимает указание классов пользователей:

- «*u*» – владелец-пользователь (*user*),
- «*g*» – владелец-группа (*group*),
- «*o*» – остальные пользователи (*others*),
- «*a*» – все вышеперечисленные группы вместе (*all*);
- операцию, которую необходимо произвести с правами доступа:
- «+» – добавить,
- «-» – убрать,

- «=» – присвоить;
- права доступа («*r*», «*w*», «*x*»).

Как и в команде *chgrp*, в *chmod* может использоваться ключ *-R*, позволяющий рекурсивно обрабатывать содержимое подкаталогов.

Права доступа по умолчанию, команда umask

Очевидно, что при создании новых файлов и каталогов они уже будут обладать определенным набором прав доступа. Эти права доступа, устанавливаемые по умолчанию, определяются значением маски прав доступа, которая устанавливается командой *umask*. При вводе команды *umask* без аргументов она выведет текущее значение маски, при использовании восьмеричного числа в качестве аргумента будет установлено новое значение.

Маска прав доступа определяет, какие права должны быть удалены из полного набора прав, т.е. маска прав доступа является в некотором роде обратным значением прав доступа по умолчанию. Например, маска 022 приведет к сбросу битов записи для группы владельца и остальных пользователей. Заметим, что для обычных файлов (не каталогов), все биты выполнения (*x*) в правах по умолчанию будут сброшены независимо от текущей маски.

Пример, демонстрирующий эффект команды *umask*:

```
$ umask
002
$ mkdir dir1
$ ls -l
drwxrwxr-x   2 john    users      1024 Apr 21 07:29 dir1
$ umask 072
$ umask
072
$ mkdir dir2
$ ls -l
drwxrwxr-x   2 john    users      1024 Apr 21 07:29 dir1
drwx---r-x   2 john    users      1024 Apr 21 07:30 dir2
```

1.4 ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическим материалом.
2. Зарегистрироваться в системе под именем, выданным преподавателем.

3. Ознакомиться со следующими наиболее используемыми командами для пользовательской работы в ОС *Unix*: *man, apropos, ls, cd, pwd, mkdir, rmdir, cp, mv, rm, cat, echo, less, touch, grep, date, history*. Определить параметры, которые следует считать основными при использовании данных команд.

4. Определить абсолютный путь своего домашнего каталога.

5. Определить значения следующих переменных окружения: *PATH*, *MANPATH*, *PAGER*.

6. Определить границы файлового пространства, где система позволяет создавать собственные файлы и каталоги (возможно использование автоматического скрипта).

7. Проверить, возможно ли вмешательство в личное файловое пространство другого пользователя.

8. Ознакомится с командами определения прав доступа к файлам и их изменения (команды *id, groups, ls -l, stat, chmod, chown, chgrp, umask*).

9. Найти запись в файле */etc/passwd*, соответствующую вашему регистрационному имени.

10. Определить свой *UID*, узнать к каким группам относится ваше регистрационное имя, объяснить вывод команд *id, groups*.

11. Определить список групп, в которые входит пользователь *root*.

12. Узнать, какими правами доступа обладают вновь создаваемые файлы и каталоги (т.е. создать новый файл и новый каталог, и просмотреть для них права доступа).

13. Определить значение *umask*, при котором создаваемые файлы и каталоги будут недоступны для чтения, записи и исполнения никому, кроме владельца.

14. Сделать свой домашний каталог видимым для всех пользователей группы *users*.

15. Создать в домашнем каталоге подкаталог *tmp*, файлы в котором сможет создавать, удалять и переименовывать любой, входящий в группу *users*, при этом содержимое этого подкаталога не должно быть видимым всем прочим пользователям.

1.5 ТРЕБОВАНИЯ К ОТЧЕТУ

Отчет должен содержать следующие разделы:

1. Титульный лист, оформленный согласно утвержденному образцу.
2. Цели выполняемой лабораторной работы.
3. Задание на лабораторную работу.

4. Описание процесса выполнения работы: для каждого действия, производимого в командной строке, в отчет следует включить:

- краткое описание действия;
- вводимая команда или команды;
- реакция системы на ввод команд (если объем выводимых данных превышает несколько строк, всю информацию включать в отчет не следует).

5. Выводы.

2 ЛАБОРАТОРНАЯ РАБОТА №2. ПРАКТИЧЕСКОЕ ЗНАКОМСТВО СО СТАНДАРТНОЙ УТИЛИТОЙ *GNU MAKE* ДЛЯ ПОСТРОЕНИЯ ПРОЕКТОВ В ОС *UNIX*

2.1 ЦЕЛЬ РАБОТЫ

Ознакомиться с техникой компиляции программ на языке программирования *C* (*C*++) в среде ОС семейства *Unix*, а также получить практические навыки использования утилиты *GNU Make* для сборки проекта.

2.2 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Изучить особенности работы с утилитой *Make* при создании проекта на языке *C* (*C*++) в ОС *Unix*, а также получить практические навыки использования утилиты *GNU Make* при создании и сборке проекта.

2.3 ОСНОВЫ ИСПОЛЬЗОВАНИЯ УТИЛИТЫ ПОСТРОЕНИЯ ПРОЕКТОВ MAKE

«Сборка» большинства программ для ОС семейства *Unix* производится с использованием утилиты *Make*. Эта утилита считывает файл (обычно носящий имя «*makefile*» или «*Makefile*»; далее, упоминая имя этого файла, будем использовать *makefile*), в котором содержатся инструкции, и выполняет в соответствии с ними действия, необходимые для сборки программы. Во многих случаях *makefile* полностью генерируется специальной программой. Например, для разработки процедур сборки используются программы *autoconf/automake*. Однако, в некоторых программах может потребоваться непосредственное создание файла *makefile* без использования процедур автоматической генерации.

Следует отметить, что существует как минимум три различных наиболее распространенных варианта утилиты *Make*: *GNU Make*, *System V make* и *Berkeley make*.

Менее распространена, но всё ещё значима утилита *smake* Йорга Шиллинга (*Jörg Schilling*), а также первоначальная утилита *make*, определяющая комплекс основных функций, общих для всех остальных утилит. Несмотря на то, что утилита *smake* не используется по умолчанию на всех системах, это хорошая реализация *make* и она

является предпочтительной в некоторых программах (особенно созданных самим Шиллингом).

2.3.1 Основные правила работы с утилитой Make

Основными составляющими любого *make*-файла являются правила (*rules*). В общем виде правило выглядит так:

```
<цель_1> ... <цель_n>: <зависимость_1>  ... <зависимость_n>
<команда_1>
...
<команда_n>
```

Цель (*target*) – это некоторый желаемый результат, способ достижения которого описан в правиле. Цель может представлять собой имя файла. В этом случае правило описывает, каким образом можно получить новую версию этого файла.

В следующем примере целью является файл *iEdit* (исполняемый файл программы некоторого гипотетического проекта текстового редактора, с главным файлом проекта *main.cpp* и дополнительными *Editor.cpp, TextLine.cpp*). Правило описывает, каким образом можно получить новую версию бинарного файла *iEdit* (скомпоновать из перечисленных объектных файлов):

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
```

Если необходимо скомпилировать проект написанный на *C++*, то можно использовать компилятор *g++*. Следует так же отметить что ключ *–o* компилятора *GCC* указывает имя конечно бинарного файла.

Цель также может быть именем некоторого действия. В таком случае правило описывает, каким образом совершается указанное действие. В следующем примере целью является действие *clean* (очистка, удаление).

```
clean:
rm *.o iEdit
```

Подобного рода цели называют псевдоцелями (*pseudo targets*) или абстрактными целями (*phony targets*).

Зависимость (*dependency*) – это некие «исходные данные», необходимые для достижения указанной в правиле цели, некоторое «предварительное условие» для достижения цели. Зависимость может представлять собой имя файла. Для того чтобы успешно достичь указанной цели, этот файл должен существовать.

В следующем правиле файлы *main.o*, *Editor.o* и *TextLine.o* являются зависимостями. Эти файлы должны существовать для того, чтобы стало возможным достижение цели – построение файла *iEdit*:

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
```

Зависимость также может быть именем некоторого действия. Это действие должно быть предварительно выполнено перед достижением цели, указанной в правиле. В следующем примере зависимость *clean_obj* является именем действия (удалить объектные файлы программы):

```
clean_all:  clean_obj
rm iEdit
clean_obj:
rm *.o
```

Для того, чтобы достичь цель *clean_all*, необходимо сначала выполнить действие (*достигнуть цели*) *clean_obj*.

Команды – это действия, которые необходимо выполнить для обновления либо достижения цели. В следующем примере командой является вызов компилятора *GCC*. Утилита *Make* отличает строки, содержащие команды, от прочих строк *make*-файла по наличию символа *табуляции* (символа с кодом 9) в начале строки:

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
```

В приведенном выше примере строка:

```
gcc  main.o Editor.o TextLine.o -o iEdit
```

должна начинаться с символа табуляции.

Общий алгоритм работы *make*

Типичный *make*-файл проекта содержит несколько правил. Каждое из правил имеет некоторую цель и некоторые зависимости. Смыслом работы *Make* является достижение цели, которую она выбрала в качестве главной цели (*default goal*). Если главная цель является именем действия (т.е. абстрактной целью), то смысл работы *Make* заключается в выполнении соответствующего действия. Если же главная цель является именем файла, то программа *Make* должна построить самую «свежую» версию указанного файла.

Выбор главной цели. Главная цель может быть прямо указана в командной строке при запуске *Make*. В следующем примере *Make* будет стремиться достичь цели *iEdit* (получить новую версию файла *iEdit*):

```
make iEdit
```

В этом примере *Make* должна достичь цели *clean* (очистить директорию от объектных файлов проекта):

```
make clean
```

Если не указывать какой-либо цели в командной строке, то *Make* выбирает в качестве главной первую, встреченную в *make*-файле цель. В следующем примере из четырех перечисленных в *make*-файле целей (*iEdit*, *main.o*, *Editor.o*, *TextLine.o*, *clean*) по умолчанию в качестве главной будет выбрана цель *iEdit*:

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
main.o: main.cpp
gcc -c main.cpp
Editor.o: Editor.cpp
gcc -c Editor.cpp
TextLine.o: TextLine.cpp
gcc -c TextLine.cpp
clean:
rm *.o
```

Схематично, «верхний уровень» алгоритма работы *Make* можно представить так:

```
make()
{
      главная_цель = ВыбратьГлавнуюЦель()
      ДостичьЦели( главная_цель )
}
```

Достижение цели. После того как главная цель выбрана, *Make* запускает «стандартную» процедуру достижения цели. Сначала в *make*-файле выполняется поиск правила, которое описывает способ достижения этой цели (функция *НайтиПравило*). Затем, к найденному правилу применяется обычный алгоритм обработки правил (функция *ОбработатьПравило*).

```
ДостичьЦели( Цель )
{
        правило = НайтиПравило( Цель )
        ОбработатьПравило( правило )
}
```

Обработка правил. Обработка правила разделяется на два основных этапа. На первом этапе обрабатываются все зависимости, перечисленные в правиле (функция *ОбработатьЗависимости*). На втором этапе принимается решение о том, нужно ли выполнять указанные в правиле команды (функция *НужноВыполнятьКоманды*). При необходимости, перечисленные в правиле команды выполняются (функция *ВыполнитьКоманды*).

```
ОбработатьПравило( Правило )
{
        ОбработатьЗависимости( Правило )
        если НужноВыполнятьКоманды( Правило )
          {
          ВыполнитьКоманды( Правило )
          }
}
```

Обработка зависимостей. Функция *ОбработатьЗависимости* поочередно проверяет все перечисленные в правиле зависимости. Некоторые из них могут оказаться целями каких-нибудь правил. Для этих зависимостей выполняется обычная процедура достижения цели (функция *ДостичьЦели*). Те зависимости, которые не являются целями, считаются именами файлов. Для таких файлов проверяется факт их наличия. При их отсутствии, *Make* аварийно завершает работу с сообщением об ошибке.

```
ОбработатьЗависимости( Правило )
{
        цикл от i=1 до Правило.число_зависимостей
          {
          если ЕстьТакаяЦель( Правило.зависимость[ i ] )
            {
            ДостичьЦели( Правило.зависимость[ i ] )
```

```
            }
        иначе
            {
            ПроверитьНаличиеФайла( Правило.зависимость[ i ] )
            }
        }
}
```

Обработка команд. На стадии обработки команд решается вопрос о том, следует ли выполнять описанные в правиле команды или нет. Считается, что нужно выполнять команды в таких случаях как:

- цель является именем действия (абстрактной целью);
- цель является именем файла и этого файла не существует;
- какая-либо из зависимостей является абстрактной целью;
- цель является именем файла и какая-либо из зависимостей, являющихся именем файла, имеет более позднее время модификации, чем цель.

В противном случае (т.е. ни одно из вышеприведенных условий не выполняется) описанные в правиле команды не выполняются. Алгоритм принятия решения о выполнении команд схематично можно представить так:

```
НужноВыполнятьКоманды( Правило )
{
    если Правило.Цель.ЯвляетсяАбстрактной()
        return  true
    //  цель является именем файла
    если ФайлНеСуществует( Правило.Цель )
        return  true
    цикл от i=1 до Правило.Число_зависимостей
        {
     если Правило.Зависимость[ i ].ЯвляетсяАбстрактной()
            return  true
        иначе
            //  зависимость является именем файла
            {
      если ВремяМодификации( Правило.Зависимость[ i ] )  >
```

```
            ВремяМодификации( Правило.Цель )
            return  true
         }
      }
   return  false
}
```

Абстрактные цели и имена файлов. Имена действий от имен файлов утилита *Make* отличает следующим образом: сначала выполняется поиск файла с указанным именем, и если файл найден, то считается что цель или зависимость являются именем файла; в противном случае считается, что данное имя является либо именем несуществующего файла, либо именем действия (различия между этими двумя вариантами не делается, поскольку они обрабатываются одинаково).

Следует отметить, что подобный подход имеет ряд недостатков. Во-первых, утилита *Make* не рационально расходует время, выполняя поиск несуществующих имен файлов, которые на самом деле являются именами действий. Во-вторых, при подобном подходе, имена действий не должны совпадать с именами каких-либо файлов или директорий, иначе *make*-файл будет выполнятся ошибочно.

Некоторые версии *Make* предлагают свои варианты решения этой проблемы. Так, например, в утилите *GNU Make* имеется механизм (специальная цель *.PHONY*), с помощью которого можно указать, что данное имя является именем действия.

2.3.2 Пример практического использования утилиты Make

Пример создания простейшего make-файла

Рассмотрим, как утилита *Make* будет обрабатывать такой *make*-файл (*makefile*):

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
main.o: main.cpp
gcc -c main.cpp
Editor.o: Editor.cpp
gcc -c Editor.cpp
TextLine.o: TextLine.cpp
```

```
gcc -c TextLine.cpp
clean:
rm *.o
```

Предположим, что в директории с проектом находятся следующие файлы:

```
main.cpp
Editor.cpp
TextLine.cpp
```

Предположим также, что программа *Make* была вызвана следующим образом:

```
make
```

Цель не указана в командной строке, поэтому запускается алгоритм выбора цели (функция *ВыбратьГлавнуюЦель*). Главной целью становится файл *iEdit* (первая цель из первого правила). Цель *iEdit* передается функции *ДостичьЦели*. Эта функция выполняет поиск правила, которое описывает обрабатываемую цель. В данном случае, это первое правило *make*-файла. Для найденного правила запускается процедура обработки (функция *ОбработатьПравило*).

Сначала поочередно обрабатываются описанные в правиле зависимости (функция *ОбработатьЗависимости*). Первая зависимость – объектный файл *main.o*. Поскольку в *make*-файле есть правило с такой целью (функция *ЕстьТакаяЦель* возвращает *true*), то для цели *main.o* запускается процедура *ДостичьЦели*.

Функция *ДостичьЦели* ищет правило, где описана цель *main.o*. Эта цель описана во втором правиле *make*-файла. Для этого правила запускается функция *ОбработатьПравило.* Функция *ОбработатьПравило* запускает процесс обработки зависимостей (функция *ОбработатьЗависимости*). Во втором правиле указана единственная зависимость – *main.cpp*. Такой цели в *make*-файле не существует, поэтому считается, что зависимость *main.cpp* является именем файла. Далее, проверяется наличие этого файла на диске (функция *ПроверитьНаличиеФайла*) – такой файл существует. На этом процесс обработки зависимостей завершается.

После обработки зависимостей, функция *ОбработатьПравило* принимает решение о том, следует ли выполнять указанные в правиле команды (функция *НужноВыполнятьКоманды*). Цели правила (файла *main.o*) не существует, поэтому команды выполнять следует. Функция *ВыполнитьКоманды* запускает указанную в правиле команду

(компилятор *GCC*), в результате чего создается файл *main.o*, так называемый объектный файл (*object file*).

Цель *main.o* достигнута (объектный файл *main.o* построен). Теперь *make* возвращается к обработке остальных зависимостей первого правила. Зависимости *Editor.o* и *TextLine.o* обрабатываются аналогично. Для них выполняются те же действия, что и для зависимости *main.o*.

После того, как все зависимости (*main.o*, *Editor.o* и *TextLine.o*) обработаны, решается вопрос о необходимости выполнения указанных в правиле команд (функция *НужноВыполнятьКоманды*).

Поскольку цель (*iEdit*) является именем файла, который в данный момент не существует, то принимается решение выполнить описанную в правиле команду (функция *ВыполнитьКоманды*).

Содержащаяся в правиле команда запускает компилятор *GCC*, в результате чего создается исполняемый (бинарный) файл *iEdit*. Таким образом главная цель (*iEdit*) достигнута. На этом программа *Make* завершает свою работу.

Рассмотрим еще один пример работы утилиты *Make* в условиях, когда для обработки описанного выше *make*-файла, будет выполнена следующая команда:

