Учебное пособие по курсу Операционные Системы и Среды (ОСиС)

Влад 'mend0za' Шахов

последнее обновление - 5 октября 2017 г.

Оглавление

Глава 1

Общие сведения о ОС UNIX

1.1. Вместо предисловия

Это пособие - почти 100% плагиат. Многие главы почти не содержат авторского текста. Это связано с тем, что автор трезво оценивает свои знания и навыки и считает, что гораздо полезнее собрать весь этот материал в одно издание из разрозненных источников, чем заново писать его. Тем более что самописная версия пособия будет заведомо хуже.

Это учебное пособие первоначально писалось для студентов 3 курса специальности информатика. Курс Операционные Системы и Среды (далее ОСиС) базируется на материале курсов ОАиП и КПиЯП и требует знаний языка Си и основ алгоритмизации.

Основная задача курса ОСиС - подготовка специалистов, овладевших пользовательским интерфейсом, архитектурой и программированием Unix-подобных систем (факультативно - элементы администрирования).

1.2. Введение

Что такое Unix? Это семейство операционных систем (ОС), обладающих сходной архитектурой и интерфесом с пользователем.

Unix как явление зародилось в начале 70-х годов и развивается до сих пор.

Основные современные варианты UNIX: Linux, BSD (FreeBSD, NetBSD, OpenBSD), AIX, HPUX, Solaris, SCO.

Важнейшие современные стандарты, обеспечивающие целостность семейства UNIX:

- POSIX - Portable Operating System Interface

- ANSI C (c89 и c99)
- Opengroup[1] Single Unix Specification Version 3 (далее SUSv3).

Примечание. Далее под словом Unix мы будем подразумевать все Unix-подобные операционные системы, если не названа конкретная система.

1.2.1. Основные черты

- Код на Си - позволяет переносить и изменять ОС.
- Многозадачная многопользовательская ОС.
- Наличие стандартов - основой всего семейства являются одинаковая структура и ряд стандартных интерфейсов.
- Простой, но мощный пользовательский интерфейс (командная строка).
- Единая древовидная файловая система. Через интерфейс файловой системы осуществляется доступ к данным, терминалам, принтерам, дискам, сети и даже к оперативной памяти.
- Большое количество программного обеспечения.

1.2.2. Структура системы

Классическая архитектура UNIX двухуровневая:

1. Ядро - управляет ресурсами компьютера и предлагает программам базовый набор услуг (системные вызовы).
2. Системные программы (управление сетью, терминалами, печатью), прикладные программы (редакторы, утилиты, компиляторы и т.д.).

Функции ядра

- *инициализация системы* - загрузка и запуск ОС
- *управление процессами и потоками*
- *управление памятью* - отображение адресного пространства на физическую память, совместное использование памяти процессами
- *управление файлами* - реализует понятие файловой системы, дерева каталогов и файлов

[1] Opengroup - организация, занятая выработкой единых стандартов на Unix-системы (www.opengroup.org). Владелец торговой марки Unix.

- *обмен данными между процессами*
  1. выполняющимися внутри одного компьютера
  2. в разных узлах сетей передачи данных
  3. а также между процессами и драйверами внешних устройств
- *программный интерфейс (API)* - обеспечивает доступ к возможностям ядра со стороны процессов пользователя через системные вызовы, оформленных в виде библиотеки функций на Си.

Системные вызовы

Ядро изолирует программы пользователя от аппаратуры. Все части системы, не считая небольшой части ядра, полностью независимы от архитектуры компьютера и написаны на Си.

Системные вызовы - это уровень, скрывающий особенности конкретного механизма выполнения на уровне аппаратуры от программ пользователя. Для программиста, системный вызов - это функция (определенная на Си), которую он вызывает в своей программе. Все низкоуровневые операции осуществляются через системные вызовы.

Подсистемы ядра

1. Файловая подсистема. Обеспечивает унифицированный доступ к файлам:
   - контроль прав доступа к файлу;
   - чтение/запись файла;
   - размещение и удаление файла;
   - перенаправление запросов к периферийным устройствам, соответствующим модулям подсистемы ввода/вывода.
2. Подсистема управления процессами.

   Запущенная на выполнение программа порождает один или несколько процессов (задач).
   Подсистема контролирует:
   - создание и удаление процессов
   - распределение системных ресурсов (памяти, вычислительных ресурсов) между процессами
   - синхронизация процессов
   - межпроцессное взаимодействие

   Специальная задача ядра *планировщик процессов* разрешает конфликты процессов в конкуренции за ресурсы.

3. Подсистема ввода/вывода. Выполняет запросы файловой системы и подсистемы управления процессами для доступа к периферийным устройствам (дискам, лентам, терминалам). Обеспечивает буферизацию данных и взаимодействует с драйверами устройств.

1.2.3. ОС UNIX для пользователя

Пользователь

С самого начала ОС UNIX разрабатывалась как интерактивная система. Чтобы начать работу, нужно войти в систему, введя со свободного терминала (консоли) свое *имя* (account name, логин) и *пароль* (password).

Человек, зарегистрированный в системе и, следовательно, имеющий учетное имя, называется *зарегистрированным пользователем системы.*

Регистрацию новых пользователей обычно выполняет администратор системы. Пользователь не может изменить свое учетное имя, но может установить или изменить пароль. Пароли находятся в отдельном файле в закодированном виде.

Все пользователи так или иначе работают с файлами. Файловая система имеет древовидную структуру. У каждого зарегистрированного пользователя есть *домашний каталог.* К нему он имеет полный доступ. К другим каталогам доступ обычно ограничен.

Интерфейс пользователя

Традиционный интерфейс - *командная строка.* После входа в систему для пользователя осуществляется запуск одной из *командных оболочек.* Общее название *shell (eng. - оболочка)*, так как они являются внешним окружением ядра системы. Оболочка - это интерпретатор комманд (встроенных и внешних) и обладает мощным встроенным языком shell scripts, позволяющим писать сложные программы.

GUI и Unix

Графический интерфейс (GUI) не является необходимым в Unix и рассматриваться не будет. Подавляющее большинство обычных операций выполняется без его участия.

Глава 2

Командная строка

2.1. Основные принципы и команды

Как следует из названия, основное назначение командной оболочки - ввод и исполнение команд. Для ввода команд служит т.н. *командная строка*, содержащая приглашение к вводу.

В примерах приглашение командной строки будем обозначать `$`.

После ввода имени и пароля, пользователь получает приглашение на ввод. После окончания работы, он выходит, набирая `exit` или `logout`.

Большинству команд можно передавать дополнительную информацию. Это делается с помощью *аргументов и ключей*.
Общий вид команды: `команда [-ключи] аргумент1 ... аргументN`

Простейшая команда - `echo`. Она просто выводит на экран свои параметры.

Ключ - 1 или более букв, перед которой ставится -(минус или дефис). Обычно ключи управляют режимами работы команды.

Аргумент - строка, передаваемая команде.

```
Пример
$kill -9 1023
kill имя команды (завершение процесса)
-9  режим (безусловное завершение процесса)
1023 номер процесса (завершаемый процесс)
$ - символ приглашения ввода (в разных системах - разный)
```

После ввода команды, система производит поиск в каталогах программ, заданных переменной *PATH* . Если команда найдена, то она будет запущена. Иногда, если PATH не содержит нужного каталога, нужно указать полный путь (см. 2.3) к программе.

Примечание. Текущий каталог НЕ ВХОДИТ в PATH. Программы из него запускаютсяь следующим образом:
`./program`

2.2. Встроенная справка

Любая Unix система обладает развитой системой справки. Она называется `man` (от англ. слова manual).

Справка запускается следующим образом: `man статья`[1]. Для перемещения используются клавиши `"вверх", "вниз"`. Для выхода нажмите `q`.

`man` делится на несколько разделов, которые пронумерованы[2]:

- 1 - Команды, которые могут быть запущены пользователем
- 2 - Системные вызовы (функции, исполняемые ядром)
- 3 - Библиотечные вызовы (функции из библиотек различных языков программирования)
- 5 - Форматы файлов и соглашения

Запись типа `bash(1)` обозначает, что справка о команде `bash` находиться в разделе 1 (команды). Бывает что справки в различных разделах называются одинаково. Тогда для получения нужной надо явно указать раздел.

```
Примеры использования man
$ man echo

Получение справки по самой команде man
$ man man

Справка из конкретного раздела
$ man 2 open

Все статьи с таким названием из всех разделов
$ man -a mount
```

Некоторые команды содержат свою собственную систему помощи[3], Она вызывается с помощью ключа `--help` или вызовом без параметров

```
Примеры
$ file
Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
Usage: file -C [-m magic]
```

[1] Будет показана *только первая* доступная статья

[2] Здесь приведены только те разделы, которые будут активно использоватьсяв рамках курса ОСиС. Описания других разделов вы сможете найти в книгах [1] и [9]

[3] в основном это касается команд написанных проектом GNU

```
$ split --help
Usage: split [OPTION] [INPUT [PREFIX]]
Output fixed-size pieces of INPUT to PREFIXaa, PREFIXab, ...; default
PREFIX is `x'.  With no INPUT, or when INPUT is -, read standard input.

  -b, --bytes=SIZE        put SIZE bytes per output file
  -C, --line-bytes=SIZE   put at most SIZE bytes of lines per output file
  -l, --lines=NUMBER      put NUMBER lines per output file
  -NUMBER                 same as -l NUMBER
      --verbose           print a diagnostic to standard error just
                            before each output file is opened
      --help              display this help and exit
      --version           output version information and exit

SIZE may have a multiplier suffix: b for 512, k for 1K, m for 1 Meg.

Report bugs to <bug-textutils@gnu.org>.
```

2.3. Перемещение по файловой системе

Файл в Unix - это основа всего. Существует высказывание "Unix - это файлы".

Имя файла может состоять из любых символов, которые можно ввести с клавиатуры. Точка не имеет особого значения - имя файла может содержать ее или нет[4]. Обычно пользователи присваивают расширения своим файлам, чтобы различать их тип. Система *не связывает типы файлов с их расширениями*. Например файл `example.txt` может быть исполняемым (программой), а может и не быть. А `/bin/sh` как правило исполняемый.

Однако у точки есть специальные значения

- точка в начале файла указывает на то, что файл - скрытый, например `.profile`
- файл `.` - ссылка на текущий каталог (см. пример относительных путей)
- файл `..` - ссылка на родительский каталог

Если вы не знаете тип файла, примените команду `file`.

[4] можно создать файл или каталог с именем из одних точек

```
$ file /home/work/OSIS/LAB/RUsak/demon1.c
/home/work/OSIS/LAB/RUsak/demon1.c: ASCII C program text, with very long lines

$ file /bin/bash
/bin/bash: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linke
```

Unix различает строчные и заглавные буквы в именах. Например `name1.txt.125` и `NAME1.TXT.125` - это разные файлы.

Файловая система ОС Unix является единой. Это значит что, все файлы находятся в рамках одной логической структуры - *дерева каталогов*. В Unix отсутствует понятие "диск" и "буква устройства".

Например - файлы с дискеты обычно находятся в `/mnt/floppy`, но могут быть и в любом другом месте.

Началом дерева является корневой каталог (корень), обозначаемый `/` .

Разделителем каталогов является прямой слэш `/` .

Путь - последовательный список каталогов, который нужно пройти, чтобы достигнуть файла или каталога.

Есть 2 вида путей: *абсолютный* и *относительный*.
Абсолютный - полный путь относительно корневого каталога.

```
Примеры абсолютных путей:
/etc/init.d/apache
/bin/sh
/usr/local/share
```

относительный - путь относительно текущего каталога

```
Примеры относительных путей:
./a.out
laba1/text.cpp
../index.html
```

2.3.1. Команды перемещения

- `cd` - перемещение между каталогами. Может использоваться в 2-х вариантах:

  ```
  перемещение по указанному пути
  $cd  /usr/local

  вернуться в домашний каталог
  $cd
  ```

- `pwd` - показать текущий каталог

```
$pwd
/home/user/OSiS/metoda
```

- `ls` - показать содержимое каталога(файлы и подкаталоги). Имеет множество ключей и режимов работы

```
Без параметров - вывод содержимого текущего каталога
$ ls
literatura.aux  Makefile    metoda.dvi  metoda.tex  part1.aux  part2.aux
literatura.tex  metoda.aux  metoda.log  metoda.toc  part1.tex  part2.tex

С аргументом - вывод содержимого этого каталога
$ ls /usr
bin   doc  games    kerberos  libexec  man                sbin   src  X11R6
dict  etc  include  lib       local    OpenOffice.org1.0  share  tmp

С ключом "-l" - полный формат вывода (с доп информацией)
$ ls -l /home/user/OSiS
итого 710
drwxrwxr-x    2 user     user         1024 Июн 11 14:11 Lectures
-rw-rw-r--    1 user     user       123686 Июн 12 12:53 lectures.rar
-rw-rw-rw-    1 user     user         1789 Июн 11 15:44 lhr10.log
-rw-r--r--    1 user     user            0 Июн 11 15:44 lkypc.pdf
-rw-rw-r--    1 user     user       438112 Апр 20  1999 lshort.dvi
-rw-r--r--    1 user     user       112747 Июн 12 12:53 lshrtdvi.zip
drwxrwxr-x    2 user     user         1024 Июн 14 18:50 metoda
-rw-rw-r--    1 user     user        40960 Июн 11 14:06 metoda.doc
```

2.3.2. Подключение других устройств(дисковода, CD-ROM)

Чтобы подключить устройство (дисковод, CD-ROM, раздел вичестера, сетевой диск), нужно указать место, куда будет отображаться его содержимое. Это место называется *точка монтирования*. Точка монтирования - это обычный каталог. После подключения (в терминах Unix - *монтирования*) каталог будет содержать файлы, расположенные на устройстве. Список подключенных устройств и монтирование - команда `mount(8)`.

Обычно поключаемые устройства отображаются на каталог `/mnt`. Например `/mnt/cdrom`, `/mnt/floppy`.

Отключение (*размонтирование*) производит отсоединение устройства от дерева каталогов. Команда `umount(8)`.

2.4. Копирование, удаление, перемещение файлов и каталогов

- `cp` - копирование

```
копирование одного файла в другой
$ cp src.file /tmp/dest.file1

копирование нескольких файлов в другой каталог
$ cp *.tex metoda.dvi /mnt/floppy

копирование каталогов
$ cp -r metoda /archive/old.Docs
```

- `rm` - удаление файлов и `rmdir` - удаление каталогов[5]

```
простое удаление
$ rm part1.aux intro.temp

удаление всех файлов из каталога
$ rm /tmp/*

удаление каталога
$ rmdir oLD.stupid.dir
```

- `mv` - переместить(переименовать)

```
переместить один файл в другой
$ mv src.file /tmp/dest.file1

перемещение нескольких файлов в другой каталог
$ mv *.tex metoda.dvi /mnt/floppy

перемещение каталогов
$ mv metoda /archive/old.Docs
```

- `mkdir` - создать каталог

```
создать 1 или более каталогов
$ mkdir /tmp/dir.12345
$ mkdir empty.DIR DiRecTorY
```

[5] работает только для *пустых* каталогов

2.5. Информация о системе

- `ps` - список запущенных процессов
- `who` - список пользователей, работающих в системе
- `date` - текущая дата и время
- `w` - общая информация о системе

```
Примеры:
Процессы на текущей консоли
$ ps
 3642 pts/1    00:00:00 bash
 4548 pts/1    00:00:00 ps

Процессы конкретного пользователя
$ ps -u user
 1766 ?        00:00:04 xterm
 1853 pts/2    00:00:13 vim
 3642 pts/1    00:00:00 bash
 4553 pts/1    00:00:00 ps

Все процессы
$ ps -ef (для BSD и Linux - ps ax )
UID        PID  PPID  C STIME TTY          TIME CMD
user      1766  1176  0 06:45 ?        00:00:04 xterm -title Terminal
user      1768  1766  0 06:45 pts/2    00:00:00 bash
user      1853  1768  0 06:56 pts/2    00:00:13 vim part2.tex
<пропущено - список 52 строки>

$ date
Сбт Июн 15 10:34:17 EEST 2002

$ who
root     tty1     Jun 15 10:24
work     tty2     Jun 15 10:24
user     pts/1    Jun 15 09:18

$ w
 10:35am  up  4:14,  3 users,  load average: 0.09, 0.04, 0.01
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
root     tty1     -                10:24am 11:25   0.02s  0.02s  -bash
work     tty2     -                10:24am 11:11   0.03s  0.03s  -bash
user     pts/1    -                 9:18am  1.00s  0.14s  0.01s  w
```

2.6. Общение между пользователями

- `write пользователь` - послать сообщение[6]

```
$ write stud11
набрать текст сообщения
нажать <ctrl+d>
```

- `talk пользователь` - двухсторонний чат
- `mail` - электронная почта

```
Послать письмо
$  mail stud7 //локальному пользователю
Subject: test
набрать текст сообщения
нажать <ctrl+d>

$ mail user@tut.by //удаленному пользователю
.....

$ mail // просмотреть свою почту
.....
```

2.7. Просмотр, создание, объединение файлов

- `cat` - вывод содержимого файла на экран
- `more` - разбиение входного файла на страницы
- `less` - просмотр файлов (выход по клавише `q`)

```
Примеры
Вывод на экран файла
$ cat file

Копирование файла
$ cat file >file2

Объединение 2-х файлов в третий
$ cat file1 file2 >end.file

Постраничный вывод файла на экран
```

[6] ctrl+d воспринимается системой как конец ввода

```
$ cat large.file |more

Ввод файла с клавиатуры
$ cat >new.text
<набирается текст>
<ctrl+d>
```

2.8. Объединение команд

Идеологически, Unix - это собрание большого количества небольших утилит, выполняющих какую-то одну локальную задачу. Но делающими свою задачу лучше всего, во всех возможных вариантах и режимах.

Фильтры - это программы, предназначенные для обработки текста тем или иным способом.

Часто они применяются в связке с другими командами, образуя *конвейеры*. Конвейер обозначается символом `|`. Его значение следующее: програма слева от конвейера передает свой вывод на вход программы справа от конвейера[7]. Простейший пример: `$ cat myFILE|more`[8]. В Unix этот метод называют *перенаправление*. Подробнее о перенаправлении - в 3.4.

