Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

А. В. Столяров, И. Г. Головин, И. А. Волкова

Операционная система Unix

методическое пособие для выполнения заданий практикума

Москва
2006

УДК 519.6

Авторы будут признательны за конструктивную критику, в том числе за сообщения об обнаруженных в тексте пособия опечатках. Адрес для связи: avst@cs.msu.ru.

Черновая версия от 28 сентября 2006 г.

Предисловие

На втором курсе факультета ВМиК семинары по практическому программированию традиционно проводятся с ориентацией на операционные системы семейства Unix. Многие студенты с такими операционными системами сталкиваются впервые.

Ограничения по времени зачастую не позволяют преподавателям раскрыть на занятиях специфические особенности Unix-подобных операционных систем. Кроме того, в силу различных причин аспекты работы с Unix, необходимые при выполнении заданий практикума на ЭВМ, оказываются разнесенными по всевозможной литературе, ориентированной на разный уровень читателя; пособий, конспективно и сжато охватывающих все основные вопросы как пользовательского интерфейса Unix, так и системы программирования Gnu (включая компиляторы семейства gcc, отладчик gdb и систему автоматической сборки Gnu Make), практически нет.

Настоящее пособие призвано заполнить образовавшуюся нишу. Пособие написано на основе опыта, приобретенного авторами в процессе ведения семинарских занятий по программированию на втором курсе факультета ВМиК МГУ; в тексте содержатся ответы на наиболее типичные вопросы, возникающие у студентов в связи с использованием Unix-подобных операционных систем при выполнении заданий практикума на ЭВМ.

Вопросы, связанные с языками программирования C и C++, системными вызовами ОС Unix и т.п. в настоящем пособии не рассматриваются; для изучения этих вопросов следует обратиться к соответствующей литературе.

1 Введение

С созданием в конце восьмидесятых годов минувшего столетия персональных компьютеров с процессором i386 стало возможным применение на персональных компьютерах операционных систем с *истинной многозадачностью*, и прежде всего — операционных систем семейства Unix. Последовавшее вскоре появление свободно распространяемых юниксоподобных систем FreeBSD и Linux, способных работать на компьютерах архитектуры i386, стало настоящей революцией представлений о персональных компьютерах.

Существует несколько десятков операционных систем, объединенных общим названием «Unix». Среди них — такие системы, как SunOS/Solaris, HPUX, AIX, Digital Unix (DEC Unix), Minix, FreeBSD, NetBSD, OpenBSD, BSDi, Linux и другие. Любители точных формулировок говорят, что словом «Unix» нельзя назвать ни одну из операционных систем, вместо этого следует говорить о *семействе* операционных систем, называемых *Unix'оподобными* (Unix-like).

Под термином «ОС Unix» мы будем понимать систему, входящую в это семейство.

Операционные системы семейства Unix объединены достаточно серьезными традициями, многие из которых заложены еще создателями первого Unix'а в начале 1970-х годов. Эти традиции выдержали проверку временем, подобную которой в мире компьютерных технологий выдерживает очень небольшое количество инноваций.

Первое, что следует отметить при переходе на Unix — это очень развитые средства командной строки. Именно путем подачи команд совершается большинство действий в Unix'е. Иногда это создает ошибочное впечатление отсутствия для Unix'а средств графического пользовательского интерфейса; на самом деле, разумеется, такие средства существуют и к настоящему моменту развиты не хуже, а во многом и лучше, чем в коммерческих операционных системах ряда Windows, MacOS и т.п. Использование командной строки обусловлено исключительно тем, что так *действительно* удобнее.

Второе, что бросается в глаза – это многообразие графических оболочек. Дело в том, что графический интерфейс пользователя (GUI) здесь не является частью операционной системы; поэтому пользователь может выбрать тот внешний вид и функциональность оконной системы, которые ему удобнее.

Третье отличие не столь заметно на первый взгляд; оно состоит в том, что подавляющее большинство программ в мире Unix распространяется в исходных текстах и часто компилируется уже на машине конечного пользователя, что делает, например, компилятор языка C обязательной частью почти любого дистрибутива Unix.

Немаловажно, что бóльшая часть программного обеспечения, доступного в мире Unix, распространяется свободно, то есть условия лицензии допускают свободное копирование и модификацию программ.

Программисты, привыкшие к интегрированным средам разработки, могут с удивлением обнаружить, что при разработке программ для ОС Unix монолитные интегрированные среды обычно не применяются. Каждый программист использует наиболее удобный ему редактор текстов для написания кода программы; многие редакторы поддерживают средства интеграции с компиляторами командной строки, отладчиком и системой автоматической сборки.

В целом процесс разработки программ в среде Unix отличается высоким комфортом для программиста, что отчасти объясняет существование большого количества свободно распространяемого программного обеспечения. Авторы пособия выражают надежду, что читателю удастся преодолеть неудобства, вызываемые на первых порах непривычностью операционной среды, и оценить несомненные достоинства ОС Unix как с пользовательской, так и с программистской точек зрения.

2 Пользовательские средства ОС Unix

В этой главе рассматриваются основные возможности пользовательского интерфейса ОС Unix: взаимодействие с командным интерпретатором Bourne Shell и редактирование файлов с помощью редакторов vim, joe и встроенного редактора оболочки Midnight Commander.

2.1 Первый сеанс

При выполнении заданий практикума вам, возможно, придется столкнуться с ОС FreeBSD или ОС Linux, причем в зависимости от условий конкретного компьютерного класса нужная вам операционная система может функционировать либо непосредственно на той машине, с которой вы работаете, либо на общем сервере, к которому вам потребуется удаленный доступ.

С точки зрения пользователя различия между этими вариантами невелики. Краткую инструкцию о том, как войти в систему, вы получите от преподавателя или от системного администратора компьютерного класса вместе с вашим входным именем (login) и паролем (password).

Итак, введите входное имя и пароль. Если вы сделали ошибку, система выдаст сообщение `Login incorrect`, которое может означать как опечатку во входном имени, так и неправильный пароль. Учтите, что регистр букв важен в обоих случаях, так что причиной неприятия системой пароля может быть, например, случайно нажатая клавиша CapsLock.

Для работы с системой вам необходимо получить приглашение командной строки. При использовании удаленного терминального доступа (например, с помощью программы `putty`) командная строка – это единственное средство работы с системой, которое вам доступно. Приглашение появится сразу после того, как вы введете верные имя и пароль.

Если вы работаете в терминальном Unix-классе и вход в систему выполняете с помощью текстовой консоли, после ввода верных имени и пароля вы также немедленно получите приглашение командной строки, однако в этом случае у вас есть возможность запустить один из возможных графических оконных интерфейсов. Это удобнее хотя бы тем, что можно открыть несколько окон одновременно. Для запуска графической оболочки X Window необходимо дать команду `startx`[1]; после этого нужно запустить один или несколько экземпляров программы `xterm`,

[1] В некоторых системах может потребоваться другая команда; за информацией обращайтесь к вашему системному администратору

которая представляет собой графическое окно, в котором запускается интерпретатор команд. Подробнее работа с графическими оболочками рассматривается в главе 3.

Возможно также, что вход в систему выполняется сразу с помощью графического интерфейса (этот вариант бывает возможен как с локальной машиной, так и с удаленным доступом). В этом случае также необходимо получить доступ к интерпретатору командной строки, запустив программу `xterm`.

Первым вашим действием в системе должна стать смена пароля. В зависимости от конфигурации системы это может потребовать команды `passwd` или `yppasswd`. Введите эту команду (без параметров). Система спросит у вас сначала старый пароль, затем (дважды) новый. Учтите, что при вводе пароля на экране ничего не отображается.

Учтите, что придуманный вами пароль должен содержать ровно восемь символов, причем в нем должны присутствовать латинские буквы верхнего и нижнего регистров, цифры и знаки препинания. Пароль не должен основываться на английском слове или на вашем входном имени. Вместе с тем, следует придумать такой пароль, который вы легко запомните. Проще всего взять какую-либо запоминающуюся фразу, содержащую знаки препинания и числительные, и построить пароль на ее основе (числительные передаются цифрами, от остальных слов берутся первые буквы, причем буквы, соответствующие существительным, берутся заглавными, остальные – строчными). Например, из пословицы «Один с сошкой, семеро с ложкой» можно «сделать» пароль `1sS,7sL.`.

2.2 Дерево каталогов. Работа с файлами

Система каталогов в ОС Unix существенно отличается от привычной пользователям MSDOS и WinXX, и наиболее заметные на первый взгляд отличия – это отсутствие букв, обозначающих устройства (что-то вроде `A:`, `C:` и т.п.), а также то обстоятельство, что имена каталогов разделяются в ОС Unix не обратной, а прямой косой чертой (`/`).

После входа в систему вы окажетесь в вашем *домашнем каталоге*. Домашний каталог – это место для хранения ваших личных файлов. Чтобы узнать имя (*путь*) текущего каталога, введите команду `pwd`:

```
$ pwd
/home/stud/s2003324
```

Узнать, какие файлы находятся в текущем каталоге, можно с помощью команды `ls`:

```
$ ls
Desktop    tmp
```

Имена файлов в ОС Unix могут содержать любое количество точек в любых позициях, т.е. например, имя `a.b..c...d....e` является вполне допустимым именем файла. При этом действует соглашение, что имена, начинающиеся с точки, соответствуют «невидимым» файлам. Чтобы увидеть все файлы, включая невидимые, можно воспользоваться командой `ls -a`:

```
$ ls -a
.  ..  .bash_history  Desktop   tmp
```

Некоторые из показанных имен могут соответствовать подкаталогам текущего каталога, другие могут иметь специальные значения. Чтобы было проще различать файлы по типам, можно воспользоваться флажком `-F`:

```
$ ls -aF
./  ../  .bash_history  Desktop/   tmp/
```

Теперь мы видим, что все имена, кроме `.bash_history`, соответствуют каталогам. Заметим, что `.` – это ссылка на сам текущий каталог, а `..` – ссылка на каталог, содержащий текущий каталог (в нашем примере это `/home/stud`).

Перейти в другой каталог можно командой `cd`:

```
$ pwd
/home/stud/s2003324
$ cd tmp
$ pwd
/home/stud/s2003324/tmp
$ cd ..
$ pwd
/home/stud/s2003324
$ cd /usr/include
$ pwd
/usr/include
$ cd /
$ pwd
/
$ cd
$ pwd
/home/stud/s2003324
```

cp	Копирование файла
mv	Переименование или перемещение файла
rm	Удаление файла
mkdir	Создание директории
rmdir	Удаление директории
touch	Создание файла или установка нового времени модификации
less	Просмотр содержимого файла с пейджингом

Таблица 1: Команды для работы с файлами

Последний пример показывает, что команда `cd` без указания каталога делает текущим домашний каталог пользователя, как это было сразу после входа в систему.

Основные команды работы с файлами перечислены в таблице 1.

Большинство команд принимает дополнительные ключи, начинающиеся со знака '-'. Так, команда `rm -r the_dir` позволяет удалить директорию `the_dir` вместе со всем её содержимым.

2.3 Редакторы текстов

Различных редакторов текстов в операционных системах семейства Unix существует несколько сотен. Ниже приводятся основные сведения о трёх из них.

Выбирая для работы редактор текстов, следует обратить внимание на то, подходит ли он для написания программ. Для этого редактор текстов должен, во-первых, работать с файлами в обычном текстовом формате; во-вторых, редактор не должен выполнять автоматического форматирования абзацев текста (т.е., например, MSWord для этой цели непригоден); и, в-третьих, редактор обязан использовать моноширинный шрифт, т.е. шрифт, в котором все символы имеют одинаковую ширину. Выяснить, удовлетворяет ли редактор этому свойству, проще всего, набрав в этом редакторе строку из десяти латинских букв `m` и под ней - строку из десяти латинских букв `i`. В редакторе, использующем моноширинный шрифт, полученный текст будет выглядеть так:

```
mmmmmmmmmm
iiiiiiiiii
```

тогда как в редакторе, использующем пропорциональный шрифт (и непригодном, вследствие этого, для программирования), вид будет

`^`	перейти в начало строки
`$`	перейти в конец строки
`x`	удалить символ под курсором
`dw`	удалить слово (от курсора до пробела или конца строки)
`dd`	удалить текущую строку
`d$`	удалить символы от курсора до конца строки
`J`	слить следующую строку с текущей (удалить перевод строки)
`i`	начать ввод текста с позиции перед текущим символом (insert)
`a`	то же, но после текущего символа (append)
`o`	вставить пустую строку после текущей и начать ввод текста
`O`	то же, но строка вставляется перед текущей
`.`	повторить последнюю операцию
`u`	отменить последнюю операцию (undo)
`U`	отменить все изменения, внесенные в текущую строку

Таблица 2: Команды редактора vim

примерно таков:

mmmmmmmmmm
iiiiiiiii

2.3.1 Редактор vim

Редактор vim (Vi Improved) является клоном классического редактора текстов для Unix-подобных операционных систем VI. Работа в редакторах этого семейства может показаться для начинающего пользователя несколько неудобной, т.к. по построению интерфейса редактор VI коренным образом отличается от привычных большинству пользователей экранных редакторов текстов с системами меню.