```
make clean
```

Цель явно указана в командной строке, поэтому главной целью становится абстрактная цель *clean*. Цель *clean* передается функции *ДостичьЦели*. Эта функция ищет правило, которое описывает обрабатываемую цель. Это будет пятое правило *make*-файла. Для найденного правила запускается процедура обработки (функция *ОбработатьПравило*).

Поскольку в правиле не указано каких-либо зависимостей, *Make* сразу переходит к этапу обработки указанных в правиле команд. Цель является именем действия, поэтому команды нужно выполнять. Указанные в правиле команды выполняются, и цель *clean*, таким образом, считается достигнутой. На этом программа *make* завершает работу.

Использование переменных

Возможность использования переменных внутри *make*-файла – очень удобное и часто используемое свойство *Make*. В традиционных версиях утилиты, переменные ведут себя подобно макросам языка *C*. Для задания значения переменной используется оператор присваивания. Например, выражение:

```
obj_list := main.o Editor.o TextLine.o
```

присваивает переменной *obj_list* значение «*main.o Editor.o TextLine.o*» (без кавычек). Пробелы между символом «=» и началом первого слова игнорируются. Следующие за последним словом пробелы также игнорируются. Значение переменной можно использовать с помощью конструкции:

```
$(имя_переменной)
```

Например, при обработке такого *make*-файла:

```
dir_list := . .. src/include
all:
echo $(dir_list)
```

на экран будет выведена строка:

```
. .. src/include
```

Переменные могут не только содержать текстовые строки, но и «ссылаться» на другие переменные. Например, в результате обработки *make*-файла:

```
optimize_flags := -O3
compile_flags  := $(optimize_flags) -pipe
all:
echo $(compile_flags)
```

на экран будет выведено:

```
-O3 -pipe
```

Во многих случаях использование переменных позволяет упростить *make*-файл и повысить его наглядность. Для того чтобы облегчить модификацию *make*-файла, можно разместить «ключевые» имена и списки в отдельных переменных и поместить их в начало *make*-файла:

```
program_name := iEdit
obj_list     := main.o Editor.o TextLine.o
$(program_name): $(obj_list)
gcc $(obj_list) -o $(program_name)
...
```

Адаптация такого *make*-файла для сборки другой программы сведется к изменению нескольких начальных строк.

Использование автоматических переменных

Автоматические переменные – это переменные со специальными именами, которые «автоматически» принимают определенные значения перед выполнением описанных в правиле команд. Автоматические переменные можно использовать для «упрощения» записи правил. Такое, например, правило:

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
```

с использованием автоматических переменных можно записать следующим образом:

```
iEdit: main.o Editor.o TextLine.o
gcc  $^  -o $@
```

Здесь *$^* и *$@* являются автоматическими переменными. Переменная *$^* означает «список зависимостей». В данном случае при вызове компилятора *GCC* она будет ссылаться на строку «*main.o Editor.o TextLine.o*». Переменная *$@* означает «имя цели» и будет в этом примере ссылаться на имя «*iEdit*». Если бы в примере была использована следующая автоматическая переменная *$<*, то она указывала бы на первое имя зависимости, т.е. в данном случае на файл *main.o*.

Иногда использование автоматических переменных совершенно необходимо, например, в шаблонных правилах.

Шаблонные правила

Шаблонные правила (*implicit rules* или *pattern rules*) – это правила, которые могут быть применены к целой группе файлов. В этом их отличие от обычных правил, описывающих отношения между конкретными файлами. Традиционные реализации *Make* поддерживают так называемую «суффиксную» форму записи шаблонных правил:

```
.<расширение_файлов_зависимостей>.<расширение_файлов_целей>:
	<команда_1>
	<команда_2>
	...
	<команда_n>
```

Например, следующее правило гласит, что все файлы с расширением «*o*» зависят от соответствующих файлов с расширением «*cpp*»:

```
.cpp.o:
gcc -c $^
```

Для современной реализации *Make* более предпочтительная следующая запись данной цели:

```
%.o:  %.cpp
gcc -c $^ -o $@
```

Следует обратить внимание на использование автоматической переменной *$^* для передачи компилятору имени файла-зависимости. Поскольку шаблонное правило может применяться к разным файлам, использование автоматических переменных – это единственный способ узнать для каких файлов задействуется правило в данный момент. Шаблонные правила позволяют упростить *make*-файл и сделать его более универсальным. Рассмотрим простой проектный файл:

```
iEdit: main.o Editor.o TextLine.o
gcc $^ -o $@
main.o: main.cpp
gcc -c $^
Editor.o: Editor.cpp
gcc -c $^
TextLine.o: TextLine.cpp
gcc -c $^
```

Все исходные тексты программы обрабатываются одинаково: для них вызывается компилятор *GCC*. С использованием шаблонных правил, этот пример можно переписать так:

```
iEdit: main.o Editor.o TextLine.o
gcc $^ -o $@
%.o: %.cpp
gcc -c $^
```

Когда *Make* ищет в файле проекта правило, описывающее способ достижения искомой цели (см. «Достижение цели», функция *НайтиПравило*), то в расчет принимаются и шаблонные правила. Для каждого из них проверяется, нельзя ли задействовать это правило для достижения искомой цели.

Пример создания более сложного make-файла

Предыдущие два примера создания *make*-файлов существенно упрощают создание проектов. Следует отметить, что работа по перечислению всех объектных файлов, составляющих программу, может быть также автоматизирована. При этом вариант создания бинарного файла типа

```
iEdit: *.o
gcc $< -o $@
```

может не сработать, т.к. в указанном случае будут учтены только существующие в данный момент объектные файлы. Особенно это актуально в случае наличия сложных заголовочных файлов (**.h*), определющих зависимости частей проекта. Для того, чтобы избежать подобного затруднения следует использовать более сложный способ, который основан на предположении, что все файлы, содержащие исходный текст, должны быть скомпилированы и скомпонованы в результирующую программу. Такой вариант методики сборки состоит из двух шагов:

Получить список всех файлов с исходным текстом программы (всех файлов с расширением «*cpp*»). Для этого следует использовать функции обработки строк, в данном случае функцию *wildcard*, которая получает список файлов с заданным шаблоном в указанном каталоге.

Преобразовать список исходных файлов в список объектных файлов (заменить расширение «*cpp*» на расширение объектных файлов «*o*»). Для этого следует воспользоваться функцией *patsubst*, которая заменяет заданную подстроку в заданной строке.

Следующий пример содержит модифицированную версию *make*-файла с использованием указанных двух шагов:

```
iEdit: $(patsubst  %.cpp,%.o,$(wildcard  *.cpp))
gcc $^ -o $@
%.o: %.cpp
gcc -c $<
main.o: main.h Editor.h TextLine.h
Editor.o:                Editor.h        TextLine.h
TextLine.o: TextLine.h
```

Список объектных файлов программы строится автоматически (цель *iEdit*). Сначала с помощью функции *wildcard* получается список всех файлов с расширением «*cpp*», находящихся в директории проекта. Затем, с помощью функции *patsubst*, полученный таким образом список

исходных файлов, преобразуется в список объектных файлов (расширение файлов меняется с «*cpp*» на «*o*»). *Make*-файл теперь стал более универсальным.

Автоматическое построение зависимостей от заголовочных файлов

Автоматизировав перечисление объектных файлов, остается проблема перечисления зависимостей объектных файлов от заголовочных файлов. Например:

```
….
main.o: main.h Editor.h TextLine.h
Editor.o:            Editor.h      TextLine.h
TextLine.o: TextLine.h
```

Перечисление зависимостей «вручную» может потребовать существенного объема работы. Не всегда достаточно просто открыть файл с исходным текстом и перечислить имена всех заголовочных файлов, подключаемых с помощью директивы «*#include*»: заголовочные файлы могут включать в себя другие заголовочные файлы и подобная «цепочка зависимостей» может быть достаточно длинной. Построение списка зависимостей можно реализовать с использованием утилиты *Make* и компилятора *GCC*.

Для совместной работы с *Make* компилятор *GCC* имеет несколько опций:

- *Ключ компиляции –M***.** Для каждого файла с исходным текстом препроцессор будет выдавать на стандартный выход список зависимостей в виде правила для программы *Make*. В список зависимостей попадает сам исходный файл, а также все файлы, включаемые с помощью директив «*#include <имя_файла>*» и «*#include "имя_файла"*». После запуска препроцессора компилятор останавливает работу и генерации объектных файлов не происходит.
- *Ключ компиляции –MM***.** Аналогичен ключу *-M*, но в список зависимостей попадает только сам исходный файл, и файлы, включаемые с помощью директивы «*#include "имя_файла"*».
- *Ключ компиляции –MD***.** Аналогичен ключу *-M*, но список зависимостей выдается не на стандартный выход, а записывается в отдельный файл зависимостей. Имя этого файла формируется из имени исходного файла путем замены его расширения на расширение «*d*». Например, файл зависимостей для файла *main.cpp* будет называться *main.d*. В отличие от ключа *-M* компиляция проходит обычным образом, а не прерывается после фазы запуска препроцессора.

- *Ключ компиляции –MMD*. Аналогичен ключу *-MD*, но в список зависимостей попадает только сам исходный файл, и файлы, включаемые с помощью директивы «*#include "имя_файла"*».

Как видно компилятор может работать двумя способами – в одном случае компилятор выдает только список зависимостей и заканчивает работу (опции *-M* и *-MM*). В другом случае компиляция происходит как обычно, только в дополнении к объектному файлу генерируется еще и файл зависимостей (опции *-MD* и *-MMD*). Предпочтительней использовать второй вариант, т.к.:

1) при изменении какого-либо из исходных файлов будет построен заново лишь один соответствующий ему файл зависимостей;

2) построение файлов зависимостей происходит «параллельно» с основной работой компилятора и практически не отражается на времени компиляции.

Из двух возможных опций *-MD* и *-MMD*, предпочтительней первая, т.к. с помощью директивы «*#include <имя_файла>*» часто включаются не только «стандартные» (например «*#include <iostream.h>*»), но и свои собственные заголовочные файлы, которые могут иногда меняться (например, «*#include «myclass.h*»).

После того как файлы зависимостей сформированы, они имеют расширение «*d*». Для того, чтобы сделать их доступными утилите *Make*, следует использовать директиву *#include*:

```
include $(wildcard *.d)
```

Следует обратить внимание на использование функции *wildcard*, т.к. конструкция:

```
include *.d
```

будет правильно работать только в том случае, если в каталоге будет находиться хотя бы один файл с расширением «*d*». Если таких файлов нет, то *Make* аварийно завершится, т.к. потерпит неудачу при попытке «построить» эти файлы. Если же использовать функцию *wildcard*, то при отсутствии искомых файлов, эта функция просто вернет пустую строку. Далее, директива *include* с аргументом в виде пустой строки, будет проигнорирована, не вызывая ошибки. Теперь новый вариант *makefile* из этого примера выглядит следующим образом:

```
iEdit: $(patsubst %.cpp,%.o,$(wildcard *.cpp))
gcc $^ -o $@
%.o: %.cpp
gcc -c -MD $<
```

```
include $(wildcard *.d)
```

Файлы с расширением «*d*» – это сгенерированные компилятором *GCC* файлы зависимостей. Вот, например, как выглядит файл *Editor.d*, в котором перечислены зависимости для файла *Editor.cpp*:

```
Editor.o: Editor.cpp Editor.h TextLine.h
```

Теперь при изменении любого из файлов – *Editor.cpp, Editor.h* или *TextLine.h*, файл *Editor.cpp* будет перекомпилирован для получения новой версии файла *Editor.o*.

Размещение файлов с исходными текстами по директориям

Приведенный выше *make*-файл вполне работоспособен и с успехом может быть использован для сборки небольших программ. Однако, с увеличением размера программы, становится не очень удобным хранить все файлы с исходными текстами в одном каталоге. В таком случае предпочтительно размещать эти файлы по разным директориям, отражающим логическую структуру проекта. Для этого нужно модифицировать *make*-файл, чтобы неявное правило

```
%.o: %.cpp
gcc -c $<
```

осталось работоспособным, используют переменную *VPATH*, в которой перечисляются все директории, в которых могут располагаться исходные тексты. В следующем примере файлы *Editor.cpp* и *Editor.h* расположены в каталоге *Editor*, а файлы *TextLine.cpp* и *TextLine.h* в каталоге *TextLine*:

```
main.cpp
main.h
Editor /
Editor.cpp
Editor.h
TextLine /
TextLine.cpp
TextLine.h
makefile
```

Вот как выглядит *makefile* для этого примера:

```
source_dirs := Editor TextLine
search_wildcard s := $(addsuffix /*.cpp,$(source_dirs))
```

```
iEdit: $(notdir $(patsubst %.cpp,%.o,$(wildcard $(search_wildcard s))))
gcc $^ -o $@
VPATH := $(source_dirs)
%.o: %.cpp
gcc -c -MD $(addprefix -I ,$(source_dirs)) $<
include $(wildcard *.d)
```

По сравнению с предыдущим вариантом *make*-файла он претерпел следующие изменения:

- для хранения списка директорий с исходными текстами, который нужно будет указывать в нескольких местах, заведена отдельная переменная *source_dirs*;
- шаблон поиска для функции *wildcard* (переменная *search_wildcard s*) строится «динамически» исходя из списка директорий *source_dirs*;
- переменная *VPATH* используется для того, чтобы согласно шаблонному правилу в указанном списке директорий мог осуществляться поиск файлов с исходными текстами;
- компилятору разрешается искать заголовочные файлы во всех директориях с исходными текстами; для этого используется функция *addprefix* добавляющая префикс-флаг «*-I*» к строке компилятора *GCC*;
- при формировании списка объектных файлов, из имен исходных файлов «убирается» имя каталога, где они расположены (с помощью функции *notdir*);
- при формировании списка исходных файлов с расширением «*cpp*» была использована функция *addsuffix*, добавляющая суффикс «*/*.cpp*» к названиям каталогов с исходными файлами, указанными в переменной *source_dirs*.

Сборка программы с разными параметрами компиляции

Часто возникает необходимость в получении нескольких вариантов программы, скомпилированных с использованием различным параметров. Типичным примером использования двух различных вариантов является использование отладочной и рабочей версии программы. В таких случаях следует использовать подход, при котором:

1) все варианты программы собираются с помощью одного и того же *make*-файла;

2) необходимые настройки компилятора осуществляется в *make*-файле с использованием параметров, передаваемых программе *Make* в командной строке (например, флаги компиляции):

```
make compile_flags="-O3 -funroll-loops -fomit-frame-pointer"
```

Таким образом, наиболее простым способом задания параметров компиляции будет внесение дополнительной переменной *compile_flags* в *makefile*. Если параметров компиляции несколько (строка с параметрами содержит пробелы), то строка со значением переменной *compile_flags* должны быть заключена в кавычки. Командный файл *makefile* с использованием параметров может выглядеть следующим образом:

```
override compile_flags := –pipe
source_dirs := Editor TextLine
search_wildcard s := $(addsuffix /*.cpp,$(source_dirs))
iEdit: $(notdir $(patsubst %.cpp,%.o,$(wildcard $(search_wildcard s))))
gcc $^ $(compile_flags) -o $@
VPATH := $(source_dirs)
%.o: %.cpp
gcc -c -MD $(addprefix -I ,$(source_dirs)) $(compile_flags) $<
include $(wildcard *.d)
```

Переменная *compile_flags* получает свое значение из командной строки, если они заданы, в противном случае используется значение по умолчанию (т.е. «*–pipe*»). Для ускорения работы компилятора к параметрам компиляции добавляется флажок *-pipe*. Следует обратить внимание на необходимость использования в примере директивы *override*, использованной для изменения переменной *compile_flags* внутри *make*-файла. В противном случае переданные флаги компиляции из командной строки не будут размещены в переменной *compile_flags*.

2.4 ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическим материалом.
2. Используя любой текстовый редактор, создать простейшую программу на языке *C* (*C++*) с использованием как минимум двух исходных файлов (с программным кодом).
3. Для автоматизации сборки проекта утилитой *Make* создать *make*-файл (см. п. «Пример создания более сложного *make*-файла»).
4. Выполнить программу (скомпилировать, при необходимости отладить).
5. Показать, что при изменении одного исходного файла и последующем вызове *Make* будут исполнены только необходимые команды компиляции (неизмененные файлы перекомпилированы не

будут) и изменены атрибуты и/или размер объектных файлов (файлы с расширением *.o*).

6. Создать *make*-файл с высоким уровнем автоматизированной обработки исходных файлов программы согласно следующим условиям:

- имя скомпилированной программы (выполняемый или бинарный файл), флаги компиляции и имена каталогов с исходными файлами и бинарными файлами (каталоги *src*, *bin* и т.п.) задаются с помощью переменных в *Makefile*;
- зависимости исходных файлов на языке *C (C++)* и цели в *make*-файле должны формироваться динамически;
- наличие цели *clean*, удалающей временные файлы.
- каталог проекта должен быть структурирован следующим образом:
    - *src* – каталог с исходными файлами;
    - *bin* – каталог с бинарными файлами (скомпилированными);
    - *Makefile*.

2.5 ТРЕБОВАНИЯ К ОТЧЕТУ

Отчет должен содержать следующие разделы:

1. Титульный лист, оформленный согласно утвержденному образцу.
2. Цели выполняемой лабораторной работы.
3. Задание на лабораторную работу.
4. Исходные тексты созданных программ, содержимое созданных *make*-файлов, иллюстрацию результатов работы.
5. Выводы.

3 ЛАБОРАТОРНАЯ РАБОТА №3. ПРАКТИЧЕСКОЕ ЗНАКОМСТВО С ПОТОКАМИ И СИНХРОНИЗАЦИЕЙ ПОТОКОВ В ОС *UNIX*

3.1 ЦЕЛЬ РАБОТЫ

Ознакомиться с подсистемой управления потоками в операционной системе *Unix* и основными программными средствами для создания, управления и удаления потоков.

3.2 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Изучить основные программные средства управления потоками ОС *Unix*, а также способы синхронизации потоков. Разработать приложения для многопоточных вычислений с использованием синхронизации посредством мьютексов, семафоров и условных переменных.

3.3 УПРАВЛЕНИЕ ПОТОКАМИ

3.3.1 Понятие потока

Поток (*thread*) можно определить как часть процесса, включающая управляющую последовательность команд, и использующая системные ресурсы этого процесса.

Существует две основных категории реализации потоков:

Пользовательские потоки – потоки, реализуемые через специальные библиотеки потоков и работающие в пользовательском пространстве.