В общем случае, фильтры читают свой *стандартный ввод* и пишут на *стандартный вывод*. Если не было перенаправления, то вводом считается клавиатура, а выводом - экран.

Часто употребимые фильтры

- `grep`[9] - поиск в файле по образцу
  `Пример: $ ls /usr/include | grep "stdlib.h"`
- `sort` - сортировка содержимого
- `wc` - статистика по файлу (кол-во букв, байт, строк и т.п.)
- `head` и `tail` - показать начало(head) и конец(tail) файла
- `tee` - одновременный вывод в файл и на экран
  `Пример: $ ls -1 /tmp | tee all.tempfiles`

[7] конвейер может применяться несколько раз

[8] То что выводит cat посылается на вход more. А more выступает здесь в качестве простого фильтра.

[9] Существует целое семейство grep-команд : grep, egrep, fgrep

2.9. Приемы эффективной работы

Оболочка `bash` обладает 3 базовыми средствами автоматизации[10], делающими работу в командной строке простой и легкой:

1. Автодополнение путей и команд

   Использование: набрать 1 или более начальных символов команды и нажать `TAB`. Если символов хватает для определения команды, то недостающие будут добавлены автоматически. Если существует более 1 подходящей команды, то при повторном нажатиии `TAB` на экран высветиться список возможных команд. Можно тогда добавить несколько букв и однозначно определить команду.

   ```
   $ la<TAB>
   lambda      last        lastb       lastlog     latex       latex2html
   $ last<TAB>
   last     lastb    lastlog
   $ lastb<ENTER>
   lastb: /var/log/btmp: No such file or directory
   Perhaps this file was removed by the operator to prevent logging lastb info
   ```

   Для поиска комманд используется переменная PATH (см. 2.1).

   Точно так же работает автодополнение для путей: вводиться кусочек пути и после нажатия <TAB> происходит дополнение пути.

   Примечание. Автодополнение не работает для ключей комманд (`ls --he<TAB>` - будет безрезультатным) и для аргументов команды `man`.

2. История команд

   Для просмотра истории - `history`. Для обращения к конкретному пункту истории - `!номер`. Можно прокручивать историю с помощью клавиш `"вверх"` и `"вниз"`.

3. Редактирование командной строки

   Возможно с помощью клавиш `"вправо"` и `"влево"`. На начало строки - `ctrl+a`, на конец - `ctrl+e`.

[10] Подробнее об автоматизации BASH - в [12], [9], man bash

Глава 3

Shell (оболочка)

3.1. Понятие оболочки

Оболочки (командные интерпретаторы, процессоры, shells) представляют собой промежуточные уровень между пользователем и ОС. Они анализируют командную строку, выполняют преобразования аргументов команд, находя и выполняя команды.

Shell это обычная прикладная программа. Она не является частью ядра, и поэтому может быть заменена на любую другую, например, на игрушку или текстовый редактор. Из оболочки может быть запущена другая оболочка (или такая же), что дает дополнительные возможности.

Виды shell: sh, csh, ksh, zsh, tcsh, ash и другие.

В курсе будет рассматриваться *Bourne Shell - sh*. Оболочки, совместимые с Bourne Shell существуют для все версий Unix. На лабораторных работах будет использоваться `bash` (Bourne Again SHell).

3.2. Bourne Shell

Сценарий (скрипт) оболочки представляет собой текстовый файл, который задает выполнение последовательности действий. Сценарий может содержать любую последовательность команд (как внутренних команд оболочки, так и внешних команд UNIX, с аргументами или без них), вызовов программ или других написанных ранее сценариев.

Про Shell можно сказать, что это и программа и язык программирования.

Работа пользователя в командной сроке ничем не отличается от выполнения длинного запутанного скрипта.

Механизм работы скрипта: при запуске скрипта из командной строки запускается копия интерпретатора, для которого вводом служит скрипт (как-будто пользователь сам вводил эти команды).

Способы запуска: по имени, bash имя.
Для запуска по имени, файл скрипта должнен иметь атрибут X (eXecutable - исполняемый). Подробнее об атрибутах будет расказано в **??**

3.2.1. Структура скриптов

`#` - то, что следует за ним(в том числе и другие `#`), является комментарием. Комментарии могут занимать всю строку или следовать за командой.
\ - обозначает, что строка продолжится на следующей строке файла.
Можно записать несколько команд в 1 строку, разделенных `;` .

3.3. Переменные

Значение переменной - строка, которая передается присваиванием.
`V1 = 5; v2 = "string"` [1]
Переменной также можно присваивать значение, которое возвращается командой.
`V3 = \`pwd\``

Получение значения

- `$имя_переменной` - в это место подставляется значение переменной.
- `$(имя_переменной)` - отделяем переменную от последующих символов.

  ```
  Пример:
  $ echo result = $(v1)2 +
  result = 52+
  ```

  В shell существует ряд *предопределенных* переменных[2]:

- `HOME` - домашний каталог пользователя
- `PATH` - путь поиска исполняемых программ
- `MAIL` - полное имя файла с почтой пользователя
- `PS1,PS2` - первичное и вторичное приглашение shell (Значок `$`, который пишется в примерах - это первичное приглашение).

[1] Значения переменных будут переходить из примера в пример
[2] Полный список встроенных и предопределенных переменных можно найти в man bash

В shell существует ряд переменных, которые определяются оболочкой по ходу выполнения скриптов. Это так называемые *встроенные* переменные:

1. `$0, $1, $2, ... ,$9` значения параметров, передаваемых скрипту из командной строки.

2. `$0` - имя самого скрипта.
   Примечание: хороший стиль программирования выдавать имя скрипта по `$0` в выдаваемом скриптом сообщении.

3. `$#` - число параметров, переданных скрипту;

4. `$*` - все параметры, переданные скрипту. Представляют собой единое слово, заключенное в кавычки.

Существуют три вида кавычек при присваивании переменных:

- `' '` - непосредственная подстановка, например: `v4='$v1'` присвоит `$v1`, а не 5.

- `" "` - подстановка после интерпретации символов \ и `$`. Например: `v5=$v1`. Переменной `v5` присвоится 5. Записи без подстановочных символов эквивалентны.

```
Например присваивание
         v6=string
         v7="string"
         v8='string'
даст одинаковый результат. Кавычки одного вида экранируют другие.
```

- `` ` ` `` - выполнение команды внутри скобок. Результат выполнения команды будет присвоен переменной.
  `Пример: $list=`ls -a``

По умолчанию все переменные локальны, то есть существуют, пока выполняется скрипт. Чтобы сделать их глобальными (для данного shell), надо задать их при помощи export. `например: export v1`

При выводе неопределенных переменных результатом будет пустая строка.

Для удаления переменной используется `unset список_переменных`.
`Пример: $ unset $v1 $v3 $v4`

Команда `set` выводит список всех установленных переменных shell.

3.4. Перенаправление ввода/вывода

.

Каждая программы, запущенная из shell, получает три открытых потока ввода/вывода, которые по умолчанию ассоциируются с терминалом. Потоки задаются номерами(*дескрипторы*):

- 0 стандартный поток ввода, ассоциируется с клавиатурой
- 1 стандартный поток вывода, ассоциируется с экраном
- 2 стандартный поток ошибок, ассоциируется с экраном

Большинство утилит Unix используют только стандартные потоки, поэтому для этих утилит можно осуществлять перенаправление.

Виды перенаправления

- `>file` - поток вывода перенаправляется в файл. `Пример: cat file1>file2`
- `>>file` - данные из потока вывода добавляются в файл.
- `<file` - получение данных для стандартного ввода из файла.
- `p1|p2` - передача вывода программы p1 на ввод программы p2 (*конвейер или неименованный канал*). `Пример: cat spisok | wc l`
- `n>file` - переключение потока с номером `n` в файл.
- `n>>file` - переключение потока с дескриптором `n` в файл, но данные добавляются в конец файла.
- `n>&m` - объединить потоки с дескрипторами `n` и `m`.
- `<<str` - конструкция *"Ввод здесь"*. Использует стандартный поток ввода до появления строки `str` во вводе и потом передает его на вход программе.

```
Примеры:
1)ls -al | wc 1>&2 1>>wc.out
2)run 2>/dev/null  подавление вывода ошибок.
```

3.5. Шаблоны(wildcard's, подстановочные символы)

Оболочка позволяет делать подстановку имен. *Подстановочный символ* заменяется оболочкой на имена файлов, если что-то в каталоге подходит под шаблон. Это полезно в случаях, когда файлов много или необходимо выбрать несколько файлов по определенному правилу.

- `*` - заменяет любое количество символов (может быть и 0) , в имени файла.
- `?` - заменяет любой символ в имени файла.
- `[символы]` - задает любой символ из диапазона.`[a-c1-3]` тоже что и `[abc123]`
- `\c` - задает символ с буквально (*экранирует*) , если с это спецсимвол(`\,',",`,# и т.д.`).

Примеры:
```
$ ls [a-d]* //все файлы, начинающиеся на a,b,c,d
$ ls x*y // все, начинающиеся на x и кончающиеся на y
$ ls *\ ? // предпоследний символ - пробел
```

3.6. Условное выполнение команд

Следующая удобная возможность - *условное выполнение*. Его суть такова: пусть ваши действия зависят от результата выполнения предыдущих. Выше уже упоминался разделитель команд `';'` (см. 3.2.1). Но при использовании `';'` последовательность команд всегда выполниться, вне зависимости от результатов работы отдельных команд (ошибок в них).

Для повышения гибкости работы, в Bourne Shell существуют следующие конструкции:

- `p1&&p2` выполняется p1, если удачно (код возврата 0[3]), то запускается p2
- `p1||p2` - выполняется p1, если неудачно (код возврата не 0), то запускается p2
- `p1&` - p1 выполняется в фоновом режиме, и shell не ждет окончания работы p1 (см. **??**, Контроль заданий). Оболочка сразу выводит приглашение на ввод.
- `(p1;p2;)` - команды выполняются последовательно в новой оболочке.
- `{p1;p2;}` - команды выполняются последовательно в текущей shell.

Примеры:
```
1)(ps; who) | more
2)mount | wc -l > mounts.number &
```

Примечание: процессы в фоновом режиме не могут использовать стандартный ввод и вывод, поэтому их надо перенаправлять. Поток ошибок работает без изменений.

[3]Коды возврата для shell - противоположны Си (см. 3.7)

3.7. Условные выражения

Синтаксис

```
if   условие
then
else
fi
```

В shell true (0) и false (не 0) имеют обратные значения по сравнению с Си.. [4]

Условные выражения можно записывать в строчку, разделяя ;.

Условием может быть результат команды. Часто используется `test` с параметрами. Наиболее употребимые значения:

- test s файл - является ли размер файла отличным от 0
- test r файл - доступен ли файл для чтения
- test f файл - существует ли файл и является ли он обычным
- test d файл - существует ли файл и является ли он каталогом

Можно записывать без `test`, используя то же значение в `[ ... ]`.

Примечание. Между `[, ], if` обязательно должны стоять пробелы!

```
Пример: следующие записи эквивалентны:
1)if test f $HOME/file.txt
then
            echo Он есть!
     fi
2)if [ -f $HOME/file.txt]
then
            echo Он есть!
     fi
3)  test f $HOME/file.txt && echo Он есть!
```

3.7.1. Сравнение строк

- `строка1 = строка2` проверка на равенство
- `строка1 != строка2` проверка на не равно
- `-n $переменная` true, если строка имеет ненулевую длину

```
Примеры:
1)if [ $v1 = abc ]; then; echo Потрясающе!
2)if [ -n $empty ]; then; echo Действительно
```

[4] Более подробно коды возврата отдельных команд описаны в man в пункте EXIT STATUS или DIAGNOSTIC или RETURN CODE

3.7.2. Сравнение чисел

Аргументами являются `$x`(значение переменной) или число[5]:

1. `$x eq $y` true, если аргументы равны
2. `$x ne $y` true, если аргументы не равны
3. `$x gt $y` true, если значение x больше значения y
4. `$x ge $y` true, если значение x больше либо равно значению y

```
Пример:  if [ $# eq 2 ]; then; echo 2 аргумента
```

3.7.3. Сложные выражения

1. `!выражение` - отрицание
2. `выражение1 a выражение2` - логическое И
3. `выражение1 o выражение2` - логическое ИЛИ

```
Пример:
1)if [ !\( $x eq $y\) ]
2)if [ $a ne 3 a $b lt $c ]
3)if [ $x = $y a \( $n lt 0 o $m gt 30\) ]
```

Примечание: Скобки экранируются (см. 3.5), так как они имеют специальный смысл для команд (последовательное выполнение в новом экземпляре shell).

3.8. Циклы

В языке shell есть несколько видов циклов. Часто употребимые из них:`for`, `while` .

- Цикл `for`[6] выполниться столько раз, сколько слов в списке. var последовательно принимает значения из списка. Список может формироваться вручную, как вывод команды (‘команда‘) или с помощью шаблонов.

  ```
  for переменная in список
  do
  ....
  done
  ```

[5]Тут наблюдается некоторое сходство с ассемблером

[6]BASH поддерживает также циклы в стиле Си: for ((i=1; $i < 10$;i++)) - с 2 открывающими и закрывающими скобками.

- Цикл `while` выполняется, пока условие не станет ложным.

```
while условие
do
...
done
```

```
Примеры:
while sleep 60
do
 who | grep mary
done

for user in `who`
do
  echo и этот $user здесь!
done

for i in * ; do echo $i; done  #эквивалент ls.
```

Элементы цикла можно записывать на отдельных строках или разделяя ';'. Эти формы записи эквивалентны.

Примечание. Если вводить в командной строке цикл, условное выражение или просто не закрыть скобку (кавычки) то при нажатии `ENTER` оболочка предложит вводить окончание команды на следующей строке.

Существует *встроенная*(см. 3.9) команда `break` для выхода из цикла.

3.9. Функции

Для упорядочивания скрипта пользователь может определить функцию:

```
имя_функции ()
{
   команды
}
```

Синтаксис и передача аргументов - как у скрипта.

```
Пример: отображает в приглашении имя каталога
mcd ()
{
cd $*
PS1 = `pwd`
}
```

Команды разделяются на *встроенные* (в оболочку) и *внешние*. Запуск встроенной команды не требует создания нового процесса.

Распространенные встроенные команды: `cd`, `pwd`, `echo`, `exit`, `set`, `unset`.

3.10. Выполнение арифметических операций

Важное замечание. В shell выполняется только целочисленная арифметика[7]!

`expr строка` преобразует строку в число. `Например: expr 23`.

Выполняются операции: `+, -, *, /, %` (деление по модулю). Их приоритет обычный.

```
Примеры:
1)a = `expr $a + 3`
2)b = `expr 2 \* 3` - символ \ отменяет специальное значение *.
```

Примечание. Числа и операции разделяются пробелами.

[7]Для плавающей точки и сложных вычислений можно использовать программу bc

Глава 4

Текстовый редактор VI

Редакторы для Unix делятся на 2 группы - редакторы командного стиля(vi, emacs, joe, ed) и меню-ориентированные (mcedit, kwriter, kword).

Редакторы командного стиля обычно работают в *консольном (текстовом)* режиме. Все действия в них выполняются подачей прямых управляющих команд, закрепленных за определенными сочетаниями клавиш. Мышь и меню в них, как правило, не используются.

Редактор `vi` присутствует как стандартный в любой Unix-подобной системе[1]. Существует несколько редакторов основанных на vi: vim, elvis.

Современные клоны vi (vim к примеру) обладают очень большой функциональностью, скрытой за аскетичным интерфейсом. Редактор vi изначально создавался как кросс-платформенный, который обязан работать на любых типах терминалов и виртуальных консолей. Все действия в нем можно осуществить не покидая основной, алфавитно-цифровой, части клавиатуры.

Примечание. Далее мы будем рассматривать редактор vim. Однако все описанные команды можно будет применить в любом vi-совместимом редакторе.

4.1. Режимы работы

В vi существует три принципиально различных режима работы:

- Командный режим (command mode)

- Режим ввода (edit mode)

- Режим построчного редактирования (ex mode)

Командный режим включается по умолчанию при запуске vi. В этом режиме нажатия клавиш *не приводят* к вводу символов, а ин-

[1] В этом качестве он внесен в стандарт Single Unix Specification

терпретируются как внутренние команды перемещения по тексту и редактирования. Поэтому попытка немедленно начать ввод текста (как в DOS/Windows) ни к чему не приведет.

Примечание. Если вы не знаете, в каком режиме находитесь, то нажмите клавишу `ESC` для перехода в командный режим.

Создание текста в командном режиме невозможно. Для этого нужно перейти в *режим ввода*. Для этого служат команды (командного режима!) `a` (от append - после текущей позиции курсора) и `i` (от insert - перед текущей позицией курсора). В режиме ввода нажатия клавиш приводят к вводу обычных символов, позволяя создавать новый текст или редактировать существующий.

Возврат в командный режим осуществляется нажатием клавиши `escape`.

Для операций с документами (файлами) предназначен *ex-режим*. Он вызывается командой `:` командного режима. После этого дается команда ex-режима. Например:

- открыть существующий файл (`:e имя_файла`)
- вставить файл в позицию курсора (`:r имя_файла`)
- записать файл (`:w`), в том числе под другим именем (`:w имя_файла`)
- выход из сохраненного файла (`:q`)
- выход с предварительным сохранением файла (`:x`)

Когда вы находитесь в ex-режиме, то в нижнем левом углу экрана появляется `:`.

Возможно совмещение команд ex-режима. Например `:wq`.

Команда ex-режима отправляется на выполнение нажатием клавиши `Enter` после чего происходит возврат в командный режим.

Примечание. Попытка загрузить новый файл (командой `:e`) или завершить работу редактора (командой `:q`) при несохраненном старом файле вызовет ошибку.

4.2. Получение помощи

Получить справку можно используя ex-команду `:help`.

Подразделы справки выделены значками `|раздел|`. Справку по ним вызывается через `:help раздел`.

Очень полезным является учебник по vim - `vimtutor`. С его помощью можно освоить основные навыки использования vim.

4.3. Запуск и остановка редактора

`Vi (vim)` может быть запущен из командной строки, с именем файла или без такового. Если указано имя файла, то редактор открывает его[2].

