

Лабораторная работа №11

Дисциплина: Операционные системы

Андреев Владислав Владимирович

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	7
4	Библиография	18
5	Выводы	19

Список таблиц

Список иллюстраций

3.1	Создание файла	7
3.2	Скрипт №1	8
3.3	Проверка работы скрипта	8
3.4	Создание файла	8
3.5	Скрипт №2	9
3.6	Проверка работы скрипта	9
3.7	Скрипт №3	10
3.8	Проверка работы скрипта	11
3.9	Создание файла	11
3.10	Скрипт №4	12
3.11	Проверка работы скрипта	12

1 Цель работы

Цель данной лабораторной работы — Изучить основы программирования в оболочке ОС UNIX/Linux. Научиться писать небольшие командные файлы.

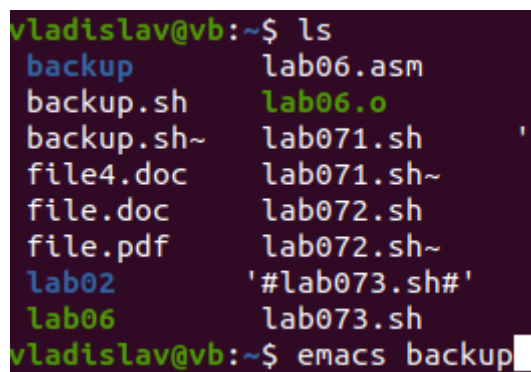
2 Задание

1. Сделать отчёт по лабораторной работе №11 в формате Markdown.
2. Изучить основы программирования в оболочке ОС UNIX/Linux.

3 Выполнение лабораторной работы

1).

Создаем файл, в котором будем писать первый скрипт, и открыла его в редакторе emacs, (команды «touch backup.sh» и «emacs backup.sh»).



```
vladislav@vb:~$ ls
backup      lab06.asm
backup.sh   lab06.o
backup.sh~  lab071.sh
file4.doc   lab071.sh~
file.doc    lab072.sh
file.pdf    lab072.sh~
lab02       '#lab073.sh#'
lab06       lab073.sh
vladislav@vb:~$ emacs backup
```

Рис. 3.1: Создание файла

Написал скрипт, который при запуске будет делать резервную копию самого себя (то есть файл, в котором содержится его исходный код) в другую директорию backup в вашем домашнем каталоге. При этом файл должен архивироваться одним из архиваторов на выбор zip, bzip2 или tar. При написании скрипта использовала архиватор bzip2.

```
#!/bin/bash

name='backup.sh'
mkdir ~/backup
bzip2 -k ${name}
mv ${name}.bz2 ~/backup/
echo "Done"
```

Рис. 3.2: Скрипт №1

Проверил работу скрипта (команда «`bash ./backup.sh`»). Проверил, появился ли каталог `backup/`, перейдя в него (команда «`cd backup/`»), посмотрел его содержимое (команда «`ls`») и просмотрела содержимое архива (команда «`bunzip2 -c backup.sh.bz2`»). Скрипт работает корректно.

```
vladislav@vb:~$ cd backup
vladislav@vb:~/backup$ ls
backup.sh.bz2
vladislav@vb:~/backup$
```

Рис. 3.3: Проверка работы скрипта

2). Создал файл, в котором буду писать второй скрипт, и открыл его в редакторе `emacs`, (команды «`touch script2.sh`» и «`emacs script2.sh`»).

```
vladislav@vb:~$ ls
backup      lab06.asm      lab074.sh
backup.sh   lab06.o        lab074.sh~
backup.sh~  lab071.sh     '#lab07.sh#'
file4.doc   lab071.sh~    lab07.sh
file.doc    lab072.sh     script2.sh
file.pdf    lab072.sh~    script2.sh~
```

Рис. 3.4: Создание файла

Написал пример командного файла, обрабатывающего любое произвольное число аргументов командной строки, в том числе превышающее десять. Например, скрипт может последовательно распечатывать значения всех переданных аргументов .


```
#!/bin/bash
echo "Args"
for arg in $@
do echo $arg
done
```

Рис. 3.5: Скрипт №2

Проверила работу написанного скрипта (команды «`bash ./script2.sh 0 1 2 3 4`» и «`./script2.sh 0 1 2 3 45 6 7 8 9 10 11`»). Вводил аргументы количество которых меньше 10 и больше 10. Скрипт работает корректно.

```
vladislav@vb:~$ bash ./script2.sh 0 1 2 3 4
Args
0
1
2
3
4
vladislav@vb:~$ bash ./script2.sh 0 1 2 3 5 6 7 8 9 10 11
Args
0
1
2
3
5
6
7
8
9
10
11
```

Рис. 3.6: Проверка работы скрипта

3). Создал файл, в котором буду писать третий скрипт, и открыл его в редакторе

```
vladislav@vb:~$ ls
backup      lab06.asm      lab06.o
backup.sh   lab071.sh     '#lab071.sh~'
file4.doc   lab071.sh~    lab072.sh
file.doc    lab072.sh     scr
file.pdf    lab072.sh~    scr
lab02       '#lab073.sh#' scr
lab06       lab073.sh     scr
```

emacs (команды «`touch script3.sh`» и «`emacs script3.sh`»).