В то же время многие программисты, работающие под Unix-системами, предпочитают использовать именно эти редакторы, т.к. для человека, умеющего использовать основные функции этих редакторов, именно этот вариант интерфейса оказывается наиболее удобным для работы над текстом программы.

В любом случае, если освоение редактора vim покажется вам чрез-

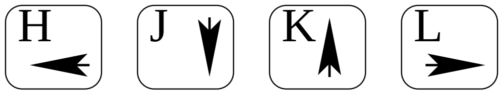

Рис. 1: Перемещение курсора в vim с помощью алфавитных клавиш

мерно сложной задачей, к вашим услугам другие редакторы текстов, два из которых описаны ниже. Для читателей, решивших обойтись без изучения vim, приведем для справки последовательность нажатия клавиш для выхода из этого редактора: если вы случайно запустили vim, практически в любой ситуации вы можете нажать `Escape`, затем набрать `:q!`, и это приведёт к выходу из редактора без сохранения изменений.

Чтобы запустить редактор vim, достаточно дать команду `vim myfile.c`. Если файла `myfile.c` не существует, он будет создан при первом сохранении изменений.

Первое, что необходимо уяснить, работая с vim – это **наличие у него двух режимов работы: режима ввода текста и режима команд**. Сразу после запуска вы оказываетесь в режиме команд. В этом режиме любые нажатия клавиш будут восприняты как команды редактору, т.е., если вы попытаетесь ввести текст, результаты могут оказаться совершенно не похожи на ожидавшиеся.

Перемещение по тексту в режиме команд возможно с помощью стрелочных клавиш, однако более опытные пользователи vim предпочитают пользоваться для этой цели символами j, k, h и l для перемещения, соответственно, вниз, вверх, влево и вправо (см. рис. 1).

Причина такого выбора в том, что в ОС UNIX стрелочные клавиши генерируют последовательность байт, начинающуюся с кода `Esc` *(0x1b); любая такая последовательность может быть воспринята редактором как требование на переход в командный режим и несколько команд-символов, причем единственный способ отличить Esc-последовательность, порожденную нажатием клавиши, от такой же последовательности, введенной пользователем – это измерение времени между приходом кода* `Esc` *и следующего за ним. При работе на медленной линии связи (например, при удаленном редактировании файла в условиях медленной или неустойчивой работы сети) этот способ может давать неприятные сбои.*

Несколько наиболее часто употребляемых команд приведены в таблице 2. Команды `i`, `a`, `o`, и `O` переводят вас в режим ввода текста. Теперь всё вводимое с клавиатуры воспринимается как текст, подлежащий вставке. Естественно, возможно использование клавиши Backspace

`:w`	сохранить редактируемый файл
`:w <name>`	записать файл под новым именем
`:w!`	сохранить, игнорируя (по возможности) флаг readonly
`:wq`	сохранить файл и выйти
`:q`	выйти их редактора (если файл не был изменен с момента последнего сохранения)
`:q!`	выйти без сохранения, сбросив сделанные изменения
`:r <name>`	прочитать содержимое файла <name> и вставить его в редактируемый текст
`:e <name>`	начать редактирование еще одного файла
`:ls`	показать список редактируемых файлов (активных буферов)
`:b <N>`	перейти к буферу номер N

Таблица 3: Файловые команды редактора vim

в её обычной роли. В большинстве случаев возможно также использование стрелочных клавиш, но в некоторых версиях vim, при некоторых особенностях настройки, а также при работе по медленной линии возможна неправильная реакция редактора на стрелки. В этом случае для навигации по тексту необходимо выйти из режима ввода.

Выход из режима ввода и возврат в режим команд осуществляется нажатием клавиши Escape.

При необходимости найти в тексте то или иное ключевое слово следует использовать (в командном режиме) последовательность `/<word>`, завершая её нажатием Enter. Так, `/myfun` установит курсор на ближайшее вхождение строки `myfun` в вашем тексте. Повторить поиск можно, введя символ `/` и сразу же нажав Enter.

Переместиться на строку с заданным номером (например, на строку, для которой компилятор выдал сообщение об ошибке) можно, набрав двоеточие, номер строки и нажав Enter.

Также через двоеточие доступны команды сохранения, загрузки файлов, выхода и т.п. (см. таблицу 3).

При одновременной работе с несколькими файлами переход между двумя последними редактируемыми файлами кроме команды `:b` также можно осуществить комбинацией клавиш `Ctrl-^`.

Отдельного упоминания заслуживают команды выделения блоков и работы с блоками. Начать выделение фрагмента, состоящего исключительно из целых строк, можно командой `V`; выделить фрагмент, состоящий из произвольного количества символов, можно с помощью команды `v`. Граница выделения устанавливается стрелками или соответствующими командами h,j,k и l.

Удалить выделенный блок можно командой d, скопировать - командой y. В обоих случаях выделение снимается, а фрагмент текста, находившийся под выделением, помещается в специальный буфер. Содержимое буфера можно вставить в текст командами `p` (после курсора) и `P` (перед курсором).

Текст может попасть в буфер и без выделения. Так, все команды, удаляющие те или иные фрагменты текста (`x`, `dd`, `dw`, `d$` и др.) помещают удалённый текст в буфер. Команды `yy`, `yw`, `y$` помещают в буфер, соответственно, текущую строку, текущее слово и символы от курсора до конца строки.

Возможности редактора vim возрастут, если вы создадите тэг-файл для ваших исходных текстов. Это делается программой ctags, например:

```
$ ctags *.c *.h
```

В результате в текущей директории появится файл tags, содержащий информацию о расположении в вашей программе деклараций и описаний. Наиболее очевидное использование этой информации - автоматический поиск в вашей программе описания функции или переменной, имя которой находится в настоящий момент под курсором. Для использования этой возможности нажмите комбинацию `Ctrl-]`.

Если программа, над которой вы работаете, компилируется и компонуется с помощью системы `make` (см.§4.3), вы можете также использовать способность vim к восприятию сообщений об ошибках и предупреждениях, выдаваемых компилятором gcc. Для этого следует запустить утилиту `make` командой vim. Например, если ваша программа собирается командой `make prog`, в командном режиме vim следует набрать `:make prog`. Если при компиляции возникнут предупреждения или ошибки, редактор автоматически найдет для вас то место в ваших исходных файлах, которое “не понравилось” компилятору. Чтобы снова увидеть полностью сообщение компилятора, относящееся к данному месту исходного файла, используйте команду `:cc`, а для перехода к следующей ошибке или предупреждению – команду `:cn`.

2.3.2 Редактор joe

Другой популярный в среде Unix редактор текстов называется JOE (от слов Jonathan’s Own Editor). Чтобы запустить редактор joe, доста-

`Ctrl-K D`	сохранить файл
`Ctrl-K X`	сохранить и выйти
`Ctrl-C`	выйти без сохранения
`Ctrl-Y`	удалить текущую строку
`Ctrl-K B`	отметить начало блока
`Ctrl-K K`	отметить конец блока
`Ctrl-K C`	скопировать выделенный блок в новое место
`Ctrl-K M`	переместить выделенный блок в новое место
`Ctrl-K Y`	удалить выделенный фрагмент
`Ctrl-K L`	найти строку по номеру
`Ctrl-Shift-'-'`	отменить последнее действие (undo)
`Ctrl-^`	снова выполнить отмененное действие (redo)
`Ctrl-K F`	поиск ключевого слова
`Ctrl-L`	повторный поиск

Таблица 4: Наиболее употребительные команды редактора joe

точно дать команду `joe myfile.c`. Если файла `myfile.c` не существует, он будет создан при первом сохранении изменений.

В отличие от редактора vim, интерфейс редактора joe покажется более похожим на привычные для большинства пользователей экранные редакторы текстов. Стрелочные клавиши, Enter, Backspace и т.п. работают в своей обычной роли, в большинстве случаев также доступна клавиша Delete. Команды редактору даются с помощью комбинаций клавиш, большинство из которых начинается с `Ctrl-K`. В частности, `Ctrl-K h` покажет в верхней части экрана памятку по наиболее употребительным командам редактора (см. таблицу 4).

2.3.3 Встроенный редактор оболочки Midnight Commander

Оболочка (файловый монитор) Midnight Commander представляет собой клон некогда популярного файлового менеджера под MSDOS, известного как Norton Commander. Запуск оболочки производится командой `mc`. Вызов встроенного редактора текстов для редактирования выбранного файла производится клавишей `F4`; если вы хотите создать новый файл, используйте комбинацию `Shift-F4`.

Интерфейс этого редактора достаточно понятен на интуитивном уровне, поэтому подробное описание мы опускаем. Ограничимся одной рекомендацией. Если не предпринять специальных мер, редактор будет вставлять в текст символ табуляции вместо групп из восьми пробелов, что может оказаться неудобным при использовании других редакторов. Единственный способ отключить такой стиль заполнения – установить опцию «Fill tabs with spaces». Чтобы добраться до диалога с настройками, нажмите F9, выберите пункт меню «Options», в нём – пункт «General».

Чтобы настройки не потерялись при выходе из Midnight Commander, сохраните их. Для этого, выйдя из редактора, нажмите F9, выберите пункт меню «Options», а в нём - пункт «Save Setup».

2.4 Права доступа к файлам

С каждым файлом в ОС Unix связано 12-битное слово, называемое «правами доступа» к файлу[2].

Младшие 9 бит этого слова объединены в три группы по три бита; каждая группа задаёт права доступа для владельца файла, для его группы и для всех остальных пользователей. Три бита в каждой группе отвечают за право чтения файла, право записи в файл и право исполнения файла.

Чтобы узнать права доступа к тому или иному файлу, можно воспользоваться командой `ls -l`, например:

```
$ ls -l /bin/cat
-rwxr-xr-x  1 root  root   14232  Feb 4 2003 /bin/cat
```

Расположенная в начале строки группа символов `-rwxr-xr-x` показывает тип файла (первый символ; минус означает, что мы имеем дело с обыкновенным файлом, буква `d` означала бы каталог и т.п.) и права доступа, соответственно, для владельца (в данном случае `rwx`, т.е. чтение, запись и исполнение), группы и всех остальных (в данном случае `r-x`, т.е. права на запись отсутствуют). Таким образом, файл `/bin/cat` доступен любому пользователю на чтение и исполнение, но модифицировать его может только пользователь `root` (т.е. администратор).

Поскольку группа из трёх бит соответствует ровно одной цифре восьмеричной системы счисления, общепринятой является практика записи слова прав доступа к файлу в виде восьмеричного числа, обычно трёхзначного. При этом младший разряд (последняя цифра) соответствует правам для всех пользователей, средняя - правам для группы и старшая (обычно она идёт самой первой) цифра обозначает права для владельца. Права на чтение соответствуют 1, права на запись - 2, права на чтение - 4; соответствующие значения суммируются, т.е., например,

[2] В английском оригинале - permissions

права на чтение и запись обозначаются цифрой 6 $(4 + 2)$, а права на чтение и исполнение - цифрой 5 $(4 + 1)$.

Таким образом, права доступа к файлу `/bin/cat` из нашего примера можно закодировать восьмеричным числом 0755[3].

Для каталогов интерпретация битов прав доступа несколько отличается. Права на чтение каталога дают возможность просмотреть его содержимое. Права на запись позволяют модифицировать каталог, т.е. создавать и уничтожать в нём файлы (причем удалить можно и чужой файл, а также такой, на который прав доступа нет, т.к. достаточно иметь права доступа на запись в сам каталог). Что касается бита прав «на исполнение», для каталога этот бит означает возможность какимлибо образом использовать содержимое каталога, в том числе, например, открывать файлы, находящиеся в каталоге. Таким образом, если на каталог установлены права чтения, но нет прав исполнения, мы можем его просмотреть, но воспользоваться увиденным нам не удастся. Напротив, если есть права исполнения, но нет прав чтения, мы можем открыть файл из этого каталога только в том случае, если точно знаем имя файла. Узнать имя мы никак не можем, т.к. возможности просмотреть каталог у нас нет.

Оставшиеся три (старших) разряда слова прав доступа называются `SetUid Bit` *(04000),* `SetGid Bit` *(02000) и* `Sticky Bit` *(01000).*

Если для исполняемого файла установить `SetUid Bit`*, этот файл будет при исполнении иметь права своего владельца (чаще всего - пользователя root) вне зависимости от того, кто из пользователей соответствующий файл запустил.* `SetGid Bit` *работает похожим образом, устанавливая эффективную группу пользователя (в отличие от эффективного идентификатора пользователя). Примером suid-программы является* `passwd`.

`Sticky Bit`*, установленный на исполняемом файле, в некоторых версиях ОС Unix обозначает, что сегмент кода программы следует оставить в памяти даже после того, как программа будет завершена; это позволяет экономить время на загрузке в память программ, исполняемых чаще других.*

Для каталогов `SetGid Bit` *означает, что, какой бы пользователь ни создал в этом каталоге файл, в качестве «группы владельца» для этого файла будет установлена та же группа, что и у самого каталога.* `Sticky Bit` *означает, что, даже если пользователь имеет право на запись в данный каталог, удалить он сможет только свои (принадлежащие ему) файлы.*

Для изменения прав доступа к файлам используется команда `chmod`[4]. Эта команда позволяет задать новые права доступа в виде вось-

---

[3]Обратите внимание, что число записано с нулём впереди; согласно правилам языка C это означает, что число записано в восьмеричной системе

[4]сокращение слов Change Mode

меричного числа, например:

```
chmod 644 myfile.c
```

устанавливает для файла `myfile.c` права записи только для владельца, а права чтения - для всех.

