

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тульский государственный университет»

Институт прикладной математики и компьютерных наук
Кафедра вычислительной техники

Утверждено на заседании кафедры
«Вычислительная техника»
«26» января 2022 г., протокол №7

Заведующий кафедрой
_____ А.Н. Ивутин

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ
К ПРАКТИЧЕСКИМ РАБОТАМ**
по дисциплине (модулю)

«Операционные системы семейства Unix»

**основной профессиональной образовательной программы
высшего образования – программы бакалавриата**

по направлению подготовки
09.03.01 «Информатика и вычислительная техника»

с направленностью (профилем)
Электронно-вычислительные машины, комплексы, системы и сети

Форма обучения: очная, заочная

Идентификационный номер образовательной программы: 090301-02-22

Тула 2021

Разработчик(и) методических указаний

Берсенов Г.Б., доцент каф. ВТ, к.т.н.

(ФИО, должность, ученая степень, ученое звание)

(подпись)

Практическая работа № 1.

Разработка командных процедур (скриптов) на языке Bash в системе Linux

1. Цели работы

Целями практической работы являются:

- закрепить теоретические знания по видам и синтаксису команд оболочки ядра системы Linux;
- получить практические навыки разработки командных процедур (скриптов) на языке Bash.

2. Теоретические сведения

Все современные системы Unix (в том числе и Linux) поставляются с различными оболочками ядра, являющимися командными интерпретаторами. Каждый интерпретатор имеет свой командный язык и использует соответствующие ему командные процедуры (скрипты) для инициализации среды, в которой будут выполняться команды пользователя:

Командный интерпретатор	Скрипт инициализации
Bourne shell (sh)	.profile
C shell (csh)	.login и .cshrc
Korn shell (ksh)	.profile и .kshrc
Bourne-again shell (bash)	.profile и .bashrc

В операционной системе Linux по-умолчанию обычно используется оболочка *bash*. Когда оболочка запускается, она инициализируется, затем выводит на экран промптер – символ или строку приглашения к вводу. Пользователь может изменить промптер, например, сделать так, чтобы он содержал имя текущего каталога и автоматически изменялся при смене каталога. После того, как пользователь введет командную строку, оболочка (или ее потомок, которого она обычно создает для выполнения команды) извлекает первое слово командной строки и ищет подпрограмму в оболочке (если команда относится к внутренним командам оболочки) либо готовую программу (обычно утилиту, находящуюся в виде файла в текущем каталоге или в одном из системных каталогов).

У команд могут быть аргументы, которые передаются запускаемой программе в виде текстовых строк. Например, командная строка

```
cp src dest
```

запускает программу *cp*, которая делает копию файла *src* и называет эту копию именем *dest*.

Аргументами могут быть флаги (обычно со знаком тире), которые управляют работой команды. Например, команда

```
head -20 file
```

напечатает первые двадцать строк файла *file* (вместо принятых по умолчанию 10 строк), а команда

```
head 20 file
```

даст указание программе *head* вывести первые 10 строк файла 20, а затем первые 10 строк файла *file*. Большинство команд Linux-систем могут принимать несколько флагов и аргументов.

Чтобы упростить указание нескольких файлов в команде, оболочка принимает так называемые групповые символы. Например, символ звездочка означает все возможные текстовые строки, так что строка

```
ls *.c
```

даст указание программе *ls* вывести список всех файлов текущего каталога, имя которых оканчивается на *.c*. Если в каталоге находятся файлы *x.c*, *y.c* и *z.c*, то данная команда будет эквивалентна команде

```
ls x.c y.c z.c
```

Другим групповым символом является вопросительный знак, который заменяет один любой символ. Кроме того, в квадратных скобках можно указать множество символов, из которых программа должна будет выбрать один. Например, команда

```
ls [ame]*
```

выводит список файлов, имя которых начинается с символов «a», «m» или «e».