Потоки уровня ядра – потоки, реализуемые через системные вызовы и работающие в пространстве ядра.

Каждый процесс имеет как минимум один поток, при этом самый первый поток, создаваемый при рождении нового процесса, принято называть начальным или главным потоком этого процесса.

Основное отличие процесса от потока заключается в способе использования системных ресурсов. Дочерний процесс практически независим от родительского, для него системой выделяется отдельное адресное пространство, и он на равных правах с родительским процессом «конкурирует» за процессорное время. При этом можно уничтожить родительский процесс, не затронув дочерний, который может выполняться и после завершения родительского процесса. В отличие от дочернего процесса, поток, порожденный данным

процессом, полностью зависим от процесса и завершение процесса влечет уничтожение всех созданных им потоков, поскольку происходит освобождение системных ресурсов, выделенных для этого процесса. Еще одним отличием является невозможность изменить права доступа для потока, тогда как дочерний процесс в редких случаях (например, при смене пароля для входа в систему) может обладать правами доступа отличными от прав родительского процесса.

Первая подсистема потоков в *Linux* появилась в 1996 году и называлась *LinuxThreads*, автором ее является *Ксавье Лерой* (*Xavier Leroy*). Разработанная им библиотека *LinuxThreads* была попыткой организовать поддержку потоков в *Linux* в то время, когда ядро системы еще не предоставляло никаких специальных механизмов для работы с потоками. Позднее разработку потоков для *Linux* вели сразу две конкурирующие группы – *NGPT* (*New Generation POSIX Threads*) и *NPTL* (*Native POSIX Thread Library*). В 2002 году группа *NGPT* фактически присоединилась к *NPTL* и теперь реализация потоков *NPTL* является стандартом *Linux*. Подсистема потоков *Linux* стремится соответствовать требованиям стандартов *POSIX*, поэтому новые многопоточные приложения *Linux* должны без проблем компилироваться на новых *POSIX*-совместимых системах.

3.3.2 Преимущества и недостатки использования потоков

Потоки часто становятся источниками программных ошибок особого рода. Эти ошибки возникают при использовании потоками разделяемых ресурсов системы (например, общего адресного пространства) и являются частным случаем более широкого класса ошибок – ошибок синхронизации. Если задача разделена между независимыми процессами, то доступом к их общим ресурсам управляет ОС, и вероятность ошибок из-за конфликтов доступа снижается. Впрочем, разделение задачи между несколькими независимыми процессами само по себе не защищает от других разновидностей ошибок синхронизации. В пользу потоков можно указать то, что накладные расходы на создание нового потока в многопоточном приложении ниже, чем накладные расходы на создание нового самостоятельного процесса. Уровень контроля со стороны родительского процесса над порожденными потоками в многопоточном приложении выше, чем уровень контроля приложения над дочерними процессами. Кроме того, в результате работы многопоточных программ, как правило, не остаются незавершенными дочерние процессы,

зависящие от родительского, (процессы-«зомби»), как может произойти при использовании многопроцессного приложения.

Преимущества и недостатки использования нескольких потоков во время выполнения программы приведены в табл. 3.1.

Таблица 3.1

Преимущества и недостатки использования потоков

Преимущества	Недостатки
Потоки удобно использовать при необходимости выполнения в процессе нескольких действий сразу (пример: одновременная обработка на сервере запросов нескольких пользователей).	Создание многопоточного приложения может потребовать значительных усилий на этапе проектирования из-за необходимости синхронизации потоков.
Ускорение работы приложений, использующих ввод, обработку и вывод данных за счет возможности распределения этих операций по отдельным потокам. Это дает возможность не прекращать выполнение программы во время возможных простоев из-за ожидания при чтении/записи данных.	Отладка многопоточной программы значительно сложнее, чем отладка однопоточной программы из-за сложности контроля исполнения различных потоков.
Как правило, переключение между потоками происходит быстрее и требует меньших затрат системных ресурсов, по сравнению с переключением между процессами.	Параллельные вычисления, реализованные в виде многопоточного приложения, на компьютере с одним процессором не обязательно будут выполняться быстрее, чем аналогичная по функциональности однопоточная программа.

3.3.3 Программирование потоков

В *Linux* каждый поток на самом деле является процессом, и для того, чтобы создать новый поток, нужно создать новый процесс. Однако для создания дополнительных потоков используются процессы особого типа. Эти процессы представляют собой обычные дочерние процессы главного процесса, но они разделяют с главным процессом адресное пространство, файловые дескрипторы и обработчики сигналов. Для обозначения процессов этого типа применяется специальный термин –

легкие процессы (*lightweight processes*). Поскольку для потоков не требуется создавать собственную копию адресного пространства (и других ресурсов) своего процесса-родителя, создание нового легкого процесса требует значительно меньших затрат, чем создание полновесного дочернего процесса.

Спецификация *POSIX* 1003.1*c* требует, чтобы все потоки многопоточного приложения имели один идентификатор, однако в *Linux* у каждого процесса, в том числе и у процессов-потоков, есть свой идентификатор.

Основные функции для работы с потоками

Для работы с потоками используются следующие основные функции:

- *pthread_create()* – создание потока;
- *pthread_join()* – блокирование работы вызвавшего функцию процесса или потока в ожидании завершения потока;
- *pthread_cancel()* – досрочное завершение потока из другого потока или процесса;
- *pthread_exit()* – завершает поток, код завершения передается функции *pthread_join()*. Данная функция подобна функции *exit()*, однако вызов *exit()* в «основном» процессе программы приведет к завершению всей программы.

Запуск и завершение потока

Потоки создаются функцией *pthread_create()*, имеющей следующую сигнатуру:

```
int pthread_create (pthread_t* tid, pthread_attr_t* attr,
void*(*function)(void*), void* arg)
```

Данная функция определена в заголовочном файле <*pthread.h*>. Первый параметр этой функции представляет собой указатель на переменную типа *pthread_t*, которая служит идентификатором создаваемого потока. Второй параметр, указатель на переменную типа *pthread_attr_t*, используется для установки атрибутов потока. Третьим параметром функции *pthread_create()* должен быть адрес функции потока. Эта функция играет для потока ту же роль, что функция *main()* – для главной программы. Четвертый параметр функции *pthread_create()* имеет тип *void**. Этот параметр может использоваться для передачи значения в функцию потока. Вскоре после вызова *pthread_create()* функция потока будет запущена на выполнение параллельно с другими потоками программы.

Новый поток запускается не сразу после вызова *pthread_create()* потому что перед тем, как запустить новую функцию потока, нужно выполнить некоторые подготовительные действия, а поток-родитель при этом продолжает выполняться. Необходимо учитывать данное обстоятельство при разработке многопоточного приложения, в противном случае возможны серьезные ошибки при выполнении программы. Если при создании потока возникла ошибка, то функция *pthread_create()* возвращает ненулевое значение, соответствующее номеру ошибки.

Функция потока должна иметь сигнатуру вида:

```
void* func_name(void* arg)
```

Имя функции может быть любым. Аргумент *arg* является указателем, который передается в последнем параметре функции *pthread_create()*. Функция потока может вернуть значение, которое затем может быть обработано другим потоком или процессом.

Функция, вызываемая из функции потока, должна обладать свойством реентерабельности или возможности повторного вхождения (этим же свойством должны обладать рекурсивные функции). Реентерабельная функция – это функция, которая может быть вызвана повторно, в то время, когда она уже вызвана (отсюда и происходит ее название). Реентерабельные функции используют локальные переменные (и локально выделенную память) в тех случаях, когда их нереентерабельные аналоги могут воспользоваться глобальными переменными.

Завершение функции потока происходит в следующих случаях:

- функция потока вызвала функцию *pthread_exit*;
- функция потока достигла точки выхода;
- поток был досрочно завершен другим потоком или процессом.

Функция *pthread_exit* объявлена в заголовочном файле *<pthread.h>* и ее сигнатура имеет вид:

```
void pthread_exit(void *retval)
```

Аргументом функции является указатель на возвращаемый объект. Нельзя возвращать указатель на стековый (не динамический) объект, объявленный в теле функции потока, либо на динамический объект, создаваемый и удаляемый в теле функции, т.к. после завершения потока все стековые объекты его функции удаляются. В итоге указатель будет содержать адрес памяти с неопределенным содержимым, что может привести к серьезной ошибке.

В случае, если необходимо дождаться завершения потока в теле родительского процесса вызывается функция *pthread_join()* следующего вида:

```
int pthread_join(pthread_t th, void** thread_return)
```

Первый аргумент *th* необходим для указания ожидаемого потока, значение этого аргумента определяется в результате выполнения функции *pthread_create()*. В качестве второго аргумента *thread_return* выступает указатель на аргумент функции *pthread_exit()*, либо NULL, если поток ничего не возвращает.

Досрочное завершение потока

Функции потоков можно рассматривать как вспомогательные программы, находящиеся под управлением функции *main()*. Точно так же, как при управлении процессами иногда возникает необходимость досрочно завершить процесс, многопоточной программе может понадобиться досрочно завершить один из потоков. Для досрочного завершения потока можно воспользоваться функцией *pthread_cancel()*:

```
int pthread_cancel(pthread_t thread).
```

Единственным аргументом этой функции является идентификатор потока – *thread*. Функция *pthread_cancel()* возвращает 0 в случае успеха и ненулевое значение (код ошибки) в случае ошибки.

Несмотря на то, что *pthread_cancel()* может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков. В теле функции потока можно не только самостоятельно выбрать порядок завершения в ответ на вызов *pthread_cancel()*, но и вовсе игнорировать этот вызов. Поэтому вызов функции *pthread_cancel()* следует рассматривать как запрос на выполнение досрочного завершения потока.

Функция *pthread_setcancelstate()* определяет, будет ли поток реагировать на обращение к нему с помощью *pthread_cancel()*, или не будет. Сигнатура функции имеет вид:

```
int pthread_setcancelstate(int state, int* oldstate).
```

Аргумент *state* может принимать два значения:

- *PTHREAD_CANCEL_DISABLE* – запрет завершения потока;
- *PTHREAD_CANCEL_ENABLE* – разрешение на завершение потока.

Во второй аргумент *oldstate* записывается указатель на предыдущее значение аргумента *state*. С помощью функции *pthread_setcancelstate()*

можно указывать участки кода потока, во время исполнения которых поток нельзя завершить вызовом функции *pthread_cancel()*:

```
//...
// участок функции, который можно досрочно завершить
//...
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
//...
// участок функции, который нельзя досрочно завершить
//...
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
//...
// участок функции, который можно досрочно завершить
//...
```

Функция *pthread_testcancel()* создает точку возможного досрочного завершения потока (точку отмены). Такие точки необходимы для корректного завершения потока, т.к. даже если досрочное завершение разрешено, поток, получивший запрос на досрочное завершение, часто может завершить работу не сразу. Если поток находится в режиме отложенного досрочного завершения (именно этот режим установлен по умолчанию), он выполнит запрос на досрочное завершение, только достигнув одной из точек отмены. Сигнатура функции *pthread_testcancel()*:

```
void pthread_testcancel().
```

В соответствии со стандартом *POSIX*, точками отмены являются вызовы многих «обычных» функций, например, *open()*, *pause()* и *write()*.

Тем не менее, выполнение потока может быть прервано принудительно, не дожидаясь точек отмены. Для этого необходимо перевести поток в режим немедленного завершения, что делается с помощью вызова функции

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL).
```

Вызов функции

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL)
```

снова переводит поток в режим отложенного досрочного завершения.

3.3.4 Синхронизация потоков

При выполнении нескольких потоков во многих случаях необходимо синхронизировать их взаимодействие. Существует несколько способов синхронизации потоков:

- взаимные исключения – мьютексы (*Mutex – MUTial EXclusion*);
- переменные состояния;
- семафоры.

Механизм использования переменных состояния и семафоров в многопоточных приложениях аналогичен механизму использования этих методов синхронизации в многопроцессных приложениях.

Механизм мьютексов представляет общий метод синхронизации выполнения потоков. Мьютекс можно определить как объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. В любой момент временем мьютексом может владеть только один поток.

Использование мьютексов гарантирует, что только один поток в некоторый момент времени выполняет критическую секцию кода. Мьютексы можно использовать и в однопоточном коде.

Доступны следующие действия с мьютексом: инициализация, удаление, захват или открытие, попытка захвата.

Объекты синхронизации потоков являются переменными в памяти процесса, соотвественно обладают живучестью объектов процесса, к которым можно обратиться так же, как к данным. Потоки в различных процессах могут связаться друг с другом через объекты синхронизации, помещенные в разделяемую память потоков, даже в случае, когда потоки в различных процессах невидимы друг для друга.

Объекты синхронизации можно также разместить в файлах, где они будут существовать независимо от создавшего их процесса.

Необходимость в синхронизации потоков возникает в следующих случаях:

1) Если синхронизация – это единственный способ гарантировать последовательность разделяемых (общих) данных.

2) Если потоки в двух или более процессах могут использовать единственный объект синхронизации совместно. При этом объект синхронизации должен инициализироваться только одним из взаимодействующих процессов, потому что повторная инициализация объекта синхронизации устанавливает его в открытое состояние.

3) Если синхронизация может гарантировать достоверность изменяющихся данных.

4) Если процесс может отобразить файл и существует поток в этом процессе, который получает уникальный доступ к записям. Как только установлена блокировка, любой другой поток в любом процессе, отображающем файл, который пытается установить блокировку, блокируется, пока запись в файл не будет закончена.

Функции синхронизации потоков с использованием мьютексов

Для синхронизации потоков с использованием мьютексов используются следующие основные функции:

- *pthread_mutex_init()* – инициализирует взаимоисключающую блокировку;
- *pthread_mutex_destroy()* – удаляет взаимоисключающую блокировку;
- *pthread_mutex_lock()* – устанавливает блокировку. В случае, если блокировка была установлена другим потоком, текущий поток останавливается до снятия блокировки другим процессом;
- *pthread_mutex_unlock()* – снимает блокировку.

Инициализация и удаление объекта атрибутов мьютекса

Атрибуты мьютекса могут быть связаны с каждым потоком. Чтобы изменить атрибуты мьютекса по умолчанию, можно объявить и инициализировать объект атрибутов мьютекса, а затем изменить определенные значения. Часто атрибуты мьютекса устанавливаются в одном месте в начале приложения, чтобы быстро найти и изменить их. После того, как сформированы атрибуты мьютекса, можно непосредственно инициализировать мьютекс.

Функция

```
int pthread_mutexattr_init (pthread_mutexattr_t* mattr)
```

используется, чтобы инициализировать объект атрибутов mattr значениями по умолчанию. Память для каждого объекта атрибутов выделяется системой поддержки потоков во время выполнения.

Пример вызова функции *pthread_mutexattr_init()*:

```
#include <pthread.h>

pthread_mutexattr_t mattr;

int ret;

/* инициализация атрибутов значениями по умолчанию */

ret = pthread_mutexattr_init(&mattr);
```

Для корректного удаления объекта атрибутов, созданного с помощью функции *pthread_mutexattr_init()*, необходимо вызвать функцию *pthread_mutexattr_destroy()*. В противном случае возможна утечка памяти, так как тип *pthread_mutexattr_t* является закрытым. Она возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример вызова функции *pthread_mutexattr_destroy()*:

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
/* удаление атрибутов */
ret = pthread_mutexattr_destroy(&mattr);
```

Область видимости мьютекса

Областью видимости мьютекса может быть либо некоторый процесс, либо вся система. В первом случае оперировать мьютексом могут только потоки, созданные процессом, в котором создан и мьютекс. Во втором случае мьютекс существует в разделяемой памяти и может быть разделен среди потоков нескольких процессов. По умолчанию мьютекс создается в области видимости процесса и обладает живучестью процесса.

Чтобы установить область видимости атрибутов мьютекса используется функция

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t* mattr, int)
```

Первый аргумент *mattr* является указателем на объект атрибутов, для которого устанавливается область видимости. Второй аргумент является константой обозначающей устанавливаемую область видимости: *PTHREAD_PROCESS_PRIVATE* для области видимости процесса и для *PTHREAD_PROCESS_SHARED* области видимости системы. Мьютекс, созданный в области видимости системы, должен существовать в разделяемой памяти.

Пример вызова функции *pthread_mutexattr_setpshared()*:

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
ret = pthread_mutexattr_init(&mattr);
/* переустановка на значение по умолчанию: private */
```

```
ret = pthread_mutexattr_setpshared (&mattr, PTHREAD_PROCESS_PRIVATE);
```

Функция

```
pthread_mutexattr_getpshared(pthread_mutexattr_t *mattr, int *pshared)
```

используется для получения текущей области видимости мьютекса потока:

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int pshared, ret;
/* получить атрибут pshared для мьютекса */
ret = pthread_mutexattr_getpshared(&mattr, &pshared);
```

Инициализация мьютекса

Функция *pthread_mutex_init()* предназначена для инициализации мьютекса:

```
int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);
```

Мьютекс, на который указывает первый аргумент *mp*, инициализируется значением по умолчанию, если второй аргумент *mattr* равен *NULL*, или определенными атрибутами, которые уже установлены с помощью *pthread_mutexattr_init()*.