Права доступа также можно задать в виде мнемонической строки вида `[ugoa][+-=][rwxsXtugo]` Буквы u, g, o и a в начале означают, соответственно, владельца (user), группу (group), всех остальных (others) и всех сразу (all). `+` означает добавление новых прав, `-` – снятие старых прав, `=` – установку указанных прав и снятие всех остальных. После знака буквы r, w, x означают, как можно догадаться, права на чтение, запись и исполнение, буква s – установку/снятие `Set`-битов (имеет смысл для владельца и группы), t обозначает `Sticky Bit`. Буква `X` (заглавная) означает установку/снятие бита исполнения только для каталогов, а также для тех файлов, на которые хотя бы у кого-нибудь есть права исполнения.

Если команду `chmod` использовать с ключом `-R`, она проведёт смену прав доступа ко всем файлам во всех поддиректориях заданной директории.

Например, команда `chmod a+x myscript` сделает файл `myscript` исполняемым; команда `chmod go-rwx *` снимет со всех файлов в текущем каталоге все права, кроме прав владельца. Очень полезной может оказаться команда

```
chmod -R u+rwX,go=rX ~
```

на случай, если вы случайно испортите права доступа в своей домашней директории; эта команда, скорее всего, приведёт всё в удовлетворительное состояние.

2.5 Перенаправления ввода-вывода в интерпретаторе Bourne Shell

Как известно, практически все программы в ОС Unix следуют соглашению, по которому поток ввода-вывода с дескриптором 0 объявляется потоком стандартного ввода, поток с дескриптором 1 — потоком стандартного вывода и поток с дескриптором 2 — потоком для вывода сообщений об ошибках.

Осуществляя обмен данными через стандартные потоки, большинство программ не делает предположений о том, с чем на самом деле связан тот или иной поток. Это позволяет использовать одни и те же

<table>
<tr><td>cmd1 > file1</td><td>запустить программу cmd1, направив её вывод в файл file1. Если файл существует, он будет перезаписан с нуля, если не существует – будет создан.</td></tr>
<tr><td>cmd1 >> file1</td><td>запустить программу cmd1, дописав её вывод в конец файла file1. Если файла не существует, он будет создан.</td></tr>
<tr><td>cmd2 < file2</td><td>запустить программу cmd2, подав ей содержимое файла file2 в качестве стандартного ввода. Если файла не существует, произойдёт ошибка.</td></tr>
<tr><td>cmd3 > file1 < file2</td><td>запустить программу cmd3, перенаправив как ввод, так и вывод.</td></tr>
<tr><td>cmd1 | cmd2</td><td>запустить одновременно программы cmd1 и cmd2, подав данные со стандартного вывода первой на стандартный ввод второй.</td></tr>
<tr><td>cmd4 2> errfile</td><td>направить поток сообщений об ошибках в файл errfile.</td></tr>
<tr><td>cmd5 2>&1 | cmd6</td><td>объединить потоки стандартного вывода и вывода ошибок программы cmd5 и направить всё на стандартный ввод программе cmd6</td></tr>
</table>

Таблица 5: Примеры перенаправлений ввода-вывода

программы как для работы с терминалом, так и для чтения из файла и/или записи в файл.

Командные интерпретаторы, в том числе классический Bourne Shell, предоставляют возможности для управления вводом-выводом запускаемых программ. Для этого используются символы `<`, `>`, `>>`, `>&` и `|` (см. таблицу 5).

Обычно в ОС Unix присутствует программа `less`, позволяющая постранично просматривать содержимое файлов, пользуясь клавишами "Стрелка вверх", "Стрелка вниз", PgUp, PgDn и др. для прокрутки. Эта же программа позволяет постранично просматривать текст, поданный ей на стандартный ввод. Использование программы `less` полезно в случае, если информация, выдаваемая какой-либо из запускаемых вами программ, не умещается на экран. Например, команда

```
ls -lR | less
```

позволит вам просмотреть список всех файлов, находящихся в текущей директории и всех её поддиректориях.

Учтите, что компилятор gcc выдаёт все сообщения об ошибках и предупреждения в стандартный поток ошибок. Поэтому, чтобы просмотреть постранично сообщения, выдаваемые в ходе компиляции, следует дать, например, такую команду:

```
gcc -Wall -g myprog.c -o myprog 2>&1 | less
```

2.6 Процессы

Список процессов, выполняющихся в настоящий момент, можно получить командой `ps`:

```
$ ps
  PID TTY          TIME CMD
 2199 pts/5    00:00:00 bash
 2241 pts/5    00:00:00 ps
```

Как видно, команда по умолчанию выдаёт только список процессов, запущенных в данном конкретном сеансе работы.

К сожалению, опции команды `ps` очень сильно отличаются в зависимости от версии (в частности, для FreeBSD и Linux). За подробной информацией следует обращаться к документации по данной конкретной ОС; здесь мы ограничимся замечанием, что команда `ps ax` выдаст список всех существующих процессов, а команда `ps axu` дополнительно выдаст информацию о владельцах процессов[5].

В некоторых случаях может оказаться полезной программа `top`, работающая интерактивно. Она выдаёт на экран список наиболее активных процессов, обновляя его один раз в секунду. Чтобы выйти из программы `top`, необходимо ввести букву `q`.

Снять процесс можно с помощью сигнала. Обычно применяют сначала сигнал `SIGTERM` (15), а если после этого процесс не завершился – сигнал `SIGKILL` (9). Сигнал `SIGTERM` может быть перехвачен программой, например, с целью корректного завершения; сигнал `SIGKILL` уничтожает процесс безусловным образом, не оставляя возможности произведения каких-либо действий по подготовке к корректному завершению.

[5]Это верно для ОС Linux и FreeBSD. В других ОС, например в SunOS/Solaris, опции команды ps имеют совершенно иной смысл

Для передачи сигнала процессу используется команда kill. По умолчанию передаётся сигнал SIGTERM, т.е., например, команда kill 2763 приведёт к тому, что процесс 2763 получит сигнал SIGTERM. Задать другой сигнал можно либо по номеру, либо по названию (TERM, KILL и т.п.). Следующие две команды эквивалентны; обе передают процессу 2763 сигнал SIGKILL:

```
kill -9 2763
kill -KILL 2763
```

2.7 Выполнение в фоновом режиме

Некоторые программы выполняются ощутимое время, при этом не требуя взаимодействия с пользователем через стандартные потоки ввода/вывода. Во время выполнения таких программ удобно иметь возможность продолжать давать команды командному интерпретатору, чтобы не тратить время.

Допустим, нам потребовался список всех файлов в файловой системе. Такой список можно получить с помощью команды ls -lR /. Естественно было бы перенаправить её вывод в файл, чтобы позднее иметь возможность его анализа. Заметим, что такая команда будет выполняться несколько минут и ждать её окончания нам бы не хотелось, поскольку эти несколько минут мы могли бы, например, использовать для набора текста в редакторе. Чтобы запустить команду в фоновом режиме, к ней следует в конце приписать символ &, например:

```
$ ls -lR / >list.txt 2>/dev/null &
[1] 2437
```

Здесь мы перенаправили поток вывода сообщений об ошибках в устройство /dev/null[6], чтобы сообщения о невозможности чтения некоторых каталогов не мешали нашей дальнейшей работе.

В ответ на нашу команду система сообщает, что задание запущено в фоновом режиме в качестве фоновой задачи №1, причем номер запущенного процесса – 2437. Текущий список выполняемых фоновых задач можно узнать командой jobs:

```
$ jobs
[1]+  Running   ls -lR / >list.txt 2>/dev/null &
```

[6]Устройство /dev/null предназначено для поглощения потоков данных, которые нам не нужны; в него можно записывать что угодно, данные при этом попросту исчезают

По окончании выполнения задачи командный интерпретатор нам об этом сообщит. В случае успешного завершения сообщение будет выглядеть так:

```
[1]+  Done      ls -lR / >list.txt 2>/dev/null &
```

В случае, если программа завершилась с кодом, отличным от нуля, сообщение будет иметь другой вид:

```
[1]+  Exit 1      ls -lR / >list.txt 2>/dev/null &
```

Наконец, если фоновый процесс снять сигналом, сообщение будет примерно таким (для сигнала SIGTERM):

```
[1]+  Terminated  ls -lR / >list.txt 2>/dev/null &
```

При отправлении сигналов процессам, являющимся фоновыми задачами данного конкретного экземпляра командного интерпретатора, можно ссылаться на номера процессов по номерам фоновых задач, добавляя к номеру символ `%`. Так, команда `kill %2` отправит сигнал `SIGTERM` второй фоновой задаче. Символ `%` без указания номера обозначает последнюю из фоновых задач.

Если задача уже запущена не в фоновом режиме и нам не хочется ждать её завершения, мы можем сделать обычную задачу фоновой. Для этого следует нажать `Ctrl-Z`, в результате чего выполнение текущей задачи будет приостановлено. Затем с помощью команды `bg`[7] приостановленную задачу можно снова поставить на выполнение, но уже в фоновом режиме.

Также возможно сделать текущей (т.е. такой, окончания которой ожидает командный интерпретатор) любую из фоновых и приостановленных задач. Это делается с помощью команды `fg`[8].

2.8 Электронная документация

Дистрибутивы ОС Unix обычно содержат большое количество документации, доступ к которой можно получить непосредственно в процессе работы. Большая часть такой документации оформлена в виде файлов, отображаемых с помощью традиционной команды `man`, либо с помощью более современной программы `info`.

[7] От английского background – фон
[8] От английского foreground

2.8.1 Команда `man`

Команда `man` служит для поиска и отображения установленной в системе справочной информации. Справочник охватывает команды ОС Unix, системные вызовы, библиотечные функции языка С (и других установленных в системе языков), форматы файлов, некоторые общие понятия и т.д.

К примеру, если вы хотите узнать все опции команды `ls`, следует дать команду

```
$ man ls
```

а если вы, допустим, забыли, в каком порядке идут аргументы системного вызова `waitpid()`, вам поможет

```
$ man waitpid
```

Программа `man` найдет соответствующий документ в системном справочнике и запустит программу его отображения. Появившийся на экране документ можно листать с помощью клавиш «стрелка вверх» и «стрелка вниз», можно использовать клавишу «пробел», чтобы пропустить сразу страницу текста. Выход из просмотра справочного документа осуществляется клавишей `q` (от слова quit).

Если нужный вам справочный документ имеет большой объем, а вам необходимо найти в нем определенное место, может оказаться удобным поиск подстроки. Это делается вводом символа `/`, после которого следует набрать строку для поиска и нажать Enter. Повторный поиск той же строки осуществляется вводом `/` и нажатием Enter (то есть саму строку можно опустить). Чтобы выполнить поиск в обратном направлении, можно воспользоваться символом `?` вместо `/`.

В некоторых случаях системный справочник может содержать более одного документа с данным именем. Так, в системе существует команда `write` и системный вызов `write`. Команда `write` вам вряд ли понадобится, так что, если вы набрали `man write`, скорее всего, вы имели в виду системный вызов; к сожалению, система этого не знает и выдаст вам совсем не тот документ, который вам нужен.

Эту проблему можно решить указанием *номера секции* системного справочника. Так, в нашем примере команда

```
$ man 2 write
```

выдаст именно документ, посвященный системному вызову `write`, поскольку секция №2 содержит справочные документы по системным вызовам. Перечислим другие секции системного справочника:

- 1 – пользовательские команды ОС Unix (такие команды, как `ls`, `rm`, `mv` и т.п. описываются в этой секции)
- 2 – системные вызовы ядра ОС Unix
- 3 – библиотечные функции языка С (к этой секции можно обратиться, например, за информацией о функции `sprintf`)
- 4 – описания файлов устройств
- 5 – описания форматов системных конфигурационных файлов
- 6 – игровые программы
- 7 – общие понятия (например, `man 7 ip` выдаст полезную информацию о программированием с использованием TCP/IP)
- 8 – команды системного администрирования ОС Unix (например, в этой секции вы найдете описание команды `mount`, предназначенной для монтирования файловых систем).

Система может содержать и другие секции, причем не обязательно обозначающиеся цифрой; так, при установке в системе интерпретатора языка Tcl его справочные страницы обычно оформляются в отдельную секцию, которая может называться “n”, “3n” и т.п.

2.8.2 Команда `info`

Если команда `man` служит для отображения отдельных документов, то команда `info` позволяет осуществлять просмотр целых наборов страниц, связанных между собой гиперссылками; иначе говоря, документы в формате `info` представляют собой гипертекст, а сама команда – браузер (просмотрщик) гипертекста, работающий в текстовом режиме.

Система `info` часто содержит более подробную информацию о том же предмете, чем `man`. Так, `man make` выдаст довольно скудную информацию об опциях командной строки команды `make`, в то время как `info make` позволит основательно изучить работу с программой `make` (см. §4.3), научиться составлять ее входные файлы и т.д.