Оболочка не должна открывать терминал (клавиатуру и монитор), чтобы прочесть с него или сделать на него вывод. Вместо этого запускаемые программы автоматически получают доступ для чтения к файлу (точнее, к устройству, поскольку в Linux каждое устройство является одним из видов файла), называемому стандартным устройством ввода (stdin), а для записи – к файлу, называемому стандартным устройством вывода (stdout), и к файлу, называемому стандартным устройством для вывода сообщений об ошибках (stderr). По умолчанию всем этим трем устройствам соответствует терминал, т.е. чтение производится с клавиатуры, а запись – на монитор. Многие Linux-программы читают данные со стандартного устройства ввода и пишут на стандартное устройство вывода. Например, команда

```
sort
```

запускает программу-утилиту, находящуюся в файле *sort*, которая читает строки с терминала (пока пользователь не нажмет комбинацию клавиш CTRL-D, чтобы обозначить конец файла), а затем сортирует их в алфавитном порядке и выводит результат на экран.

Стандартные ввод и вывод можно перенаправить, что является очень полезным свойством. Для этого используются символы «<» и «>» соответственно. Например, команда

```
sort <in >out
```

заставляет программу *sort* взять в качестве входного файла *in* и направить вывод в файл *out*. Поскольку стандартный вывод сообщений об ошибках не был перенаправлен, то все сообщения об ошибках попадут на экран. Программа, которая считывает данные со стандартного устройства ввода, выполняет определенную обработку этих данных и записывает результат в поток стандартного вывода, называется фильтром.

Рассмотрим следующую командную строку, состоящую из трех отдельных команд:

```
sort <in >temp; head -30 <temp; rm temp
```

Сначала запускается программа *sort*, которая принимает данные из файла *in* и записывает результат в файл *temp*. Когда она завершает свою работу, оболочка запускает программу *head*, дав ей указание вывести первые 30 строк из файла *temp* на стандартное устройство вывода, которым по умолчанию является терминал. По завершению вывода временный файл *temp* удаляется.

Часто используются командные строки, в которых первая программа в командной строке формирует вывод, используемый второй программой в качестве входа. В приведенном выше примере для этого использовался временный файл *temp*. Однако система Linux (как и Unix) предоставляет для этого более простой способ. В командной строке

```
sort <in | head -30
```

используется вертикальная черта, называемая символом канала. Она означает, что вывод программы *sort* должен использоваться в качестве входа для программы *head*, что позволяет обойтись без создания, использования и удаления временного файла.

Набор команд, соединенных символом канала, называется конвейером. Он может содержать произвольное количество команд. Пример четырехкомпонентного конвейера показан в следующей строке:

```
grep ter *.t | sort | head -20 | tail -5 >foo
```

Здесь в стандартное устройство вывода записываются все строки, содержащие строку «ter» во всех файлах, оканчивающихся на *.t*, после чего они сортируются. Первые 20 строк выбираются

программой *head*, которая передает их программе *tail*, записывающей последние пять строк (из переданных двадцати) в файл *foo*.

Linux является универсальной многозадачной системой. Один пользователь может одновременно запустить несколько программ, каждую в виде отдельного процесса. Синтаксис оболочки для запуска фонового процесса состоит в использовании амперсанда в конце строки. Таким образом, строка

```
wc -l <file1 >file2 &
```

запустит программу *wc* для подсчета количества символов/слов/строк, которая сосчитает число строк (флаг -l) во входном файле *file1* и запишет результат в файл *file2*, но будет делать это в фоновом режиме. Как только команда будет введена пользователем, оболочка выведет символ приглашения к вводу и будет готова к обработке следующей команды.

Конвейеры также могут выполняться в фоновом режиме, например:

```
sort <file1 | head &
```

Можно одновременно запустить в фоновом режиме несколько конвейеров.

Список команд оболочки может быть помещен в файл, а затем можно будет запустить оболочку с этим файлом в качестве стандартного входа. В этом случае оболочка выполнит перечисленные в этом файле команды одну за другой – точно так же, как если бы эти команды вводились с клавиатуры. Файлы, содержащие команды оболочки, называются сценариями оболочки.

Сценарии оболочки могут присваивать значения переменным оболочки и затем считывать их. Они также могут иметь параметры и использовать конструкции *if*, *for*, *while* и *case*. Таким образом, сценарии оболочки представляют собой настоящие программы, написанные на языке оболочки.