`Пример: $ vi ~/texts/newtext.txt`

Команда vim без имени файла откроет редактор vim и выведет заставку.

Для выхода из редактора нажмите `:q` или `:wq` (см. "Режимы работы" 4.1).

4.4. Перемещение по тексту

В командном режиме существуют следующие команды[3]:

- `h` - курсор влево на 1 символ
- `l` - курсор вправо на 1 символ
- `j` - курсор вниз на 1 строку
- `l` - курсор вверх на 1 строку

Также, есть расширенные команды, действующие с блоками текста.

- `w и W` - перемещение вперед на "маленькое слово"[4] и т.н. "большое слово"[5]
- `b и B` - перемещение назад на "маленькое слово" и "большое слово"
- `0 и $` - на начало и на конец строки
- `( и )` - на начало предложения и его конец

Вообще, для многих команд vi характерно наличие парных элементов - в нижнем и верхнем регистрах одной клавиши (`e` и `E`, `w` и `W`); действие второй команды из пары как бы расширяет действие первой.

Команды навигации vi могут использоваться с численными аргументами.

Например команда `5h` переместит курсор на 5 символов влево (считая символ в позиции курсора), а команда `3B` на 3 "больших" слова назад.

Для перемещения на конкретную строку, можно использовать следующую команду ex-режима: `:N`, где N - номер строки.

---

[2] Если файл не существует, то создается новый

[3] Обычно работают и стрелки на клавиатуре, но не стоит полагаться на них

[4] Отдельное слово, отделенное пробелом, знаками препинания, +, -

[5] Обязательно отделенное пробелом

4.5. Ввод и редактирование текста

Для создания текста необходимо перейти в режим ввода.

Для этого служат следующие команды:

- `i и I` - ввод в позиции курсора или в начале строки
- `a и A` - ввод после курсора или в конце строки

Текст можно изменять и в режиме ввода, используя `DEL` и `BACKSPACE`, но часто удобнее использовать команды редактирования.

Команды редактирования предназначены для изменения существующего текста без перехода в режим ввода:

- `x` - удаление одиночного символа
- `dd` - удаление строки
- `dw` - удаление слова
- `d)` - удаление предложения

Как и команды перемещения, команды редактирования можно использовать с численными аргументами. Так команда `5dd` удалит текущую строку и еще 4 строки ниже ее, а `3dw` удалит три слова считая текущее.

4.6. Копирование и вставка

В vim для этих целей существует отдельный режим - выделения, Visual Selection (см. `man vim`). Однако в большинстве случаев мы можем обойтись стандартными командами:

- `p` - вставить из буфера.
- `yy` - скопировать строку в буфер
- `yw` - скопировать текущее слово в буфер
- `y)` - скопировать предложение
- `y}` - скопировать абзац

*Примечание.*В буфер также попадает все удаленное с помощью команд x, dd, dw и им подобных. Таким образом эти команды могут служить для копирования с удалением.

Все вышеперечисленные команды могут использоваться с численным аргументом. Например: `3p` - 3 раза вставить содержимое буфера.

4.7. Откат действий

Действие ошибочно введенных команд редактирования может быть отменено командой `u` (сокращенно от undo). Повторное нажатие - отмена предыдущего действия, и так далее. Для возврата (redo) ошибочно отмененной операции используется `control+r`.

4.8. Поиск и замена

Для поиска по тексту служит команда `/` (прямой слэш). При вводе этого символа в командном режиме в нижней строке появляется символ `/`, после которого вы можете ввести образец для поиска. Это может быть текстовая строка или *регулярное выражение* (см. 5). После нажатия `ENTER` в тексте будут подсвечены [6] все возможные вхождения строки поиска и курсор перейдет к первому доступному найденому фрагменту вниз по тексту.

Для поиска следующих вхождений строки поиска, существует команды `n` (вниз по тексту) и `N` (вверх по тексту).

Для поиска и замены текстовых фрагментов, в том числе и с использованием регулярных выражений, предназначена команда ex-режима `:s`(substitute). Формат команды:
`:#s/pattern/string/опция`
где `#` - интервал строк (через `,` или `;` - см. `:help cmdline-ranges`).

Примечание. опции в стандартном vi не поддерживаются.

Часто употребимые опции: `c` - подтверждение каждой замены, `g` - замена всех вхождений в строке.

Примечание. Поиск и замена в vi возможны только для последовательности символов, составляющих 1 строку. Заменяющая последовательность символов тоже должна образовывать 1 строку.

4.9. Вызов внешних команд

Редактор vim часто называют средой, так как он позволяет полноценнно работать в системе, не выходя из редактора.

Из vi можно запускать внешние программы с помощью команды ex-режима `:! cmdlline`:

Пример
```
:! ls -l

:! man bash
```

[6] Это справедливо только для vim

Так работать гораздо удобнее, так как нет надобности постоянно входить и выходить из редактора. При запуске командной строки ее вывод будет сохранен на экране до нажатия пользователем клавиши ENTER.

Глава 5

Регулярные выражения. sed

Регулярные выражения (regular expression или regexp) - специальные строки символов, которые задаются для поиска совпадающих фрагментов. Иначе говоря это способ описания наборов букв.

Простейшим набором является слово, но регулярное выражение может включать и глобальные символы, заменяющие другие символы. Все UNIX-программы, осуществляющие поиск в тексте, используют регулярные выражения. Если слово или фраза описаны регулярным выражением, говорят, что они соответствуют регулярному выражению.

Регулярные выражения - мощное, гибкое и эффективное средство обработки текстов. Универсальные шаблоны регулярных выражений сами по себе напоминают миниатюрный язык программирования, предназначенный для описания и анализа текста. При дополнительной поддержке со стороны конкретной утилиты или языка программирования регулярные выражения способны вставлять, удалять, выделять текстовые данные любого вида и выполнять практичяески любые операции над ними.

Регулярные выражения расширяют принципы *метасимволов (шаблонов или wildcards)*.

Некоторые программы используют регулярные выражения в чистом виде (grep, egrep). Но чаще всего регулярные выражения используются внутри специальных языковых конструкций, т. н. "оберток".

5.1. Структура регулярных выражений

Регулярное выражение состоит из двух типов символов. Специальные символы называются *метасимволами*. Все остальные символы (то есть обычный текст), называются *литералами*.

Регулярные выражения можно рассматривать как самостоятельный язык, в котором литералы выполняют функции слов, а метасимволы -

функции грамматических элементов. Слова по определенным правилам объединяются сграмматическими элементами и создают конструкции, выражающие некоторую мысль.

Для примера: существует утилита `grep`[1]. При запуске программе `grep` передается регулярное выражение и список просматриваемых файлов. Она сопоставляет regexp с каждой строкой файла и выводит только те строки, в которых было найдено совпадение.

```
$ grep 'cat' file1.text
```

Если в нашем выражении ($\ulcorner cat \lrcorner$ [2]) не используются метасимволы, оно фактически превращается в стредство "простого поиска текста". Будут найдены и выведены все строки файла, содержащие три стоящие подряд буквы `c, a` и `t`. Среди них будут выведены строки, в которых встречается слово (к примеру) `vacation`. Даже если в строке нет слова `cat`, последовательность букв `c a t` в слове `vacation` все равно считается успешно найденной. Необходимо только наличие указанных символов.

5.2. Правила для регулярных выражений

Существует всего два универсальных правила для регулярных выражений:

1. Предпочтение отдается тому совпадению, которое начинается раньше.

2. Квантификаторы (см. 5.5) всегда работают максимально. Если некоторый элемент может совпадать переменное число раз, механизм всегда пытается найти максимальное число повторений.

Признаки хорошо написанного регулярного выражения:

- регулярное выражение должно совпадать там где нужно и нигде более
- регулярное выражение должно быть понятным и управляемым
- оно должно быть эффективным (быстро приводить к совпадению или несовпадению в зависимости от результатов поиска)

[1] Утилиты семейства grep предназначены для поиска текста по шаблонам регулярных выражений

[2] Для grep рекомендуется заключать regexp в кавычки, так как некоторые метасимволы имеют для оболочки специальные значения и выражение может работать некорректно

5.3. Диалекты регулярных выражений

В разных программах регулярные выражения выполняют разные функции, поэтому наборы метасимволов и другие возможности, поддерживаемые программами, также различаются.

К примеру, диалекты регулярных выражений в sed, perl и grep имеют значительное число отличий между собой. Более того, различные варианты grep тоже могут использовать разные диалекты.

Далее изложение будет придерживаться диалекта `sed`, с указанием отличий от других реализаций регулярных выражений.

5.4. Метасимволы

Существует несколько типов метасимволов, выполняющих разные функции. Значение некоторых из них различно в разных частях выражения (или зависит от контекста).

5.4.1. Начало и конец строки

⌜^⌟ (крышка) и ⌜$⌟(доллар) представляют собой начало и конец проверяемой строки.

Примеры:

1. ⌜`^cat`⌟ находит все строки, в начале которых находиться `cat`.
2. ⌜`^cat$`⌟ находит все строки, которые состоят только из `cat`
3. ⌜`^$`⌟ пустая строка

Особенность ⌜^⌟ и ⌜$⌟ в том, что они совпадают с определенной *позицией* строки, а не с символами текста.

5.4.2. Символьные классы

Совпадение с одним символом из нескольких возможных

При помощи конструкции ⌜[...]⌟, называемой *символьным классом* (character class), можно перечислить символы, которые могут находиться в данной позиции текста.

Примеры:

1. ⌜$gr[ea]y$⌟. Это обозначает "найти символ `g`, за которым идет `r`, за которым следуют `e` или `a` и все это завершается символом `y`".
2. ⌜$[Ss]eparate$⌟. возможная смена регистра в первой букве

Количество символов в классе может быть любым. Например, класс ⌜[123456]⌟ совпадает с любой из перечисленных цифр.

В контексте (внутри) символьного класса *метасимвол символьного класса* - обозначает интервал символов; так выражение ⌜[1 − 6]⌟ эквивалентно предыдущему примеру. Классы ⌜[0 − 9]⌟ и ⌜[a − z]⌟ обычно используются для поиска цифр и символов нижнего регистра соответственно.

Символьный класс может содержать несколько интервалов, поэтому класс ⌜[0123456789abcdefABCDEF]⌟ записывается в виде ⌜[0−9a−fA−F]⌟. Интервалы также можно записывать вместе с литералами: ⌜[0 − 9_!.?A − Z]⌟ (совпадет со всеми цифрами, буквами в верхнем регистре и знаками подчеркивания, точки, восклицательного и вопросительного знаков).

Примечание 1. Дефис выполняет функции метасимвола только внутри символьного класса - в остальных случаях он совпадает с обычным дефисом в строке.

Примечание 2. Правила, определяющие состав поддерживаемых метасимволов (и их функции) внутри класса и за его пределами, полностью различны.

Примечание 3. Дефис не интерпретируется как метасимвол, если он находиться на первой позиции класса, например ⌜[-./]⌟.

Инвертированые символьные классы

Есле вместо ⌜[...]⌟ используется запись ⌜[^...]⌟, класс совпадает с любыми символами *не входящими* в приведенный список. Пример: ⌜q[^u]⌟.

Префикс ^ инвертирует список - вместо того, чтобы перечислять символы, принадлежащие классу, перечисляются символы, не входящие в него.

Примечание. Инвертированный класс означает "совпадение с символами не входящими в список", а не "несовпадение с символами, входящими в список". Поэтому инвертированный класс удобно рассматривать как сокращенную форму записи для обычного класса, включающего все символы, *кроме* перечисленных.

5.4.3. Один произвольный символ

Метасимвол ⌜.⌟ (точка) представляет собой сокращенную форму записи для символьного класса, содержащего *все* символы. Применяется в тех случаях, когда в некоторых позициях регулярного выражения могут находиться произвольные символы.

Пример: пусть надо найти дату, которая может быть записана в формате `07/04/76`, `07-06-76` или `07.06.76`. Самый простой вариант - ⌜07.04.76⌟. Но такое выражение будет совпадать и со строкой `19 207304 7639`.

Выражение ⌜07[−./]04[−./]76⌟ обеспечевает более точное совпадение, но его труднее читать и записывать.

При построении регулярных выражений часто приходиться идти на компромис с точностья за счет знания текста. Если вы уверены, что в тексте ⌜07.04.76⌟ наверняка не вызовет нежелательных совпадений, то этим вариантом вполне можно воспользоваться.

Замечание. *Знание целевого текста - важный фактор, обеспечивающий эффективное использование регулярных выражений*

5.4.4. Выбор

Одно из нескольких выражений

Очень удобный символ ⌜\|⌟ [3] обозначает "или". Он позволяет объединить несколько регулярных выражений в одно, совпадающее с любым из выражений-компонентов.

Например, ⌜*Erik*⌟ и ⌜*Bobby*⌟ - два разных выражения, а ⌜*Erik*\|*Bobby*⌟ - одно выражение, совпадающее с любой из этих строк. Подвыражения, объединенные этим способом, называются *альтернативами* (alternatives).

Конструкция выбора всегда является высокоуровневой (то есть обладающей очень низким приоритетом).

Вернемся к примеру 1 из 5.4.2 ⌜*gr*[*ae*]*y*⌟. Это выражение можно записать также в виде ⌜*gray*\ | *grey*⌟ или даже ⌜`gr\(a\|e\)y`⌟[4]. Здесь круглые скобки `\(` и `\)` отделяют конструкцию выбора от остального выражения. Без скобок ⌜*gra* | *ey*⌟ будет означать "⌜*gra*⌟ или ⌜*ey*⌟".

Выражение внутри скобок может быть как угодно сложным, но "снаружи" оно воспринимается как единое целое.

Примечание. Не путайте конструкцию выбора с символьным классом. Класс ⌜*abc*⌟ и конструкция выбора ⌜`\(a\|b\|c\)`⌟ фактически обозначают одно и то же, но это не для общего случая. Символьный класс совпадает ровно с одним символом, каким бы длинным или коротким не был список допустимых символов. С другой стороны, конструкция выбора может содержать альтернативы произвольной длинны, совершенно не связанные друг с другом длиной текста: ⌜`\(1.000.000\|million\|thousand*thousand\)`⌟. В отличие от символьных классов, конструкции выбора не могут инвертироваться.

5.4.5. Границы слов

Одна из распространенных проблем заключается в том, что искомое слово встречается внутри других слов. Для явного указания начала и конца слова используются *метапоследовательности* ⌜\ <⌟ и ⌜\ >⌟

[3] Для Perl и egrep - |

[4] Для диалекта sed. Для egrep или perl это будет gr(a|e)y.

Как и якоря `^` и `$`, эти метапоследовательности не соотвествуют конкретным символам.

Примеры:

1. ⌜`\<cat\>`⌟ - найти отдельное слово `cat`
2. ⌜`\<free`⌟ - найти слово, начинающееся с `free`, к примеру `freeware`
3. ⌜`ed\>`⌟- найти слова, заканчивающиеся на `ed`

Примечание Сами по себе символы ⌜<⌟ и ⌜>⌟ метасимволами не являются. Они приобретают особый смысл только в сочетании с обратным слэшем \.

5.5. Квантификаторы

Квантификаторы регулируют количество экземпляров повторяющегося элемента. Сами по себе, квантификаторы не являются щаблонами символов в тексте, но поставленные после символа или выражения в скобках, указывают, сколько раз может повторяться этот символ или выражение.

Квантификаторы руководствуются критерием максимального совпадения и пытаются найти совпадение как можно большей длины.

5.5.1. Необязательные элементы

Метасимвол ⌜`\?`⌟ [5] (вопросительный знак) означает "необязательный символ". Он ставиться после символа, который может находиться в данной позиции текста, но наличие которого не требуется для успешного совпадения. Вопросительный знак относиться *только* к символу, расположенному непосредственно перед ним.

Пример: ⌜`colou\?r`⌟

Пример2; Пусть нам надо найти дату, содержащую четвертый день месяца. На английском, это будет выглядеть так: `4` или `4th` или `fourth` - ⌜$fourth \mid 4 \mid 4th$⌟. Вторую половину выражения можно сократить до ⌜`4\(th\)\?`⌟. Получим ⌜`fourth\|4\(th\)\?`⌟.

Таким образом, квантификатор ⌜`\?`⌟ может присоединяться и к выражениям в скобках.

5.5.2. Повторение

- Метасимвол ⌜`\+`⌟ [6]обозначает "один или несколько экземпляров непосредственно предшествуюших элементов".

[5] Для Perl, egrep - ?.
[6] Для perl и egrep +

- Метасимвол ⌜*⌟ обозначает "любое количество экземпляров элемента (в том числе и нулевое)".

Иначе говоря, ⌜*⌟ означает "найти столько экземпляров сколько возможно, но при необходимости обойтись и без них". Конструкция ⌜\+⌟ имеет похожий смысл, но при отсутствии хотя бы одного экземпляра сопоставление завершается неудачей.

Примеры:

1. ⌜^[0-9]\+⌟ - строка, начинающаяся с одной или более цифр
2. ⌜^[0-9]*$⌟ - строка, содержащая в себе только цифры(может быть и пустой)
3. ⌜.*⌟ - любое количество любых символов
4. ⌜*⌟ и ⌜+⌟ могут следовать за скобками: ⌜\(th\)i\+⌟ - одно и более сочетаний букв `th` подряд
5. ⌜␣\+⌟ - один или более пробелов

5.5.3. Интервал

Конструкция вида ⌜...\{min,max\}⌟[7] называется *интервальным* квантификатором.

Например, выражение ⌜...\{3,12\}⌟ совпадает до 12 раз, если это возможно, но может ограничиться и всего 3 совпадениями. Запись `\{0,1\}` эквивалентна метасимволу `\?`, `\{1,\}` - `+`.

5.6. Круглые скобки и обратные ссылки

Мы уже знакомы с двумя применениями круглых скобок:

- ограничение области действия `|` (см. 5.4.4)
- группировка символов для применения квантификаторов (см. 5.5)

Существует еще одно применение круглых скобок. Круглые скобки могут "запоминать" текст, который совпал с подходящим в них подвыражением.

Обратные ссылки позволяют искать новый текст, который совпадает с другим текстом в предшествующей части регулярного выражения, причем на момент написания выражения этот текст *неизвестен*.

Круглые скобки "запоминают" текст, а специальный метасимвол ⌜\1⌟ представляет этот текст (каким он бы не был) в оставшейся части регулярного выражения.