При работе с программой `info` можно пролистывать текст клавишами стрелок и пробелом. Клавиши `n` и `p` позволяют перейти, соответственно, к следующей и предыдущей странице гипертекста. Гиперссылки в тексте заключены между знаком `*` и `::`, например:

```
* Reading Makefiles::    How makefiles are parsed
```

Если позиционировать курсор на словах `Reading Makefiles` и нажать Enter, произойдет переход по данной гиперссылке. Если необходимо перейти назад, можно воспользоваться клавишей `l` (от слова last).

Выход из программы `info` производится по клавише `q` (от слова quit) или нажатием `Ctrl-C`.

2.9 Дополнительные возможности

2.9.1 Командные файлы в Bourne Shell

Интерпретатор Bourne Shell позволяет осуществлять не только работу в режиме диалога с пользователем, но и выполнение программ, называемых командными файлами (скриптами). Файл с программой, предназначенной для исполнения интерпретатором Bourne Shell, должен начинаться со строки

```
#!/bin/sh
```

Переменные в языке Bourne Shell имеют имена, состоящие из латинских букв, цифр, знака подчеркивания и начинающиеся всегда с буквы. Значением переменной может быть любая строка символов. Чтобы присвоить переменной значение, необходимо написать оператор присваивания, например:

```
I=10
MYFILE=/tmp/the_file_name
MYSTRING="Here are several words"
```

Обратите внимание, что в имени переменной, а также вокруг знака равенства (символа присваивания) не должно быть пробелов, в противном случае команда будет расценена не как присваивание, а как обычная команда, в которой знак присваивания – один из параметров.

Для обращения к переменной используется знак `$`, например:

```
echo $I $MYFILE $MYSTRING
```

В результате выполнения этой команды будет напечатано:

```
10 /tmp/the_file_name Here are several words
```

При необходимости скомпоновать слитный текст из значений переменных можно имена переменных заключать в фигурные скобки, например:

```
echo ${I}abc
```

Эта команда напечатает:

```
10abc
```

Для выполнения арифметических действий используется конструкция `$(( ))`. Например, команда

```
I=$(( $I + 7 ))
```

увеличит значение переменной I на семь.

С помощью встроенной в интерпретатор Bourne Shell команды `test` можно осуществлять проверку выполнения различных условий. Если заданное условие выполнено, команда завершится с нулевым (успешным) кодом возврата, в противном случае – с единичным (неуспешным). Синонимом команды test является символ открывающей квадратной скобки, причем сама команда в этом случае воспрнимает символ закрывающей квадратной скобки в качестве своего параметра (как знак окончания выражения), что позволяет наглядно записывать проверяемое выражение, заключая его в квадратные скобки. Приведём несколько примеров.

```
[ -f "file.txt" ]
            # существует ли файл с именем file.txt
[ "$I" -lt 25 ]
            # значение переменной I меньше 25
[ "$A" = "abc" ]
            # значение переменной A является строкой abc
[ "$A" != "abc" ]
            # значение переменной A не является строкой abc
```

Это можно, например, использовать в операторе ветвления:

```
if [ -f "file.txt" ]; then
   cat "file.txt"
else
   echo "Файл file.txt не найден"
fi
```

Заметим, что то же самое можно было написать и иначе, без использования квадратных скобок, но это менее наглядно:

```
if test -f "file.txt" ; then
   cat "file.txt"
else
   echo "Файл file.txt не найден"
fi
```

В качестве команды, проверяющей условие, может фигурировать не только test, но и любая другая команда. Например:

```
if gcc -Wall -g myprog.c -o myprog; then
   echo "Компиляция прошла успешно"
else
   echo "При компиляции произошла ошибка"
fi
```

Кроме оператора ветвления, язык Bourne Shell поддерживает и более сложные конструкции, в том числе циклы. Например, следующий фрагмент напечатает все числа от 1 до 100:

```
I=0
while [ $I -le 101 ]; do
  echo $I
  I=$(( $I + 1 ))
done
```

Использовать информацию об успешности выполнения команды можно также с помощью так называемых логических связок `&&` и `||`, означающих, соответственно, логические операции "и" и "или". При этом логическому значению "истина" соответствует успешное завершение команды, а значению "ложь" – неуспешное. Командная строка

```
$ cmd1 && cmd2
```

заставит интерпретатор выполнить сначала команду `cmd1`, а затем в случае ее **успешного** завершения – команду `cmd2`; в случае неуспешного завершения первой команды вторая выполняться не будет. Наоборот, командная строка

```
$ cmd1 || cmd2
```

подразумевает запуск `cmd2` в случае **неуспешного** завершения `cmd1`.

Приоритет логических связок между собой – традиционный (то есть "и" приоритетнее, чем "или". В то же время, приоритет операций "конвейер" и перенаправлений ввода-вывода выше, чем приоритет логических связок; так, команда

```
$ cmd1 && cmd2 | cmd3
```

представляет собой связку между командой `cmd1` *и конвейером* `cmd2 | cmd3` *как целым. Значение «истинности» конвейера определяется успешностью или неуспешностью выполнения последней из составляющих его команд.*

Очередность применения операций, как обычно, можно изменить использованием круглых скобок, например:

```
$ (cmd1 && cmd2) | cmd3
```

В этом примере стандартный вывод команд `cmd1` *и* `cmd2` *(если, конечно, она вообще будет выполняться), будет направлен на стандартный ввод* `cmd3`.

Для более подробной информации о программировании на языке Bourne Shell следует обратиться к специальной литературе [1].

2.9.2 Протоколирование сеанса работы (команда script)

При выполнении заданий практикума часто требуется представить протокол сеанса работы с программой, т.е. текст, включающий как информацию, вводимую пользователем, так и информацию, выдаваемую программой. Это легко сделать с помощью команды `script`.

Чтобы начать протоколирование, запустите команду `script` с одним параметром, задающим имя файла протокола. Для окончания протоколирования нажмите `Ctrl-D` ("конец файла"). Например:

```
$ script my_protocol.txt
Script started, file is my_protocol.txt
$ ls
a1.c   Documents   my_protocol.txt tmp
$ echo "abc"
abc
$ [Ctrl-D]
Script done, file is my_protocol.txt
```

Файл `my_protocol.txt` теперь содержит протокол сеанса работы:

```
Script started on Wed Nov 17 16:31:54 2005
$ ls
a1.c   Documents   my_protocol.txt tmp
$ echo "abc"
abc
$
Script done on Wed Nov 17 16:32:14 2005
```

2.9.3 Команды head и tail

При работе в среде ОС Unix, особенно в процессе разработки программ, могут оказаться очень полезными команды `head` и `tail`. Эти команды служат для просмотра содержимого в начале и в конце файла, соответственно (по умолчанию показывается 10 первых или последних строк). Так, команда

```
$ head prog.c
```

покажет 10 первых строк файла `prog.c`, что может быть полезно, например, чтобы узнать, какие заголовочные файлы в него включаются.

Командам `head` и `tail` можно указать и несколько файлов за один раз. Например, можно быстро просмотреть начало всех файлов с программами на языке C в текущей директории, выполнив команду

```
$ head *.c *.h
```

На экран (точнее, в стандартный вывод) будут выведены первые 10 строк[9] каждого файла из текущей директории с расширением `.c` или `.h`. При этом перед выдачей информации из очередного файла выводится имя этого файла.

Количество строк, которые следует вывести, можно задать в командной строке. Например, команды

```
$ head -25 prog.c
$ tail -25 prog.c
```

выдадут, соответственно, 25 первых или 25 последних строк файла `prog.c`.

Команда `tail` особенно удобна для отслеживания появления новой информации в конце файла, в который постоянно производится вывод (например, в файле протокола). Так, команда

```
$ tail -f mylog.txt
```

выдаст последние 10 строк файла `mylog.txt`, а затем будет выводить новые строки по мере того, как они появляются в файле. Завершить работу команды можно стандартным образом – нажав комбинацию клавиш `Ctrl-C`.

Важным свойством этих команд (как и многих других) является то, что они могут работать как фильтры[10]. Например, команда

```
$ gcc -c sh.c 2>&1 | head -20
```

выведет на экран первые 20 строк диагностики компилятора `gcc` (в случае, если есть ошибки, конечно). *Заметим, что здесь понадобилось объединить стандартный поток сообщений об ошибках (2) со потоком стандартного вывода (1) с помощью конструкции* `2>&1` *(см. § 2.5, табл. 5), поскольку конвейер перенаправляет только поток стандартного вывода.*

[9] В случае, если в файле меньше 10 строк, будет выведен весь файл

[10] Под фильтом понимается программа, читающая некий поток данных со своего стандартного ввода, выполняющая те или иные преобразования и выдающая результаты в поток стандартного вывода; именно из таких программ строятся конвейеры.

2.9.4 Команда cat

Эта команда копирует содержимое файлов, указанных в командной строке, в поток стандартного вывода. Например, можно просмотреть содержимое файла без обращения к текстовому редактору, используя следующий конвейер:

```
$ cat sh.c | less
```

Напомним, что команда less используется для постраничного просмотра текста.

В зависимости от опций команда `cat` может преобразовывать свой ввод. Например, команда

```
$ cat -n sh.c
```

выдаст содержимое файла `sh.c`, пронумеровав все его строки, начиная с единицы. Команда

```
$ cat -v fl.txt
```

распечатает файл `fl.txt`, показывая все управляющие символы (т.е. символы с кодами из дапазона 0–31), кроме LF (перевод строки) и TAB (табуляция), и символы с кодом больше 127, в пригодном для восприятия виде. Управляющие символы показываются в виде ^СИМВОЛ, причем код символа получается из кода соответствующего управляющего символа прибавлением 64 (0x40). Например, символу с кодом 0 соответствует `^@`, с кодом 1 – `^A`, с кодом 2 – `^B`, с кодом 13 – `^M` и т.д. (символ `@` имеет код 64, латинская заглавная буква `A` – 65 и т.д.)

Символы с кодами больше 127 (из т.н. национальной половины кодового набора) показываются в виде M-СИМВОЛ, причем код символа получается из кода соответствующего национального символа вычитанием 129 (0x80). При этом, если получаемый СИМВОЛ относится к категории управляющих (и, как следствие, не имеет визуального образа), то он изображается в виде ^СИМВОЛ, как было объяснено выше. Например, если файл `binary_file` содержит следующую информацию (побайтно в 16-ичном виде):

```
80 FF D0 0A
```

то команда

```
$ cat -v binary_file
```

выдаст следующую строку

```
M-^@M-^?M-P
```

так как код `@` – 0x40, код `?` – 0x3F, код заглавной латинской буквы `P` – 0x50.

Отметим, что и команда `cat` может работать в режиме фильтра, т.е. читать информацию из потока стандартного ввода. Это происходит, если в командной строке не задано ни одного имени файла; в этом случае команда просто копирует поток ввода в поток вывода. С учетом вышеописанных возможностей преобразования такой режим команды может оказаться весьма полезен. Также команда `cat` без параметров (то есть в режиме буквального копирования ввода на вывод) оказывается полезной в связке с другими программами; так, связка

```
$ ./prog1 | (head -5 ; cat > /dev/null )
```

позволяет увидеть первые пять строк выдачи команды `prog1`, при этом дав программе доработать до конца (при отсутствии команды `cat` программа `prog1` получила бы сигнал `SIGPIPE` после окончания выполнения `head` и была бы, скорее всего, снята с выполнения).

2.9.5 Команды wc и yes

Эти команды удобно использовать, например, для отладки консольных приложений.

Команда `wc` подсчитывает количество строк, слов и байтов в файле для каждого файла, специфицированного в командной строке:

```
$ wc /etc/resolv.conf
8  15 131 /etc/resolv.conf
```

Это означает, что файл `/etc/resolv.conf` содержит 8 строк, 15 слов и 131 байт. Как и другие программы, `wc` может использоваться в режиме фильтра для подсчета символов, строк и слов в потоке стандартного ввода (в этом случае имя файла не печатается):

```
$ cat /etc/resolv.conf | wc
      8       15      131
```

Перечислим некоторые пции команды wc:

- `-c` – подсчитывать количество байтов
- `-m` – подсчитывать количество символов (заметим, что количество символов может быть меньше количества байтов для некотоых кодировок, например, UNICODE или UTF-8)

- `-w` – подсчитывать количество слов
- `-l` – подсчитывать количество строк
- `-L` – вычислить максимальную длину строки

Отметим, что под словом понимается последовательность символов, ограниченная пробельными символами (т.е. пробелами, табуляциями, переводами строк и т.п.)