Оболочки системы Linux позволяют компилировать и выполнять программы, написанные на различных языках программирования. В качестве примера рассмотрим программы *hello.c*, *add.c* и *add.cpp*, написанные на языках C и C++ соответственно:

```
// hello.c
#include <stdio.h>
using namespace std;
int main ()
{
    printf ("Hello, world!\n");
    return 0; }

// add.c
#include <stdio.h>
using namespace std;
int main ()
{
    int x, y;
    printf("Input 2 numburs:  ");
    scanf ("%d%d", &x, &y);
    printf("Summa = %d\n", x+y);
    return 0;
}

// add.cpp
#include <iostream>
using namespace std;
int main ()
{
    int x, y;
    cout<<"Input 2 numburs:  ";
    cin>>x>>y;
    cout<<"Summa = "<<x+y<<endl;
    return 0;
}
```

Для компиляции и выполнения программы *hello.c* в консольном окне (из командной строки) используются следующие команды:

```
cc hello.c
a.out
```

Здесь по команде *cc* создается исполнимый файл *a.out*, который затем запускается на выполнение. Если в переменной *PATH* среды интерпретатора отсутствует текущий каталог, в

команде запуска на выполнение необходимо указать полный или относительный путь к файлу a.out, например,

```
/home/virtualuser/lab2/a.out
```

или

```
./a.out.
```

Для добавления текущего каталога в список путей поиска файлов, находящийся в переменной PATH, достаточно ввести команду

```
PATH=$PATH:.
```

При создании многомодульных программ, т.е. программ, состоящих из нескольких исходных файлов, каждый из модулей отдельно компилируется в объектный модуль (файл с расширением .o), а затем выполняется компоновка объектных модулей в загрузочный модуль. В среде системы Linux используется флажок -c для указания компилятору (cc, gcc или g++) о том, что необходимо выполнить только компиляцию модуля. Например, для модуля add.cpp могут быть выполнены следующие команды

```
g++ -c add.cpp
g++ add.o
a.out
```

В команде cc или g++ можно с помощью флага -o указать имя файла, в который необходимо записать полученный в результате компиляции и компоновки загрузочный модуль программы, например

```
cc -o prog hello.c
prog
```

Интерпретатор команд позволяет записывать в одной строке несколько команд, например

```
cc hello.c; a.out
```

В этом примере исполнимый файл a.out будет запускаться даже в том случае, если компиляция завершится неудачно и файл a.out не будет создан либо запустится загрузочный модуль, полученный когда-то ранее. Для предотвращения запуска следующей команды при неуспешном завершении предыдущей эти команды должны быть соединены связкой &&, например

```
cc hello.c&&./a.out
```

Файл a.out автоматически создается компилятором как исполнимый (с флажком x). Если же мы создаем командную процедуру (скрипт) обычным текстовым редактором, то она не будет иметь флага x и для ее запуска необходимо использовать команду sh. Например, команда sh comf1 запускает выполнение скрипта comf1.

Аналогичный результат будет получен, если скрипт comf1 сделать исполнимым с помощью команды chmod, а затем запустить его выполнение без использования команды sh, например

```
chmod +x comf1
./comf1
```

Если comf1 будет содержать команды

```
file=hello.c
file1=""
str="Begin compile"
echo file;echo $file;echo $str
```

то при его выполнении будет получен следующий вывод в консольное окно:

```
file
hello.c
Begin compile
```

Внутри скрипта к фактическим параметрам обращаются, используя формальные позиционные параметры \$1, \$2, ..., \$9, а также специальные переменные \$0, \$#, \$@, \$*. Переменная \$0 содержит имя выполняемой shell-программы (скрипта), \$# - число аргументов в команде, а \$@ и \$* - список этих аргументов.