Функция *pthread_mutex_init()* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример использования функции *pthread_mutexattr_init()*:

```
#include <pthread.h>
pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;
/* инициализация мьютекса значением по умолчанию */
ret = pthread_mutex_init(&mp, NULL);
```

Когда мьютекс инициализируется, он находится в открытом (разблокированном) состоянии. Статически определенные мьютексы могут инициализироваться непосредственно значениями по умолчанию с помощью константы *PTHREAD_MUTEX_INITIALIZER*. Пример инициализации:

```
/* инициализация атрибутов мьютекса по умолчанию*/
```

```
ret = pthread_mutexattr_init(&mattr);
/* смена значений mattr с помощью функций */
ret = pthread_mutexattr_setpshared
(&mattr, PTHREAD_PROCESS_SHARED);
/* инициализация мьютекса произвольными значениями */
ret = pthread_mutex_init(&mp, &mattr);
```

Запирание (захват) мьютекса

Функция *pthread_mute_lock()* используется для запирания мьютекса. Аргументом функции является адрес запираемого мьютекса. Если мьютекс уже заперт, вызывающий поток блокируется и мьютекс ставится в очередь приоритетов. Когда происходит возврат из *pthread_mute_lock()*, мьютекс запирается, а вызывающий поток становится его владельцем. Функция *pthread_mute_lock()* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_mutex_lock(&mp);
```

Для открытия мьютекса используется функция *pthread_mutex_unlock()*. При этом мьютекс должен быть закрыт, а вызывающий поток должен быть владельцем мьютекса, то есть тем, кто запирал мьютекс. Пока любые другие потоки ждут доступа к мьютексу, поток-владелец мьютекса, находящийся в начале очереди, не блокирован. Функция *pthread_mutex_unlock()* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_mutex_unlock(&mp);
```

Существует способ захвата мьютекса без блокирования потока. Функция *pthread_mutex_trylock()* пытается провести запирание мьютекса. Она является неблокирующей версией *pthread_mutex_lock()*. Если мьютекс уже закрыт, вызов возвращает ошибку, однако поток, вызвавший эту функцию, не блокируется. В противном случае, мьютекс

закрывается, а вызывающий процесс становится его владельцем. Функция *pthread_mutex_trylock()* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_ mutex_trylock(&mp);
```

Захват через мьютекс не должен повторно инициализироваться или удаляться, пока другие потоки могут его использовать. Если мьютекс инициализируется повторно или удаляется, приложение должно убедиться, что в настоящее время этот мьютекс не используется.

Удаление мьютекса

Функция *pthread_mutex_destroy()* используется для удаления мьютекса в любом состоянии. Функция *pthread_mutex_destroy()* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_mutex_destroy(&mp);
```

Иерархия блокировок

Иногда может возникнуть необходимость одновременного доступа к нескольким ресурсам. При этом возникает проблема, заключающаяся в том, что два потока пытаются захватить оба ресурса, но запирают соответствующие мьютексы в различном порядке.

В приведенном ниже примере два потока запирают мьютексы 1 и 2 и возникает тупик при попытке запереть один из мьютексов.

Таблица 3.2

Пример «тупика» для двух потоков

Поток 1	Поток 2
/ использует ресурс 1 */* *pthread_mutex_lock(&m1);* */* теперь захватывает* *ресурсы 2+1*/* *pthread_mutex_lock(&m2);*	*/* использует ресурс 2 */* *pthread_mutex_lock(&m2);* */* теперь захватывает* *ресурсы 1+2*/* *pthread_mutex_lock(&m1);*

Наилучшим способом избежать проблем является запирание нескольких мьютексов в одном и том же порядке во всех потоках. Эта техника называется иерархией блокировок: мьютексы упорядочиваются путем назначения каждому своего номера. После этого придерживаются правила – если мьютекс с номером *n* уже заперт, то нельзя запирать мьютекс с номером, меньше *n*.

Если блокировка всегда выполняется в указанном порядке, тупик не возникнет. Однако, эта техника может использоваться не всегда поскольку иногда требуется запирать мьютексы в порядке, отличном от порядка их номеров.

Чтобы предотвратить тупик в этой ситуации, лучше использовать функцию *pthread_mutex_trylock()*. Один из потоков должен освободить свой мьютекс, если он обнаруживает, что может возникнуть тупик.

Ниже проиллюстрировано использование условной блокировки:

```
// Поток 1:
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m2);
/* обработка */
/* нет обработки */
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
// Поток 2:
for (; ;) {
pthread_mutex_lock(&m2);
if (pthread_mutex_trylock(&m1)==0)
/* захват! */
break;
/* мьютекс уже заперт */
pthread_mutex_unlock(&m2);
}
/* нет обработки */
pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);
```

В примере выше, поток 1 запирает мьютексы в нужном порядке, а поток 2 пытается закрыть их по-своему. Чтобы убедиться, что тупик не

возникнет, поток 2 должен аккуратно обращаться с мьютексом 1; если поток блокировался, ожидая мьютекс, который будет освобожден, он, вероятно, только что вызвал тупик с потоком 1. Чтобы гарантировать, что это не случится, поток 2 вызывает *pthread_mutex_trylock()*, который запирает мьютекс, если тот свободен. Если мьютекс уже заперт, поток 2 получает сообщение об ошибке. В этом случае поток 2 должен освободить мьютекс 2, чтобы поток 1 мог запереть его, а затем освободить оба мьютекса.

Синхронизация с использованием семафора

Семафор предназначен для синхронизации потоков по действиям и по данным и в общем случае способ использования семафора сходен со способом использования мьютексов.

Семафор (S) – это защищенная переменная, значения которой можно опрашивать и менять только при помощи специальных операций $P(S)$ и $V(S)$ и операции инициализации. Семафор может принимать целое неотрицательное значение. При выполнении потоком операции P над семафором S значение семафора уменьшается на 1 при $S>0$, или поток блокируется, «ожидая на семафоре», при $S=0$. При выполнении операции $V(S)$ происходит пробуждение одного из потоков, ожидающих на семафоре S, а если таковых нет, то значение семафора увеличивается на 1. В простом случае, когда семафор работает в режиме 2-х состояний ($S>0$ и $S=0$), его алгоритм работы полностью совпадает с алгоритмом мьютекса.

Как следует из вышесказанного, при входе в критическую секцию поток должен выполнять операцию $P(S)$, а при выходе из критической секции операцию $V(S)$.

Прототипы функций для манипуляции с семафорами описываются в файле *<semaphore.h>*. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий:

- *int sem_init(sem_t* sem, int pshared, unsigned int value);* – инициализация семафора *sem* значением *value*. В качестве *pshared* всегда необходимо указывать 0.
- *int sem_wait(sem_t* sem);* – «ожидание на семафоре». Выполнение потока блокируется до тех пор, пока значение семафора не станет положительным. При этом значение семафора уменьшается на 1.
- *int sem_post(sem_t* sem);* – увеличивает значение семафора *sem* на 1.
- *int sem_destroy(sem_t* sem);* – уничтожает семафор *sem*.
- *int sem_trywait(sem_t* sem);* – неблокирующий вариант функции *sem_wait*. При этом вместо блокировки вызвавшего потока

функция возвращает управление с кодом ошибки в качестве результата работы.

Синхронизация с использованием условной переменной

Условная переменная позволяет потокам ожидать выполнения некоторого условия (события), связанного с разделяемыми данными. Над условными переменными определены две основные операции: информирование о наступлении события и ожидание события. При выполнении операции «информирование» один из потоков, ожидающих значения условной переменной, возобновляет свою работу.

Условная переменная всегда используется совместно с мьютексом. Перед выполнением операции «ожидание» поток должен заблокировать мьютекс. При выполнении операции «ожидание», указанный мьютекс автоматически разблокируется. Перед возобновлением ожидающего потока выполняется автоматическая блокировка мьютекса, позволяющая потоку войти в критическую секцию, после критической секции рекомендуется разблокировать мьютекс. При подаче сигнала другим потокам рекомендуется функцию «сигнализации» так же защитить мьютексом.

Прототипы функций для работы с условными переменными содержатся в файле *pthread.h*. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий.

- *pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* attr);* – инициализирует условную переменную *cond* с указанными атрибутами *attr* или с атрибутами по умолчанию (при указании 0 в качестве *attr*).
- *int pthread_cond_destroy(pthread_cond_t* cond);* – уничтожает условную переменную cond.
- *int pthread_cond_signal(pthread_cond_t* cond);* – информирование о наступлении события потоков, ожидающих на условной переменной *cond*.
- *int pthread_cond_broadcast(pthread_cond_t* cond);* – информирование о наступлении события потоков, ожидающих на условной переменной *cond*. При этом возобновлены будут все ожидающие потоки.
- *int pthread_cond_wait(pthread_cond_t* cond , pthread_mutex_t* mutex);* – ожидание события на условной переменной cond.

Рассмотренных средств достаточно для решения разнообразных задач синхронизации потоков. Вместе с тем они обеспечивают взаимоисключение на низком уровне и не наполнены семантическим смыслом. При непосредственном их использовании легко допустить

ошибки различного рода: забыть выйти из критической секции, использовать примитив не по назначению, реализовать вложенное использование примитива и т.д. При этом операции с мьютексами, семафорам и условными переменными оказываются разбросанными по всему программному коду приложения, что повышает вероятность появления ошибки и усложняет ее поиск и устранение.

3.3.5 Компиляция многопоточной программы

Для компиляции и сборки многопоточной программы необходимо иметь следующее:

- Стандартный компилятор *C* (*cc, gcc, g*++т.д.);
- Файлы заголовков: *<thread.h>, <pthread.h>, <errno.h>, <limits.h>, <signal.h>, <unistd.h>*;
- Библиотеку реализации потоков (*libpthread*);
- Другие библиотеки, совместимые с многопоточными приложениями (*libc*, *libm*, *libw*, *libintl*, *libnsl*, *libsocket*, *libmalloc*, *libmapmalloc* и др.).

Файл заголовка *<pthread.h>*, используемый с библиотекой *lpthread*, компилирует код, который является совместимым с интерфейсами многопоточности, определенными стандартом *POSIX* 1003.1*c*.

Для компиляции программы, использующей потоки и реентерабельные системные функции[1] необходимо дополнительно указать в строке вызова компилятора следующие аргументы:

-D_REENTERANT –lpthread.

Команда компиляции *-D* включает макрос *_REENTERANT*. Этот макрос указывает, что вместо обычных функций стандартной библиотеки к программе должны быть подключены их реентерабельные аналоги. Реентерабельный вариант библиотеки *glibc* написан таким образом, чтобы реализованные в ней реентерабельные функции как можно меньше отличались от их обычных аналогов. Также в строке вызова компилятора могут дополнительно указываться пути для поиска заголовочных файлов (ключ «*-I*») и путь для поиска библиотек (ключ –*L*). Для компоновщика указывается «*-l*», что программа должна быть связана с библиотекой *libpthread*, которая содержит все специальные функции, необходимые для работы с потоками.

[1] Рентабельныя функция – функция, которая может быть вызвана повторно. Рентабельные функции используют локальные переменные и локально выделяемую память в тех случаях, когда их нереентабельные аналоги могут пользоваться глобальными переменными.

3.3.6 Особенности отладки многопоточной программы

Отладка многопоточной программы сложнее, чем отладка однопоточной. Ниже приведены наиболее типичные ошибки исходного кода, которые могут вызвать ошибки исполнения в многопоточных программах:

1) Доступ к глобальной памяти без использования механизмов синхронизации.

2) Создание тупиков, вызванных двумя потоками, пробующими получить права на одну и ту же пару глобальных ресурсов в различном порядке (один поток управляет одним ресурсом, а второй управляет другим ресурсом и ни один может продолжать выполнение до освобождения нужного им ресурса).

3) Попытка повторно получить доступ к уже захваченному ресурсу (рекурсивный тупик).

4) Создание скрытого промежутка при синхронизации. Это происходит, когда сегмент кода, защищенный механизмом синхронизации, содержит вызов функции, которая освобождает и затем повторно создает механизм синхронизации прежде, чем управление возвращается к вызывающему потоку. В результате вызывающему «кажется», что глобальные данные были защищены, хотя это не так.

5) Невнимание к тому факту, что потоки по умолчанию создаются с типом *PTHREAD_CREATE_JOINABLE* и ресурсы таких потоков должны быть утилизированы (возвращены родительскому процессу) посредством вызова функции *pthread_join()*. Обратите внимание, что *pthread_exit()* не освобождает выделенную память.

6) Создание глубоко вложенных, рекурсивных обращений и использование больших автоматических массивов может вызвать проблемы, потому что многопоточные программы имеют более ограниченный размер стека, чем однопоточные.

7) Определение неадекватного размера стека или использование стека не по умолчанию.

3.3.7 Примеры практической реализации

Пример программы с использованием потока

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
int i = 0;
void* thread_func(void *arg) {
```

```
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        for (i=0; i < 4; i++)  {
                printf("I'm still running!\n");
                sleep(1);
        }
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
        pthread_testcancel();
        printf("YOU WILL NOT STOP ME!!!\n");
}
int main(int argc, char * argv[]) {
        pthread_t thread;
        pthread_create(&thread, NULL, thread_func, NULL);
        while (i < 1) sleep(1);
        pthread_cancel(thread);
        printf("Requested to cancel the thread\n");
        pthread_join(thread, NULL);
        printf("The thread is stopped.\n");
return EXIT_SUCCESS;
}
```

Пример реализации многопоточной программы

Ниже приведен пример программы, использующей сразу несколько потоков.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define NUM_THREADS 6
void *thread_function(void *arg);
int main() {
int res;
int lots_of_threads;
```

```
pthread_t a_thread[NUM_THREADS];
void *thread_result;
srand ( (insigned)time(NULL) );
for (lots_of_threads = 0; lots_of_threads < NUM_THREADS;
lots_of_threads++)
        {
          res = pthread_create (&(a_thread[lots_of_threads]),NULL,
                      thread_function, (void *)&lots_of_threads);
          if (res != 0) {
                        perror("Thread creation failed");
                        exit(EXIT_FAILURE);
                      }
          sleep(1);
}
printf("Waiting for threads to finish...\n");
for (lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0;
lots_of_threads--)
        {
           res = pthread_join (a_thread[lots_of_threads],&thread_result);
           if (res == 0) printf("Picked up a thread\n");
                                  else perror("pthread_join failed");
}
printf("All done\n");
exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
        int my_number = *(int *)arg;
        int rand_num;
        printf ("thread_function is running. Argument was %d\n",
my_number);
        rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
```

```
        sleep(rand_num);
        printf ("Bye from %d\n", my_number);
        pthread_exit(NULL);
}
```

Пример использования мьютекса для контроля доступа к переменной

В приведенном ниже примере функция *increment_count()* использует мьютекс, чтобы гарантировать атомарность (целостность) модификации разделяемой переменной *count*. Функция *get_count()* использует мьютекс, чтобы гарантировать, что переменная *count* атомарно считывается.

```
#include <pthread.h>
pthread_mutex_t count_mutex;
long count;
void increment_count() {
        pthread_mutex_lock(&count_mutex);
        count = count + 1;
        pthread_mutex_unlock(&count_mutex);
}

long get_count() {
        long c;
        pthread_mutex_lock(&count_mutex);
        c = count;
        pthread_mutex_unlock(&count_mutex);
        return (c);
}
```

Пример многопоточной программы с синхронизацией с использованием мьютексов

Для иллюстрации примера многопоточной программы с синхронизацией между потоками посредством мьютекса выбрана задача возведения в квадрат значений элементов матрицы 2×2.

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

```
#include <unistd.h>
#include <math.h>
#define SIZE_I 2
#define SIZE_J 2
float X[SIZE_I][SIZE_J];
float S[SIZE_I][SIZE_J];
int count = 0;          // глобальный счетчик
struct DATA_ {
    double x;
    int i;
    int z;
};
typedef struct DATA_ DATA;
pthread_mutex_t lock; //Исключающая блокировка
// Функция для вычислений
double f(double x) { return x*x; }
// Потоковая функция для вычислений
void *calc_thr (void *arg) {
    DATA* a = (DATA*) arg;
    X[a->i][a->z] = f(a->x);
    // установка блокировки
    pthread_mutex_lock(&lock);
    // изменение глобальной переменной
    count ++;
    // снятие блокировки
    pthread_mutex_unlock(&lock);
    delete a;
    return NULL;
}
// Потоковая функция для ввода
void *input_thr(void *arg) {
    DATA* a = (DATA*) arg;
```

```
	printf("S[%d][%d]:", a->i, a->z);
	scanf("%f", &S[a->i][a->z]);
	delete a;
	return NULL;
}
int main() {
//массив идентификаторов потоков
pthread_t thr[ SIZE_I * SIZE_J ];
// инициализация мьютекса
pthread_mutex_init(&lock, NULL);
DATA *arg;
// Ввод данных для обработки
for (int i=0;i<SIZE_I; i++) {
	for (int z=0; z<SIZE_J; z++) {
		arg = new DATA;
		arg->i = i;
		arg->z = z;
// создание потока для ввода элементов матрицы
pthread_create (&thr[ i* SIZE_J + z ], NULL, input_thr, (void *)arg);
}	// for (int z=0; z<SIZE_J; P ++z)
}	// for (int i=0;i<SIZE_I; P ++i)
// Ожидание завершения всех потоков ввода данных
// идентификаторы потоков хранятся в массиве thr
for(int i = 0; i < SIZE_I*SIZE_J; i++) pthread_join (thr[i], NULL);
// Вычисление элементов матрицы
pthread_t thread;
printf("Start calculation\n");
for (int i=0;i<SIZE_I; i++) {
	for (int z=0; z<SIZE_J; z++) {
		arg = new DATA;
		arg->i = i;
		arg->z = z;
```

```
            arg->x = S[i][z];
            // создание потока для вычислений
            pthread_create (&thread, NULL, calc_thr, (void *)arg);
            // перевод потока в отсоединенный режим
            pthread_detach(thread);
            // for (int z=0; z<SIZE_J; z++)
    }    // for (int i=0;i<SIZE_I; i++)
    do {
    // Основной процесс "засыпает" на 1с
    sleep(1);
    // Проверка состояния вычислений
    printf("finished %d threads.\n", count);
    } while ( count < SIZE_I*SIZE_J);
    // Вывод результатов
    for (int i=0;i<SIZE_I; i++) {
        for (int z=0; z<SIZE_J; z++) {
            printf("X[%d][%d] = %f\t", i, z, X[i][z]);
    }
    printf("\n");
    }
    // удаление мьютекса
    pthread_mutex_destroy(&lock);
    return 0;
    }
```

Пример многопоточной программы с синхронизацией с использованием семафора

Ниже приведен пример программы использующей семафоры для синхронизации записи данных в один файл результатов.

```
#include "main.h"
#include <iostream.h>
#include <semaphore.h>
#include <fstream.h>
```

```
#include <stdio.h>
#include <error.h>
void* WriteToFile(void*);
int errno;
sem_t psem;
ofstream qfwrite;

int main(int argc, char **argv) {
pthread_t tidA,tidB;
int n;
char filename[] = "./rezult.txt";
qfwrite.open(&filename[0]);
sem_init(&psem,0,0);
sem_post(&psem))
        pthread_create(&tidA,NULL,&WriteToFile,(void*)100));
        pthread_create(&tidB,NULL,&WriteToFile,(void*)100));
        pthread_join(tidA,NULL));
        pthread_join(tidB,NULL));
sem_destroy(&psem));
qfwrite.close();
}
void* WriteToFile(void *f){
int max = (int)f;
for (int i=0; i<=max; i++)
 {
     sem_wait(&psem);
     qfwrite<<pthread_self()<<"-writetofilecounter i="<<i<<endl;
     qfwrite<<flush;
     sem_post(&psem);
 }
return NULL;
}
```

Пример многопоточной программы с синхронизацией с использованием условных переменных

Ниже приведен фрагмент программы использующей семафоры для синхронизации записи (writer) и чтения (reader) данных в буфер data и из него, емкость буфера – 1 запись.

```
#include "main.h"
#include <iostream.h>
#include <semaphore.h>
#include <fstream.h>
#include <stdio.h>
#include <error.h>
…
int full;
pthread_mutex_t mx;
pthead_cond_t cond;
int data;
void *writer(void *)
{
        while(1)
        {
           int t= write_to_buffer ();
           pthread_mutex_lock(&mx)
           while(full) {
               pthread_cond_wait(&cond, &mx);
            }
        data=t;
        full=1;
        pthread_cond_signal(&mx);
        pthread_mutex_unlock(&mx);
        }
return NULL;
}
```

```
void * reader(void *)
{
        while (1)
        {
           int t;
           pthread_mutex_lock(&mx);
           while (!full)
           {
              pthread_cond_wait(&cond, &mx);
            }
          t=data;
          full=0;
          pthread_cond_signal(&mx);
          pthread_mutex_unlock(&mx);
          read_from _buffer();
        }
return NULL;
}
…
```

3.4 ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическим материалом.
2. Разработать три многопоточные программы с использованием минимум двух потоков и различных средств синхронизации. Например: два потока записывают и читают информацию из одного файла; два потока увеличивают значение общей переменной; два потока с различной частотой считывают и записывают данные в общий буфер памяти.
3. Необходимо обеспечить синхронизированную работу потоков в критической секции с использованием:

- мьютексов;
- семафоров;
- условных переменных.