Команда `yes`, наверное, самая простая из всех рассмотренных: она просто построчно выводит свои аргументы в стадартный поток вывода, причем делает это в бесконечном цикле , т.е. пока не произойдет ошибка вывода, либо процесс не будет завершен (по команде `kill`, по сигналу завершения с клавиатуры – `Ctrl-C`, и т.п.). Например, команда

```
$ yes Hello, world
```

будет бесконечно (и достаточно быстро) выводить

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
...
```

Изначально команда `yes` предназначалась для формирования потока ввода для программ, задающих много вопросов и ожидающих ответа “yes” или “no”.

2.9.6 Команда grep

Команда `grep` предназначена для поиска в текстовых файлах строк, удовлетворяющих заданным условиям. Эта команда может претендовать на звание самой сложной из разобранных здесь, хотя частные (и частые) случаи её использования весьма просты. Например, команда

```
$ grep include *.c
```

отыскивает строки, содержащие слово `include` во файлах с расширением `.c` из текущей директории. Для каждой найденной строки в стандартный поток вывода выводится имя файла, двоеточие и сама найденная строка:

```
hel.c:#include <fcntl.h>
hel.c:#include <sys/stat.h>
kr.c:#include <stdio.h>
kr.c:#include "mod1.h"
```

В случае, если grep читает текст со стандартного ввода (т.е. используется как конвейерный фильтр), либо в командной строке задан всего один файл, на выводе имя файла и двоеточие не печатаются:

```
$ cat *.c | grep include
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include "mod1.h"
```

В общем случае команда grep отыскивает в вводимом тексте (со стандартного потока ввода, либо из файлов, специфицированных в командной строке) строки, удовлетворяющие некоторому шаблону. Шаблон задается так называемым регулярным выражением, позволяющим описывать весьма нетривиальные поисковые условия. За недостатком места мы не будем описывать здесь синтаксис и семантику регулярных выражений; заинтересованные читатели легко найдут полное описание в соответствующей документации. Простейшим случаем регулярного выражения является последовательность символов (как в примере выше). Если эта последовательность содержит пробелы или символы, имеющие специальное значение для командного интерпретатора, их можно заключить в одинарные апострофы:

```
$ grep 'include <' *.c
hel.c:#include <fcntl.h>
hel.c:#include <sys/stat.h>
kr.c:#include <stdio.h>
```

Естественно, поиском явно указанной цепочки символов возможности `grep` не ограничиваются. Для иллюстрации отметим два часто используемых специальных символа в регулярных выражениях: символ `^` соответствует началу строки, а символ `$` – концу строки. Так, команда

```
$ grep ' $' *.c
```

поможет отыскать во всех файлы в вашей программе такие строки, в конце которых остались невидимые пробелы (отметим, что присутствие таких пробелов считается нежелательным).

Команда `grep` имеет достатчно большое количество опций, среди которых отметим только две: `-v` заставляет искать строки, **не** соответствующие заданному выражению; `-r` позволяет производить рекурсивный поиск по всем файлам, содержащимся в заданных директориях. Так, команда

```
$ grep -v '^ ' myfile
```

позволяет выбрать из файла `myfile` все строки, **кроме** тех, которые начинаются с пробела; команда

```
$ grep -r '^void very_strange_function(' *
```

позволит найти файл, в котором имеется строка, начинающаяся с цепочки

```
void very_strange_function(
```

среди всех видимых файлов текущей директории и всех ее поддиректорий; такая команда может оказаться крайне полезной, если вы, например, забыли, в каком из файлов вашей программы описана функция `very_strange_function`, имеющая тип возвращаемого значения `void`.

Отметим также, что команда grep завершается успешно, если была найдена хотя бы одна строка, удовлетворяющая шаблону, и неуспешно иначе. Данное обстоятельство используется при комбинировании команды grep с другими командами, например, `find` (см. ниже), а также при программировании на языке Bourne Shell в управляющих конструкциях (см. § 2.9.1).

2.9.7 Команда find

Эта команда предназначена для рекурсивного поиска файлов в директориях. Вкупе с описанной выше командой grep, которая анализирует содержимое файлов, команда find представляет собой очень мощное и гибкое средство поиска файлов, превосходящее по своим возможностям специализированные средства поиска, встроенные в файл-менеджеры ОС семейства Windows.

Рассмотрим работу команды на ряде примеров (как всегда, полное описание возможностей команды можно найти в оперативной документации с помощью команд `man` или `info`; см. §2.8).

```
$ find .
```

выводит **все** файлы из текущей директории и рекурсивно из поддиректорий. Точно так же работают команды

```
$ find . -print
```

или

```
$ find -print
```

Команда

```
$ find /home/igor
```

выводит все файлы из директории `/home/igor` и всех ее поддиректорий.

Команда

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

выведет имена всех файлов с расширением `.c` из текущей директории и ее поддиректорий, например:

```
./hel.c
./kr.c
./SQLTest/sqlcl.c
./SQLTest/sqlsrv.c
```

С помощью команды

```
$ find ~ -name core
```

можно найти все файлы с именем `core` в вашей домашней директории (напомним, что командный интерпретатор преобразует символ ~ в имя домашней директории текущего пользователя).

Команда

```
$ find . -name "*.c" -exec grep include {} \;
```

отыскивает и выводит в стандартный поток вывода все строки, содержащие слово `include`, из файлов с расширением `.c`, находящихся в текущей директории или в её поддиректориях. Разберем эту команду подробнее. Точка означает, что `find` должен перебрать все файлы из текущей директории и из её поддиректорий. Предложение `-name "*.c"` означает, что отбираются только файлы с именем, удовлетворяющим заданной маске, т.е. с расширением `.c`. `-exec include {} \;` означает, что вместо вывода имени файла для каждого отобранного имени файла запускается команда `grep` со строкой `include` и именем файла в качестве аргументов, т.е. имя отобранного файла подставляется вместо `{}`. Конец команды, которую должен выполнить `-exec`, обозначается символом “`;`”, который мы экранируем обратной косой чертой, чтобы интерпретатор не воспринял его как разделитель двух команд.

Результат команды grep выводится в стандартный поток вывода (при этом grep работает в режиме фильтра, т.е. имя файла перед строкой не печатается).

Если мы хотим увидеть также имя файла, в котором встретилась найденная строка, то можно выдать следующую команду:

```
$ find . -name "*.c" -exec grep include {}  \; -print
```

В результате будет выдано

```
#include <fcntl.h>
#include <sys/stat.h>
./hel.c
#include "stdio.h"
./kr.c
```

т.е. после запуска команды `grep` (и её успешного завершения!) по опции -print выполняется вывод имени файла.

В общем случае, в команде `find` нужно задать два объекта: местоположение файлов и выражение, вычисляемое для каждого имени файла.

Местоположение файлов – это список файловых имен (допускаются маски имен файлов). Если файловое имя, указанное в команде или подходящее под маску, соответствует поддиректории, то выбираются (рекурсивно) все файлы из этой поддиректории. По умолчанию в качестве местоположения выбирается текущая директория (.). Например:

```
$ find /home/igor/*.h /usr/include/*.h /usr/local/include/*.h
```

Здесь перебираются все файлы с расширением `.h` из директорий `/home/igor`, `/usr/include` и `/usr/local/include`. Файлы из поддиректорий выбираться не будут, за исключением маловероятного случая, когда в указанных директориях существует поддиректория с расширением h. В этом случае будут выбраны ВСЕ файлы из данной поддиректории, включая файлы из её поддиректорий.

Ключевое значение для понимания работы команды find играет понятие выражения. Для каждого выбранного файла это выражение вычисляется, причем в процессе вычисления выражения над файлом могут выполняться какие-либо команды (как в примере с командой grep выше). По умолчанию выражением является опция `-print`, результатом вычисления которой будет вывод полного имени файла в стандартный поток вывода. Значением каждого выражения является логическое значение (истина или ложь).

Простейшим случаем выражения является опция (начинающаяся со знака минус). Примеры опций:

- `-print` – выводит полное имя файла. Всегда истинна.
- `-name NAME` – истинна, если имя файла (без полного пути) соответствует NAME. Например, `-name core` выбирает файлы с именем `core`; `-name "*.c"` выбирает файлы с расширением `.c`.
- `-perm MODE` – истинна для файлов, права доступа к которым совпадают с константой `MODE`. Например, `-perm 666` выбирает только неисполняемые файлы, доступные на чтение и запись для любого пользователя.
- `-exec COMMAND ;.` – при вычислении выполняет команду `COMMAND`. При этом выбранное имя файла подставляется вместо комбинации символов `{}` Истинность опции определяется успехом ("истина") или неуспехом ("ложь") выполнения команды COMMAND. Обычно символ ";", ограничивающий выполняемую команду, приходится экранировать (то есть ставить перед ним символ \), чтобы интерпретатор командной строки не принял его за свой спецсимвол. Пример опции `-exec` разобран выше.

Существует большое количество опций команды `find`, которые, в частности, позволяют отбирать файлы по временам (создания, модификации и т.п.), правам доступа, владельцам, задавать более тонкие, чем показанные выше, критерии отбора имени файла (регулярные выражения, игнорирование регистра, глубина поиска в поддиректориях, тип файловой системы) и т.д. и т.п.

Выражения (опции) могут комбинироваться с помощью логических операций. По умолчанию, в качестве операции берется коньюнкция, т.е. `expr1 expr2` означает, что выражение `expr2` будет выполняться, только если выражение `expr1` истинно. Именно поэтому в команде

```
$ find . -exec grep main {} \; -print
```

опция `-print` будет вычислена (выполнена) и полное имя файла выведено, только если команда `grep` нашла в выбранном файле (а выбираются все файлы из текущей директории и её поддиректорий) строку `main`. Если строка не найдена, то и имя не будет выведено. В случае, если мы по ошибке поместим опцию `-print` перед опцией `-exec`, то будут выведены имена всех файлов, перемежаемые строками, содержащими `main`.

Разумеется, кроме коньюнкции поддерживаются и другие логические операции, а именно отрицание, обозначаемое восклицательным знаком, и дизъюнкция, обозначаемая `-o`. Например, команда

```
$ find /usr/include \! -name "*.h"
```

выведет все имена файлов из `/usr/include`, не имеющие расширения `.h` (обратите внимание на пробелы до и после "`\!`").

Команда

```
$ find . -name "*.cpp" -o -name "*.c"
```

выведет все имена файлов с расширениями `.c` и `.cpp`. Если опустить операцию `-o`, то ни одного имени не будет выведено, поскольку две опции `-name` окажутся (по умолчанию) связаны коньюнкцией.

Отметим, что логические операции имеют традиционный приоритет, поэтому команда

```
$ find . -name "*.cpp" -o -name "*.c" -exec grep main {} \; -print
```

выведет строки, содержащие `main`, только для файлов с раширением `.c` (т.к. приоритет коньюнкции выше приоритета дизъюнкции):

```
int main()
./kr.c
int main()
./hel.c
```

Чтобы `grep` выполнялся для обоих расширений, необходимо условие с `-o` заключить в круглые скобки, при этом не забыв их заэкранировать, чтобы интерпретатор командной строки не принял их за свои символы. Результирующая команда выглядит страшновато, но делает все, что нужно (снова обратите внимание на пробелы – они значащие!):