[7] В egrep и perl она выглядит как ...{min,max}

В выражение можно включить несколько пар круглых скобок и ссылаться на совпавший текст с помощью ⌜\1⌟ , ⌜\2⌟, ⌜\3⌟ и т.д. Пары скобок нумеруются в соответствии с порядковым номером открывающей скобки справа налево.

Пример. Пусть нам надо найти повторяющиеся слова. Если известно конкретное слово, то можно включить его в шаблон, например, ⌜*the the*⌟. Но все пары слов проверить таким образом невозможно. Нам надо найти одно "обобщенное" слово, а потом указать искать то же самое. Заменим ⌜*the*⌟ регулярным выражением для обобщенного слова - ⌜`[A-Za-z]\+`⌟ и запомним его в круглых скобках - ⌜`\([A-Za-z]\+\)`⌟. Добавим выражение для пробелов - ⌜␣\+⌟. Теперь объединим полученные выражения и добавим обратную ссылку ⌜`\([A-Za-z]\+\)`␣\+\1⌟. И последнее - обозначим границы слов[8] - ⌜`\<\([A-Za-z]\+\)`␣\+\1\>⌟.

5.7. Экранирование

Чтобы включить в выражение символ, который совпадает с метасимволом, необходимо выполнить *экранирование*. Экранирование выполняется с помощью символа \.

Например: метасимвол "точка" ⌜.⌟ совпадает с любым символом. Чтобы получить обычную точку, надо записать ⌜\.⌟, которая называется "экранированной" (escaped) точкой.

Экранирование может выполняться со всеми стандартными метасимволами, кроме метасимволов символьных классов.

5.8. sed

sed (sequential или stream editor) неинтерактивный (поточный) редактор текста. Он служит для выполнение анализа и преобразования текста. Фактически sed - это продвинутый текстовый фильтр. Имеет свой входной язык, тесно связанный с регулярными выражениями.

Очень удобен для использования в скриптах оболочек как средство обработки текстов.

Далее будет описываться GNU sed.

sed можно использовать двумя основными способами:

```
sed [-n] [-e] 'команды редактирования' входной_файл
```

и

```
sed [-n] -f сценарий входные_файлы
```

[8] Иначе будут найдены не только повторяющиеся слова, но и сочетания, когда буква, завершающая слово, является первой для следующего.

Чаще используется первый способ.

Параметры:

- -f cmdfile прочитать сценарий из файла
- -n блокирование вывода, кроме явно задаваемого из сценария

Если команд несколько, то они разделяются `;`.

Входные файлы: редактируемый входной поток. Если не указывать имя файла, то sed будет работать со стандартным вводом. Результат выводится в стандартный вывод и обычно перенаправляется в файл или конвейер. Если входных файлов несколько, то они объединяются в один буфер, с которым и идет потом работа.

Строки в буфере пронумерованы. Если sed применяется к нескольким файлам, то номера строк будут продолжаться. Если первый файл содержит 200 строк, то адресом первой строки следующего файла будет 201.

Примечание. входные файлы *не изменяются.*

Схема работы:

Входные файлы (stdin) считываются в область шаблонов (pettern buffer), после этого к буферу последовательно применяются команды, и затем результат выводится для сохранения или дальнейшей обработки.

Внимание: крайне не рекомендуется перенапрвлять результат в исходные файлы. Это приводит к непредсказуемым результатам.

5.8.1. Общий вид команды

`[адрес1[, адрес2]] функция [аргументы]`

Функция: представляет собой букву команды. Единственный обязательный параметр. Например `'p'`.

Адрес: может быть номер строки, регулярное выражение, $ (последняя строка).

Если в адресе задается регулярное выражение, то оно задает все строки, соответствующие регулярному выражению. Регулярное выражение берется в `/` (прямой слэш), то есть `/regexp/`.

Если не заданы адреса, то обрабатываются все строки буфера.

Адресный интервал - это пара адресов, разделенная `","` и включающая все строки, начиная со строки, соответствующей первому адресу, до строки, соответствующей второму адресу включительно. Если второй адрес раньше первого, то обрабатывается только первая строка, соответствующая первому адресу.

При добавлении символа `!` после адреса смысл меняется на противоположный: обрабатываются все строки, не лежащие в интервале.

Примеры: `1,4; 1,$; 2,6!`.

5.8.2. Команды sed

Описаны только часто используемые команды.

Замена

`s/regexp/replacement/flags`.

`s` - буква команды (замена, подстановка - substitute), `regexp` - строка поиска, то есть то, что заменится на `replacement`.

Флаги:

- `g` заменить все вхождения
- `w` file записать изменения в файл
- `p` после замены вывести строку на экран (обычно используется с ключом sed `-n`).

```
Примеры:
1. $ sed 's/sun/moon/g' myfile
2. $ sed '1,4 !s/sun/moon/g' myfile
3. $ echo Жужжали бабочки | sed -n 's/\(жж\)\(али\)/Гуж<&>\2\1/p'
# Результат: ЖуГуж<жжали>алижж бабочки
4. $ sed '/^Example/,/ED$/s/first/second/g'
# если несколько совпадающих строк, то редактируются все вхождения.
5. $ sed 's/Sunday/Monday/gw' changes
# все Sunday заменяются
```

Удаление строк

`d`

```
Примеры:
1. $ sed '4,5 d' file
2. $ sed '/sun/ !d' file.txt
# удаление всех строк, кроме содержащих sun.
3. $ sed '/sun/,/moon/ d' myfile
# удаляется диапазон от первой строки, содержащей sun
до первой строки, содержащей moon
```

Вывод на экран

`p`

Обычно используется с `sed -n` (иначе строки будут выводиться два раза).

```
Примеры:
$ sed -n '/stroka/i !p'.
$ sed -n ' 1,4 p'
# вывести строки с 1 по 4 включительно
```

Трансляция символов

`y/source_chars/dest_chars/`

Замена символов по принципу "один к одному" (строки должны быть одной длины). Пример: `$ sed 'y/abc/ABC/' file`

Запись в файл

`w file` - пишет буфер в файл.

Вставка файла

`r file` - вставка в выходной поток файла. Если его нет, то вставляется файл нулевой длины (без ошибки).

Вставка строк

- `адрес a` помещает за обрабатываемой строкой
- `адрес i` выводит до указанной строки

Примечание: a и i разрешают использовать только один адрес.

```
Пример:
$ cat script
  3 a\
Здесь добавлена\
строка
$ who | sed -f script
 root
 stud1
 stud10
        Здесь добавлена
        строка
 stud11
```

Символ экранирования "\" необходим, чтобы скрыть все символы конца строки кроме последнего.

Глава 6

Файловая система ОС UNIX

С точки зрения пользователя в ОС UNIX существует два типа объектов: файлы и процессы.

Все данные хранятся в виде файлов, доступ к периферийным устройствам осуществляется через чтение/запись в специальные файлы.

При запуске программы ядро загружает соответствующий исполняемый файл, создает образ процесса и передает ему управление.

Во время выполнения процесс может считывать или писать данные в файл. С другой стороны, вся функциональность ОС определяется выполнением соответствующих процессов.

Таким образом, понятия файловой системы и процессов тесно взаимосвязаны.

6.1. Базовые сведения о файловой системе

В UNIX файлы организованы в виде *древовидной структуры* (дерева), называемой *файловой системой* (FS или file system).

Каждый файл имеет имя, определяющее его расположение в дереве FS.

Корнем дерева является *корневой каталог* (root directory), имеющий имя "/".

Имена всех файлов, кроме "/", содержат *путь - список каталогов, которые надо пройти, чтобы достичь файла.* Все доступное файловое пространство объединено в единое дерево каталогов, корнем которого является каталог "/". Таким образом, полное имя любого файла начинается с "/". Полное имя файла не содержит идентификатора устройства (HDD, CD-ROM или удаленного компьютера в сети), на котором он фактически находится. Символ "/" является разделителем в структуре каталогов.

Каждый файл имеет связанные с ним *метаданные* (хранящиеся в индексных дескрипторах - *inode*), содержащие все характеристики файла и позволяющие ОС выполнять операции над ним.

Метаданные хранят *права доступа*, *владельца-пользователя* и *владельца-группу*, указатели на дисковые блоки, хранящие данные. *В метаданных нет* сведений об *имени файла*.

6.2. Типы файлов

В UNIX существует шесть типов файлов, различающихся по строение и поведению при выполнении операций над ними:

6.2.1. Обычный файл (regular file)

Это наиболее общий тип файлов, содержащий данные в некотором формате. Для ОС это просто последовательность байт. Интерпретация содержимого производится прикладной задачей.
Пример: текстовый файл, двоичные данные, исполняемый файл. Их можно просматривать командами `cat имя` и `less имя`.

6.2.2. Каталог (directory)

Это файл, содержащий имена находящихся в нем файлов, а также указатели на метаданные этих файлов, позволяющие ОС производить операции над ними.

Каталоги определяют положение файла в дереве файловой системы, так как сам файл не содержит информации о своем местонахождении. Каталоги образуют дерево.

Пример:

Номер inode	Имя файла
10245	.
12432	..
8672	file1.txt
12567	first
19678	report

Для работы с каталогами используются команды: `ls` с ключами `-a` и `-l`, `cd`, `mkdir`, `rm`, `rmdir`, `mv`.

Первые два байта в каждой строке каталога являются единственной связью между именем файла и его содержимым. Именно поэтому *имя файла в каталоге называют связью*. Оно связывает имя в иерархии каталогов с индексным дескриптором и, тем самым, с информацией.

6.2.3. Специальный файл устройства (special device file)

Обеспечивает доступ к физическому устройству. Различают символьные и блочные файлы устройств. Доступ к устройствам происходит путем открытия, чтения/записи в специальный файл устройства. *Символьные файлы* позволяют небуферизованный обмен данными (посимвольно), а *блочные* - обмен пакетами определенной длины - блоками. К некоторым устройствам доступ возможен как через символьные, так и через блочные файлы.

Для создания файлов устройств используется команда `mknod`.

6.2.4. FIFO или именованный канал (named pipe)

Используется для связи между процессами. Подробно будет рассмотрен при описании системы межпроцессного взаимодействия (см. **??**).

6.3. Связь (ссылка)

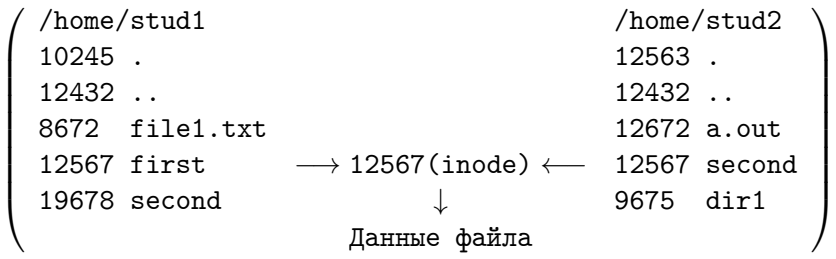
6.3.1. Жесткая ссылка

Связь имени файла с его данными называется *жесткой ссылкой* (hard link). Имена жестко связаны с метаданными и, соответственно, с данными файла, в то время, как файл существует независимо от того, как его называют в файловой системе. Такая система позволяет одному файлу иметь несколько имен в файловой системе.

```
Пример:
$ pwd
/home/stud1
$ln first /home/stud2 second
# создание жесткой ссылки.
```

Все жесткие ссылки на файл абсолютно равноправны.

Файлы `first` и `second` будут отличатся только именем в файловой системе. Изменения, внесенные в любой из этих файлов, затронут и другой, так как они ссылаются на одни и те же данные. Даже при переносе файлов в другой каталог все равно они будут жестко связаны.



Файл существует в системе до тех пор, пока существует хотя бы одна жесткая связь, указывающая на него, то есть пока у него есть хотя бы одно имя. Например, простое удаления файла `second` не удаляет данные. Их можно достать через `first`.

В выводе команды `ls -l` вторая колонка показывает количество жестких связей файла.

Таким образом, жесткая связь не принадлежит к особому типу файлов, а является естественной формой связи имени файла с его метаданными.

Жесткие ссылки можно создать командой `ln` (link).

6.3.2. Символическая ссылка

Особый тип связи - символическая связь, позволяющая косвенно адресовать файл, в отличие от жесткой, обращающейся напрямую. Символическая ссылка содержит в себе имя файла, на который ссылается, а не его данные.

Физическое расположение файлов различно. Размер `symfirst` - длина имени файла, на который ссылается символическая связь. ОС работает с `symfirst` не так, как с обычным файлом: при обращении к нему появятся данные `first`.

6.3.3. Сокет (socket)

Используются для межпроцессного взаимодействия. Будут подробнее рассмотрены в соответствующей теме (см. **??**.

6.4. Структура файловой системы

Все Unix-системы имеют сходную систему расположения и именования файлов и каталогов. Использование общепринятых имен файлов и структуры каталогов в UNIX-подобных ОС облегчает работу и перенос. Нарушение структуры ведет к нарушениям в работе.

Корневой каталог `"/"` является основой FS. Все остальные файлы и каталоги располагаются в рамках структуры, порождаемой корневым каталогом.

Абсолютное или полное имя файла определяет точное местонахождение файла в структуре файловой системы. Начинается с `"/"` (в корневом каталоге) и содержит полный путь подкаталогов, которые нужно пройти, чтобы достичь файла.

Относительное имя определяет местонахождение файла через текущий каталог. Никогда не начинается с `"/"`.

Каталог-предок - это тот, который содержит другой каталог. Две точки (`..`) как имя каталога всегда относятся к каталогу, содержащему

текущий каталог. Корневой каталог не имеет предка. Каталог, находящийся в другом каталоге, называется *каталогом-потомком или подкаталогом*. К текущему каталогу можно обратиться по имени `"."`. Например, `./file1`.

Домашним или начальным каталогом называется область, котрая выделяется каждому пользователю и в которой он может хранить свои файлы и программы. К своему домашнему каталогу пользователь может обратиться по имени `~` (тильда). Например, `~/file.txt`.

6.4.1. Основные каталоги

1. `/bin` - наиболее часто употребляемые файлы и утилиты.
2. `/dev` - содержит специальные файлы устройств, являющиеся интерфейсом доступа к периферийным устройствам. Может содержать подкаталоги, группирующие устройства по типам. Например, `/dev/dsk` - доступ к дискам.
3. `/etc` - системные конфигурационные файлы и утилиты. Иногда утилиты отсюда выносятся в `/sbin` и `/usr/sbin`.
4. `/lib` - библиотеки Си и других языков программирования. Часть библиотек - в `/usr/lib`.
5. `/lost`found+ - "каталог потерянных файлов", то есть потерявших свое имя при сбое, но существующих на диске.
6. `/mnt` - для временного связывания (монтирования) физических файловых систем к корневой для получения единой структуры.
7. `/home` - каталоги пользователей.
8. `/usr`
   - `/usr/bin` - утилиты;
   - `/usr/include` - заголовочные файлы Си;
   - `/usr/man` - справочная система;
   - `/usr/local` - дополнительные программы;
   - `/usr/share` - файлы, разделяемые между различными программами.
9. `/var` - временные файлы сервисных подсистем (печати, почты, новостей).
10. `/tmp` - каталог временных файлов. Обычно открыт на запись для всех пользователей системы.

6.5. Атрибуты файлов

6.5.1. Владельцы файлов

Группой называется определенный список пользователей системы. Пользователь может быть членом нескольких групп, одна из которых является первичной, а остальные - дополнительными.

`/etc/passwd` - список всех пользователей и их первичных групп; `/etc/group` - список всех групп и их дополнительных пользователей. В UNIX любой файл имеет двух владельцев:

1. владельца-пользователя
2. владельца-группу.

При этом владелец-пользователь не обязательно принадлежит владельцу-группе.

Команда `ls -l` выводит информацию о владельцах в третью и четвертую колонки. Для изменения владельцев используются команды: `chown новый_влад. имя_файла`. Например: `chown sys something.doc`. `chgrp новый_влад. имя_файла`. Например: `chgrp adm something.doc`.

Сменить владельца-пользователя может либо текущий владелец, либо администратор (root). Сменить владельца-группу может либо владелец-пользователь для группу, к которой он сам принадлежит (POSIX), либо администратор.

6.5.2. Права доступа к файлам

У каждого файла существуют атрибуты, называемые правами доступа. В UNIX существует три базовых типа доступа:

1. `u` (user) для владельца-пользователя
2. `g` (group) для владельца-группы
3. `o` (other) для всех остальных
4. `a` (all - объединяет 3 предыдущих класса). Для всех классов пользователей

В каждом из этих классов установлены три основных права доступа:

1. `r` (read) право на чтение
2. `w` (write) право на запись
3. `x` (execute) право на выполнение

В первой колонке вывода команды `ls -l` можно просмотреть установленные права.

```
Пример:
$ ls -l
 - r w - r - - r w x     1   stud1    students  ...  example.program
 0 1 2 3 4 5 6 7 8 9
0 - тип  файла: - обычный; d каталог; l символическая ссылка;
    c,b символьный/блочный файл устройств.
1-3 - права доступа для владельца-пользователя.
4-6 - права доступа для владельца-группы.
7-9 - права доступа для остальных.
```

Права может изменять владелец-пользователь и(или) администратор. Для изменения прав доступа используется команда `chmod`:

$$\texttt{chmod} \quad \begin{bmatrix} u \\ g \\ o \\ a \end{bmatrix} \begin{bmatrix} + \\ - \\ = \end{bmatrix} \begin{bmatrix} r \\ w \\ x \end{bmatrix} \quad \texttt{файлы}$$

$+$ добавить права к текущим
$-$ отнять права от текущих
$=$ обнулить права и присвоить новые

```
Пример:
$ chmod a+w text
# добавить разрешение писать всем пользователям;
$ chmod go=r text
# установить только одно право на чтение для всех кроме владельца-пользователя;
$ chmod g+x-r program
# добавить для группы право на выполнение и отнять у нее право читать;
$ chmod u+w, og+r-w text2;
```

Возможно также задание прав через числовой формат в восьмеричной системе счисления.

Пример: `chmod 666 *`.

6.5.3. Значение прав доступа

Для обычных фалов - очевидно: право на чтение надо, чтобы прочитать файл, право на запись, чтобы иметь возможность файл изменить, а право на выполнение, чтобы запустить программу или скрипт.