4. Убедиться в результативности применения средств синхронизации потоков, сравнив результаты работы программ с использованием и без использования средств синхронизации.

3.5 ТРЕБОВАНИЯ К ОТЧЕТУ

Отчет должен содержать следующие разделы:

1. Титульный лист, оформленный согласно утвержденному образцу.
2. Цели выполняемой лабораторной работы.
3. Задание на лабораторную работу.
4. Исходные тексты созданных программ.
5. Результаты работы программ с использованием средств синхронизации. Результаты работы программ без использования средств синхронизации.
6. Выводы (с пояснением различий результатов работы программ при использовании и без использования средств синхронизации).

4 ЛАБОРАТОРНАЯ РАБОТА №4. ПРАКТИЧЕСКОЕ ЗНАКОМСТВО С ПРОЦЕССАМИ, ПЕРЕДАЧЕЙ ДАННЫХ МЕЖДУ ПРОЦЕССАМИ И ИХ СИНХРОНИЗАЦИЕЙ

4.1 ЦЕЛЬ РАБОТЫ

Практическое знакомство с объектом процесс, основными механизмами передачи данных между процессами, а также синхронизацией взаимодействующих процессов в ОС *Unix*.

4.2 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Изучить базовые возможностей оболочки *bash* ОС *Unix* по управлению процессами (заданиями). Разработать приложения реализующие схему «клиент»-«сервер» с использованием средств межпроцессного взаимодействия: семафоров, разделяемой памяти, программных каналов и одной очереди сообщений.

4.3 ПРОЦЕССЫ И МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ

4.3.1 Понятие процесса

При описании общих свойств ОС в некотором общеупотребительном, смысле, часто применяют слова «программа» и «задание»: «…вычислительная система исполняет одну или несколько программ, ОС планирует задания, программы могут обмениваться данными и т.д.». При этом одни и те же слова обозначали и объекты в статическом состоянии, не обрабатывающиеся вычислительной системой (например, совокупность файлов на диске), и объекты в динамическом состоянии, находящиеся в процессе исполнения. Термины «программа» и «задание» предназначены для описания статических, неактивных объектов. Программа же в процессе исполнения является динамическим, активным объектом.

Для более детального ознакомления с особенностями функционирования современных компьютерных систем необходимо иметь более точную и однозначную терминологию.

Не существует взаимно-однозначного соответствия между процессами и программами:

- в некоторых ОС для работы программ может организовываться более одного процесса;

- один и тот же процесс может исполнять последовательно несколько различных программ.

Итак, понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т.д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением ОС.

Изменением состояния процессов занимается ОС, совершая операции над ними.

Основные операции над процессами удобно объединить в три пары:

- создание процесса – завершение процесса (одноразовые);
- приостановка процесса (перевод из состояния исполнение в состояние готовность) – запуск процесса (перевод из состояния готовность в состояние исполнение);
- блокирование процесса (перевод из состояния исполнение в состояние ожидание) – разблокирование процесса (перевод из состояния ожидание в состояние готовность).

Необходимо помнить, что существует еще одна (непарная) операция: изменение приоритета процесса.

У процесса выделяют следующие контексты:

1) регистровый (содержимое всех регистров процессора);
2) системный (запись в таблице процессов, управляющая информация о процессе и пр.);
3) пользовательский (код и данные).

Совокупность всех вышеуказанных контекстов называют просто контекстом процесса, в любой момент полностью характеризующим процесс.

4.3.2 Межпроцессное взаимодействие

Для повышения эффективности функционирования вычислительной системы обеспечивают два вида взаимодействия процессов:

- псевдопараллельное (исполнение на одной вычислительной системе);
- параллельное (исполнение на разных вычислительных системах).

Существуют различные причины кооперации процессов:

- повышение скорости работы (один процесс в ожидании, другой выполняет полезную работу, направленную на решение общей задачи);
- совместное использование данных (использование различными процессами одной и той же динамической базы данных или разделяемого файла);
- модульная конструкция какой-либо системы (например, микроядерный способ построения ОС, когда взаимодействие процессов осуществляется путем передачи сообщений через микроядро);
- для удобства работы пользователя (например, при одновременном редактировании и отладке программы, процессы редактора и отладчика должны взаимодействовать).

Различают два вида процессов:

- кооперативные (влияют на взаимное поведение путем обмена информацией);
- независимые (деятельность процессов остается неизменной при любой принятой информации).

По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории:

1) сигнальные;
2) канальные;
3) разделяемая память.

В случае *сигнального* обмена передается минимальное количество информации, достаточное для извещения процесса о наступлении события.

При *канальном* обмене информацией «общение» процессов происходит через линии связи, предоставленные ОС. Объем передаваемой информации в этом случае в единицу времени ограничен пропускной способностью линий связи.

При использовании процессами *разделяемой памяти* совместно используется некоторая область адресного пространства, формируемая ОС. Этот способ обмена информацией представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе, но требует при использовании повышенной осторожности.

Различают два способа адресации при обмене информацией между процессами:

- *прямую* (процессы осуществляют операции обмена данными явно указывая имя или номер этих процессов);

- *непрямую* (данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных с адресом, откуда они затем могут быть изъяты каким-либо другим процессом).

Прямая адресация может быть двух типов:

- *симметричной*: процессы (принимающие и передающие данные) указывают имена своих партнеров по взаимодействию, при этом ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные;
- *асимметричной*: только один из взаимодействующих процессов указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе.

Следует выделить две различные модели передачи данных по каналам связи:

- *поток ввода-вывода* (не важна структура данных, не осуществляется их интерпретация; процесс, прочитавший 100 байт из линии связи, не знает, были ли они переданы одновременно, пришли от одного процесса или от разных. Примером такой модели является «*pipe*» (*пайп или канал*));
- *сообщения* (на передаваемые данные налагается некоторая структура, весь поток информации разделяется на отдельные сообщения, вводя между данными, по крайней мере, границы сообщений).

Наиболее простой вариант пайпа (канала), а именно неименованный канал, создает оболочка *Unix* (например, *bash*) между программами, запускаемыми из командной строки, разделенными символом «|». Например, командная строка

```
dmesg | less
```

создает канал от программы *dmesg* к *less*, выводящей отладочные сообщения ядра при загрузке, к программе постраничного просмотра *less*.

4.3.3 Основы оперирования процессами в оболочке bash

Задания и процессы

Многие командные оболочки (включая *bash* и *tcsh*) имеют функции управления заданиями (*job control*). Управление заданиями позволяет запускать одновременно несколько команд или заданий (*job*s) и

осуществлять управление ими. Прежде чем говорить об этом более подробно, следует рассмотреть понятие процесс (*process*).

Каждый раз при запуске программы стартует некоторый процесс. Вывести список протекающих в настоящее время процессов можно командой *ps*, например, следующим образом:

```
/home/larry# ps
  PID TT STAT  TIME COMMAND
   24  3 S     0:03 (bash)
  161  3 R     0:00 ps
/home/larry#
```

Номера процессов (*process ID*, или *PID*), указанные в первой колонке, являются уникальными номерами, которые система присваивает каждому работающему процессу. Последняя колонка, озаглавленная *COMMAND*, указывает имя работающей команды. В данном случае в списке указаны процессы, которые запустил сам пользователь *larry*. В системе работает еще много других процессов, их полный список можно выдать командой *ps -aux*. Однако среди команд, запущенных пользователем *larry*, есть только *bash* (командная оболочка для пользователя *larry*) и сама команда *ps*. Видно, что оболочка *bash* работает одновременно с командой *ps*. Когда пользователь ввел команду *ps*, оболочка *bash* начала ее исполнять. После того, как команда *ps* закончила свою работу (таблица процессов выведена на экран), управление возвращается процессу *bash*. Тогда оболочка *bash* выводит на экран приглашение и ждет новой команды.

Работающий процесс также называют заданием (*job*). Здесь и далее не будем делать различия между этими понятиями. Следует отметить, что обычно процесс называют «заданием», когда имеют ввиду управление заданием (*job control*). Управление заданием – это функция командной оболочки, которая предоставляет пользователю возможность переключаться между несколькими заданиями.

В большинстве случаев пользователи запускают только одно задание – это будет та команда, которую они ввели последней в командной оболочке. Однако, используя свойство управления заданиями, можно запустить сразу несколько заданий и, по мере надобности, переключаться между ними.

Управление заданиями может быть полезно, если, например, вы редактируете большой текстовый файл и хотите временно прервать редактирование, чтобы сделать какую-нибудь другую операцию. С помощью функций управления заданиями можно временно покинуть

редактор, вернуться к приглашению командной оболочки и выполнить какие-либо другие действия. Когда они будут сделаны, можно вернуться обратно к работе с редактором и обнаружить его в том же состоянии, в котором он был покинут. У функций управления заданиями есть еще много полезных применений.

Передний план и фоновый режим

Задания могут быть либо на переднем плане (*foreground*), либо фоновыми (*background*). На переднем плане в любой момент времени может быть только одно задание. Задание на переднем плане – это то задание, с которым происходит взаимодействие пользователя; оно получает ввод с клавиатуры и посылает вывод на экран (если ввод или вывод не перенаправили куда-либо еще). Напротив, фоновые задания не получают ввода с терминала; как правило, такие задания не нуждаются во взаимодействии с пользователем.

Некоторые задания исполняются очень долго и во время их работы не происходит ничего интересного. Пример таких заданий – компилирование программ, а также сжатие больших файлов. Нет никаких причин смотреть на экран и ждать, когда эти задания выполнятся. Такие задания следует пускать в фоновом режиме. В это время можно работать с другими программами.

Задания также можно (временно) приостанавливать (*suspend*). Затем приостановленному заданию можно дать указание продолжать работу на переднем плане или в фоновом режиме. При возобновлении исполнения приостановленного задания его состояние не изменяется – задание продолжает выполняться с того места, где его остановили.

Прерывание задания – действие отличное от приостановки задания. При прерывании (*interrupt*) задания процесс погибает. Прерывание заданий обычно осуществляется нажатием соответствующей комбинации клавиш обычно: это *Ctrl-C*. Восстановить прерванное задание никаким образом невозможно. Следует также знать, что некоторые программы перехватывают команду прерывания, так что нажатие комбинации клавиш *Ctrl-C* может не прервать процесс немедленно. Это сделано для того, чтобы программа могла уничтожить следы своей работы прежде, чем она будет завершена. На практике, некоторые программы прервать таким способом нельзя.

Перевод заданий в фоновый режим и уничтожение заданий

Начнем с простого примера. Рассмотрим команду *yes*, которая на первый взгляд покажется бесполезной. Эта команда посылает

бесконечный поток строк, состоящих из символа «*y*» в стандартный вывод. Попробуем, как работает эта команда:

```
/home/larry# yes
y
y
y
y
y
```

Последовательность таких строк будет бесконечно продолжаться. Уничтожить этот процесс можно нажатием клавиши прерывания, которая обычно является комбинацией *Ctrl-C*. Поступим теперь иначе. Чтобы на экран не выводилась эта бесконечная последовательность перенаправим стандартный вывод команды *yes* на */dev/null*. Как отмечалось выше, устройство */dev/null* действует как «черная дыра»: все данные, посланные в это устройство, пропадают. С помощью этого устройства очень удобно избавляться от слишком обильного вывода некоторых программ.

```
/home/larry# yes > /dev/null
```

Теперь на экран ничего не выводится. Однако и приглашение командной оболочки также не возвращается. Это происходит потому, что команда *yes* все еще работает и посылает свои сообщения, состоящие из букв *y* на */dev/null*. Уничтожить это задание также можно нажатием клавиш прерывания.

Допустим теперь, что вы хотите, чтобы команда *yes* продолжала работать, но при этом и приглашение командной оболочки должно вернуться на экран, так чтобы вы могли работать с другими программами. Для этого можно команду *yes* перевести в фоновый режим, и она будет там работать, не общаясь с вами.

Один способ перевести процесс в фоновый режим – приписать символ «*&*» к концу команды. Пример:

```
/home/larry# yes > /dev/null &
\verb+[1] 164+
/home/larry#
```

Как видно, приглашение командной оболочки опять появилось. Однако, что означает «*[1] 164*»? И действительно ли команда *yes* продолжает работать?

Сообщение «*[1]*» представляет собой номер задания (*job number*) для процесса *yes*. Командная оболочка присваивает номер задания каждому исполняемому заданию. Поскольку *yes* является единственным исполняемым заданием, ему присваивается номер 1. Число «*164*» является идентификационным номером, соответствующим данному процессу (*PID*), и этот номер также дан процессу системой. Как мы увидим дальше, к процессу можно обращаться, указывая оба этих номера.

Итак, теперь у нас есть процесс команды *yes*, работающий в фоне, и непрерывно посылающий поток из букв *y* на устройство */dev/null*. Для того, чтобы узнать статус этого процесса, нужно исполнить команду *jobs*, которая является внутренней командой оболочки.

```
/home/larry# jobs
[1]+  Running              yes >/dev/null  &-
/home/larry#
```

Мы видим, что эта программа действительно работает. Для того, чтобы узнать статус задания, можно также воспользоваться командой *ps*, как это было показано выше.

Для того, чтобы прервать работу задания, используется команда *kill*. В качестве аргумента этой команде дается либо номер задания, либо *PID*. В рассмотренном выше случае номер задания был 1, так что команда

```
/home/larry# kill %1
```

прервет работу задания. Когда к заданию обращаются по его номеру (а не PID), тогда перед этим номером в командной строке нужно поставить символ процента («*%*»).

Теперь введем команду *jobs* снова, чтобы проверить результат предыдущего действия:

```
/home/larry# jobs
[1]+  Terminated           yes >/dev/null
/home/larry#
```

Фактически задание уничтожено, и при вводе команды *jobs* следующий раз на экране о нем не будет никакой информации.

Уничтожить задание можно также, используя идентификационный номер процесса (*PID*). Этот номер, наряду с идентификационным номером задания, указывается во время старта задания. В нашем примере значение *PID* было 164, так что команда

```
/home/larry# kill 164
```

была бы эквивалентна команде

/home/larry# kill %1

При использовании *PID* в качестве аргумента команды *kill* вводить символ «*%*» не требуется.

Приостановка и продолжение работы заданий

Предложим еще один метод, с помощью которого процесс можно перевести в фоновый режим. Процесс запускается обычным образом (на переднем плане), затем приостанавливается командой *stop*, а потом запускается повторно в фоновом режиме.

Запустим сначала процесс командой *yes* на переднем плане, как это делалось раньше:

/home/larry# yes > /dev/null

Как и ранее, поскольку процесс работает на переднем плане, приглашение командной оболочки на экран не возвращается.

Теперь вместо того, чтобы прервать задание комбинацией клавиш *Ctrl-C*, задание можно приостановить (*suspend*, буквально – «подвесить»). «Подвешенное» задание не будет уничтожено; его выполнение будет временно остановлено до тех пор, пока оно не будут возобновлено. Для приостановки задания надо нажать соответствующую комбинацию клавиш, обычно это *Ctrl-Z*.

```
/home/larry# yes > /dev/null
{ctrl-Z}
[1]+  Stopped               yes >/dev/null
/home/larry#
```

Приостановленный процесс попросту не выполняется. На него не тратятся вычислительные ресурсы процессора. Приостановленное задание можно запустить выполняться с той же точки, как будто бы оно и не было приостановлено.

Для возобновления выполнения задания на переднем плане можно использовать команду *fg* (от слова «*foreground*» – передний план).

```
/home/larry# fg
yes >/dev/null
```

Командная оболочка еще раз выведет на экран название команды, так что пользователь будет знать, какое именно задание он в данный момент запустил на переднем плане. Приостановим это задание еще раз нажатием клавиш *Ctrl-Z*, но в этот раз запустим его в фоновый режим командой *bg* (от слова «*background*» – фон). Это приведет к тому, что

данный процесс будет работать так, как если бы при его запуске использовалась команда с символом «*&*» в конце (как это делалось в предыдущем разделе):

```
/home/larry# bg
[1]+ yes &>/dev/null  &
/home/larry#
```

При этом приглашение командной оболочки возвращается. Сейчас команда *jobs* должна показывать, что процесс *yes* действительно в данный момент работает; этот процесс можно уничтожить командой *kill*, как это делалось раньше.