```
$ find . \( -name "*.cpp" -o -name "*.c" \) -exec grep main {} \; -print
int main()
./kr.c
int main()
./hel.c
int main()
./kr.cpp
```

В заключение заметим, что приведенные примеры демонстрируют только очень небольшую часть функциональности команды find, но и их достаточно, чтобы ощутить мощность и выразительность этой команды и, в общем случае, концепции командной строки Unix.

3 Графическая оболочка X Window

3.1 Основные понятия X Window

Одним из основных отличий Unix от остальных популярных современных операционных систем является проведение четкого различия между тремя фундаментальными понятиями: **операционная система**, **графическая подсистема** и **графический интерфейс пользователя**.

Операционная система – это комплекс программ для управления ресурсами компьютера на логическом и физическом уровне.

Графическая подсистема (ГП) – это комплекс программ для управления вводом и отображением графической информации. ГП использует сервисы операционной системы (драйвер клавиатуры, мыши, сетевые сервисы, файловую системуи т.п.), специфические драйверы устройств (графический видеоадаптер, планшет) и предоставляет прикладным программам интерфейс для работы с графической информацией. ГП в Unix работает под управлением ОС и использует ОС, но не является при этом частью ОС. Так, нередко компьютеры (например, сервера) под управлением Unix вообще не содержат ГП. Это существенно упрощает как профессиональную работу с ОС, так и разработку прикладных и системных программ.

Интерфейс пользователя (ИП) – это прежде всего набор понятий и правил взаимодействия пользователя-человека с компьютерной системой. Например, в ИП входят понятия меню, главного окна приложения (содержащего рамку, меню, кнопки управления окном – т.н. элементы оформления окна), командной кнопки, стандарты оформления меню, окон, кнопок, курсоров (тени, границы, шрифты и др.), правила взаимодействия пользователя с меню и главным окном (как управлять размерами и положением окна, как управлять меню с помощью клавиатуры и/или мыши) и многие другие вещи. Если ИП использует графическую подсистему, то его называют графическим интерфейсом пользователя (ГИП)[11]. В ОС Unix есть возможность выбора как между типами ИП (например, можно использовать многооконный интерфейс для алфавитно-цифровых терминалов, основанный на библиотеке curses[12]), так и между вариациями конкретных ГИП (можно модели-

[11] Соответствующий английский термин – Graphical user interface, GUI

[12] Библиотека curses (современная версия называется ncurses) включает в себя функции управления экраном алфавитно-цифрового терминала, основанные на окнах; реализация функций curses пользуется базой данных об escape-последовательностях, поддерживаемых различными типами терминалов, и позволя-

ровать существующие стандарты ГИП, либо создавать свои варианты). Разумеется, кроме понятий и правил в ГИП также входят программы и библиотеки, реализующие эти понятия и правила.

Рассмотрим подробнее ГП и ГИП, используемые в настоящее время в Unix.

Установившимся стандартом графической подсистемы для Unix стала так называемая X Window System (далее – X Window). Основными понятиями X Window являются X-сервер, X-протокол и X-клиент.

X-сервер – это специальная программа, выполняющая основные функции ГП. X-сервер отвечает за вывод на графический дисплей текстовых строк, визуализацию базовых графических примитивов (точка, отрезок, дуга, прямоугольник, курсор и т.д.), управление цветом (палитрой) и т.п. Запросы на эту деятельность X-сервер получает от прикладных программ, использующих графическую подсистему. Такие прикладные программы называются X-клиентами (как и полагается серверу, X-сервер может работать одновременно с несколькими клиентами). Также X-сервер занимается вводом информации от пользователя – через клавиатуру и мышь. Важно понимать, что X-сервер **не занимается** обработкой и интерпретацией введенной информации; всю эту информацию он передает X-клиентам, которые и ответственны за эти функции. "Революционность"идеи X Window (эта система начала разрабатываться в 80-ых годах прошлого века) состоит в том, что общение X-клиентов и X-сервера может происходить по сети: в частности, в качестве транспортного протокола может использоваться TCP/IP, а в качестве прикладного протокола – так называемый X-протокол, который и описывает взаимодействие прикладных программ (X-клиентов) и X-сервера. Заметим, что в такой схеме и X-сервер, и X-клиенты могут функционировать как на разных архитектурах компьютеров, так и в различных операционных системах (что и было одной из целей разработчиков X Window System). Можно запустить программу – X-сервер под Unix, MS Windows, Mac OS, VMS и другими операционными системами (реализации X-серверов существуют практически для всех операционных систем и архитектур) и работать с программами (X-клиентами), работающими на другом компьютере (кстати, не обязательно под управлением Unix).

Отметим на всякий случай еще раз, что **под X-сервером понимается программа, собственно осуществляющая отображение графических объектов на экране, при этом такое отображение**

ет программам работать одинаково на терминалах разных моделей и с различными размерами экрана

считается *услугой*, которую оказывает X-сервер прикладным программам.

X-сервер под Unix – это обычный процесс, который, правда, имеет определенные привилегии для доступа к соответствующему оборудованию (например, памяти видеоадаптера).

Далее мы будем рассматривать только X Window для Unix. В настоящее время наиболее распространенными реализациями являются свободно распространяемые XFree86 и X.org.

Важным понятием X Window является оконный менеджер (ОМ). Эта программа отвечает за работу с элементами обрамления окна, способами создания, уничтожения, перемещения и изменения размера окон, определяет общий внешний вид дисплея (так называемое корневое окно), специфику работы с мышью и т.д. Оконный менеджер реализует значительную часть функциональности ГИП. С одной стороны, оконный менеджер – это обычная клиентская программа (X-клиент); с другой стороны, X-сервер особым образом взаимодействует с ним; в частности, нельзя запустить несколько оконных менеджеров для дисплея – ОМ должен быть единственным. Unix предлагает большой выбор оконных менеджеров; к этому вопросу мы вернемся чуть позже.

Итак, оконный менеджер берет на себя внешнее оформление и работу с главными окнами приложений, использующих ГИП. Остальную часть функциональности ГИП берет на себя клиентское приложение. X-клиенты не реализуют X-протокол непосредственно, используя вместо этого специальную библиотеку, называемую Xlib, функции из которой и взаимодействуют с X-сервером через сеть. Xlib – это низкоуровневая библиотека, позволяющая использовать все возможности X Window, подобно языку ассемблера, который позволяет непосредственно использовать все возможности архитектуры компьютера. Ценой такой мощности служит в обоих случаях большая сложность и трудоемкость создания программ. Существует ряд библиотек, «надстраивающих» Xlib и позволяющих более эффективно и быстро создавать программы с ГИП. В настоящее время среди программистов, пишущих на C и C++, наиболее популярными являются библиотеки GTK+ (ориентированная на оболочку GNOME) и Qt (ориентированная на оболочку GNOME). Однако возможности X Window можно использовать и на языках, отличных от C/C++, например, Lisp, Ada, Tcl/Tk, Python и др.

Одной из самых популярных X-клиентских программ является `xterm` – эмулятор алфавитно-цифрового дисплея для X Window. В работе может быть удобно завести одновременно несколько экземпляров процесса `xterm`, каждый из которых порождает своё окно, в котором запускает копию интерпретатора командной строки. В одном окне мы можем запускать редактор vim, в другом – выполнять трансляцию и отладку, в третьем – запускать тестовые программы и т.д.

Вообще, профессиональные пользователи Unix (программисты и системные администраторы) обычно все действия с файлами и запуском программ осуществляют с использованием средств командной строки; при работе с X Window эти средства доступны с помощью программы `xterm` и других подобных программ. Разумеется, для ОС Unix существуют и чисто графические оболочки, позволяющие запускать программы и обрабатывать файлы (копировать их, переименовывать, стирать и т.п.), по интерфейсу напоминающие файловые менеджеры ОС Windows и MacOS. Следует, однако, понимать, что человек, владеющий средствами командной строки ОС Unix, любое действие выполнит с ее помощью существенно (иногда в десятки раз) быстрее, нежели с помощью графического интерфейса.

Заметим, что процесс программирования в ОС Unix, безусловно, требует владения средствами командной строки, т.е. программист (в отличие от конечных пользователей) изучить средства командной строки попросту вынужден. Поскольку данное пособие ориентировано на студентов программистских специальностей, позволим себе настоятельно порекомендовать читателю хотя бы первое время (два-три месяца) воздержаться от использования графических файловых менеджеров и прочих подобных программ при работе с ОС Unix; преимущества командной строки вскоре станут для вас очевидны.

Сказанное никоим образом не означает отказа от использования графической подсистемы как таковой. Работа с использованием X Window оказывается удобнее традиционной работы с алфавитно-цифровой консолью хотя бы тем, что позволяет запустить неограниченное количество `xterm`'ов одновременно. Кроме того, многие выполняемые с помощью компьютера задачи требуют возможности отображения графики; к таким задачам, например, относятся обработка фотографий и видео, просмотр документов, содержащих иллюстрации, компьютерная верстка и т.п.

3.2 Запуск X Window и выбор оконного менеджера

В зависимости от конфигурации конкретной машины, система X Window может оказаться уже запущена, либо вам необходимо будет запустить ее самостоятельно. Обычно это делается с помощью команды `startx`, которая, в свою очередь, запускает программу `xinit`.

Возможно, что в вашей локальной сети присутствует машина, выполняющая роль сервера приложений на основе `xdm`*; к такой машине можно подключиться с помощью штатных средств X Window таким образом, что все ваши программы будут выполняться на этой (удаленной) машине, а на вашем локальном рабочем месте будет осуществляться только отображение их графических окон, т.е. ваш компьютер будет играть роль X-терминала. Чтобы проверить, есть ли в вашей сети xdm-сервера, попробуйте дать команду*

```
$ X -broadcast
```

Если xdm-сервер в вашей сети действительно есть, после переключения в графический режим вы увидите приглашение к вводу входного имени и пароля для входа на этот сервер. Если xdm-серверов в сети больше одного, сначала вы увидите их список с предложением выбрать, услугами какого из них вы желаете воспользоваться.

Если в течение 15–20 секунд после перехода в графический режим ничего подобного не произошло – скорее всего, xdm-сервер в вашей сети отсутствует.

Команда `startx` вместе с X-сервером запустит для вас тот или иной оконный менеджер. В некоторых системах оконный менеджер можно выбрать из меню, появляющегося сразу после запуска X Window, в других системах выбор конкретного оконного менеджера определяется конфигурацией.

Если конфигурация системы вас по тем или иным причинам не устраивает, вы можете настоить ее по своему вкусу путем создания в домашней директории файла с именем `.xinitrc` (либо его редактирования, если такой файл уже есть). Обратите внимание на точку перед именем файла, она важна.

По сути `.xinitrc` представляет собой командный файл, из которого запускаются прикладные программы, включая и сам оконный менеджер. Программа `xinit` запускает X-сервер, а затем, соответствующим образом установив переменные окружения, начинает выполнение команд из `.xinitrc`. Завершение `xinitrc` означает завершение сеанса работы с X Window, при этом процесс X-сервера завершается.

Простейший пример `.xinitrc` может выглядеть примерно так:

```
xterm &
twm
```

В этом случае сначала будет запущен `xterm` (его мы запускаем на всякий случай, чтобы можно было работать, даже если оконный менеджер имеет неудобную конфигурацию), после чего – стандартный оконный менеджер `twm`. Обратите внимание, что `xterm` запускается в фоновом режиме (для этого в конце первой строки поставлен знак &). Это сделано, чтобы не ожидать его завершения для запуска twm.

Оконный менеджер twm достаточно примитивен. Хотя многие пользователи Unix используют именно его[13], вы, возможно, захотите попробовать другие оконные менеджеры. В вашей системе могут быть установлены, кроме twm, такие оконные менеджеры, как fvwm, icewm, wmaker и другие; кроме того, вы можете обнаружить и более развитые ОМ, такие как KDE и GNOME[14]. Чтобы воспользоваться icewm, измените ваш .xinitrc:

```
xterm &
icewm
```

Если вы предпочитаете KDE, ваш файл может выглядеть так:

```
xterm &
startkde
```

Выбор конкретного оконного менеджера среди множества доступных зависит в основном от ваших личных предпочтений. Так, стандартный ОМ twm выглядит достаточно аскетично, тогда как KDE по вычурности своего графического дизайна, пожалуй, превосходит системы семейства Windows. В то же время KDE весьма требователен к ресурсам и ощутимо «тормозит» даже на сравнительно быстрых компьютерах, тогда как аскетичные twm, fvwm и даже достаточно развитый в дизайнерском отношении icewm работают с приемлемой скоростью даже на компьютерах с процессорами Pentium-1 (KDE на таких машинах лучше и не пробовать запускать).

Отметим, что KDE предназначен в основном для пользователей, не умеющих и не желающих работать в командной строке. Профессиональные программисты и системные администраторы обычно предпочитают оконные менеджеры попроще.

3.3 Работа с классическими оконными менеджерами

К классическим ОМ мы относим twm, fvwm и некоторые другие.

[13]Надо сказать, что twm в системе присутствует практически всегда, чего о других оконных менеджерах не скажешь

[14]Всего существует несколько десятков оконных менеджеров, так что приведенный список не может претендовать на полноту

Следует отметить, что любой оконный менеджер имеет весьма развитые средства настройки, позволяющие существенно изменить его поведение, так что дать исчерпывающую инструкцию по работе с каким-либо из оконных менеджеров на уровне «нажмите такую-то клавишу для получения такого-то результата» было бы затруднительно. В этом параграфе мы ограничимся общими рекомендациями, которые позволят вам быстро освоить работу с вашим оконным менеджером в той конфигурации, которая установлена в вашей системе. При желании вы можете изменить любые настройки оконного менеджера; за инструкциями по этому поводу следует обратиться к технической документации.

Итак, первое, что можно посоветовать после запуска оконного менеджера – это попытаться понять, каким образом в нем высветить главное меню. Во всех классических ОМ главное меню выдается, если щелкнуть левой кнопкой мыши **в любом свободном месте экрана** (то есть в таком месте, которое не закрыто никакими окнами). Отметим, что в классических ОМ не поддерживается ничего похожего на «пиктограммы на рабочем столе», так что запуск программ можно осуществить либо через меню, либо с помощью командной строки, доступ к которой можно получить, запустив программу `xterm` или аналогичную. Обычно `xterm` или какой-то его аналог имеется либо в самом главном меню, либо в его подменю, называемом «terminals», «shells» и т.п. Если у вас уже есть одно окошко с командной строкой, можно запустить новый экземпляр `xterm`, дав команду

```
$ xterm &
```

Обратите внимание на символ `&`. Программу `xterm` мы запускаем в фоновом режиме, чтобы старый экземпляр `xterm` (с помощью которого мы даем команду) не ждал его завершения: в противном случае запуск нового `xterm` теряет смысл, ведь у нас во время его работы не будет возможности пользоваться старым экземпляром.

Программа `xterm` имеет развитую систему опций. Например,