Примечание. Для успешного запуска скрипта необходимо установить атрибут r, чтобы командный интерпретатор мог построчно считывать текст скрипта.

Для каталогов и символических связей интерпретация прав доступа проводится по-другому.

Права символических ссылок совпадают с файлом, на который она указывает. На самой ссылке стоит 777 (всем все) и это не имеет значения.

Для каталогов `r` позволяет получить имена (и только имена) файлов, находящихся в данном каталоге. `X` позволяет "выполнить" каталог,

то есть заглянуть в метаданные и получить полную информацию о каталоге.

```
Пример:
$ chmod u+r-x dir1
$ ls dir1      - выполнится
$ ls -l dir1  - Permission denied
$  cd dir1     - Permission denied (надо x).
```

r и x для каталога действуют независимо (одно не требует другого).

```
Пример:
$ mkdir dark_dir
$ chmod a-r+w dark_dir
$ ls dark_dir  -выполниться
$ ls -l        -нет
$ cat file1
# yes (заранее зная имя файла, можно обратиться к нему).
```

Атрибут w должен быть установлен для того, чтобы можно было изменять каталог: создавать и удалять файлы. Для удаления файла из каталога достаточно иметь установленный атрибут w для каталога, в котором он находился, а права файла при этом не учитываются.

6.5.4. Последовательность проверки прав

1. если вы администратор (root), доступ разрешен. Права не проверяются.

2. если операция запрашивается владельцем, идет проверка его прав. В соответствии с ними ему разрешается выполнение операции или нет.

3. если операция запрашивается пользователем, входящим в группу, владеющую файлом, идет проверка его прав. Соответственно, он либо получает разрешение, либо нет.

4. аналогично для всех остальных пользователей.

```
Пример:
----rwr--  2  stud1   students ... file1
stud1 в доступе будет отказано, но он, как владелец,
может в любой момент сменить права доступа.
```

6.5.5. Дополнительные атрибуты файла

Для обычных файлов:

- `t` - *"sticky bit"* (бит липучка)- сохранить образ выполняемого файла в памяти после выполнения (устаревший аттрибут)
- `s` - set UID, *SUID* - установить права у процесса, как у запущенного файла, а не как у пользователя, запустившего программу (по умолчанию)
- `s` - set GID, *SGID* - то же для группы
- `1` - блокирование - в каждый момент времени с файлом может работать только одна задача

Для каталогов:

- `t` - пользователь может удалять только те файлы и каталоги, которыми владеет или имеет право на запись;
- `s` для создаваемых файлов группа-владелец наследуется от каталога-предка (а не от первичной группы пользователя, создающего файл).

Дополнительные атрибуты также устанавливаются с помощью `chmod`.

Глава 7

Процессы

Процесс - это экземпляр выполняющейся программы.

Программа - совокупность файлов, будь то исходники, объектные файлы либо выполняемый файл.

Для запуска программы на выполнение ОС должна создать *окружение* или среду выполнения задачи, куда относится ресурсы памяти, возможность доступа к устройствам ввода/вывода и различным системным ресурсам.

Процесс состоит из инструкций, выполняемых процессором, данных и информации о выполняемой задаче, такой, как размещенная память, открытые файлы и статус процесса.

Программа может породить более одного процесса. Пользователи могут запускать несколько экземпляров одной программы. Например, количество BASH - эквивалентно числу пользователей. Таким образом UNIX - многозадачная ОС.

Выполнение процесса заключается в точном следовании набору инструкций, который никогда не передает управление набору инструкций другого процесса. Процесс взаимодействует со своими данными и стеком, но ему не доступны чужие данные и стек.

Процессы изолированы друг от друга. В то же время, процессы имеют возможность обмениваться друг с другом данными с помощью системы межпроцессного взаимодействия *(IPC)*.

Виды IPC:

1. сигналы
2. каналы
3. разделяемая память
4. семафоры
5. сообщения
6. файлы.

7.1. Типы процессов

7.1.1. Системные процессы

Системные процессы являются частью ядра и всегда расположены в оперативной памяти. Системные процессы не имеют соответствующих им программ в виде исполняемых файлов и запускаются особым образом при инициализации ядра системы.

```
Примеры:
1)диспетчер свопинга
2)диспетчер памяти
и другие.
```

Выполняемые инструкции и данные системных процессов находятся в ядре, таким образом, они могут обращаться к функциям и данным, не доступным извне (ядра).

Процесс `init` можно также отнести к системным, хотя он запускается из файла. Он прародитель всех процессов системы. `init` запускается первым после загрузки ядра и запускает все остальные инициализационные задачи на выполнение.

7.1.2. Демоны

Демоны - неинтерактивные процессы, запускаемые обычным образом, и выполняются в фоновом режиме. Они не связаны ни с одним пользовательским сеансом и не могут непосредственно управляться пользователем. Обеспечивают работу различных подсистем:

- печати
- сетевого доступа
- терминального доступа
- почта
- web-сервера
- СУБД.

7.1.3. Прикладные процессы

Прикладные процессы - все остальные процессы. Как правило, порождаются в рамках пользовательского сеанса.

`Пример: ls, BASH.`

Пользовательские процессы могут выполняться как в интерактивном, так и в фоновом режиме, но время из жизни (выполнения) ограничено сеансом работы пользователя. При выходе из системы все пользовательские процессы будут уничтожены.

Примечание: интерактивные процессы монопольно владеют терминалом, и, пока такой процесс не завершит выполнение, пользователь не может работать с другими приложениями. (Кроме случаев, когда есть режим запуска других процессов из этого интерактивного процесса.)

7.2. Атрибуты процессов

Атрибуты позволяют ОС эффективно управлять работой процесса.

Просмотр атрибутов процесса: `ps -ef`.

7.2.1. Идентификатор процесса

Идентификатор процесса - *Process ID (PID)* - каждый процесс имеет уникальный идентификатор, позволяющий ядру системы различать процессы. При создании нового процесса, ядро присваивает ему следующий свободный идентификатор. Присвоение PID - по возрастающей, то есть PID нового процесса больше, чем PID процесса, созданного перед ним.

Если PID достиг максимального значения, следующий процесс получит минимальный свободный и цикл повторяется.

Когда процесс завершает работу - ядро освобождает занятый им PID.

7.2.2. Родительский процесс

Идентификатор родительского процесса -Parent Process ID (PPID) - PID процесса, породившего данный.

7.2.3. Приоритет процесса

Приоритет процесса (nice number) - относительный приоритет процесса, учитываемый планировщиком при определении очередности запуска. Чем меньше число, тем больше приоритет (nice - приятный, то есть чем более "приятный" процесс, тем меньше он загружает CPU).

Фактическое распределение ресурсов - приоритет выполнения: динамически изменяется ядром во время выполнения. Относительный - постоянен, но может изменяться администратором или пользователем с помощью nice.

7.2.4. Терминальная линия

Терминальная линия (TTY) - терминал или псевдотерминал, ассоциированный с процессом.

Примечание. Демоны не имеют ассоциированного терминала.

7.2.5. Идентификаторы пользователей

Реальный (RID) и эффективный (EUID) идентификаторы пользователя. RID - идентификатор пользователя, запустившего этот процесс. EUID служит для определения прав доступа процессак системным ресурсам (в первую очередь к файловой системе.) Обычно RID=EUID, то есть процесс имеет те же права, что и пользователь, запустивший его. RID!=EUID, когда на программе установлен бит SUID. Тогда EUID=UID, то есть процесс получает те же права, что и у владельца исполняемого файла (например, администратор).

7.2.6. Идентификаторы групп

Реальный (RGID) и эффективный (EGID) идентификаторы группы. RGID=GID первичной группы пользователя, запустившего процесс. EGID служит для определения прав доступа пользователя по классу доступа группы. По умолчанию RGID=EGID, кроме SGID, установленного на команду, тогда EGID=GID группы-владельца команды.

7.3. Жизненный путь процессов

Процесс в UNIX создается системным вызовом `fork(2)`.

Процесс, сделавший вызов fork(2), называется *родительским*, а вновь созданный - *дочерним*. Новый процесс является точной копией породившего его процесса.

Примечание. новый процесс имеет те же инструкции и данные, что и родитель. Более того, выполнение родительского и дочернего начнется с одной и той же инструкцией, следующей за системным вызовом fork. Единственное их отличие - идентификатор PID.

Каждый процесс имеет одного родителя, но может иметь несколько потомков.

Для запуска задачи, то есть загрузки новой программы, процесс должен сделать вызов `exec(3)`. При этом новый процесс не порождается, а исполняемый код нового процесса полностью замещается кодом запускаемой программы. Тем не менее сохраняются значения переменных окружения, назначение стандартных потоков ввода/вывода и ошибок, а также приоритет процесса.

В UNIX запуск на выполнение новой программы часто связан с порождением нового процесса. Таким образом, процесс сначала выполняет fork, порождая дочерний процесс, который затем выполняет exec, полностью замещая родительский процесс. Такая процедура запуска называется fork-and-exec.

Бывают ситуации, когда достаточно одного вызова fork без последующего exec. В этом случае исполняемый код родительского и дочернего процессов должен содержать логическое ветвление для родительского и дочернего процессов. (fork возвращает PID порожденного процесса в родительский и ноль - в дочерний.)

Все процессы создаются через вызов fork. Запуск осуществляется либо по fork-and-exec, либо с помощью exec. Прародителем всех процессов является init или распределитель процессов.

7.4. Сигналы

Сигналы являются способом передачи уведомления о возникновении какого-либо события. Сигнал может идти от одного процесса другому или от ядра ОС какому-либо процессу.

Сигналы - простейшая форма IPC.

Например, при делении на ноль процессу посылается сигнал SIGFPE, а при нажатии Ctrl+C на терминале текущему процессу посылается сигнал SIGINT.

Для отправки сигналов используется команда `kill`:
`$ kill sig_no pid`, где
sig_no - номер или символьное название сигнала;
pid - идентификатор процесса, которому посылается сигнал.

Пользователь может посылать сигналы только тем процессам, владельцем которых он является, то есть RID и EUID совпадают с UID пользователя. Администратор (root) может посылать сигналы всем процессам.

```
Пример: посылка сигнала процессу, только что запущенному в фоновом режиме
$ back_fone_prog &
$ kill $! (по умолчанию посылается SIGTERM, номер 15).
```

При получении сигнала процесс может реагировать следующим образом:

1. игнорировать сигнал.
   Замечание: не следует игнорировать аппаратно вызванные сигналы, например, SIGFPE.

2. действие по умолчанию. Обычно это завершение работы.

3. перехватить сигнал и самостоятельно обработать его. Например, перехват SIGINT позволит удалить все tmp-файлы и корректно завершить выполнение.

Исключение: SIGKILL и SIGSTOP нельзя ни перехватить, ни игнорировать.

Возможны ситуации, когда процесс не реагирует на SIGKILL:

1. процессы-зомби. Фактически он завершился, но осталась запись в системной таблице процессов.

2. процессы, ожидающие недоступные ресурсы NFS. Например, процессы пишущие данные в файл удаленного компьютера, который уже отключился. Проблему решают посылкой SIGINT или SIGQUIT.

3. процесс, ожидающий завершения операции с устройством, например, перемотка ленты или перепозиционирование головки CD-ROM на порченном диске.

Сигналы используются не только для завершения работы процессов, но и могут иметь специфическое значение для приложения. (Это не относится к SIGKILL и SIGSTOP, потому что их нельзя перехватить.) К примеру, системные демоны - proxy servers, smtp (pop, imap), СУБД, bind при получении сигнала SIGHUP должны перечитать свои конфигурационные файлы и рестартовать.

Глава 8

Среда программирования Unix

8.1. Unix-way программирования

8.2. Unix как единая среда разработки (IDE)

Из высказывания в ru.unix.prog Алексея Махоткина из ru.unix.prog FAQ :

> » Q: Какие есть IDE (integrated development environments) под Unix? Ну
> » чтобы компилятор, среда редактирования, отладчик и прочее - были все
> » вместе?
>
> ...
>
> UNIX сам по себе является Integrated Development Environment.
>
> В "обычных" IDE есть бинарник-интегратор, который вызывает в лучшем случае внешние утилиты, а в худшем случае – свою реализацию каждой функцию из DLL или прямо зашитую в бинарник.
>
> В UNIX таким бинарником-интегратором является shell (Emacs считается shell'ом в данном случае). Для выполнения каждой функции вызываются специально написанные динамически выполняемые модули, такие как make, cc, ld, и т. д.
>
> Преимущество в этом такое же, как преимущество математических функций высшего порядка перед "обычными" функциями.

> Например, функция "отслеживать зависимости" чаще всего реализуется с помощью make, но можно также легко использовать, скажем, cook, или же переключаться между GNU Make и BSD Make по вкусу. Точно такая ситуация с используемыми редактором, компилятором, etc. Более того, сам по себе shell является "функцией высшего порядка", и легко может быть заменен.
>
> Кроме того, так как пространство функций практически неограниченно, то IDE "Unix" обеспечивает также заранее не предусмотренные функции высшего порядка, например, различную автогенерацию кода, поддержку тестирования и т. п.

Другими словами, командная строка Unix (shell) и является IDE для Unix. Подобный подход позволяет не зацикливаться на програмных решениях одного производителя (разработчика). Любой компонент, воспринимаемый как часть IDE (компилятор, отладчик, компоновщик, редактор, ассемблер, утилиты сборки и тестирования проекта, система контроля версий) может быть заменен на другой. Эти компоненты мы условно можем назвать *Инструментальные средства Unix*.

8.3. Низкоуровневый доступ к системе

Базой UNIX-системы является компилятор Си (cc), библиотека libc и ядро.

Все версии UNIX предоставляют строго определенный, ограниченный набор входов в ядро ОС, через которые прикладные задачи получают доступ к базовым услугам UNIX. Эти точки входа называются *системными вызовами (system calls)*.

Системный вызов определяет функцию, выполняемую ядром ОС от имени процесса, выполнившего вызов. Syscall является интерфейсом самого низкого уровня взаимодействия прикладных процессов с ядром.

Язык системного программирования - Си. Характерная особенность Unix - ассемблер практически не используется. Более того, часть прерываний и регистров просто недоступна из program space. Ассемблер применяется только для написания драйверов устройств и ядра (платформенно-зависимые их части).

Библиотека *libc* - это набор интерфейсов к системным вызовам и различных функций, работающих поверх системных вызовов. Для программиста различие между *системным вызовом* и *библиотечной функцией* лишь в том, как они взаимодействуют с ядром. Системный вызов сразу уходит в пространство ядра и там выполняется. Библиотечная функция выполняется в пространстве процесса (хотя конечно может и делать системные вызовы в ходе выполнения).

8.4. Принципы разработки программ для Unix

> Чтобы плавать, надо плавать. Мао Цзе Дун (из красных книжечек председателя Мао)

За 30 лет существования вокруг Unix сформировалась своя культура: слэнг, традиции и опыт, передаваемый между поколениями программистов. Принципы - это набор философских высказываний, подводящий итоги и суммирующий опыт тысяч человеко-лет разработки.

Из Tao Of The Unix Programming (by Eric S. Raymond):

- Правило модульности: Пишите простые части, соединяемые ясными интерфейсами.
- Правило ясности: Ясность лучше чем изощренность.
- Правило соединения: Проектируйте программы чтобы они могли взаимодействовать с другими программами.
- Правило разделения: Отделяйте алгоритмы от механизмов и интерфейсы от движков.
- Правило простоты: Разрабатывайте просто. Используйте сложные конструкции только тогда, когда без этого не обойтись.
- Правило умеренности: Пишите большую программу, только если ясно, что больше ничего не поможет.
- Правило прозрачности: Пишите наглядно, чтобы сделать просмотр и отладку программы легче.
- Правило надежности: Надежность - следствие ясности и простоты.
- Правило представления: Храните знания в данных, а программная логика должна быть надежной и тупой.
- Правило наименьшего удивления: Когда разрабатываете интерфейсы, делайте их как можно более предсказуемыми.
- Правило тишины: Когда программе нечего сказать нового, она должна молчать.
- Правило восстановления: Когда программа должна ошибиться, она должна шумно и долго об этом вопить. Настолько часто, насколько это возможно.
- Правило экономии: Время программиста - дорогое, экономьте его, взваливая как можно больше задач на компьютер (автоматизация).

- Правило генерации кода: Избегайте ручного кодирования. Пишите программы для генерации других программ всегда, когда возможно.
- Правило оптимизации: Делайте прототипы перед полировкой кода. Заставьте код работать, прежде чем оптимизировать.
- Правило многообразия: Не доверяйте всем претензиям на "единственно верное решение"
- Правило рамширяемости: Проектируйте с прицелом на будущее. Оно может наступить быстрее чем вы думаете.

Философия Unix в одном уроке.
keep It Simple, Stupid - оставь это простым, тупица.

Глава 9

Инструментальные средства разработчика

9.1. Компилятор Си

Компилятор языка Си (C compliler или cc) - неотъемлемая часть системы. С него начинается разработка любой версии Unix или перенос на новую платформу существующей. Ядро и базовые утилиты системы написаны на Си[1].

Компилятор реализован как утилита командной строки. Он вызывается командой `cc`. Существует большое количество системных компиляторов Си [2] (gcc в Linux и FreeBSD, собственные компиляторы в большинстве коммерческих версий Unix), поэтому `cc` будет указывать на компилятор по умолчанию для нашей системы.

```
типовые ключи компилятора
$cc foo.c
компилирование и сборка (линковка) программы из 'foo.c'
создаётся исполняемый файл 'a.out'

$cc -c foo.c
только компилировать. Будет получен объектный модуль 'foo.o'

$cc -o exec_foo foo.c
создается исполняемый файл 'exec_foo' (вместо 'a.out')

$cc foo.c bar.o
```

[1] Более того, сам язык Си был создан для разработки ядра Unix. Авторы языка Си являются также первыми разработчиками Unix.

[2] В SUSv3(POSIX) опредено общее подмножество ключей и параметров, которые должен поддерживать компилятор Си. Этому набору следуют все системные компиляторы. Естественно, что каждый из них имеет и свои дополнительные параметры.

```
слинковать 2 объектных файла в исполняемый ('a.out')
```

9.2. make

Make - стандартное средство, применяемое для сборки программных проектов. Является универсальной программой для решения задач автоматической генерации и изменения файлов с учетом зависимостей.