Для того, чтобы приостановить задание, работающее в фоновом режиме, нельзя пользоваться комбинацией клавиш *Ctrl-Z*. Прежде, чем приостанавливать задание, его нужно перевести на передний план командой *fg* и лишь потом приостановить. Таким образом, команду *fg* можно применять либо к приостановленным заданиям, либо к заданию, работающему в фоновом режиме.

Между заданиями в фоновом режиме и приостановленными заданиями есть большая разница. Приостановленное задание не работает – на него не тратятся вычислительные мощности процессора. Это задание не выполняет никаких действий. Приостановленное задание занимает некоторый объем оперативной памяти компьютера, хотя оно может быть перенесено в «своп». Напротив, задание в фоновом режиме выполняется, использует память и совершает некоторые действия, которые, возможно, требуются пользователю, но он в это время может работать с другими программами.

Задания, работающие в фоновом режиме, могут пытаться выводить некоторый текст на экран. Это будет мешать работать над другими задачами. Например, если ввести команду

```
/home/larry# yes &
```

(стандартный вывод не был перенаправлен на устройство */dev/null*), то на экран будет выводится бесконечный поток символов *y*. Этот поток невозможно будет остановить, поскольку комбинация клавиш *Ctrl-C* не воздействует на задания в фоновом режиме. Для того, чтобы остановить эту выдачу, надо использовать команду *fg*, которая переведет задание на передний план, а затем уничтожить задание комбинацией клавиш *Ctrl-C*.

Сделаем еще одно замечание. Обычно командой *fg* и командой *bg* воздействуют на те задания, которые были приостановлены последними (эти задания будут помечены символом «+» рядом с номером задания,

если ввести команду *jobs*). Если в одно и то же время работает одно или несколько заданий, задания можно помещать на передний план или в фоновый режим, задавая в качестве аргументов команды *fg* или команды *bg* их идентификационный номер (*job ID*). Например, команда

```
/home/larry# fg %2
```

помещает задание номер 2 на передний план, а команда

```
/home/larry# bg %3
```

помещает задание номер 3 в фоновый режим. Использовать *PID* в качестве аргументов команд *fg* и *bg* нельзя. Более того, для перевода задания на передний план можно просто указать его номер. Так команда

```
/home/larry# %2
```

будет эквивалентна команде

```
/home/larry# fg %2
```

Важно помнить, что функция управления заданием принадлежит оболочке. Команды *fg*, *bg* и *jobs* являются внутренними командами оболочки.

4.3.4 Механизмы межпроцессного взаимодействия в ОС Unix

При решении задачи синхронизации процессов и их взаимодействия посредством различных механизмов, предоставляемых ОС, может потребоваться использование следующих системных вызовов:

- создание, завершение процесса, получение информации о процессе: *fork()*, *exit()*, *getpid()*, *getppid()* и т.д.;
- синхронизация процессов: *signal()*, *kill()*, *sleep()*, *alarm()*, *wait()*, *pause()*, *semop()*, *semctl()*, *semcreate()* и т.д.;
- создание информационного канала, разделяемой памяти, очереди сообщений и работа с ними: *pipe()*, *mkfifo()*, *read()*, *write()*, *msgget()*, *shmget()*, *msgctl()*, *shmctl()* и т.д.

Механизм межпроцессного взаимодействия (*Inter-Process Communication Facilities – IPC*) включает:

- средства, обеспечивающие возможность синхронизации процессов при доступе к совместно используемым ресурсам – *семафоры* (*semaphores*);
- средства, обеспечивающие возможность посылки процессом сообщений другому произвольному процессу – *очереди сообщений* (*message queries*);

- средства, обеспечивающие возможность наличия общей для процессов памяти – *сегменты разделяемой памяти* (*shared memory segments*);
- средства, обеспечивающие возможность «общения» процессов, как родственных так и нет, через пайпы или каналы (*pipes*).

Наиболее общим понятием *IPC* является ключ, хранимый в общесистемной таблице и обозначающий объект межпроцессного взаимодействия, доступный нескольким процессам. Обозначаемый ключом объект может быть очередью сообщений, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип *keyt*, состав которого зависит от реализации и определяется в файле *<sys/types.h>*. Ключ используется для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту.

Семафоры

Для работы с семафорами поддерживаются три системных вызова:

- *semget()* для создания и получения доступа к набору семафоров;
- *semop()* для манипулирования значениями семафоров (системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров);
- *semctl()* для выполнения разнообразных управляющих операций над набором семафоров.

Прототипы перечисленных системных вызовов описаны в файлах

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

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

Системный вызов *semget()* имеет следующий синтаксис:

```
int semid = semget (key_t key, int count, int flag);
```

параметрами которого является ключ или уникальное имя сегмента (*key*), набора семафоров и дополнительные флаги (*flag*), определенные в *<sys/ipc.h>*, число семафоров в наборе семафоров (*count*), обладающих одним и тем же ключом. Системный вызов возвращает идентификатор набора семафоров *semid*. Живучесть такого семафора определяется живучестью ядра, т.е. объект семафор будет уничтожен тогда, и только тогда, когда произойдет перезагрузка ядра либо его принудительно удалят. После вызова *semget()* индивидуальный семафор идентифицируется идентификатором набора семафоров и номером семафора в этом наборе. Флаги системного вызова *semget()* приведены ниже в таблице:

Таблица 4.1

Флаги системного вызова *semget()*

Флаг	Описание
IPC_CREAT	вызов semget создает новый семафор для данного ключа. Если флаг *IPC_CREAT* не задан, а набор семафоров с указанным ключом уже существует, то обращающийся процесс получит идентификатор существующего набора семафоров.
IPC_EXLC	флаг *IPC_EXLC* вместе с флагом *IPC_CREAT* предназначен для создания (и только для создания) набора семафоров. Если набор семафоров уже существует, *semget* возвратит -1, а системная переменная errno будет содержать значение *EEXIST*.

Младшие 9 бит флага задают права доступа к набору семафоров, флаги прав доступа приведены в таблице:

Таблица 4.2

Константы режима доступа при создании нового объекта *IPC*

Константа	Описание
S_IRUSR	Владелец – чтение
S_IWUSR	Владелец – запись
S_IRGRP	Группа – чтение
S_IWGRP	Группа – запись
S_IROTH	Прочие – чтение
S_IWOTH	Прочие – запись

Таким образом, флаг создания семафора можно указать так:

```
int flag = S_IRUSR | S_IWUSR | S_IRGRP | IPC_CREAT;
```

Системный вызов *semctl()* имеет формат

```
int semctl (int semid, int sem_num, int command, union semun arg);
```

где *semid* – это идентификатор набора семафоров, *sem_num* – номер семафора в группе, *command* – код операции, а *arg* – указатель на структуру, содержимое которой интерпретируется по разному, в зависимости от операции.

Объединение имеет вид:

```
union semun
{
int val; /* устанавливает значение семафора только для SETVAL */
struct semid_ds *buf;
```

```
            /* используется командами IPC_STAT и IPC_SET */
    unsigned short *array; /* используется командами SETALL и GETALL */
};
```

Объединение *semun* всегда должен быть переопределен в глобальной секции программы. Структура *semid_ds* выглядит следующим образом:

```
struct semid_ds {
    struct ipc_perm sem_perm; /* разрешения на операции */
    struct sem *sem_base; /* указатель на массив семафоров в наборе */
    ushort sem_nsems;              /* количество семафоров */
    time_t sem_otime; /* время последнего вызова semop() */
    time_t sem_ctime; /* время создания последнего IPC_SET */
};
```

Вызов *semctl ()* позволяет:

- уничтожить набор семафоров или индивидуальный семафор в указанной группе (*IPCRMID*);
- вернуть значение отдельного семафора (*GETVAL*) или всех семафоров (*GETALL*);
- установить значение отдельного семафора (*SETVAL*) или всех семафоров (*SETALL*);
- вернуть число семафоров в наборе семафоров (*GETPID*).

Основным системным вызовом для манипулирования семафором является

```
int semop (int semid, struct sembuf *op_array, int count);
```

где *semid* – это ранее полученный дескриптор группы семафоров, *op_array* – массив структур *sembuf*:

```
struct sembuf {
    short sem_num;  /* номер семафора: 0,1,2..n */
    short sem_op;   /* операция с семафором */
    short sem_flg;  /* флаги операции: 0, IPC_NOWAIT, SEM_UNDO */
};
```

определенных в файле *<sys/sem.h>* и содержащих описания операций над семафорами группы, а *count* – количество элементов массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора.

Если указанные в массиве *op_array* номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов последовательно меняет значение семафора (если это возможно) в соответствии со значением поля «операция». Возможны три случая:

1. Отрицательное значение *sem_op*.
2. Положительное значение *sem_op*.
3. Нулевое значение *sem_op*.

Если значение поля операции *sem_op* отрицательно, и его абсолютное значение меньше или равно значению семафора *semval*, то ядро прибавляет это отрицательное значение к значению семафора. Если в результате значение семафора стало нулевым, то ядро активизирует все процессы, ожидающие нулевого значения этого семафора. Если же значение поля операции *sem_op* по абсолютной величине больше семафора *semval*, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и усыпляет текущий процесс до наступления этого события.

Если значение поля операции *sem_op* положительно, то оно прибавляется к значению семафора *semval*, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются в терминологии *Unix*).

Если значение поля операции *sem_op* равно нулю и значение семафора *semval* также равно нулю, выбирается следующий элемент массива *op_array*. Если же значение семафора *semval* отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания. При использовании флага *IPCNOWAIT* ядро ОС *Unix* не блокирует текущий процесс, а лишь сообщает в ответных параметрах о возникновении ситуации, приведшей к блокированию процесса при отсутствии флага *IPCNOWAIT*.

Именованные и неименованные каналы (пайпы)

Операционные системы семейства Unix всегда поддерживают два типа однонаправленных каналов:

- неименованные каналы;
- именованные каналы *FIFO*.

Неименованные каналы – это самая первая форма *IPC* в *Unix* (1973), главным недостатком которых является отсутствие имени, вследствие чего они могут использоваться для взаимодействия только родственными процессами. В *Unix System* третьей редакции (1982) были добавлены каналы *FIFO,* которые называются именованными каналами. Аббревиатура *FIFO* расшифровывается как «*first* in, *first out*» – «первым

вошел, первым вышел», то есть эти каналы работают как очереди. Именованные каналы в *Unix* функционируют подобно неименованным – они позволяют передавать данные только в одну сторону. Однако в отличие от программных каналов каждому каналу *FIFO* сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же *FIFO*. Доступ и к именованным каналам, и к неименованным организуется с помощью обычных функций *read()* и *write()*.

FIFO создается вызовом *mkfifo()*:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
/* возвращает 0 при успешном выполнении, -1 при ошибке */
```

Здесь *pathname* – обычное для *Unix* полное имя файла, которое и будет именем *FIFO*.

Аргумент *mode* указывает битовую маску разрешений доступа к файлу (табл. 4.2), аналогично второму аргументу команды *open*. В табл. 4.2 приведены шесть констант, определенных в заголовке *<sys/stat.h>*. Эти константы могут использоваться для задания разрешений доступа и к *FIFO*.

Функция *mkfifo* действует как *open*, вызванная с аргументом *mode=O_CREAT | O_EXCL*. Это означает, что создается новый канал *FIFO* или возвращается ошибка *EEXIST*, в случае если канал с заданным полным именем уже существует. Если не требуется создавать новый канал, вызывайте *open* вместо *mkfifo*. Для открытия существующего канала или создания нового в том случае, если его еще не существует, вызовите *mkfifo*, проверьте, не возвращена ли ошибка *EEXIST*, и если такое случится, вызовите функцию *open*.

Команда *mkfifo* также создает канал *FIFO*. Ею можно пользоваться в сценариях интерпретатора или из командной строки.

Живучесть каналов определяется живучестью процессов, т.е. канал будет существовать до тех пор, пока он не будет принудительно закрыт либо не останется ни одного процесса работающего с каналом.

После создания канал *FIFO* должен быть открыт на чтение или запись с помощью либо функции *open*, либо одной из стандартных функций открытия файлов из библиотеки ввода-вывода (например, *fopen*). *FIFO* может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись, поскольку именованные каналы могут быть только односторонними. Например,

случай взаимодействия между двумя процессами может представить следующим образом (рис. 4.1):

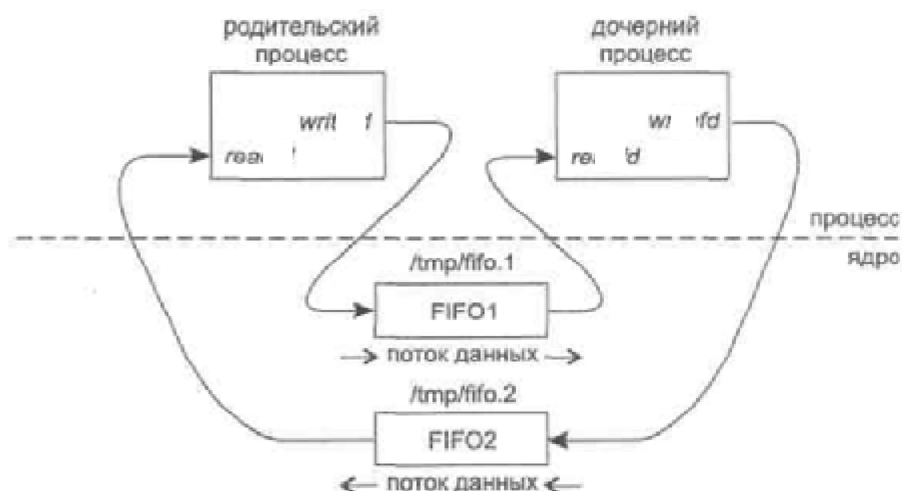


Рис. 4.1 Взаимодействие двух процессов посредством каналов FIFO

При записи в программный канал или канал *FIFO* вызовом *write* данные всегда добавляются к уже имеющимся, а вызов *read* считывает данные, помещенные в программный канал или *FIFO* первыми. При вызове функции *lseek* для программного канала или *FIFO* будет возвращена ошибка *ESPIPE*.

Неименованные каналы создаются вызовом *pipe()* и предоставляют возможность только однонаправленной (односторонней) передачи данных:

```
#include <unistd.h>
int fd[2];
pipe(fd);
/* возвращает 0 в случае успешного завершения, -1 - в случае ошибки;*/
```

Функция возвращает два файловых дескриптора: *fd[0]* и *fd[l]*, причем первый открыт для чтения, а второй – для записи.

Хотя канал создается одним процессом (рис. 4.2), он редко используется только этим процессом, каналы обычно используются для связи между двумя процессами (родительским и дочерним) следующим образом: процесс создает канал, а затем вызывает *fork*, создавая свою копию – дочерний процесс (рис. 4.3). Затем родительский процесс закрывает открытый для чтения конец канала, а дочерний, в свою очередь, – открытый на запись конец канала (рис. 4.4). Это обеспечивает одностороннюю передачу данных между процессами (рис. 4.5)

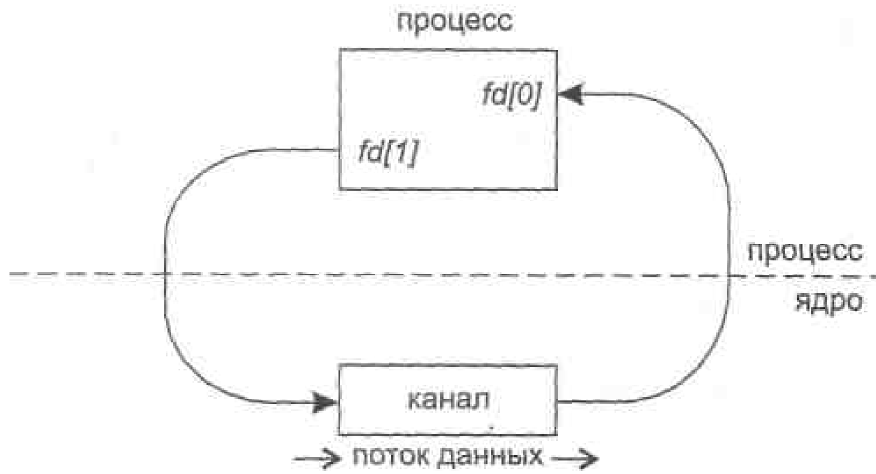


Рис. 4.2 Функционирование канала для случая одиночного процесса

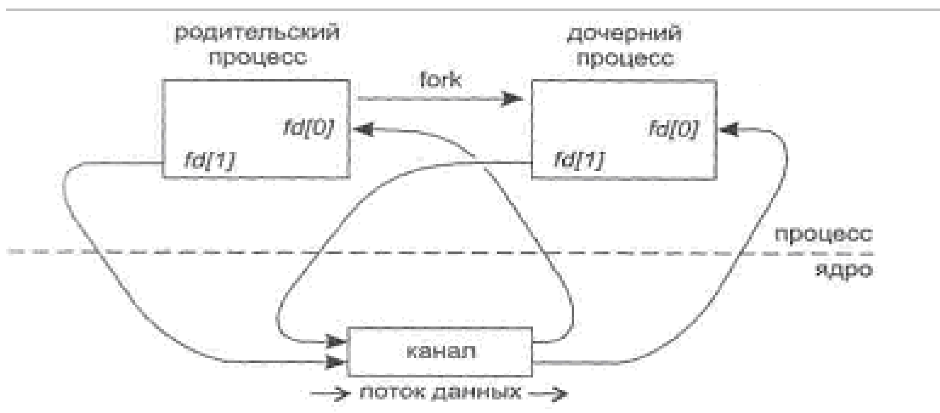


Рис. 4.3 Функционирование канала после создания дочернего процесса (после вызова fork)

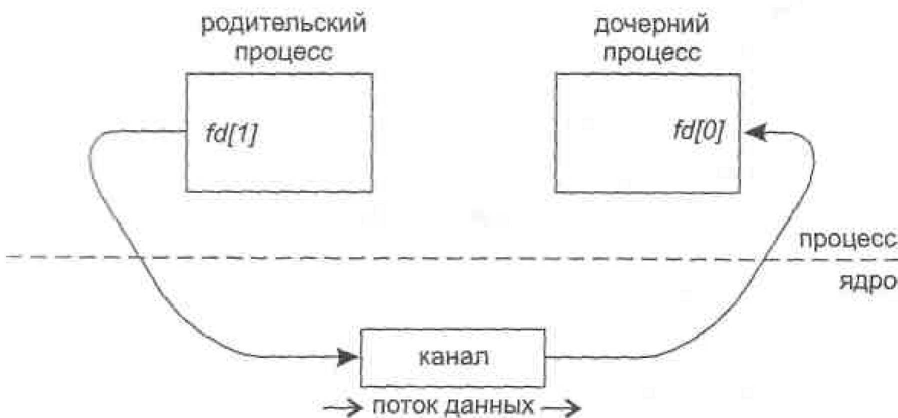


Рис. 4.4 Функционирование канала между двумя процессами

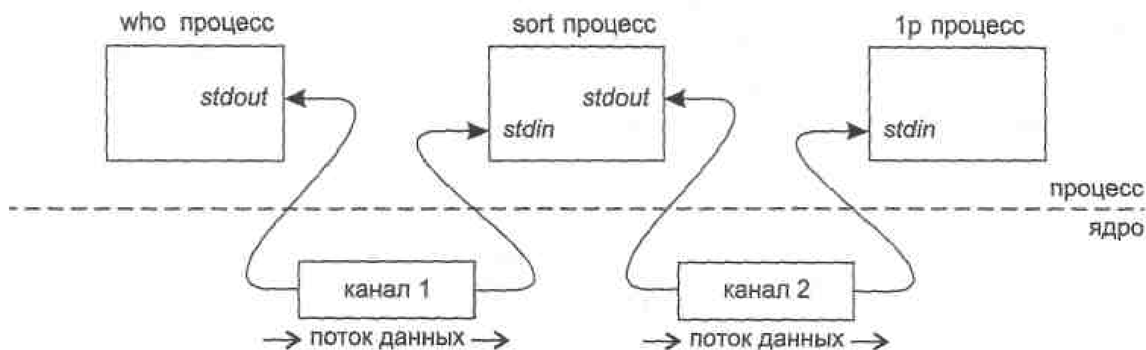


Рис. 4.5 Функционирование каналов между тремя процессами в конвейерной обработке

При вводе команды типа:

who | sort | 1p

интерпретатор команд *Unix* выполняет вышеописанные действия для создания трех процессов с двумя каналами между ними (рис. 4.5).

Интерпретатор также подключает открытый для чтения конец каждого канала к стандартному потоку ввода, а открытый на запись – к стандартному потоку вывода.

Все рассмотренные выше неименованные каналы были однонаправленными (односторонними), то есть позволяли передавать данные только в одну сторону. При необходимости передачи данных в обе стороны нужно создавать пару каналов и использовать каждый из них для передачи данных в одну сторону. Этапы создания двунаправленного неименованного канала *IPC* следующие:

- создаются каналы 1 (*fd1[0]* и *fd1[1]*) и 2 (*fd2[0]* и *fd2[0]*);
- вызов *fork();*
- родительский процесс закрывает доступный для чтения конец канала 1 (*fd1[0]*);
- родительский процесс закрывает доступный для записи конец канала 2 (*fd2[1]*);
- дочерний процесс закрывает доступный для записи конец канала 1 (*fd1[1]*);
- дочерний процесс закрывает доступный для чтения конец канала 2 (*fd2[0]*).

4.3.5 Очереди сообщений

Для обеспечения возможности обмена сообщениями между процессами механизм очередей поддерживается следующими системными вызовами:

- *msgget()* для образования новой очереди сообщений или получения дескриптора существующей очереди;
- *rnsgsnd()* для постановки сообщения в указанную очередь сообщений;
- *rnsgrcv()* для выборки сообщения из очереди сообщений;
- *rnsgctl()* для выполнения ряда управляющих действий.

Прототипы перечисленных системных вызовов описаны в файлах

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

По системному вызову *msgget()* в ответ на ключ (*key*), определяющий уникальное имя очереди, и набор флагов (полностью аналогичны флагам в системном вызове *semget()*). Вызовом *msgget()* ядро, либо создает новую очередь сообщений в ядре и возвращает пользователю идентификатор созданной очереди, либо находит элемент таблицы очередей сообщений ядра, содержащий указанный ключ, и возвращает соответствующий идентификатор очереди:

```
int msgqid = msgget(key_t   key,   int   flag).
```

Таким образом, очередь сообщения обладает живучестью ядра. Для помещения сообщения в очередь служит системный вызов *msgsnd()*:

```
int rnsgsnd (int msgqid, void *rnsg, size_t size, int flaq),
```

где *msg* – это указатель на структуру длиной *size*, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив-сообщение, причем размер пользовательских данных вычисляется следующим образом: *size = sizeof(msg) – sizeof(long)*. Структура *msg* всегда имеет вид:

```
struct  rnsq {
lonq rntype; /*   тип   сообщения   */
char mtext[SOMEVALUE]; /*текст сообщения  */
};
```

Поле типа *long* всегда должно быть первым в структуре, далее могут следовать в любом порядке пользовательские данные, в этом случае ядро не накладывает ограничение на тип данных, только на их длину (зависящую от реализации системы). Параметр *flag* определяет действия ядра для вызвавшего потока при чтении очереди или выходе за пределы допустимых размеров внутренней буферной памяти. Если *flag = 0*, то при отсутствии сообщения в очереди поток блокируется.

Если *flag* = *IPCNOWAIT*, то поток не блокируется и при отсутствии сообщения возвращается ошибка *ENOMSG*.

Условиями успешной постановки сообщения в очередь являются:

- наличие прав процесса по записи в данную очередь сообщений;
- не превышение длиной сообщения заданного системой верхнего предела;
- положительное значение указанного в сообщении типа сообщения.

Если же оказывается, что новое сообщение невозможно буферизовать в ядре по причине превышения верхнего предела суммарной длины сообщений, находящихся в данной очереди сообщений (флаг *IPCNOWAIT* при этом отсутствует), то обратившийся процесс откладывается (усыпляется) до тех пор, пока очередь сообщений не разгрузится процессами, ожидающими получения сообщений или очередь не будет удалена или вызвавшей поток не будет прерван перехватываемым сигналом.

Для приема сообщения используется системный вызов *msgrcv()*:

```
int rnsgrcv (int msgqid, void *msg, size_t size, long rnsg_type, int flag);
```

Аргумент *rnsg_type* задает тип сообщения, которое нужно считать из очереди:

- если значение равно 0, то возвращается первое сообщение в очереди, т.е. самое старое сообщение;
- если тип больше 0, то возвращается первое сообщение, тип которого равен указанному числу;
- если тип меньше нуля, возвращается первое сообщение с наименьшим типом, значение которого меньше либо равно модулю указанного числа.

Значение *size* в данном случае указывает ядру, что возвращаемые данные не должны превышать размера указанного в *size*.

Системный вызов *msgctl()* позволяет управлять очередями сообщений

```
int msgctl (int msgqid, int command, struct msgid_ds *msg_stat);
```

и используется:

- для опроса состояния описателя очереди сообщений (*command* = *IPCSTAT*) и помещения его в структуру *msgstat*;
- изменения его состояния (*command* = *IPCSET*), например, изменения прав доступа к очереди;
- для уничтожения указанной очереди сообщений (*command* = *IPCRMID*).

4.3.6 Работа с разделяемой памятью

Для работы с разделяемой памятью используются системные вызовы:

- *shmget()* создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;
- *shmat()* подключает сегмент с указанным описателем к виртуальной памяти обращающегося процесса;
- *shmdt()* отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;
- *shmctl()* служит для управления разнообразными параметрами, связанными с существующим сегментом.

Прототипы перечисленных системных вызовов описаны в файлах

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

После того, как сегмент разделяемой памяти подключен к виртуальной памяти процесса, этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи.

Системный вызов

```
int shmid = shmget (key_t key, size_t size, int flag);
```

на основании параметра *size* определяет желаемый размер сегмента в байтах. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является идентификатор существующего сегмента, причем параметр *size* должен быть в этом случае равен 0. В противном случае создается новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера. Живучесть объектов разделяемой памяти определяется живучестью ядра. Создание сегмента не означает немедленного выделения под него основной памяти и это действие откладывается до выполнения первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Флаги *IPCCREAT* и *IPCEXCL* аналогичны рассмотренным выше.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову *shmat()*:

```
void *virtaddr = shmat(int shmid, void *daddr, int flags);
```

Параметр *shmid* – это ранее полученный идентификатор сегмента, а *daddr* – желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является фактический виртуальный адрес начала сегмента. Если значением *daddr* является *NULL*, ядро выбирает наиболее удобный виртуальный адрес начала сегмента. Флаги системного вызова *shmat()* приведены ниже в таблице.

Таблица 4.3

Флаги системного вызова *shmat()*

Флаг	Описание
SHM_RDONLY	Ядро подключает участок памяти только для чтения
SHM_RND	Определяет, если возможно, способ обработки ненулевого значения *daddr*.

Для отключения сегмента от виртуальной памяти используется системный вызов *shmdt()*:

```
int shmdt (*daddr);
```

где *daddr* – это виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова *shmat()*.

Системный вызов *shmctl()*:

```
int shmctl (int shmid, int command, struct shmid_ds *shrn_stat);
```

по синтаксису и назначению системный вызов полностью аналогичен *msgctl()*.

4.3.7 Примеры практической реализации

Семафоры

Программа *semsyn*, исходный код которой приведен в листинге 4.1, создает семафор и два процесса, синхронизирующихся с помощью созданного семафора. В программе дочерний процесс является главным, он блокирует и разблокирует семафор, родительский процесс ждет освобождения семафора.

Листинг № 4.1

Программа для синхронизации двух процессов посредством семафора *System V*.

```
#include <unistd.h>
#include <stdio.h>
#include <error.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <fcntl.h>
#include <time.h>
#include <iostream.h>
#define MAXLINE 128
#define SVSEM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define SKEY 1234L // идентификатор семафора
union semun {
   int val;
   struct semid_ds *buf;
   ushort *array;
};
int var;
int main(int argc, char **argv) {
char filename[] = "./rezult.txt";
pid_t pid; // идентификатор дочернего процесса
time_t ctime; // переменная времени
int oflag, c, semid;
struct tm *ctm;
union semun arg;
struct semid_ds seminfo;
struct sembuf psmb;
unsigned short *prt = NULL;
    var = 0;
    oflag = SVSEM_MODE | IPC_CREAT; // флаг семафора
    printf("Parent: Creating semaphore...\n");
    semid = semget(SKEY, 1, oflag); // создание семафора
    arg.buf = &seminfo;
    printf("Parent: Getting info about semaphore (not required, for
example)...\n");
    semctl(semid, 0, IPC_STAT, arg); //получение инф. о семафоре
```

```
	g.buf->sem_ctime;
	ctm = localtime(&ctime);
	printf("%s %d %s %d %s %d %s","Parent: Creating time - ",
	ctm->tm_hour,":",ctm->tm_min,":",ctm->tm_sec,"\n");
	arg.val = 5;
	printf("%s %d %s","Parent: Setting value \"",arg.val, "\" to
semaphores...\n");
	semctl(semid, 0, SETVAL, arg); // установка значения семафора
	printf("Parent: Creating child process...\n");
    if ((pid = fork()) == 0) { // child process ;
		printf("		Child: Child process was created...\n");
		struct sembuf csmb;
		unsigned short semval;
		union semun carg;
		int oflag = SVSEM_MODE | IPC_EXCL;
		printf("		Child: Opening semaphore...\n");
		int smd = semget(SKEY, 1, oflag); // открытие семафора
		csmb.sem_num = 0;
		csmb.sem_flg = 0;
		csmb.sem_op = -1;
		printf("		Child: Locking semaphore...\n");
		semop(smd,&csmb,1); // блокировка семафора
		printf("		Child: Do something...\n");
		// работа процесса в защищенном режиме
		sleep(2);
		// работа процесса в защищенном режиме закончена
		printf("		Child: Done something...\n");
		carg.buf = NULL;
		carg.array = &semval;
		semctl(smd,0,GETALL,carg); // получение значения семафора
		semval = *carg.array;
		printf("%s %d %s","		Child: Semaphore value = ",semval,"\n");
		csmb.sem_num = csmb.sem_flg = 0;
		csmb.sem_op = -semval;
```

```
        printf("            Child: Unlocking semaphore...\n");
        semop(smd,&csmb,1);
        printf("            Child: Terminating child process...\n");
        exit(0);
    }

printf("Parent: Waiting for unlocking semaphore...\n");
psmb.sem_num = psmb.sem_flg = psmb.sem_op = 0;
semop(semid,&psmb,1);
printf("Parent: Semaphore is unlocked...\n");
printf("Parent: Waiting for SIGCHILD...\n");
waitpid(pid,NULL,0);
printf("Parent: Deleting semaphore...\n");
semctl(semid, 0, IPC_RMID);
exit(0);
}
```

Запуск приведенной выше программы происходит следующим образом:

```
semsyn
Parent: Creating semaphore...
Parent: Getting info about semaphore (not required, for example)...
Parent: Creating time -  13 : 14 : 6
Parent: Setting value " 5 " to semaphore...
Parent: Creating child process...
        Child: Child process was created...
        Child: Opening semaphore...
        Child: Locking semaphore...
        Child: Do something...
Parent: Waiting for unlocking semaphore...
        Child: Done something...
        Child: Semaphore value =  4
        Child: Unlocking semaphore...
Parent: Semaphore is unlocked...
```

```
Parent: Waiting for SIGCHILD...
        Child: Terminating child process...
Parent: Deleting semaphore...
```

Во время работы программы создается семафор с живучестью ядра:

```
ipcs –s
------ Semaphore Arrays --------
key       semid     owner     perms     nsems
0x000004d2 425986    root      644       1
```

Разделяемая память

Программа *shmget*, текст которой приведен в листинге 4.2, создает сегмент разделяемой памяти, принимая из командной строки полное имя произвольного файла и длину сегмента.

Листинг № 4.2

Программа для создания сегмента разделяемой памяти *System V* указанного размера.

```
#include <stdio.h>
#include <error.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <stdlib.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
int c, id, oflag;
char *ptr;
size_tlength;
oflag = SVSHM_MODE | IPC_CREAT; // флаг создания семафора
while ( (c = getopt(argc, argv, "e")) != -1) {
    switch (c) { // просмотр ключей командной строки
        case 'e':
        oflag |= IPC_EXCL;
        break;
```

```
        }
    }
    if (optind != argc - 2)
       {
            printf("usage: shmget [ -e ] <path_to_file> <length>");
            return 0;
        }
    length = atoi(argv[optind + 1]);
    id = shmget(ftok(argv[optind], 0), length, oflag);
    ptr = (char*) shmat(id, NULL, 0);
    return 0;
    }
```

Вызов *shmget* создает сегмент разделяемой памяти указанного размера. Полное имя, передаваемое в качестве аргумента командной строки, преобразуется в ключ *IPC System V* вызовом функции *ftok*. Если указан параметр *–e* командной строки и в системе существует сегмент с тем же именем, запуски программы завершатся по ошибке. Если известно, что сегмент уже существует, то в командной строке должна быть указана нулевая длина сегмента памяти.

Вызов *shmat* подключает сегмент к адресному пространству процесса, после чего программа завершает работу. В связи с тем, что разделяемая память *System V* обладает «живучестью ядра», то сегмент разделяемой памяти при этом не удаляется.

В листинге 4.3 приведен текст программы *shmrmid*, которая вызывает функцию *shmctl* с командой *IPC_RMID* для удаления сегмента разделяемой памяти из системы.

Листинг № 4.3

Программа для удаления сегмента разделяемой памяти *System V* из системы.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <error.h>
#include <fcntl.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
```

```
{
int id;
if (argc != 2)
   {
         printf("usage: shmrmid <path_to_file>");
         return 0;
    }
id = shmget(ftok(argv[1], 0), 0, SVSHM_MODE);
shmctl(id, IPC_RMID, NULL);
return 0;
}
```

В листинге 4.4 приведен исходный код программы *shmwrite*, которая заполняет сегмент разделяемой памяти последовательностью значений 0,1, 2,..., 254, 255,0,1 и т.д. Сегмент разделяемой памяти открывается вызовом *shmget* и подключается вызовом *shmat*. Его размер может быть получен вызовом *shmctl* с командой *IPC_STAT*.

Листинг № 4.4

Программа для записи в сегмент разделяемой памяти *System V*.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <error.h>
#include <fcntl.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
    int i, id;
struct shmid_ds buff;
unsigned char *ptr;
if (argc != 2)
   {
         printf("usage: shmwrite <path_to_file>");
         return 0;
   }
```

```
id = shmget(ftok(argv[1], 0), 0, SVSHM_MODE);
ptr = (unsigned char*) shmat(id, NULL, 0);
shmctl(id, IPC_STAT, &buff);
/* 4set: ptr[0] = 0, ptr[1] = 1, etc. */
for (i = 0; i < buff.shm_segsz; i++) *ptr++ = i % 256;
return 0;
}
```

Программа *shmread*, текст которой приведен в листинге 4.5, проверяет последовательность значений, записанную в разделяемую память программой *shmwrite*.

Листинг № 4.5

Программа для проверки значений в сегменте разделяемой памяти.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <error.h>
#include <fcntl.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
int i, id;
struct shmid_ds buff;
unsigned char c, *ptr;
if (argc != 2)
   {
        printf("usage: shmread <path_to_file>");
        return 0;
   }
id = shmget(ftok(argv[1], 0), 0, SVSHM_MODE);
ptr = (unsigned char*) shmat(id, NULL, 0);
shmctl(id, IPC_STAT, &buff);
/* check that ptr[0] = 0, ptr[1] = 1, and so on. */
for (i = 0; i < buff.shm_segsz; i++)
```

```
if ( (c = *ptr++) != (i % 256)) printf("ptr[%d] = %d", i, c);
return 0;
}
```

Рассмотрим результат запуска приведенных выше программ при работе с разделяемой памятью. В начале создается сегмент разделяемой памяти длиной 1234 байта. Для идентификации сегмента используем полное имя исполняемого файла */tmp/test1*. Это имя будет передано функции *ftok*:

```
shmget /tmp/test1 1234
ipcs -bmo
IPC status from <running system> as of Thu Jan 8 13:17:06 1998
T  ID   KEY       MODE      OWNER  GROUP  NATTCH  SEGSZ
Shared Memory:
m  1    0x0000fl2a  --rw-r--r--  rstevens otherl  0    1234
```

Программа *ipcs* запускается для того, чтобы убедиться, что сегмент разделяемой памяти действительно был создан и не был удален по завершении программы *shmcreate*.