```
$ xterm -bg black -fg gray &
```

запустит эмулятор терминала на черном фоне с серыми буквами (тот же набор цветов обычно используется в текстовой консоли).

Классическим в системе X Window считается поведение, при котором фокус ввода находится в том окне, на котором в настоящий момент находится курсор мыши[15]. Такое соглашение позволяет, на-

[15]Это отличается от поведения в Windows, где для перемещения фокуса ввода необходимо выбрать окно, щелкнув по нему мышкой, при этом окно обязательно показывается полностью

пример, работать с окном, которое частично скрыто другими окнами, не вытаскивая его на верхний уровень. Такое поведение (именуемое FocusFollowsMouse), однако, может оказаться непривычным; при необходимости его можно сменить на более привычное, называемое ClickToFocus, с помощью настроек вашего оконного менеджера. В любом случае, следите внимательно за положением курсора мыши.

Показать полностью (поднять на верхний уровень) окно, которое частично скрыто, можно, «щелкнув» мышью по его **заголовку** (а не по любому месту окна). Ваши настройки могут предусматривать и обратную операцию («утопить» окно, показав то, что под ним) – обычно это делается «щелчком» **правой** кнопки мыши по заголовку. Для перемещения окна по экрану также служит его заголовок: достаточно навести на заголовок курсор мыши, нажать (и не отпускать) левую кнопку, выбрать новое положение окна и отпустить кнопку. Если заголовка окна не видно (например, он скрыт под другими окнами), ту же операцию можно проделать с помощью вертикальных и горизонтальных частей рамки окна, кроме выделенных участков в уголках рамки; эти угловые участки служат для изменения размеров окна, то есть при протягивании их мышкой перемещается не всё окно, а только тот его уголок, который вы захватили.

Если вы потеряли нужное вам окно, обычно его можно легко отыскать, щелкнув **правой** кнопкой мыши в свободном месте экрана. Обычно при этом выдается меню, состоящее из списка существующих окон.

В большинстве случаев оконные менеджеры поддерживают так называемый виртуальный рабочий стол (virtual desktop), состоящий из нескольких «виртуальных экранов», на каждом из которых можно разместить свои окна. Это бывает удобно, если вы работаете одновременно с большим количеством окон. Карта виртуального рабочего стола, на которой изображены виртуальные экраны, обычно находится в правом нижнем углу экрана; чтобы переключиться на нужный вам виртуальный экран, достаточно щелкнуть мышью в соответствующем месте карты.

Из окон, в которых отображается некий текст, обычно можно скопировать этот текст в другие окна. Для этого достаточно выделить текст мышью; в большинстве программ, работающих под управлением X Window, нет специальной операции «copy»: копируется ровно тот текст, который выделен.

Вставка выделенного текста осуществляется **третьей** (средней) кнопкой мыши. Если на вашей мышке только две кнопки, обычно система настраивается на имитацию третьей кнопки путем нажатия одновременно правой и левой кнопок. Если же ваша мышь оснащена «колесом» для скроллинга, обратите внимание на то, что обычно на это колесо можно нажать сверху вниз, не прокручивая его; в этой ситуации колесо работает как обычная (третья) кнопка, что вам, собственно, и требуется.

3.4 Работа в среде KDE

Один из оконных менеджеров, называемый KDE (K Desktop Environment), предоставляет графическую среду, интерфейс которой подобен пользовательским интерфейсам MS Windows, Mac OS и т.п.; KDE поддерживает привычные для пользователей этих систем понятия рабочего стола, пиктограмм, панели задач и др..

На рабочем столе KDE располагаются ярлыки запуска наиболее часто используемых программ. Принцип их установки на рабочий стол и использования такой же, как в Windows. Пользователь имеет право удалить любой ярлык или добавить свой для запуска той или иной программы.

Рекомендуется прямо во время первого сеанса вынести на рабочий стол или в панель рабочего стола пиктограмму, соответствующую программе «Konsole», имеющую вид компьютерного монитора. Это позволит быстро получать доступ к командной строке Unix.

Внизу рабочего стола располагается панель (desktop panel), предназначенная для управления функциями рабочего стола, индикации состояния запущенных программ в KDE, а также для размещения кнопок запуска часто используемых программ. Эту панель с помощью мыши можно переместить на любую сторону рабочего стола, а также убрать с экрана с помощью кнопок со стрелками, расположенными с правой и с левой сторон панели. Наиболее просто настроить внешний вид KDE и панели, если щелкнуть правой кнопкой мыши на рабочем столе KDE и в появившемся контекстном меню выбрать пункт «Настроить Рабочий Стол» («Configure Desktop...»). Для того, чтобы какие-либо изменения вступили в силу, надо нажать кнопку «Применить» («Apply»).

Самая важная кнопка на панели рабочего стола – кнопка входа в главное меню, на которой традиционно изображается буква «K». Эта кнопка всегда располагается слева. В различных версиях и главные меню могут различаться, но основные принципы их построения везде одинаковы. Роль этой кнопки аналогична роли кнопки «Пуск» («Start») в системах семейства Windows.

Справа от кнопки главного меню располагаются кнопки вызова наиболее важных или часто вызываемых программ. В разных версиях KDE набор этих кнопок может различаться, кроме того, пользователь может удалить и добавить кнопки для запуска любых программ. Пиктограммы кнопок также не имеют стандарта, хотя некоторое единообразие в них все же есть.

На панели рабочего стола KDE есть специальные кнопки в виде прямоугольника, поделенного на пронумерованные прямоугольнички (обычно их 4). Каждый прямоугольничек является кнопкой переключения рабочего стола, что дает пользователю возможность разложить окна используемых им в данном сеансе работы программ по разным экранам.

В центре панели рабочего стола располагается панель задач, где запущенные пользователем программы показаны кнопками (прямоугольниками) с названиями программ.

На правой части панели рабочего стола KDE расположены кнопки для вызова системных команд и индикации состояния ряда служебных программ или процессов. В правом углу панели всегда располагается циферблат системных часов. Щелчок левой кнопки мыши в этой области вызывает окно с календарем.

Для манипуляций с файлами и запуска программ в KDE можно воспользоваться файловым менеджером Konqueror, который по функциональности аналогичен файловому менеджеру Windows. Интерфейс окна программы в режиме просмотра файловой системы вполне стандартный и привычный: на левой панели окна представлено дерево файловой системы, а справа - файлы и подкаталоги в выбранном каталоге. Между панелями (или слева) обычно расположена панель управления с кнопками для переключения режима работы панелей. С помощью кнопок можно, например, переключить левую панель в режим отображения вкладок, в режим журнала, в режим проигрывателя, перейти в домашний каталог, в корневой каталог, получить доступ к сервису, к сети. Правая панель, в зависимости от настроек, может отображать пиктограммы каталогов и зарегистрированных файлов или показывать информацию о каталогах и файлах в виде текстовых строчек. При работе с графическими файлами на правой панели вместо пиктограмм могут выводиться иконки с изображением содержимого файла.

Следует учитывать, что те же операции над файлами быстрее и удобнее можно проводить средствами командной строки Unix.

4 Инструментарий программиста

4.1 Компилятор gcc/g++

Компиляторы семейства GCC (Gnu Compiler Collection) являются компиляторами командной строки, т.е. все необходимые действия задаются при запуске компилятора и выполняются уже без непосредственного участия пользователя. Это, в частности, позволяет использовать компилятор в командных файлах (скриптах).

Команда `gcc` предназначена для компиляции программ на языке C, а команда `g++` – на языке C++[16].

Имена файлов, подлежащих компиляции и линковке, компилятор принимает с командной строки. Кроме того, компилятор воспринимает большое количество опций. Вам обязательно понадобятся следующие из них:

- `-o` <*filename*> задает имя исполняемого файла, в который будет записан результат компиляции (если не указать эту опцию, результат компиляции будет помещен в файл `a.out`.).

- `-Wall` приказывает компилятору выдавать все разумные предупредительные сообщения (warnings). **Обязательно всегда используйте эту опцию**, она поможет вам сэкономить немало времени и нервов.

- `-ggdb` и `-g` используются для включения в результирующие файлы разнообразной отладочной информации (информации, используемой отладчиком, включая имена переменных и функций, номера строк исходных файлов и т.п.). Опция `-ggdb` снабжает файлы расширенной отладочной информацией, понятной только отладчику `gdb`. Если вам кажется, что что-то не в порядке с отладчиком, попробуйте использовать опцию `-g`.

- `-c` указывает компилятору, что результатом должна быть не вся программа, а отдельный ее модуль. В этом случае имя файла для объектного модуля можно не задавать, оно будет сгенерировано автоматически заменой расширения на `.o`.

[16]На самом деле, используется один и тот же компилятор; оба имени являются обычно символическими ссылками на исполняемый файл компилятора. Поведение компилятора зависит от того, по какому имени его вызвали; прежде всего, различие выражается в наборе стандартных библиотек, подключаемых по умолчанию при сборке исполняемого файла.

- `-O`*n* задает уровень оптимизации. `n=0` означает отсутствие оптимизации (значение по умолчанию). Для получения более эффективного объектного кода рекомендуется использовать опцию `-O2`. Учтите, что оптимизация может затруднить работу с отладчиком.

- `-ansi` приказывает компилятору работать в соответствии со стандартом ANSI C.

- `-pedantic` заставляет компилятор строже относиться к соблюдению стандарта.

- `-E` останавливает компилятор после проведения стадии макропроцессирования. Результат макропроцессирования выдается на стандартный вывод. Эта опция может быть полезна, если ваши макроопределения повели себя не так, как вы ожидали, и хочется понять, что на самом деле происходит.

- `-D` позволяет с командной строки (т.е. без изменения исходных файлов) определить в программе некий макросимвол. Это полезно, если в вашей программе используются директивы условной компиляции и требуется, не изменяя исходных файлов, быстро откомпилировать альтернативную версию программы. Например, `-DDEBUG=2` имеет такой же эффект, какой дала бы директива `#define DEBUG 2` в начале исходного файла.

- `-l` позволяет подключить к программе библиотеку функций. Так, если в вашей программе используются математические функции (`sin`, `exp` и другие), необходимо при компиляции задать ключ `-lmath`; в некоторых вариантах ОС Unix (например, в SunOS/Solaris) при использовании сокетов вам понадобится также ключ `-lnsl`.

- `-MM` анализирует заданные исходные файлы и строит информацию об их взаимозависимостях. О том, как использовать полученную информацию, рассказывается в §4.3.

Итк, чтобы откомпилировать программу, написанную на языке C и целиком находящуюся в файле `prog.c`, следует дать команду

```
gcc -g -Wall prog.c -o prog
```

При этом результат компиляции будет помещен в файл `prog` в текущей директории.

Чтобы откомпилировать программу, состоящую из нескольких модулей `mod1.c`, `mod2.c`, `mod3.c` и главного файла `prog.c`, следует сначала откомпилировать все модули:

```
$ gcc -g -Wall -c mod1.c
$ gcc -g -Wall -c mod2.c
$ gcc -g -Wall -c mod3.c
```

и получить объектные файлы `mod1.o`, `mod2.o`, `mod3.o`. После этого для компиляции основного файла и сборки готовой программы следует дать команду

```
$ gcc -g -Wall mod1.o mod2.o mod3.o prog.c -o prog
```

4.2 Отладчик gdb

Отладчик `gdb` (Gnu DeBugger) позволяет отлаживать программу в интерактивном режиме, пользуясь интерфейсом командной строки, а также анализировать причины "смерти" программы по созданному системой core-файлу.

Учтите, что для нормальной работы отладчика **необходимо**, чтобы все модули вашей программы были откомпилированы с ключем `-ggdb` или `-g` (см. §4.1). В некоторых случаях нормальной работе отладчика может помешать включенный при компиляции режим оптимизации, так что перед отладкой оптимизацию лучше отключить.

4.2.1 Пошаговое выполнение программы

Чтобы запустить отладчик для программы, исполняемый файл которой называется `prog`, следует дать команду

```
$ gdb prog
```

Отладчик сообщит свою версию и некоторую другую информацию, после чего выдаст приглашение своей командной строки, обычно выглядящее так: `(gdb)`.

Основные команды отладчика:

- `run` осуществляет запуск программы в отладочном режиме. Перед запуском целесообразно задать точки останова (см. ниже). Если вы затрудняетесь определить, где именно следует приостановить выполнение программы, поставьте точку останова на функцию `main()`.

- `list` показывает на экране несколько строк программы, предшествующих текущей и идущих непосредственно после текущей.

- `break` позволяет задать точку приостановки выполнения программы (breakpoint). Точка останова может быть задана именем функции, номером строки в текущем файле, либо выражением <имя-файла>:<номер-строки>, например `file1.c:73`.

- `inspect` позволяет просмотреть значение переменной (в том числе и заданной сложным выражением вроде `*(a[i+1].p)`).

- `backtrace` или `bt` показывает текущее содержимое стека, что позволяет узнать последовательность вызовов функций, приведшую к текущему состоянию программы.

- `frame` позволяет сделать текущим один из фреймов, показанных командой `backtrace`, что дает возможность исследовать значения переменных в этом фрейме и т.п.

- `step` позволяет выполнить одну строку программы. Если в строке содержится вызов функции, текущей строкой станет первая строка этой функции (т.е. процесс трассировки зайдет внутрь функции).

- `next` подобна команде `step`, с тем отличием, что вход в тела вызываемых функций не производится.

- `until` <номер-строки> позволяет выполнять программу до тех пор, пока текущей не окажется строка с указанным номером.

- `call` позволяет выполнить вызов произвольной функции.

- `cont` позволяет продолжить прерванное выполнение программы.

- `help` позволит узнать подробнее об этих и других командах отладчика.

- `quit` завершает работу отладчика (можно также воспользоваться комбинацией клавиш `Ctrl-D`).

4.2.2 Анализ причин аварийного завершения по core-файлу

Часто ошибки в программе приводят к ее аварийному завершению, при котором система создает так называемый `core`-файл. При этом выдается сообщение

```
Segmentation fault (core dumped)
```

Это означает, что в текущей директории аварийно завершенного процесса создан файл с именем `core` (или `prog.core`, где `prog` – имя вашей программы), в который система записала содержимое сегмента данных и стека программы на момент ее аварийного завершения.

С помощью отладчика gdb можно проанализировать этот файл, узнав, в частности, при выполнении какой строки программы произошла авария, откуда и с какими параметрами была вызвана функция, содержащая эту строку, каковы были значения переменных на момент аварии и т.д.

Чтобы запустить отладчик в режиме анализа core-файла, необходимо дать команду:

```
gdb prog prog.core
```

где `prog` – имя исполняемого файла вашей программы, а `prog.core` – имя созданного системой core-файла[17]. Очень важно при этом, чтобы в качестве исполняемого файла выступал именно тот файл, при исполнении которого получен core-файл. Так, если уже после получения core-файла вы перекомпилируете свою программу, анализ core-файла, скорее всего, приведет к ошибочным результатам.

Сразу после запуска отладчика в режиме анализа core-файла рекомендуется дать команду `backtrace` (или просто `bt`).

4.2.3 Анализ причин зацикливания

Если ваш процесс зациклился, не торопитесь его убивать. С помощью отладчика можно понять, какой фрагмент кода выполняется в настоящий момент, и проанализировать причины зацикливания. Для этого нужно *присоединить* отладчик к существующему процессу. Определите номер процесса с помощью команды `ps`. Допустим, имя исполняемого файла вашей программы – `prog`, и она выполняется как процесс номер 12345. Тогда команда запуска отладчика должна выглядеть так:

[17]Подобное имя core-файла характерно для ОС FreeBSD. В ОС Linux файл, скорее всего, будет называться просто core.