В качестве примера рассмотрим командную процедуру comf2 для трансляции и выполнения программ на языке C:

```
echo "Command = " $0
echo "List of arguments = " $@
echo "Number of arguments = " $#
for i in $@
do
    if cc $i
    then ./a.out
    fi
done
```

В скрипте comf2 используется условный оператор if и оператор цикла с перечислением for. Командной переменной i присваиваются поочередно имена из списка аргументов - имена файлов с исходными текстами программ. Если трансляция и компоновка завершаются успешно (команда cc дает нулевое значение), полученный загрузочный модуль считывается из файла a.out и выполняется. Например:

```
$ sh comf2 hello.c add.c
Command = comf2
List of arguments = hello.c add.c
Number of arguments = 2
cc hello.c
./a.out
Hello, world!
cc add.c
./a.out
Input 2 numburs: 2 2
Summa = 4
```

Отметим, что вместо

```
for i in $@
```

в скриптах обычно пишут

```
for i
```

поскольку в обоих случаях i поочередно принимает значения позиционных параметров.

Скрипт comf3 в отличие от comf2 компилирует программы, написанные как на C, так и на C++:

```
for i
do
    case $i in
        *.c)echo $i;cc $i;;
        *.cpp)echo $i;g++ $i;;
        *)echo "$i - bad file name";;
    esac
    a.out
done
```

Оператор case в зависимости от типа файла в списке аргументов выполняет команду cc или g++.

При компиляции программ (посредством команд cc или g++ с флажком -c) в текущем каталоге появляются файлы объектных модулей. Некоторые компиляторы не создают новые объектные модули, если не удалены старые одноименные объектные модули. Для их удаления можно использовать командную процедуру comf4:

```
set - *
for i
do
    if test -f $i
    then case $i in
        *.o)rm $i
        echo "Remove file $i" ;;
    esac
    fi
done
```

Встроенная команда `set` присваивает позиционным параметрам `$1`, `$2`, ... скрипта имена файлов текущего каталога. В каждом цикле командой `test` проверяется, является ли файл `$i` обычным файлом (не каталогом). Если команда `test` дает нулевое значение, то оператор `case` проверяет, содержит ли файл объектный модуль (тип файла - `.o`). Обнаруженный файл с объектным модулем уничтожается, а его имя выдается в консольное окно.

Значения командных переменных можно определить не только оператором присваивания, но и оператором чтения с клавиатуры `read`. При выполнении следующего скрипта (командной процедуры `comf5`) в каждом цикле выполняется программа, имя которой введено пользователем по команде `read`, или программа, находящаяся в файле `a.out`, если введено пустое имя (сразу введен символ `<Enter>`):

```
str2="Execute program and/or <Enter>?"
file1=""
while echo $str2; read file1 || exit
do
    if [ "$file1" = "" ]
    then echo a.out; a.out
    else $file1
fi
done
```

Команда `exit`, используемая для завершения скрипта и соответственно цикла `while`, выполняется, если команда `read` дает ненулевой код завершения. Обычно команда `read` дает ненулевой код завершения, если введен символ конца файла `<Ctrl-z>`. Команда `test` (вместо слова `test` здесь используются квадратные скобки) проверяет наличие имени (цепочки символов) в переменной `file1`.

3. Задание на работу

1. Ознакомиться с командным интерфейсом операционной системы Linux.
2. Переписать в личный каталог командные процедуры для компиляции, компоновки и выполнения программ. Выполнить отладку и проверку работы командных процедур.

4. Порядок выполнения работы

4.1. Загрузить операционную систему Linux гостевой виртуальной машины и после появления приглашения `login` войти в систему.

4.2. Запустить терминал и с помощью команды `pwd` вывести в консольное окно путь к начальному (домашнему) каталогу пользователя.

4.3. Запустить поочередно остальные интерпретаторы, используя команды `sh`, `csh`, `bash`, `ksh`. Для каждого интерпретатора командой `set` вывести значения параметров среды и найти командные процедуры (скрипты) инициализации, устанавливающие значения этих параметров при входе пользователя в систему (скрипты `.profile` или `.login`) и при запуске очередного интерпретатора (скрипты типа `.*shrc`). Проверить выполнение команд (`help`, `history` и др.), а затем командой `exit` завершить работу очередного интерпретатора.