Запуская программу *shmwrite* можно заполнить содержимое разделяемой памяти последовательностью значений. Затем с помощью программы *shmread* проверяется содержимое сегмента разделяемой памяти:

```
shmwrite shmget
shmread shmget
shmrmid shmget
ipcs -bmo
IPC status from <running system> as of Thu Jan 8 13:17:06 1998
T  ID   KEY  MODE  OWNER  GROUP  NATTCH  SEGSZ
Shared Memory:
```

Удалить разделяемую память можно вызвав:

```
shmrmid /tmp/test1
```

Программные каналы

В листинге 4.6 приведена программа *pipes*, создающая два процесса и обеспечивающая двустороннюю связь процессов посредством неименованных каналов.

Листинг № 4.6
Программа для взаимодействия родственных процессов посредством неименованных каналов.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <iostream.h>
#include <strings.h>
#include <fstream.h>
#define MAXLINE 128
void server(int,int), client(int,int);
int main(int argc, char **argv) {
int      pipe1[2],pipe2[2]; // идентификаторы каналов
pid_t    childpid = 0;
printf("Parent: Creating pipes...\n");
pipe(pipe1);
pipe(pipe2);
printf("Parent: Pipes created...\n");
printf("Parent: Creating child process...\n");
if ((childpid = fork()) == 0) { // child process starts
   printf("Child:      Child process created...\n");
   close(pipe1[1]);
   close(pipe2[0]);
   printf("Child:      Starting server...\n");
   server(pipe1[0], pipe2[1]);
   printf("Child:      Terminating process...\n");
   exit(0);
   }
   // parent process
   close(pipe1[0]);
   close(pipe2[1]);
   sleep(2);
   printf("Parent:     Starting client...\n");
```

```
   client(pipe2[0],pipe1[1]);
   printf("Parent: Waiting for child porecess to terminate a zombie...\n");
   waitpid(childpid, NULL, 0);
   printf("Parent: Zombie terminated...\n");
   return 0;

}
void server(int readfd, int writefd) {
char str[MAXLINE];
strcpy(str,"some string to transmit");
ssize_t n = 0;
printf("%s %s %s","Child:          Server: Tranferting string
to client - \"",str,"\"\n");
write(writefd, str, strlen(str));
sleep(1);
printf("Child:          Server: Waiting for replay from client...");
while ((n = read(readfd,str,MAXLINE)) > 0)
{
     str[n] = 0;
     printf("%s %s %s","Received OK from client - \"",str,"\"\n");
     break;
}

printf("Child:          Server was terminated...\n");
return;
}

void client(int readfd, int writefd) {
ssize_t n = 0;
char buff[MAXLINE];
while ((n = read(readfd, buff, MAXLINE)) > 0 )
   {
         buff[n] = 0;
```

```
            printf("%s %s %s","Client: Recieved string from server: \"",buff,"\"\n");
            break;
   }
printf("Parent: Client: Sending OK to server\n");
sleep(1);
strcpy(buff,"sends OK from client");
write(writefd, buff, strlen(buff));
return;
}
```

В листинге 4.7 и 4.8 приведены исходные коды программ организующих межпроцессное взаимодействие посредством именованных каналов.

Листинг № 4.7

Программа «сервер» для взаимодействия через *FIFO*.

```
#include <unistd.h>
#include <stdio.h>
#include <error.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <iostream.h>
#include <strings.h>
#include <fstream.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#define MAXLINE 128
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define FIFO1 “/tmp/fifo.1”
#define FIFO2 “/tmp/fifo.2”

int main(int argc, char **argv) {
int         readfd = -1,writefd = -1;
pid_t     childpid = 0;
ssize_t n;
```

```
char str[MAXLINE];
    strcpy(str," some string to transmit ");
    cout<<"Creating pipes..."<<endl;
    unlink(FIFO1);
    unlink(FIFO2);
    if (mkfifo(FIFO1, FILE_MODE) == EEXIST) cout<<"\n Pipes is exists"<<endl;
    if (mkfifo(FIFO2, FILE_MODE) == EEXIST) cout<<"\n Pipes is exists"<<endl;
    cout<<"Pipes created..."<<endl;
    writefd = open(FIFO2, O_WRONLY);
    if ((writefd != -1)) {
        cout<<"Transmitting the string..."<<endl;
        write(writefd,str,strlen(str));
        readfd = open(FIFO1, O_RDONLY);
        cout<<"Waiting for respond..."<<endl;
        while ((n = read(readfd,str, MAXLINE)) > 0) {
            str[n] = 0;
            cout<<"Received string - \""<<str<<"\""<<endl;
            break;
           }
    close(readfd);
    close(writefd);
    unlink(FIFO1);
    unlink(FIFO2);
} else cout<<"Can't open pipes..."<<endl;
return 1;
}
```

Листинг № 4.8

Программа «клиент» для взаимодействия через *FIFO*.

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

```
#include <sys/wait.h>
#include <iostream.h>
#include <strings.h>
#include <fstream.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#define MAXLINE 128
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"

int main(int argc, char **argv) {
int readfd = -1,writefd = -1;
pid_t childpid = 0;
ssize_t n = 0;
char str[MAXLINE];
    ofstream fsw("./rezult.txt");
    fsw<<"Opening pipes..."<<endl;
while (1)
    {
        readfd = open(FIFO2, O_RDONLY, 0);
        if (readfd != -1)  {
             fsw<<"Pipes opened..."<<endl;
             fsw<<"Waiting for respond..."<<endl;
        while ((n = read(readfd,str, MAXLINE)) > 0) {
             str[n] = 0;
             fsw<<"Received string - \""<<str<<"\""<<endl;
             break;
            }
    strcpy(str,"Ok from other process");
    writefd = open(FIFO1, O_WRONLY, 0);
    fsw<<"Transmitting the string - \""<<str<<"\""<<endl;
```

```
        write(writefd,str,strlen(str));
        close(readfd);
        close(writefd);
        break;
    }
    sleep(1);
    }
    fsw.close();
    return 1;
    }
```

Рассмотрим результат запуска приведенных выше программ, использующих неименованные каналы:

```
pipes
Parent: Creating pipes...
Parent: Pipes created...
Parent: Creating child process...
Child:   Child process created...
Child:   Starting server...
Child:   Server: Tranferting string to client - " some string to transmit "
Child:   Server: Waiting for replay from client...Received OK from client - " sends OK from client "
Child:   Server was terminated...
Child:   Terminating process...
Parent: Creating pipes...
Parent: Pipes created...
Parent: Creating child process...
Parent: Starting client...
Client: Recieved string from server: " some string to transmit "
Parent: Client: Sending OK to server
Parent: Waiting for child porecess to terminate a zombie...
Parent: Zombie terminated...
```

Программы, взаимодействующие через каналы *FIFO*, запускаются следующим образом:

```
client &
```

```
Opening pipes...
Pipes opened...
Waiting for respond...
Received string - " some string to transmit "
Transmitting the string - "Ok from other process"
server
Creating pipes...
Pipes created...
Transmitting the string...
Waiting for respond...
Received string - "Ok from other process"
[1]+  Exit 1                ./pn  (wd: ~/makegnu/ipc/pipe_name/2/bin)
(wd now: ~/makegnu/ipc/pipe_name/1/bin)
```

Очереди сообщений

В листинге 4.9 приведена программа *msgcreate*, создающая очередь сообщений. Параметр командной строки *-e* позволяет указать флаг *IPC_EXCL*. Полное имя файла, являющееся обязательным аргументом командной строки, передается функции *ftok*. Получаемый ключ преобразуется в идентификатор функцией *msgget*.

Листинг № 4.9

Программа для создания очереди сообщений *System V*.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <error.h>
#include <unistd.h>
#include <fcntl.h>
#define SVMSG_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
int c, oflag, mqid;
oflag = SVMSG_MODE | IPC_CREAT;
while ( (c = getopt(argc, argv, "e")) != -1) {
```

```
    switch (c) {
        case 'e':
        oflag |= IPC_EXCL;
        break;
    }
}
if (optind != argc - 1)
    {
        printf("usage: msgcreate [ -e ] <path_to_file>");
        return 0;
    }
mqid = msgget(ftok(argv[optind], 0), oflag);
return 0;
}
```

Программа *msgsnd* приведена в листинге 4.10. Она помещает в очередь одно сообщение заданной длины и типа. В программе создается указатель на структуру *msgbuf* общего вида, а затем путем вызова *calloc* выделяется место под реальную структуру (буфер записи) соответствующего размера.

Листинг № 4.10

Программа для помещения сообщения в очередь сообщений *System V*.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <error.h>
#include <fcntl.h>
#define MSG_W (S_IWUSR)
int main(int argc, char **argv)
{
int mqid;
size_t len;
```

```
long type;
struct msgbuf *ptr;
if (argc != 4)
    {
        printf("usage: msgsnd <path_to_file><#bytes><type>");
        return 0;
    }
len = atoi(argv[2]);
type = atoi(argv[3]);
mqid = msgget(ftok(argv[1], 0), MSG_W);
ptr = (msgbuf*) calloc(sizeof(long) + len, sizeof(char));
ptr->mtype = type;
msgsnd(mqid, ptr, len, 0);
return 0;
}
```

В листинге 4.11 приведен текст программы *msgrcv*, считывающей сообщение из очереди. В командной строке может быть указан параметр *-n*, отключающий блокировку, а параметр *-t* может быть использован для указания типа сообщения в функции *msgrcv*.

Листинг № 4.11

Программа для считывания сообщения из очереди *System V*.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#define MSG_R (S_IRUSR | S_IRGRP | S_IROTH)
#define MAXMSG (8192 + sizeof(long))
int main(int argc, char **argv)
{
int c, flag, mqid;
long type;
ssize_t n;
```

```
struct msgbuf *buff;
type = flag = 0;
while ( (c = getopt(argc, argv, "nt:")) != -1) {
    switch (c) {
        case 'n':
            flag |= IPC_NOWAIT;
            break;
        case 't':
            type = atol(optarg);
            break;
        }
}
if (optind != argc - 1)
    {
         printf("usage: msgrcv [ -n ][ -t type ]<path_to_file>");
         return 0;
}
mqid = msgget(ftok(argv[optind], 0), MSG_R);
buff = (msgbuf*) malloc(MAXMSG);
n = msgrcv(mqid, buff, MAXMSG, type, flag);
printf("read %d bytes, type = %ld\n", n, buff->mtype);
return 0;
}
```

Как показано в листинге 4.12, для удаления очереди сообщений вызывается функция *msgctl* с командой *IPC_RMID*.

Листинг № 4.12

Программа для удаления очереди сообщений *System V*.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <fcntl.h>
#include <error.h>
int main(int argc, char **argv)
{
```

```
int mqid;
if (argc != 2)
   {
         printf("usage: msgrmid <path_to_file>");
         return 0;
   }
mqid = msgget(ftok(argv[1], 0), 0);
msgctl(mqid, IPC_RMID, NULL);
return 0;
}
```

Результат запуска приведенных выше программ для случая с тремя сообщениями в очереди:

```
msgcreate /tmp/no/such/file
ftok error for pathname "tmp/no/such/file" and id 0: No such file or directory
touch /tmp/testl
msgcreate /tmp/testl
msgsnd /tmp/testl 1 100
msgsnd /tmp/testl 2 200
msgsnd /tmp/testl 3 300
ipcs -qo
IPC status from <running system> as of Sat Jan 10 11:25:45 1998
T   ID    KEY       MODE      OWNER   GROUP   CBYTES  QNUM
Message Queues:
  q   100    OxOOOOИ3e   --rw-r–r–  rstevens otherl  6     3
```

Сначала происходит попытка создания очереди с помощью имени несуществующего файла. Пример показывает, что файл, указываемый в качестве аргумента функции *ftok* обязательно должен существовать. Затем создается файл */tmp/testl* и используется его имя при создании очереди сообщений. После этого в очередь помещаются три сообщения длиной 1, 2 и 3 байта со значениями типа 100, 200 и 300. Программа *ipcs* показывает, что в очереди находятся 3 сообщения общим объемом 6 байт.

Теперь с помощью аргумента *type* при вызове *msgrcv* считываются сообщения в произвольном порядке:

```
msgrcv -t 200 /tmp/testl
read 2 bytes, type – 200
```

```
msgrcv -t -300 /tmp/testl
read 1 bytes, type = 100
msgrcv /tmp/testl
read 3 bytes, type = 300
msgrcv -n /tmp/testl
msgrcv error: No message of desired type
```

В первом примере запрашивается сообщение с типом 200, во втором примере – сообщение с наименьшим значением типа, не превышающим 300, а в третьем – первое сообщение в очереди. Последний запуск *msgrcv* иллюстрирует действие флага *IPC_NOWAIT*.

Удалить очередь можно вызвав:

```
msgrmid /tmp/test1
```

4.4 ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическим материалом.
2. Запустить несколько заданий (например, команд просмотра файлов *less*), возвращаясь в командную строку комбинацией клавиш *Ctrl-Z* и изучить действие команд *ps, jobs, fg, bg, kill, killall.*
3. Обеспечить синхронизацию процессов и передачу данных между ними на примере двух приложений «клиент» и «сервер», создав два процесса (два исполняемых файла) – процесс «клиент» (первый исполняемый файл) и процесс «сервер» (второй исполняемый файл). С помощью механизмов межпроцессного взаимодействия обеспечить передачу информации от «клиента» к «серверу» и наоборот. В качестве типа передаваемой информации можно использовать: данные, вводимые с клавиатуры; данные, считываемые из файла; данные, генерируемые случайным образом и т.п.
4. Обмен данными между процессами «клиент»-«сервер» осуществить следующим образом:

- Задание №1: с использованием программных каналов (именованных либо неименованных, по указанию преподавателя);
- Задание №2: с использованием (по указанию преподавателя) одного из перечисленных вариантов:
  - разделяемая память (обязательна синхронизация процессов, например, с помощью семафоров);
  - очередь сообщений.

4.5 ТРЕБОВАНИЯ К ОТЧЕТУ

Отчет должен содержать следующие разделы:

1. Титульный лист, оформленный согласно утвержденному образцу.
2. Цели выполняемой лабораторной работы.
3. Задание на лабораторную работу.
4. Исходные тексты созданных программ с пояснениями, иллюстрирующими ход выполнения работы.
5. Выводы.

ЗАКЛЮЧЕНИЕ

Лабораторный практикум по курсу «Операционные системы» предназначен для дисциплины «Операционные системы» студентов АВТФ направления 230100 «Информатика и вычислительная техника». Материал предоставляет возможность получить практические навыки при работе в одной из разновидностей ОС *Unix – Linux* и ее основными подсистемами путем знакомства с их теоретическим материалом и выполнения практической составляющей в рамках 4-х лабораторных работ, посвященных приобретению практических навыков работы в ОС *Linux*.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Робачевский А.М. Операционная система UNIX. – СПб.: БХВ-Санкт-Петербург, 1999.– 528 с.
2. Стен Келли-Бутл. Введение в UNIX. – М.: «Лори», 1995. – 600 с.
3. Стивенс У. UNIX: взаимодействие процессов. – СПб.: Питер, 2003. – 576 с.
4. Стивенс У.Р., Феннер Б., Рудофф Э.М. Unix: разработка сетевых приложений. 3-е изд. – СПб.: Питер, 2007. – 1039 с.
5. Free Software Foundation. GNU Make Manual [Электронный ресурс]. – режим доступа: http://www.gnu.org/software/make/manual/ (12.11.2007).
6. Red Hat, Inc. Red Hat Linux 9, Red Hat Linux Customization Guide [Электронный ресурс]. – режим доступа: https://www.redhat.com/docs/manuals/linux/RHL-9-Manual/custom-guide/ (20.10.2007).
7. Средства параллельного программирования для ОС Linux [Электронный ресурс]. – режим доступа: http://www.opennet.ru/docs/RUS/linux_parallel/. (12.11.2007).
8. Программирование для Linux. Потоки. [Электронный ресурс].– режим доступа: http://www.citforum.ru/programming/unix/threads/ (12.11.2007).
9. Курячий Г. В. Учебный курс «Операционная система Unix» [Электоронный ресурс]. – режим доступа: http://www.intuit.ru/department/os/osunix/ (20.11.2007).
10. Курячий Г. В., Маслинский К. А. Учебный курс «Операционная система Linux» [Электронный ресурс]. – режим доступа: http://www.intuit.ru/department/os/linux/ (20.11.2007).
11. Олифер В. Г. Сетевые операционные системы: учебное пособие / В. Г. Олифер, Н. А. Олифер. – СПб.: Питер, 2003. – 538 с.
12. Гордеев А. В. Операционные системы : учебник / А. В. Гордеев. – 2-е изд. – СПб. : Питер, 2004. – 416 с.

Учебное издание

ЗАМЯТИН Александр Владимирович,

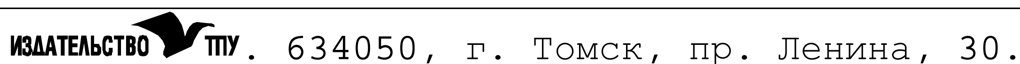
СИДОРОВ Дмитрий Владимирович

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Лабораторный практикум

Подписано к печати 00.00.2008. Формат 60х84/8.
Бумага «Снегурочка».
Печать XEROX. Усл.печ.л. 000. Уч.-изд.л. 000.
Заказ XXX. Тираж XXX экз.

Томский политехнический университет
Система менеджмента качества
Томского политехнического университета сертифицирована
NATIONAL QUALITY ASSURANCE по стандарту ISO 9001:2000


. 634050, г. Томск, пр. Ленина, 30.