Схема работы: make читает файл с описанием проекта (makefile) и, интерпретируя его содержание, выполняет определенные действия.

Makefile - текстовый файл, описывает отношения между файлами проекта и действия, необходимые для его сборки.

9.2.1. Запуск

Make является утилитой командной строки и запускается командой `make`. При запуске `make` проверяет наличие файлов `makefile`, `Makefile` и если и запускает на обработку первый из найденных Make-файлов. Можно явно указать какой Makefile использовать ключом `-f`.

9.2.2. Формат и использование make-файлов.

Основной элемент - *правила (rules)*.

```
Общий вид:
<цель 1> <цель 2> ?<цель n>:<зависимость 1> <завис-ть 2>?<завис-ть n>
        <команда 1>
        <команда 2>
        ?
        <команда n>
```

Цель (target) - некий желаемый результат, способ достижения которого описан в правиле. Цель может быть именем файла.

Примечание. Перед командами вставляется табуляция, чтобы `make` отличал их от целей.

```
Пример1: цель как имя файла
iEdit: main.o Editor.o
         gcc main.o Editor.o -o iEdit
Пример описывает, как можно получить исполняемый файл из объектных модулей.
```

Цель может быть именем некоторого действия, тогда правило описывает, как совершается указанное действие.

```
Пример2: цель как имя действия
clean:
        rm *.o iEdit
```

Такие цели называют *абстрактными* (phony targets) или псевдоцелями (pseudo targets).

Зависимость (dependency) - это 'исходные данные', необходимые для достижения указанной в правиле цели. Это предварительные условия достижения цели. Зависимостью может быть имя файла или имя действия. В примере1 main.o и Editor.o - зависимости. Файлы должны существовать, чтобы можно было собрать iEdit.

```
Пример3:
clean_all: clean_obj
        rm iEdit
clean_obj:
        rm *.o
 Для достижения clean_all необходимо выполнить действие clean_obj.
```

Команда - действия, которые надо выполнить для обновления или достижения цели. Перед командой должен быть символ табуляции (код 9). Так make определяет команды.

Типичный makefile, который содержит несколько правил, у каждого правила есть некоторая цель и зависимости.

```
Пример4:
1.  iEdit: main.o Editor.o
2.              gcc main.o Editor.o -o iEdit
3.       main.o: main.cpp
4.          gcc -c main.cpp
5. Editor.o: Editor.cpp
6.         gcc - Editor.cpp
7. clean:
8.         rm *.c
```

Смысл работы - достижение главной цели (default goal). Если цель - имя действия (абстрактная), то выполняется действие. Если главная цель - имя файла, то make строит самую свежую версию. Главная цель обычно задается как параметр make: make iEdit, make clean. Если make вызывается без параметров, то в качестве главной берется первая встреченная цель. (В примере это iEdit). Обычно задают цель *all* как цель по умолчанию.

Алгоритм работы:

1. выбор главной цели
2. достижение цели
3. обработка правил
4. обработка зависимостей

Достижение цели - проверяет зависимости и потом определяет, надо ли запускать команды. При вызове make iEdit определяет, что главная цель - iEdit. Правило ее достижения - строки 1,2. Обрабатывая правило iEdit, определяем, что зависит от main.o и Editor.o. Для этих зависимостей существуют правила (3,4) и (5,6). main.o зависит от main.cpp. Если нет еще объектного файла, но существует файл .cpp, то запускается компиляция. Аналогично и для Editor.o. Для clean зависимостей нет и make сразу переходит к выполнению.

Инкрементная сборка - перекомпилируется только то, что было изменено. Для файлов .c и .cpp обычно указываются как зависимости .h файлы.

9.2.3. Переменные make.

Присвоение: `имя = строка` (можно с пробелами).

Получение значения переменной: `$(имя)`. Значение - текстовая строка, может содержать ссылки на другие переменные.

```
Пример:
obj_list = main.o Editor.o
# присвоение;

$(obj_list)
# получение значения

1)dir_list = . .. src/include
all:
        echo $(dir_list)
2)optimize_flags  = -03
compile_flags = $(optimize_flags) -pipe
all:
        echo $(compile_flags)
Результат: -03 -pipe
3)program_name = iEdit
obj_list = main.o Editor.o TextLine.o
$(program_name) : $(obj_list)
        gcc $(obj_list) -o $(program_name)
```

Примечание. Значение переменной вычисляется в момент использования.

Часто используемые переменные:

1. `CC` - указать компилятор по умолчанию.
2. `CFLAGS` - параметры компиляции
3. `LDFLAGS` - параметры линковки объектных файлов

Автоматические переменные

- `$^` - список зависимостей, разделённых пробелами
- `$@` - имя цели (файла). Если у нас несколько целей (см 9.2.4), эта переменная принимает значение той цели, для которой выполняется шаблон в конкетный запуск
- `$<` - имя первой зависимости

```
Пример:
$(program_name):$(obj_name)
        gcc $^ -o $@
```

9.2.4. Шаблонные правила

Шаблонные правила (implicit или pattern rules) применяются к группе файлов.

Синтаксис:

```
.<расширение_файлов_завис.> .<расширение_файлов_целей>:
        <команда 1>
        <команда 2>
        ?
        <команда n>
```

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

9.3. Системы управления версиями. CVS

Очень часто над программой работает больше одного человека. Выходят различные версии программ. И существует потребность как то упорядочивать внесение изменений и дополнений. Для этого служат системы управления версиями. Из используемых сейчас можно назвать CVS, Subversion, RCS, Monotone, Arch, GIT и SourceSafe.

`CVS` - *Conhurent Versions Systems* (система управления конкурирующими версиями).

9.3.1. Репозиторий

Репозиторий CVS (или хранилище) хранит полную копию всех файлов и каталогов под управлением CVS, включая все сделанные

Обычно Вы никогда не получаете доступ к файлам в CVS напрямую. Используются команды CVS для получения копии в "рабочий каталог"

и далее работа идёт над копией. После внесения изменений - юзер вносит изменения в репозиторий. После этого, в хранилище сохраняется информация о сделанных изменениях, времени внесения изменений и другая подобная информация.

Для указания, какой из репозиторев используется - применяется переменная окружения CVSROOT, либо явно указывается с помощью ключа -d.

Примеры:

```
$ CVSROOT=/var/cvs; export CVSROOT
$ cvs checkout module/project
или
$ cvs -d /var/cvs module/project
```

Кроме локальных репозиториев - очень часто используются удалённые (сетевые). Для них необходимо указать адрес и (иногда) - способ доступа.

```
$ cvs -d server1:/root checkout sdir1
```

Подробнее об этом - в **??** или в её русском переводе на http://opennet.ru

9.3.2. Получение рабочей копии исходников

9.3.3. Сохранение результатов и версионирование

9.3.4. Коллективная работа над проектом

9.4. Библиотека Си (libc)

libc содержит 2 части: *системные вызовы* и *библиотечные функции*.

Системные вызовы определены, как функции языка Си (независимы от фактической реализации в ядре). В UNIX каждый системный вызов имеет соответствующую функцию (или функции) с тем же именем, хранящуюся в стандартной библиотеке Си. Функции из библиотеки выполняют преобразования аргументов и вызов соответствующего кода ядра. Таким образом, библиотечный код - только оболочка, фактические инструкции находятся в ядре.

Функции общего значения - также часть библиотеки, но не являются системными вызовами. Функции общего назначения и системные вызовы - основа среды программирования UNIX.

В отличие от других библиотек, libc линкуется с каждым приложением, написанном на Си.

Информация о системных вызовах и функциях содержится в 2 и 3 разделах `man` соотвественно. В различных системах различный набор системных вызовов, поэтому некоторые функции могут быть реализованы как библиотечные в одной системе и как системные вызовы в другой.

libc полностью включает в себя библиотеки, определенные в ANSI C (stdio, math, assert). Как следствие: один из основных методов сделать программу переносимой - это написать ее на ANSI C. Такая программа будет компилироваться и работать на всех unix-системах.

Подробнее интерфейсы libc будут рассмотрены в главах, посвященных архитектуре и межпроцессному взаимодействию.

Глава 10

Процессы и сигналы.IPC

В UNIX процессы выполняются в собственном адресном пространстве и изолированы друг от друга, таким образом сведены к минимуму возможности влияния процессов друг на друга.

Но существует необходимость взаимодействия процессов. Для этого требуется:

- обеспечить средства взаимодействия
- исключить нежелательное влияние одного процесса на другой

Взаимодействие решает следующие задачи:

1. передача данных
2. совместное использование данных
3. извещения

Решать проблему взаимодействия средствами самих процессов в рамках многозадачной системы опасно.

10.1. Типы IPC

- сигналы
- каналы
- FIFO (именованные каналы)
- сообщения (очереди сообщений)
- семафоры
- разделяемая память
- сокеты

10.2. Сигналы

Простейший вид IPC. Позволяют уведомлять процесс или группу процессов о наступлении некоторого события.

Группа процессов - любой процесс принадлежит определенной группе процессов. У каждой группы есть свой уникальный идентификатор. Лидер группы - процесс, PID которого совпадает с ID группы. Обычно группа наследуется процессом от родителя.

Процесс может покинуть группу и создать свою.

Управляющий терминал - процесс может быть связан с терминалом, который называется управляющим. Все процессы группы имеют один и тот же управляющий терминал.

Специальный файл устройства `/dev/tty` связан с управляющим терминалом процесса. Драйвер для этого псевдоустройства перенаправляет запросы на фактический терминальный драйвер, который может быть различным для разных процессов.

Сигнал - механизм вызова определенной процедуры при наступлении некоторого события.

Есть две фазы в использовании сигналов: генерация (отправление), доставка и обработка. В промежутке - ожидание доставки.

Примечание: сигналы не могут накапливаться, то есть в любой конкретный момент времени дожидаться обработки могут только разнотипные сигналы.

Причины оправления сигналов:

- особые ситуации (например, деление на 0)
- терминальные прерывания (нажатия клавиш Del, Ctrl+Z, Ctrl+C, отключение терминала)
- другие процессы
- управление заданиями (для командных интерпретаторов)
- квоты (превышение процессом квот)
- уведомления (процесс запрашивает информацию о готовности устройства)
- алармы.

Над сигналами можно выполнить три действия:

1. изменение реакции на сигнал (обработка)
2. блокирование сигнала - откладывание обработки на время выполнения критических участков кода

3. посылка сигнала.

Примечание: сигналы не могут непосредственно переносить информацию.

Каждому сигналу присвоено мнемоническое имя (например, SIGINT), которое указывает, для чего обычно используется сигнал этого типа. Имена сигналов определены в `<signal.h>`.

10.3. Посылка сигналов

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

int kill (pid_t pid, int sig);
```

`Пример: kill (7421, SIGTERM).`

Процесс может посылать сигналы самому себе.

Получение своего PID (7.2.1) - `pid=getpid();`

Получение своего PPID (7.2.2) - `ppid=getppid();`

Ограничения: EUID (7.2.5) или RID (7.2.5) процесса, посылающего сигнал, должны совпадать с EUID и RID процесса-адресата. Для администратора таких ограничений нет.

При неудачном вызове kill возвращает -1 и переменной errno присваивается значения: EPERM (нельзя послать чужому процессу), ESRCH (такого процесса нет) , EINVAL (sig содержит неверный номер сигнала).

10.3.1. Смысл параметра PID

Pid==0 - сигнал посылается всем процессам группы, к которой принадлежит процесс, пославший сигнал;

Pid==-1 и если EUID не администратора, то посылается всем процессам, RID которых равен EUID посылающего процесса, включая и его (если его RID=EUID);

Pid==-1 и EUID администратора, то сигнал посылается всем процессам, кроме некоторых системных;

Pid<0 и не равен -1 - посылается всем процессам, идентификатор группы которых равен по модулю PID, включая пославший процесс, если он также входит в эту группу.

10.3.2. Посылка сигнала самому себе

```
#include<signal.h>

int raise (int sig)
```

вызывающему процессу посылается сигнал. В случае успеха возвращает 0. Например, `raise (SIGKILL)`.

10.4. Таймеры

`setitimer` - установка разных таймеров (3 вида).

```
#include<unistd.h>

unsigned int alarm (unsigned int secs);
```

`secs` - время в секундах, на которое устанавливается таймер. После истечения времени процессу посылается `SIGALRM`.

`Пример: alarm(60).`

Выключение таймера `alarm(0).`

Вызов таймера не накапливается. Вызов следующего отменяет предыдущий.

```
#include<unistd.h>

int pause (void);
```

приостанавливает выполнение процесса до получения любого сигнала, часто используется вместе с `alarm`.

10.4.1. Нормальное и аварийное завершение

Большинство сигналов вызывают нормальное завершение (normal termination). Похоже на вызов процессом функции `exit`.

Некоторые сигналы (SIGABRT, SIGBUS, SIGQUIT, SIGILL и другие) инициируют аварийное завершение со сбросом значения переменных, регистров и т.д. в core (dump).

10.5. Имена сигналов (по алфавиту, выборочно)

Определены в `<signal.h>`

- SIGABRT - прерывание процесса (abort), посылается процессу при вызове системного вызова abort. Core dump.
- SIGALRM - сигнал таймера.
- SIGBUS - аппаратная ошибка на шине. Аварийное завершение.
- SIGCHLD - останов или завершение дочернего процесса. Игнорируется.

- SIGCONT - продолжение работы остановленного процесса (обратный для SIGSTOP).
- SIGHUP - освобождение линии, посылается процессам, подключенным к терминальной линии, при отключении терминала или при завершении работы лидера сеанса членам группы.
- SIGILL - недопустимая команда процессора. Core dump.
- SIGINT - нормальное прерывание программы (Ctrl+C). Посылается всем процессам сеанса.
- SIGKILL - немедленное уничтожение процесса. Не перехватывается.
- SIGPIPE - попытка записи в канал или сокет, для которого принимающий процесс уже завершил работу.
- SIGPROF - сигнал профилирующего таймера.
- SIGQUIT - завершение программы. Похож на SIGINT, но завершение аварийное.
- SIGSEGV - некорректный адрес памяти. Аварийный сброс.
- SIGSTOP - сигнал останова. Управление заданиями. Нельзя перехватить.
- SIGSYS - ошибочный системный вызов.
- SIGTERM - программный сигнал завершения. Используется для корректного завершения процесса.
- SIGTSTP - терминальный сигнал остановки (Ctrl+Z). Похож на SIGSTOP, но можно перехватить.
- SIGTTIN - попытка ввода с терминала фоновым процессом. Остановка процесса.
- SIGTTOU - попытка вывода на терминал фоновым процессом. Остановка процесса.
- SIGURG - поступление в буфер сокета срочных внеочередных данных.
- SIGUSR1, SIGUSR2 - зарезервированы для пользовательских задач. По умолчанию - ничего не происходит.
- SIGVTALRM - виртуальный таймер.
- SIGXCPU - превышение лимита процессорного времени. Core dump.
- SIGXFSZ - превышение лимита на размер файла.

10.6. Наборы сигналов

Набор сигналов - это список сигналов, которые необходимо передать системному вызову.

Тип `sigset_t` в `<signal.h>`, его размер позволяет поместиться всем сигналам, определенным в системе.

Выбор сигналов - либо из полного, удаляя ненужные, либо из пустого набора, добавляя необходимые.

10.6.1. Инициализация набора

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
```

10.6.2. Добавление и удаление сигналов

```
int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);
```

10.6.3. Типовой сценарий работы с набором

```
sigset_t mask1, mask2;

sigemptyset(&mask1);

sigaddset(&mask1, SIGINT);
sigaddset(&mask1, SIGQUIT);

sigfillset(&mask2);
sigdelset(&mask2, SIGCHLD);
```

10.7. Обработчик сигналов

После определения списка можно задать обработку сигналов:

```
#include<signal.h>

int sigaction(int signo, const struct sigaction *act, struct sigaction
*oact);
```

`signo` - сигнал, для которого задается действие.
`act` - определяем обработчик.

oact - если не NULL, то в эту структуру сохранится старый обработчик.

10.8. Разбор структуры sigaction

```
struct sigaction {
void (*sa_handler)(int); //функция обработчика
sigset_t sa_mask; //сигналы, блокируемые во время обработки данного
int sa_flags; //флаги, влияющие на поведение сигнала
viod (*sa_sigaction)(int, siginfo_t*, void*);
};
```

Толкование:
sa_handler:

1. SIG_DFL - константа обработки по умолчанию.
2. SIG_IGN - константа игнорирования. Не может применятся для SIGSTOP и SIGKILL.
3. Адрес функции, принимающей аргумент типа int (sa_handler=f1). Она будет вызываться при получении сигнала, а signo передается как параметр.

Управление передается функции из любого места программы, а после возврата из нее выполнение будет продолжено с точки, в которой было прервано.

- sa_mask - сигналы из этого набора будут игнорироваться во время выполнения функции-обработчика (sa_handler).
- sa_flags - изменение характера реакции:

1. SA_RESETHAND - после возврата из обработчика вернуть обработчик по умолчанию SIG_DFL.
2. SA_SIGINFO - обработчику передается дополнительная информация и вместо sa_handler используется sa_sigaction.
3. SA_RESTART - повтор прерванного сигналом системного вызова.

10.9. Ненадежные сигналы (устаревшая версия)

```
#include<signal.h>

void (*signal(int sig, void(*disp)(int))) (int);
```

`sig` - номер сигнала.
`disp` - `SIG_DFL`, `SIG_IGN` или функция-обработчик.
`Пример: signal(SIGINT, SIG_IGN);`

10.10. Блокирование сигналов

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

- `how` - указывается, какое действие надо выполнить:
  1. `SIG_MASK` - установить блокирование сигналов по списку;
  2. `SIG_UNBLOCK` - отмена блокирования сигналов по списку;
  3. `SIG_BLOCK` - добавление списка к текущим блокируемым.
- `set` - набор сигналов.

`Пример:`

```
sigset_t set1;

sigfillset(&set1);

sigprocmask(SIG_SETMASK, &set1, NULL);

//критический участок - непрерываемый

sigprocmask(SIG_UNBLOCK, &set, NULL);
```

Примечание: можно варьировать наборы и степень защищенности кода.