```
gdb prog 12345
```

При подключении отладчика выполнение программы будет приостановлено, однако вы сможете при необходимости продолжить его командой `cont`. В случае повторного зацикливания можно приказать отладчику вновь приостановить выполнение нажатием комбинации `Ctrl-C`.

4.3 Утилита make

При сборке ваших программ могут оказаться полезны возможности, предоставляемые утилитой make. Кратко говоря, эта утилита позволяет автоматически строить одни файлы на основании других (например, исполняемые файлы на основании исходных текстов программы) в соответствии с заданными правилами. При этом make отслеживает даты последней модификации файлов и производит перестроение только тех целевых файлов, для которых исходные файлы претерпели изменения.

Существует несколько различных версий утилиты make. Изложенное в данном параграфе соответствует варианту Gnu Make; именно этот вариант используется в большинстве дистрибутивов Linux. При работе с ОС FreeBSD для вызова Gnu Make необходимо использовать команду `gmake` вместо `make`. В случае, если в системе такой команды нет, обратитесь к системному администратору с просьбой её установить.

Правила для утилиты make задаются в файле `Makefile`, который утилита ищет в текущей директории.

4.3.1 Простейший Makefile

Допустим, ваша программа состоит из главного модуля `main.c`, содержащего функцию `main()`, а также из дополнительных модулей `mod1.c` и `mod2.c`, имеющих заголовочные файлы `mod1.h` и `mod2.h`. Соответственно, для сборки исполняемого файла (назовем его `prog`) необходимо дать следующие команды:

```
$ gcc -g -Wall -c mod1.c -o mod1.o
$ gcc -g -Wall -c mod2.c -o mod2.o
$ gcc -g -Wall main.c mod1.o mod2.o -o prog
```

Первые две команды даются для компиляции дополнительных модулей. Полученные в результате файлы `mod1.o` и `mod2.o` используются в третьей команде для сборки исполняемого файла.

Допустим, мы уже произвели компиляцию программы, после чего внесли изменения в файл `mod1.c` и хотим получить исполняемый файл с учетом внесенных изменений. При этом нам надо будет дать только две команды (первую и третью), поскольку перекомпиляции модуля `mod2` не требуется.

Чтобы подобные ситуации отслеживались автоматически, мы можем использовать утилиту `make`. Для этого напишем следующий `Makefile`:

```
mod1.o: mod1.c mod1.h
        gcc -g -Wall -c mod1.c -o mod1.o

mod2.o: mod2.c mod2.h
        gcc -g -Wall -c mod2.c -o mod2.o

prog: main.c mod1.o mod2.o
        gcc -g -Wall main.c mod1.o mod2.o -o prog
```

Поясним, что файл состоит из так называемых *целей* (в нашем случае таких целей три - `mod1.o`, `mod2.o` и `prog`). Описание каждой цели состоит из заголовка и списка команд. Заголовок цели – это **одна** строка, начинающаяся всегда с первой позиции (т.е. перед ней не допускаются пробелы и т.п.). В начале строки пишется имя цели (обычно это просто имя файла, который необходимо построить). Оставшаяся часть заголовка отделяется от имени цели двоеточием. После двоеточия перечисляется, от каких файлов (или, в более общем случае, от каких целей) зависит построение файла. В данном случае мы указали, что модули зависят от их исходных текстов и заголовочных файлов, а исполняемый файл – от основного исходного файла и от двух объектных файлов.

После строки заголовка идет список команд (в нашем случае все три списка имеют по одной команде). Строка команды **всегда начинается с символа табуляции**, причем замена табуляции пробелами недопустима и ведет к ошибке. Утилита `make` считает признаком конца списка команд первую строку, начинающуюся с символа, отличного от табуляции.

Имея в текущей директории вышеописанный Makefile, мы можем для сборки нашей программы дать команду `make prog`.

4.3.2 Переменные

В предыдущем параграфе описан Makefile, в котором можно обнаружить несколько повторяющихся фрагментов. Так, строка параметров компилятора “-g -Wall” встречается во всех трех целях. Помимо необходимости повторения одного и того же текста, мы можем столкнуться с проблемами при модификации. Предположим, нам понадобится задать компилятору режим оптимизации кода (флаг -O2). Для этого нам пришлось бы внести совершенно одинаковые изменения в три разный строки файла. В более сложном случае таких строк может понадобиться несколько десятков и даже сотен.

Аналогичная проблема встанет, например, если мы захотим произвести сборку другим компилятором.

Решить проблему позволяет введение *make-переменных*. Обозначим имя компилятора C переменной CC, а общие параметры компиляции – переменной CFLAGS[18].

Тогда наш Makefile можно переписать следующим образом:

```
CC = gcc
CFLAGS = -g -Wall -ansi -pedantic

mod1.o: mod1.c mod1.h
        $(CC) $(CFLAGS) -c mod1.c -o mod1.o

mod2.o: mod2.c mod2.h
        $(CC) $(CFLAGS) -c mod2.c -o mod2.o

prog: main.c mod1.o mod2.o
        $(CC) $(CFLAGS) main.c mod1.o mod2.o -o prog
```

4.3.3 Предопределенные переменные и псевдопеременные

Существуют определенные соглашения об именах переменных, причем некоторым переменным утилита make присваивает значения сама, если соответствующие значения не заданы явно.

Вот некоторые традиционные имена переменных:

- CC – команда вызова компилятора языка C;
- CFLAGS – параметры для компилятора языка C;
- CXX – команда вызова компилятора языка C++;
- CXXFLAGS – параметры для компилятора языка C++;

[18]Причины выбора именно таких обозначений станут ясны из дальнейшего изложения.

- CPPFLAGS – параметры препроцессора (обычно сюда помещают предопределенные макропеременные);
- LD – команда вызова системного линкера (редактора связей);
- MAKE – команда вызова утилиты make со всеми параметрами.

По умолчанию переменные CC, CXX, LD и MAKE имеют соответствующие значения, справедливые для данной системы и в данной ситуации. Значения остальных перечисленных переменных по умолчанию пусты.

Таким образом, при написании Makefile из предыдущего параграфа мы могли бы пропустить строку, в которой задается значение переменной CC, в надежде, что соответствующее значение переменная получит без нашей помощи.

Кроме таких переменных общего назначения, в каждой цели могут использоваться так называемые *псевдопеременные*. Перечислим наиболее интересные из них:

- $@ – имя текущей цели;
- $< – имя первой цели из списка зависимостей;
- $^ – весь список зависимостей.

С использованием этих переменных мы можем переписать наш Makefile следующим образом:

```
CFLAGS = -g -Wall

mod1.o: mod1.c mod1.h
	$(CC) $(CFLAGS) -c $< -o $@

mod2.o: mod2.c mod2.h
	$(CC) $(CFLAGS) -c $< -o $@

prog: main.c mod1.o mod2.o
	$(CC) $(CFLAGS) $^ -o $@
```

4.3.4 Обобщенные цели

Как можно заметить, в том варианте Makefile, который мы написали в конце предыдущего параграфа, правила для сборки обоих дополнительных модулей оказались совершенно одинаковыми. Можно пойти дальше и задать одно обобщенное правило для построения объектного файла для любого модуля, написанного на языке C, исходный файл которого имеет имя с суффиксом `.c`, а заголовочный файл – имя с суффиксом `.h`:

```
%.o: %.c %.h
        $(CC) $(CFLAGS) -c $< -o $@
```

Если теперь задать список дополнительных модулей с помощью переменной, получим следующий вариант Makefile:

```
OBJMODULES = mod1.o mod2.o
CFLAGS = -g -Wall -ansi -pedantic

%.o: %.c %.h
        $(CC) $(CFLAGS) -c $< -o $@

prog: main.c $(OBJMODULES)
        $(CC) $(CFLAGS) $^ -o $@
```

Теперь для добавления к программе нового модуля нам достаточно добавить имя его объектного файла к значению переменной `OBJMODULES`.

Если перечисление модулей через имена объектных файлов представляется неестественным, можно заменить первую строку Makefile следующими двумя строками:

```
SRCMODULES = mod1.c mod2.c
OBJMODULES = $(SRCMODULES:.c=.o)
```

Запись `$(SRCMODULES:.c=.o)` означает, что необходимо взять значение переменной `SRCMODULES` и в каждом входящем в это значение слове заменить суффикс `.c` на `.o`.

4.3.5 Псевдоцели

Утилиту `make` можно использовать не только для построения файлов, но и для выполнения, вообще говоря, произвольных действий. Добавим к нашему файлу две дополнительные цели:

```
run: prog
        ./prog

clean:
        rm -f *.o prog
```

Теперь по команде `make run` утилита `make` произведет, если необходимо, сборку нашей программы и запустит ее. С помощью же команды `make clean` мы можем очистить рабочую директорию от объектных и исполняемых файлов (например, если нам понадобится произвести сборку программы с нуля).

Такие цели обычно называют *псевдоцелями*, поскольку их имена не обозначают имен создаваемых файлов.

4.3.6 Автоматическое отслеживание зависимостей

В более сложых проектах модули могут использовать заголовочные файлы других модулей, что делает необходимой перекомпиляцию модуля при изменении заголовочного файла, не относящегося к этому модулю. Информацию о том, какой модуль от каких файлов зависит, можно задать вручную, однако этот способ приведет к трудностям в больших программах, поскольку программист при модификации исходных файлов может случайно забыть внести изменения в Makefile.

Более правильным решением будет поручить отслеживание зависимостей компьютеру. Утилита `make` позволяет наряду с обобщенным правилом указать список зависимостей для построения конкретных модулей. Например:

```
%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@

mod1.o: mod1.c mod1.h mod2.h mod3.h
```

В этом случае для построения файла `mod1.o` будет использовано обобщенное правило (поскольку никаких команд в цели `mod1.o` мы не указали), но список зависимостей будет использован из цели `mod1.o`.

Списки зависимостей можно построить с помощью компилятора. Чтобы получить строку, подобную последней строке вышеприведенного примера, необходимо дать команду

```
gcc -MM mod1.c
```

Если результат выполнения такой команды перенаправить в файл, то этот файл можно будет включить в наш Makefile директивой `include`. Эта директива имеет специальную форму со знаком `-`, при использовании которой `make` не выдает ошибок, если файл не найден. Если использовать для файла зависимостей имя `deps.mk`, соответствующая директива будет выглядеть так:

```
-include deps.mk
```

Более того, если предусмотреть цель для генерации файла, включаемого такой директивой, например:

```
deps.mk: $(SRCMODULES)
        $(CC) -MM $^ > $@
```

утилита `make`, прежде чем начать построение любых других целей, будет пытаться построить включаемый файл.

Отметим, что такое поведение нежелательно для псевдоцели clean, поскольку для очистки рабочей директории от мусора построение файлов зависимостей не нужно и только отнимает лишнее время. Чтобы избежать этого, следует снабдить директиву `-include` условной конструкцией, исключающей эту строку из рассмотрения, если единстенной целью, заданной в командной строке, является цель clean. Это делается с помощью директивы `ifneq` и встроенной переменной `MAKECMDGOALS`:

```
ifneq (clean, $(MAKECMDGOALS))
-include deps.mk
endif
```

Окончательно Makefile будет выглядеть так:

```
SRCMODULES = mod1.c mod2.c
OBJMODULES = $(SRCMODULES:.c=.o)
CFLAGS = -g -Wall -ansi -pedantic

%.o: %.c %.h
        $(CC) $(CFLAGS) -c $< -o $@

prog: main.c $(OBJMODULES)
        $(CC) $(CFLAGS) $^ -o $@

ifneq (clean, $(MAKECMDGOALS))
-include deps.mk
endif

deps.mk: $(SRCMODULES)
        $(CC) -MM $^ > $@

clean:
        rm -f *.o prog
```

Список литературы

[1] С. Баурн. Операционная система UNIX. М.:Мир, 1986.

[2] А. М. Робачевский. Операционная система UNIX. Изд-во «BHV–Санкт-Петербург», Санкт-Петербург, 1997.

Содержание