4.4. Перейти в каталог `lab3` и ознакомиться с содержимым находящихся в нем текстовых файлов `hello.c`, `add.c` и `add.cpp`, содержащих программы, написанные на языках C и C++ соответственно. Посредством команд `gcc` и `g++` откомпилировать и выполнить эти программы.

4.5. Создать в каталоге `lab3` подкаталог `lab31`. Сделать каталог `lab31` текущим и выполнить копирование в него текстовых файлов `hello.c`, `add.c` и `add.cpp`.

4.6. Выполнить командную процедуру `comf1`, используя команду `sh`. Повторить выполнение командной процедуры `comf1`, предварительно сделав ее исполнимой посредством команды `chmod`.

4.7. Переписать из родительского каталога в текущий каталог командные процедуры `comf2`, `comf3`, `comf4`, `comf5`, `clg` и повторить для них пункт 5.6. Для проверки работы скриптов использовать копии файлов `hello.c`, `add.c` и `add.cpp`, находящихся в каталоге `lab3`.

5. Оформление отчета

Отчет должен содержать:

- тексты командных процедур;
- примеры выполнения скриптов.

Примечание: не надо включать в отчет результаты выполнения команды set для интерпретаторов, а результаты выполнения скриптов привести без использования отладочных флажков.

6 Контрольные вопросы

- 6.1. Какие значения имеют переменные среды PS1 и PS2 в каждом из командных интерпретаторов системы Linux?
- 6.2. Какое значение имела переменная PATH среды при выполнении лабораторной работы?
- 6.3. Какие формальные параметры имеет скрипт?
- 6.4. Укажите назначение различных флажков команд ss и gss.
- 6.5. Как работает в скриптах цикл с перечислением?
- 6.6. Укажите назначение различных флажков команды test?
- 6.7. Как завершить работу скрипта comf5?
- 6.8. Перечислите команды строкового редактора ed.
- 6.9. Как выполняется отладка командных скриптов?
- 6.10. Какое назначение скриптов инициализации оболочки системы Linux?

Библиографический список

1. Робачевский, А.М. Операционная система UNIX : учеб.пособие / А.М.Робачевский, С.А.Немнюгин, О.Л.Стесик. — 2-е изд., перераб. и доп. — СПб. : БХВ-Петербург, 2005. — 635с.
2. Магда, Ю.С. UNIX для студента / Ю.С.Магда. — СПб. : БХВ-Петербург, 2007. — 480с.
3. Рочкинд, М.Д. Программирование для UNIX / М.Д.Рочкинд; пер.с англ. под общ. ред. В.В.Вшивцева. — 2-е изд. — СПб. : БХВ-Петербург, 2005. — 704с.
4. Моли, Molay B. Unix/Linux: теория и практика программирования : пер.с англ. / Б.Моли. — М. : КУДИЦ-ОБРАЗ, 2004. — 576с.
5. Современные операционные системы / Э. С. Таненбаум, Х. Бос. — 4-е изд. — СПб. : Питер, 2017. — 1120 с.
6. 15. Таненбаум Э. С. Современные операционные системы — 3-е изд. — СПб. : Питер, 2012. — 1120 с.
7. Реймонд, Э.С. Искусство программирования для Unix / Э.С.Реймонд; пер.с англ. и ред. В.А.Швеца. — М. и др. : Вильямс, 2005. — 543с.
8. Стивенс У. UNIX: взаимодействие процессов. — СПб. : Питер, 2002. — 576 с.
9. Митчелл М., Оулдем Д., Самьюэл А. Программирование для Linux. Профессиональный подход. - М.: "Вильямс", 2002. - 288 с.
10. К.Кристиан. Введение в операционную систему UNIX. - М.: Финансы и статистика, 1985.
11. Р.Готье. Руководство по операционной системе UNIX. - М.: Финансы и статистика, 1986.
12. М.Банахан, Э.Раттер. Введение в операционную систему UNIX. - М.: Радио и связь, 1986.
13. П.Браун. Введение в операционную систему UNIX. - М.: Мир, 1987.