Глава 11

IPC. Сокеты. Как писать сетевое ПО.

Раздел написан по материалам цикла статей "Сетевое программирование в Unix" ("Сетевые Решения", 2003(12), 2004(1), 2004(2), 2004(3), см http://nestor.minsk.by/sr/abc/114.html).

11.1. Начальные сведения

Сокеты (sockets) впервые появились в начале 80-х в составе 4.2BSD как программный интерфейс к стеку TCP/IP. С тех пор их часто называют "BSD сокеты". И до и после были попытки создавать сетевые API, но сокеты прижились лучше всего. Возможно потому что при их создании учитывались характерные особенности Unix как среды программирования: файла, как основы взаимодействия процессов и принципа "keep it simple stupid" (KISS).

Кстати, в русскоязычной литературе и сленге имеется множество вариантов перевода и произношения термина "socket": сОкет, сокЕт, гнездо (!!!) и т.п. словотворчество. Весь этот зоопарк обозначает одно и то же.

В основе сокетов лежит модная концепция "клиент-сервер". Сервер реализует некоторый сервис, клиент его использует. Два процесса, используя сеть, обмениваются данными. При создании соединения, с каждой из сторон создается сокет. Протоколы TCP/IP отвечают за передачу между ними данных. Чтобы определить кто и куда передает информацию, бы ли введены два понятия: *адрес и порт*.

Адрес - сетевой адрес компьютера, *порт* - это номер сокета на конкретном компьютере. Причем каждый порт связан с конкретным процессом и поэтому операционная система знает кому отдавать пришедшие данные. *Сокет - это пара адрес:порт*.

В итоге - передача данных использует 2 сокета, по одному на каждого участника обмена информацией. Между ними устанавливается вир-

туальный канал и нет больше об этом заботы в чешуйчатой зеленой голове программиста. Что вписано в сокет на одной стороне, то появится на другой, и наоборот.

Для иллюстрации: поглядеть 'netstat -natp' (в linux)

Работа с сокетом идет через файловый дескриптор, что очень характерно для Unix. Однако есть 2 вида передачи информации: с предварительным соединением и без. Мы подробно прожуем и проглотим оба вида передачи. Так же очень основательно будет рассмотрен серверный и клиентский режимы работы.

11.1.1. Всё есть файл

Работа с сокетом ведется через дескриптор, являющийся также и дескриптором файла. То есть уже открытое соединение ничем не отличается с точки зрения программиста от открытого файла или терминала. Разработчик работает с сокетом теми же методами, которыми он работает с обычным (regular) файлом. Несколько отклоняясь от темы, замечу, что то же самое мы наблюдаем в отношении и других специальных типов файлов (pipe, fifo, block и character devices).

Это подымает термин «файл» на совершенно новый уровень понимания: "мы работаем с этим как с файлом и плевать, что там в реализации". Файл может быть физическим устройством, сокетом, каналом, каталогом, текстовым файлом, но мы только открываем их по разному. Чтение/запись и управление режимами работы - через одни и те же системные вызовы.

Потрясающий по своей красоте и выразительности пример: запись компакт диска по сети в 1 строку.

```
$cat 4.9-i386-disc1.iso|ssh user@host.localdomain cdrecord -
```

Здесь используется сразу многие типы файлов, объединенных командой shell в единое целое. `cat` читает файл с диска и выводит на экран. Оболочка перенаправляет вывод `cat` через `pipe` на вход команды `ssh`.

`ssh` запускает на удаленном хосте команду записи дисков. Причем таким образом чтобы cdrecord кушал данные со стандартного ввода.

`ssh` доставляет ему этот ввод через сокет. Еще один неявный момент - `cdrecord` использует файл блочного устройства резака для записи полученного образа компакта. Подсчитаем: regular file, pipe, socket, block device - итого 4 различных вида файлов.

Нашему примеру можно придать изящный финальный штрих - избавится от временного файла в лице образа компакта и создавать его на лету:

```
$mkisofs /home/user2/for_write|ssh user@host.localdomain cdrecord -
```

Нет временных файлов, только безостановочное движение потока данных из одного состояния в другое, подкрепленное могучей концепцией "бывают разные файлы, но мы работаем с ними одинаково, просто пишем и читаем".

11.1.2. Передача с предварительным соединением и без

Есть две модели передачи информации посредством сокетов. С установлением соединения и без оного.

Мы в основном будем останавливаться на первой.

1. Наиболее важные отличия connection-oriented соединения (назовем его условно TCP):

   - требует установления соединения перед передачей;
   - гарантирует доставку данных (создание виртуального канала);
   - передача идет потоком данных. TCP/IP гарантирует то что данные дойдут так, как посылались;
   - основан на протоколе TCP.

2. connectionless (условно назовем его UDP):

   - не требует соединения;
   - ничего не гарантирует;
   - передача идет пакетами, которые могут пропадать, дублироваться и т.п. Контроль за целостностью информации целиком в руках программиста;
   - основан на протоколе UDP.

Итого:

TCP имеет бОльшие накладные расходы чем UDP. Но UDP проще организован, и если нужна высокая скорость и компьютеры стоят в одном сегменте сети, то UDP очень хорошо подойдет. В свою очередь, TCP больше подходит для работы по ненадежным каналам.

Прибегнув к скользкой аналогии,
UDP - выстрел вслепую, пули могут пропасть, а могут и размножится по дороге.
TCP - телефон: что сказано с обеих сторон, то и услышано. Конечно если ворона не села на провод и не перекусила его.

11.2. Основные системные вызовы

Аннотированный список системных вызовов, тем или иным образом качающихся сокетов (в списке есть как socket-specific, так и общие вызовы, связанные с файлами)

- accept(2) - принять соединение (используется сервером);
- bind(2) - связать сокет с конкретным адресом (обычно используется только сервером);
- connect(2) - соединиться с удаленным сервером (клиентский);
- close(2) - закрыть файл или сокет;
- listen(2) - начать прослушивание сокета (серверный);
- read(2) - чтение данных из файла;
- recv(2) - чтение из сокета;
- select(2) - проверка изменения статуса открытых файлов;
- send(2) - послать данные через сокет;
- shutdown(2) - закрыть соединение;
- socket(2) - создать сокет;
- write(2) - запись данных в файл.

Примечание: в Solaris, унаследовавшей реализацию сокетов от System V Release 4, сокеты не входят в ядро, а реализованы библиотекой. Поэтому man по ним содержится в 3-м разделе.

11.3. Простейший TCP-клиент

Как ни странно - чаще всего в его роли выступает небезызвестный `telnet`. Утилита `telnet` есть во всех мне известных реализациях стека tcp/ip. Набираем `telnet <host> <port>` и наслаждаемся прямым общением с сервером. Благо классические TCP-based протоколы - текстовые.

[tcpclient.cpp]

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <arpa/inet.h>
5 #include <string.h>
6 #include <unistd.h>
```

```
7 #include <stdio.h>
8 int main() {
9 struct sockaddr_in addr;
10 memset(&addr,0,sizeof(addr));
11 addr.sin_family=AF_INET;
12 addr.sin_port=htons(1212);
13 addr.sin_addr.s_addr=inet_addr("127.0.0.1");
14 int sock;
15 sock=socket(PF_INET,SOCK_STREAM,0);
16 connect(sock,(sockaddr *)&addr,sizeof(addr));
17 char buf[80+1];
18 memset(buf,0,81);
19 read(sock,buf,80);
20 puts(buf);
21 shutdown(sock,0);
22 close(sock);
23 return 0;
24 }
```

Эта программка соединяется с localhost, порт 1212, считывает последовательность байт и выводит ее на экран.

А теперь самое вкусное - обгладывание костей, любезно предоставленных нашим информационным спонсором. Строки 1-4: заголовочные файлы. Содержат объявления функций и типов данных для работы через сетевые сокеты. 5 и 7 строки - библиотечные функции Си. 6 строка - заголовочный файл стандартных системных вызовов.

9, 10, 11-13 строки - объявление, зачистка и заполнение структуры, содержащей адрес соединения. Обращаю особое внимание на то, как инициализируется порт и адрес назначения. Адрес и порт хранятся в особом формате, называемом *network byte order*. Прямое присваивание обычно ничего не даст, потому что архитектура i386 и network byte order не совпадают. Макрос `htons(3)` занимается преобразованием числа в сетевой вид. Функция `inet_addr(3)` преобразует строковое значение IP-адреса в числовое. `AF_INET` из 11 строки - указание типа сокетов (сетевые в нашем случае).

И наконец, ключевой момент. 15 строка - создание сокета (TCP-сокета естественно) и 16 строка - соединение с сервером. В `connect(2)` используется указатель на адрес , потому что размер структуры с адресом может варьироваться в зависимости от типа сокета (сетевой или локальный).

Далее идет считывание (19 строка) и вывод на экран (20 строка) полученного.

21 и 22 строки - закрыть соединение и освободить файловый дескриптор, им использованный.

11.4. TCP-сервер

В качестве примера использована реализация сервиса daytime. Суть протокола daytime: выдать время и закрыть соединение со стороны сервера.

[daytime.cpp]

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <arpa/inet.h>
5 #include <string.h>
6 #include <time.h>
7 #include <unistd.h>
8 #include <stdio.h>
9 char * daytime() {
10 time_t now;
11 now=time(NULL);
12 return ctime(&now);
13 }
14 int main() {
15 struct sockaddr_in addr;
16 memset(&addr,0,sizeof(addr));
17 addr.sin_family=AF_INET;
18 addr.sin_port=htons(1212);
19 addr.sin_addr.s_addr=INADDR_ANY;//inet_addr("127.0.0.1");
20 int sock, c_sock;
21 sock=socket(PF_INET,SOCK_STREAM,0);
22 bind(sock,(struct sockaddr *)&addr,sizeof(addr));
23 listen(sock,5);
24 for (;;) {
25 c_sock=accept(sock,NULL,NULL);
26 char buf[81];
27 memset(buf,0,81);
28 strncpy(buf,daytime(),80);
29 write(c_sock,buf,strlen(buf));
30 shutdown(c_sock,0);
31 close(c_sock);
32 puts("answer tcp");
33 }
34 return 0;
35 }
```

Пойдем по порядку. Сервер отличается от клиента режимом работы. Он должен:

1. занять адрес и порт (22 строка), `bind(2)`;
2. включить прослушивание порта (23 строка), `listen(2)`;
3. принимать и обрабатывать соединения (24-33 строки), `accept(2)`.

Подчеркнем отличия от клиента: в 19 строке указывается специальная константа `INADDR_ANY`. Она обозначает, что соединения будут приниматься на всех интерфейсах. Можно задать и конкретный адрес (в стиле TCP-клиента). Так работают многие сервисы, которым можно задать адрес привязки в конфиге (apache, squid).

С 24 по 33 строку у нас бесконечный цикл. До первого `Ctrl+C`, естественно. Сервер до полной и окончательной победы занимается приемом соединений.

Очень интересная особенность: *каждое входящее соединение порождает свой сокет* (см. 25 строку). Контрольный сокет (`sock`) и клиентский сокет (`c_sock`) - разделены. Обмен данными идет через `c_sock`. Соединения принимаются на контрольный (`sock`).

Есть два важных следствия разделения на контрольный сокет и сокеты соединений:

- упрощение работы программиста - нет необходимости отслеживать, кто и куда прислал данные, разные по сути понятия - выделены;
- возможность распараллеливания обработки соединений (подробнее об этом расскажу позже).

В примере используется простая последовательная обработка. Пока предыдущее соединение не обработано - входящие соединения ждут своей очереди. Это имеет смысл только для простых случаев - когда время обработки запроса мало и задержка для вновь поступающих невелика.

9-13 строки - получение текущего времени. Эта функция используется в строке 28 для формирования результата.

11.5. UDP

Вот мы и подошли к финальной части букваря, к протоколу UDP и методам работы с ним. Начнем традиционно, с клиента, и закончим, как обычно, сервером. Напоминаю, что UDP - это модель передачи данных без образования соединения и без проверки корректности передачи. Посему если информация пропала - никто не виноват. Но если канал передачи надежный - почему бы и не использовать UDP? Данные передаются так называемыми датаграммами (datagram) без сохранения порядка при переносе к получателю.

11.5.1. UDP-клиент

К сожалению стандартных и широко распространенных программок типа telnet не обнаружено. Поэтому в этой роли выступает накарябаный на колене за вечер корявый шедевр. Встречайте:

```
[udp_client.cpp]
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <arpa/inet.h>
5 #include <string.h>
6 #include <unistd.h>
7 #include <stdio.h>

8 int main() {
9 struct sockaddr_in server, client={AF_INET,INADDR_ANY,INADDR_ANY};

10 memset(&server,0,sizeof(server));

11 server.sin_family=AF_INET;
12 server.sin_port=htons(1212);
13 server.sin_addr.s_addr=inet_addr("127.0.0.1");

14 int sock;

15 sock=socket(PF_INET,SOCK_DGRAM,0);
16 bind(sock,(sockaddr *)&client,sizeof(client));

17 char buf[81];
18 memset(buf,0,81);
19 strcpy(buf,"request");
20 sendto(sock,&buf,strlen(buf),0,(sockaddr *)&server,sizeof(server));
21 memset(buf,0,81);
22 recvfrom(sock,buf,80,0,NULL,0);
23 puts(buf);

24 return 0;
25 }
```

Эта мини-программа посылает слово ”request” на порт 1212 по адресу 127.0.0.1 (localhost) и читает, что же ответил сервер. с 1 по 14 строки - стандартная сокетная обвязка, общая для TCP и UDP. У нас клиент, поэтому в 12 и 13 задаем порт и адрес назначения.

В 15 обратите особое внимание на константу `SOCK_DGRAM` - мы задаем тип сокета как датаграмный.

16 - уже специфична. Мы явно выполняем привязку адреса и порта к созданному сокету. 9 строка явно указывает, что программисту фиолетово, какой порт занять и через какой из сетевых интерфейсов отсылать данные. Ядро поймет это указание правильно и выдаст порт случайным образом.

20 и 22 строки - следующие специфичные для UDP элементы. Системные вызовы `sendto(2)` и `recvfrom(2)` предназначены для отправки и получения сообщений в/из сокета. Их можно использовать даже если сокет находится в несоединенном состоянии. Кстати, несмотря на природу `SOCK_DGRAM`, здесь можно использовать `connect(2)` и заменить `sendto(2)` на `write(2)` и `recvfrom(2)` на `read(2)`. Будет сделана виртуальная привязка сокета к пункту назначения, но соединение не будет устанавливаться, т.к. сокет остается по прежнему `SOCK_DGRAM`.

11.5.2. UDP-сервер

Опять же daytime сервер в DGRAM-реинкарнации. Ждет датаграмму, при приходе оной извлекает адрес отправителя и посылает в ответ текущее время сервера.

[udp_server.cpp]

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <arpa/inet.h>
5 #include <string.h>
6 #include <time.h>
7 #include <unistd.h>
9 #include <stdio.h>

10 char * daytime() {
11 time_t now;
12 now=time(NULL);
13 return ctime(&now);
14 }

15 int main() {
16 struct sockaddr_in addr;

17 memset(&addr,0,sizeof(addr));
```

```
18 addr.sin_family=AF_INET;
19 addr.sin_port=htons(1212);
20 addr.sin_addr.s_addr=INADDR_ANY;

21 int sock, c_sock;

22 sock=socket(PF_INET,SOCK_DGRAM,0);
23 bind(sock,(struct sockaddr *)&addr,sizeof(addr));
24 for (;;) {
25 struct sockaddr from;
26 unsigned int len=sizeof(from);
27 char buf[81];
28 memset(buf,0,81);
29 recvfrom(sock,&buf,80,0,&from,&len);
30 printf("udp incoming:%s",buf);

31 memset(buf,0,81);
32 strncpy(buf,daytime(),80);

33 sendto(sock,buf,strlen(buf),0,&from,len);
34 puts("answer udp");
35 }

36 return 0;
37 }
```

Все меньше и меньше остается незнакомых элементов в программах. Будь я графоманом и/или любителем гонораров - расписывал бы каждую строку :).

В этом примере мы заострим внимание всего на двух строках - 29 и 33. При вызове `recvfrom(2)` системным вызовом заполняются параметры 5 и 6. Адрес отправителя и размер структуры для его хранения. Так сервер узнает о существовании клиента и о наличии у него желания пообщаться. И в 33 строке идет подача данных в ответ.

11.5.3. Объедки анализа

Наблюдательные дети наверное заметили, что клиент и сервер у UDP практически идентичны по набору используемых вызовов. Отличие всего в двух маленьких детальках, а дьявол всегда прячется в деталях.

Первая деталька: присваивание адреса и порта сокету: сервер явно указывает порт, клиент - саботирует. Ведь чтобы обратиться к серверу - за ним должен быть зафиксирован порт.

Вторая деталька: порядок вызовов `recvfrom(2)` и `sendto(2)`. Клиент сначала отсылает, потом принимает, сервер, наоборот, принимает а

затем отсылает. Клиент-сервер as is: клиент начинает, сервер реагирует на действия.

11.6. Модели организации серверов

В жизни программиста, пишущего для интерфейса сокетов, через некоторое время неотвратимо наступает переломный момент. Написание приложения, обрабатывающего несколько подключений одновременно, или работающего под серьезной нагрузкой. Или хитрого клиента, который одновременно общается с несколькими серверами.

Существует N-ое количество способов организации нетривиальных серверов и клиентов. Наиболее распространенных из них я и коснусь. Материал будет обильно нашпигован информацией из W. Richard Stevens "Unix. Network Programming. Networking API" и ru.unix.prog FAQ. Поэтому не буду отсылать на конкретные источники, а просто последовательно излагать.

11.6.1. "естественный" и "правильный" способы организации сервера

Чем же руководствоваться при выборе подходящего способа организации сервера? Ответ - здравым смыслом, конечно. Непременно сначала подумать, а не бросаться создавать отстойные приложения.

Первый критерий: подумайте, как и сколько времени обрабатывается 1 соединение.

Если у приложения простой протокол запрос-ответ с минимальной обработкой и задержкой на сервере, простой последовательный сервер будет в самый раз. Клиенты просто не заметят разницы :)

11.6.2. Последовательный сервер

Последовательный - это когда все запросы обрабатываются друг за другом последовательно (см сервис daytime):

```
24 for (;;) {
25 c_sock=accept(sock,NULL,NULL);
26 char buf[81];
27 memset(buf,0,81);
28 strncpy(buf,daytime(),80);
29 write(c_sock,buf,strlen(buf));
30 shutdown(c_sock,0);
31 close(c_sock);
32 puts("answer tcp");
33 }
```

Пока текущий не обработан, следующий запрос на соединение стоит в очереди. Размер очереди указывается в `listen(2)`. Очень просто и, как ни странно, дьявольски эффективно. Никаких дополнительных накладных расходов на управление соединениями. Но случись задержка в обработке сервером запроса - и остальные клиенты нервно курят в стороне. Длительные операции - увы - не для него.

Как же маштабируемость? Ответ - нуль. Понапихали 4 процессора в машину? Ничего не улучшится :). Все одно, последовательно пилит на одном, ОС бессильна что-либо изменить. Итого: эффективен, но для простых вещей.

11.6.3. один процесс == один клиент (простой и prefork)

Мысль начинающего сетевого программиста общается сама с собой по очевидной схеме:
- Ага: надо много клиентов одновременно обрабатывать...
- Давайте каждому соединению создадим новый процесс!
- Да, это круто, пельмень!

Так вылазит первая, самая очевидная схема (”естественная”): сервер использует несколько процессов, каждый из которых обслуживает по одному клиенту.

Преимущества этой модели очевидны:

1. простая как ванильная сушка;
2. хорошо маштабируется с ростом числа процессов;
3. ошибка в 1 процессе не ведет к отказу всей программы.

Но тут есть и отрицательные стороны. Процесс - это достаточно тяжелый объект ОС. При большом количестве клиентов мы получаем значительную загрузку системы в целом вплоть до отказа. Переключения контекста и связанная с ними черновая работа ОС вполне могут подсадить на задницу процессор любой мощности (при действительно большом количестве клиентов).

[fork_server.cpp]

```
1 for (;;) {
2 c_sock=accept(sock,NULL,NULL);
3 if ( !fork() ) {
4 puts("incoming tcp\n");
5 char time_str[81];
6 memset(time_str,0,81);
7 strncpy(time_str,daytime(),80);
8 write(c_sock,time_str,strlen(time_str));
```

```
9 puts("answer tcp\n");
10 shutdown(c_sock,0);
11 close(c_sock);
12 exit(0);
13 }
14 }
```

Как же это работает? После прихода соединения (строка 2), запускается новый процесс через `fork(2)` (строка 3). Головной поток выполнения снова вернулся к фазе `accept(2)`, новый же процесс обрабатывает соединение и завершает работу. Почему `fork(2)` в `if`? Выкурите `man 2 fork` до полного просветления.

Кстати, приведенный пример - не единственный возможный способ реализации модели «1 процесс == 1 клиент». Существует также улучшенный (и более сложный вариант) организации под названием *prefork*. Суть его следующая: после фазы `listen(2)` мы запускаем N процессов. В каждом из них - последовательный сервер (см пункт 4.1.1). Вся толпа из N серверов занимается обслуживанием одного и того же принимающего порта. Механизм прост: если одновременно несколько процессов делают `accept(2)` на одинаковый порт, то ОС погружает всех в спячку до прихода соединения. После прихода соединения - осуществляется т.н. ”побудка” и один из процессов побеждает в соц соревновании за сокет. Остальные снова уходят в спячку. Управление этим режимом полностью осуществляется операционной системой. Плюсы - все уже запущено, только начать обрабатывать. Минус - при количестве процессов >200 начинаются потери в производительности (проверено на людях). Система начинает терять время на обдумывании, кому из 200 процессов отдать на заклание обработку. Как вариант - можно сделать блокировку `accept(2)` семафором или блокировкой. Тогда только 1 процесс делает `accept(2)` и разруливание ”побудки” не нужно.

11.6.4. один поток == один клиент

Идея проста: меняешь в предыдущем разделе процессы на потоки и защищаешь общие данные. Эффективность подобного решения - величина неизвестная. В разных версиях различных ОС потоки реализованы по разному. Но в общем случае считается, что поток весит меньше чем процесс, и переключение контекста между потоками менее накладно.

```
1 for (;;) {
2 pthread_mutex_lock(&mutex_tcp);
3 c_sock=accept(sock,NULL,NULL);
4 pthread_mutex_unlock(&mutex_tcp);
5 pthread_create(&tid,NULL,&deliver_tcp,&c_sock);
6 }
```

Здесь применена защита `accept(2)` исключающим семафором (как иллюстрация к идее о prefork, описанной в предыдущем разделе, примененная и к потокам). 2 и 4 строки - блокировка и раз-блокировка мутексом. В строке 5 мы запускаем поток на выполнение функции `deliver_tcp` с параметром `c_sock`.

11.6.5. Однопроцессная Finite State Machine и мультиплексирование

«Потоки, как и объектно ориентированное программирование - это такая "серебряная пуля". Человеку, которому лень думать над тем, что собственно надлежит сделать, разбиение программы на потоки или использование объектов кажется естественным. А потом он начинает наступать на не очевидные грабли. Потому что на самом деле это не пуля, а крылатая ракета на жидком водороде. Дальнобойность и убойная сила - поразительные, но обслуживания требует ох какого квалифицированного. И стоит дорого. Поэтому там, где можно обойтись пулей из автомата (в данном случае - конечного) надо обходиться пулей. А крылатой ракетой стрелять только тогда, когда после анализа других тактических вариантов стало ясно, что здесь ничего другое не поможет.» (Виктор Вагнер, из переписки в ru.unix.prog.)

В среде специалистов наиболее уважаемым вариантом является реализация с использованием FSM (по русски - на конечном автомате). Конечный автомат - это старая математическая абстракция, описывающая машину с некоторым количеством состояний и переходов между ними.

Ключевая особенность Unix, делающая возможным работу FSM-механизма - средства опроса состояния сокета (через select, poll, kevent/kqueue или epoll).

Сервер опрашивает состояние контрольного сокета и сокетов пришедших соединений. В случае активности на каком-нибудь из них - производит необходимые действия и возвращается в состояние опроса. Если объяснение признано невнятным, вот вам атомный пример.

```
Однопоточный TCP и UDP echo-server с использованием опроса состояния сокетов че

...
1 struct sockaddr_in addr;
/* здесь инициализация addr */

2 int sock, c_sock, u_sock;
/* здесь включение tcp-сервера на sock и udp-сервера на u_sock */

/* определение наборов дескрипторов файлов и их максимального размера */
```

```
3 fd_set rfds, afds;
4 int nfds=getdtablesize();
5 FD_ZERO(&afds);
6 FD_SET(sock,&afds);
7 FD_SET(u_sock,&afds);

8 for (;;) {
9 memcpy(&rfds, &afds, sizeof(rfds));
10 select(nfds, &rfds, NULL,NULL,NULL);

11 if ( FD_ISSET(sock, &rfds) ) {
12 c_sock=accept(sock,NULL,NULL);
13 puts("incoming tcp");
14 FD_SET(c_sock,&afds);
15 continue;
16 }

17 for (int fd=0; fd<nfds; ++fd)
18 if ( fd == u_sock && FD_ISSET(u_sock,&rfds) ) {
19 struct sockaddr from;
20 unsigned int len=sizeof(from);
21 char buffer[81];
22 memset(buffer,0,81);
23 int size=recvfrom(u_sock,&buffer,80,0,&from,&len);
24 sendto(u_sock,buffer,size,0,&from,len);
25 printf("answer udp:%s",buffer);

26 } else if ( fd != sock && FD_ISSET(fd,&rfds) ) {
27 char buffer[81];
28 memset(buffer,0,81);
29 int len=read(fd,buffer,80);
30 if (len <=0) {
31 puts("connection closed");
32 close(fd);
33 FD_CLR(fd, &afds);
34 continue;
35 }

36 write(fd,buffer,len);
37 printf("answer tcp:%s",buffer);
38 } // if and for
39 } // for
```

Echo-сервер делает весьма прозаическую вещь: все, что к нему при-

ходит, отсылается обратно отправителю. Как в tcp, так и в udp. Данная реализация работает одновременно с множественными TCP-соединениями и с присылаемыми UDP-пакетами в одном единственном процессе.

Насладимся же разбором исходников! :) С строки 3 по 7 появляется новый персонаж нашего повествования: набор дескрипторов файлов (`fd_set`). Его зовут на помощь, когда надо оперировать не 1 сокетом за раз, а целой пачкой. Для начала заполним его контрольным TCP- и UDP-сокетом. Это начальное состояние нашей FSM - `afds`, в котором определены два сокета - `sock` и `u_sock`.

10 строка - вот и опрос состояния сокетов. В начале их у нас два - `sock` и `u_sock`. По мере выполнения итераций цикла их число может меняться. `select(2)` изменяет значения `rfds` - потому в строке 9 мы восстанавливаем его каждый раз из `afds`.

А вот дальше - черная работа. Проверяем, кто из сокетов дернулся:
- sock? Надо принимать соединение и добавлять новый сокет к afds (строки 11-16);
- `u_sock`? Вычитать UDP пакет и вернуть его содержимое отправителю (строки 18-25);
- один из клиентских TCP-сокетов? Прочитать новый кусочек данных и вернуть их назад (строки 26-38).

Теперь немножко о select(2), пожилом, но полным сил мастодонте Unix API.

Параметры 2, 3, 4 - указатели на наборы дескрипторов файлов. Каких файлов - неважно (см. 1 часть цикла "Сетевое программирование в Unix"). Можете одновременно опрашивать stdout, socket и pipe - система с удовольствием схавает и отработает. Соответственно наборы обозначают дескрипторы "доступные для чтения", "доступные для записи", "те, в которых случилась ошибка". Вполне могут отличаться друг от друга, если вам надо сотворить нетривиальную вещь. Часть из них может быть NULL.

Последний, 5-й, параметр - это время, после которого select закончит работать, если ничего не произошло с дескрипторами из наборов. Когда время NULL - ждет вечно, до наступления событий.

И 1-й параметр - это число, равное значению максимального дескриптора из набора +1. Такой параметр выполняет простую функцию - ограничивает количество просматриваемых дескрипторов указанным числом. Чтобы не зацепить лишние, не используемые программой открытые файлы (это замедлит работу). Вобщем select(2) смотрит 0,1,2,3,..n дескрипторы в поисках активности на них. Это то слабое место, в которое обоснованно тыкают пальцем фанаты других способов опроса состояния файла.

Что же можно сказать об этой модели в общем? Наиболее эффективна с точки зрения использования CPU. При наличии долгоиграющих

блокирующих операций может быть затыки с параллельным приемом других соединений. В этом случае медленные операции можно вынести в отдельные потоки/процессы и решить тем самым затык. Или поработать операционной системой - разбить большую операцию на куски и выполнять ее, попутно отвлекаясь на остальную работу. Как уже стало очевидно, модель с FSM требует очень тщательного программирования.

11.6.6. Смешанные модели

Вот мы и изучили все базовые типы серверов. Можно заняться их скрещиванием. Например устойчива к сбоям модель "многопроцессность+FSM". Или производительна (на Linux) "многопоточность+FSM". Снова повторюсь. Нет "самой лучшей модели", есть наиболее подходящая к вашим условиям.

11.7. Unix-domain сокеты или локальные сокеты

Адрес сокетов в домене UNIX - имя файла. При вызове bind файл создаётся в файловой системе.

```
Пример сервера (AF_UNIX)
1. sockfd = socket (AF_UNIX, SOCK_DGRAM, 0);
2. unlink ("./echo.server");
3. bzero (&serv_addr, sizeof(serv_addr));
4. strcpy(serv_addr.sun_path, "./echo.server");
5. serv_addr.sun_family = AF_UNIX;
6. saddrlen = sizeof
   (serv_addr.sun_family)+strlen(serv_addr.sun_path);
7. bind (sockfd, (struct sockaddr *) &serv_addr, saddrlen);

1. sockfd = socket (AF_UNIX, SOCK_DGRAM, 0);
2. bzero (&serv_addr, sizeof(serv_addr));
3. strcpy(serv_addr.sun_path, "./echo.server");
4. serv_addr.sun_family = AF_UNIX;
5. saddrlen = sizeof
   (serv_addr.sun_family)+strlen(serv_addr.sun_path);
6. strcpy(clnt_addr.sun_path, "./tmp/clnt.xxxx");
```

Глава 12

System V IPC

К семейству System V IPC относятся 3 метода взаимодействия:

- очереди сообщений
- разделяемая память
- семафоры

Характеризуются общим способом адресации и схожими системными вызовами для работы с ними. Хранятся в пространстве ядра. После создания - не требуют существования создавшего процесса.

Область видимости структур IPC - вся система и не имеют счётчика ссылок. Не адресуются как имена на файловой системе, соотвественно не могут управлятся как файлы с помощью стандартных вызовов. Управляются только своими syscalls. Используют свои user-space утилиты для просмотра и удаления (ipcs, ipcrm).

Поскольку структуры System V IPC - не файлы, то к ним не может быть применено мультиплексирование (**??**, **??**). Это усложняет использование более чем 1 структуры одновременно или совместно с вводом/выводом из/в файл/устройтво.

12.1. Идентификаторы и ключи

Каждая структура IPC (очереди, семафоры, разделяемая память) адресуется в ядре положительным целочисленным идентификатором.

При создании структуры IPC ключ должен быть указан.

Все три типа адресуются ключём типа `key_t`. `key_t` определён в `<sys/types.h>` как целое (по меньшей мере - 32разрядное). Ключ переводится в идентификатор ядром.

Обычно генерируется функцией `ftok` по пути к файлу и произвольному идентификатору :

```
#include <sys/types.h>
key_t ftok(const char * pathname, int proj_id);
```

Существуют различные пути для обеспечения встречи клиента и сервера в одной IPC структуре:

1. Сервер создаёт новую структуру IPC указывая `IPC_PRIVATE` и сохраняет ключ куда-либо для клиента (например в файл). Также может использоваться при отношениях "предок-потомок", тогда ключ передаётся при fork порождённому процессу.

2. Через общий заголовочный файл, задав ключ как число. Сервер создаёт структуру IPC указывая этот ключ. Клиент получает доступ. Проблема - ключ уже может использоваться кем-то ещё.

3. Клиент и сервер могут согласится на путь к файлу и идентификатор проекта (от 0 до 255) и вызвать `ftok()` для перевода пути и идентификатора в ключ.

12.2. Права доступа

System V IPC связывает структуру `ipc_perm` с каждой структурой IPC. Она определяет *владельца и права доступа* (аналогично доступу к файлу).

```
struct ipc_perm {
uid_t uid; // uid владельца
gid_t gid; // uid группы
uid_t cuid; // uid создателя
gid_t cgid; // gid создателя
mode_t mode; // режимы доступа
ulog seq; // число использований (TODO, неясный смысл)
key_t key; // ключ
};
```

Все поля кроме `seq` задаются при создании структуры IPC. С помощью вызовов `msgctl, semctl`, `shmctl` мы можем поменять `uid`, `gid` и `mode` позднее.

12.3. Очереди сообщений

Очередь сообщений - это связанный список сообщений, хранящийся в ядре и определяемый идентификатором очереди (Queue ID).

Очередь можно создать или открыть уже существующую через `msgget()`.

Таблица 12.1: Права доступа System V IPC объектов

Permission	Message Queue	Semaphore	Shared Memory
user-read	`MSG_R`	`SEM_R`	`SHM_R`
user-write	`MSG_W`	`SEM_A`	`SHM_W`
group-read	`MSG_R>>3`	`SEM_R>>3`	`SHM_R>>3`
group-write	`MSG_W>>3`	`SEM_A>>3`	`SHM_W>>3`
other-read	`MSG_R>>6`	`SEM_R>>6`	`SHM_R>>6`
other-write	`MSG_W>>6`	`SEM_A>>6`	`SHM_W>>6`

Новые сообщения добавляются в конец очереди через `msgsnd()`. Каждое сообщение имеет тип (положительное целое число), длину, и данные этой длины, которые передаются в `msgsnd()` при добавлении в очередь.

Сообщения принимаются из очереди через `msgrcv`. Мы можем принимать сообщения основываясь на их типе.

12.3.1. Создание очереди

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

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

Литература

[1] А. Робачевский. Операционная система UNIX. Второе издание - BHV, 2005.

[2] Э. Немет, Г. Снайдер и др. UNIX: Руководство системного администратора. Для профессионалов. 3-е изд. - BHV, 2002

[3] Эрик Реймонд. Искусство программирования для Unix (The Art of Unix Programming). - Вильямс, 2005.

[4] Грэм Гласс, Кинг Эйблс. Unix для программистов и пользователей. - BHV, 2004.

[5] Б. Моли. Unix/Linux: теория и практика программирования. - Кудиц-образ, 2004.

[6] Б. Керниган, Р. Пайк. UNIX. Программное окружение. Символ, 2003

[7] Дж. Фридл. Регулярные выражения. Библиотека программиста. - Питер, 2001.

[8] Арнольд Роббинс. Linux. Программирование в примерах. - Кудиц-образ, 2005.

[9] К. Петцке. LINUX. От понимания к применению. - ДМК, 2000.

[10] У. Стивенс. UNIX: разработка сетевых приложений. - Питер, 2003.

[11] У. Стивенс. UNIX: взаимодействие процессов. - Питер, 2002.

[12] ASPLinux. Руководство пользователя. - ASPLinux, 2001.

[13] М. Бах. Архитектура операционной системы UNIX. http://www.lib.ru.

[14] А. Соловьев. Sed и awk. Учебное пособие.

[15] Программирование на shell (Unix).

[16] Стен Келли-Бутл. Введение в Unix. - Лори, 1995.

[17] Rassel S. Sage. Приемы профессиональной работы в UNIX. http://www.citforum.ru.

[18] Дж. МакМален. UNIX. - Компьютер, 1996.

[19] А. Шевель. Linux. Обработка текста. Специальный справочник. - Питер, 2001

[20] Free Software Foundation. CVS book. 1993-2004.

[21] Б. Керниган, Р. Пайк. Практика программирования. - Вильямс, 2004

GNU Free Documentation License

Version 1.2, November 2002

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed

under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary

formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the

front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.