Написал командный файл – аналог команды `ls` (без использования самой этой

команды и команды `dir`). Он должен выдавать информацию о нужном каталоге и выводить информацию о возможностях доступа к файлам этого каталога.

```
n= "$1"
for i in ${n}/*
do
    echo "$i"
    if test -f $i
    then echo "File"
    fi
    if test -d $i
    then echo "dir"
    fi
    if test -r $i
    then echo "Access reading"
    fi
    if test -w $i
    then echo "Access writing"
    fi
    if test -x $i
    then echo "Access done"
    fi
done
```

Рис. 3.7: Скрипт №3

Далее проверил работу скрипта (команда «./script3.sh~»). Скрипт работает корректно.

```

vladislav@vb:~$ bash ./script3.sh
./script3.sh: строка 2: : команда не найдена
/bin
dir
Access reading
Access done
/boot
dir
Access reading
Access done
/cdrom
dir
Access reading
Access done
/dev
dir
Access reading
Access done
/etc
dir
Access reading
Access done
/home
dir
Access reading
Access done
/lib
dir
Access reading

```

Рис. 3.8: Проверка работы скрипта

4). Для четвертого скрипта создал файл (команда «touch script4.sh») и открыла его в редакторе emacs (команда «emacs script4.sh»).

```

vladislav@vb:~$ ls
backup      lab06.asm      lab074.sh      script4.sh
backup.sh   lab06.o        lab074.sh~     script4.sh~

```

Рис. 3.9: Создание файла

Написал командный файл, который получает в качестве аргумента командной строки формат файла (.txt, .doc, .jpg, .pdf и т.д.) и вычисляет количество таких файлов в указанной директории. Путь к директории также передаётся в виде

аргумента командной строки.

```
#!/bin/bash
n="$1"
shift
for a in $@
do
    sum=0
    for i in ${n}/*.${a}
    do
        if test -f "$i"
        then let sum=sum+1
        fi
    done
    echo "$sum files in dir $n with extension $a"
done
```

Рис. 3.10: Скрипт №4

Проверил работу написанного скрипта (команда «bash ./script4.sh ~ pdf sh txt doc»), а также создав дополнительные файлы с разными расширениями (команда «touch file.pdf file1.doc file4.doc»). Скрипт работает корректно.

```
vladislav@vb:~$ bash ./script4.sh ~ pdf doc txt sh
1 files in dir /home/vladislav with extension pdf
2 files in dir /home/vladislav with extension doc
0 files in dir /home/vladislav with extension txt
9 files in dir /home/vladislav with extension sh
vladislav@vb:~$
```

Рис. 3.11: Проверка работы скрипта

Ответы на контрольные вопросы:

1). Командный процессор (командная оболочка, интерпретатор команд shell) – это программа, позволяющая пользователю взаимодействовать с операционной системой компьютера. В операционных системах типа UNIX/Linux наиболее часто используются следующие реализации командных оболочек: 1. оболочка Борна (Bourne shell или sh) – стандартная командная оболочка UNIX/Linux, содержащая базовый, но при этом полный набор функций; 2. C-оболочка (или csh) – надстройка на оболочке Борна, использующая подобный синтаксис команд с возмож-

ностью сохранения истории выполнения команд; 3. оболочка Корна (или ksh) – напоминает оболочку C, но операторы управления программой совместимы с операторами оболочки Борна; 4. BASH – сокращение от BourneAgainShell (опять оболочка Борна), в основе своей совмещает свойства оболочек C и Корна (разработка компании FreeSoftwareFoundation).

2). POSIX (Portable Operating System Interface for Computer Environments) – набор стандартов описания интерфейсов взаимодействия операционной системы и прикладных программ. Стандарты POSIX разработаны комитетом IEEE (Institute of Electrical and Electronics Engineers) для обеспечения совместимости различных UNIX/Linux подобных операционных систем и переносимости прикладных программ на уровне исходного кода. POSIX – совместимые оболочки разработаны на базе оболочки Корна.

3). Командный процессор bash обеспечивает возможность использования переменных типа строка символов. Имена переменных могут быть выбраны пользователем. Пользователь имеет возможность присвоить переменной значение некоторой строки символов. Например, команда «mark=/usr/andy/bin» присваивает значение строки символов /usr/andy/bin переменной mark типа строка символов. Значение, присвоенное некоторой переменной, может быть впоследствии использовано. Для этого в соответствующем месте командной строки должно быть употреблено имя этой переменной, которому предшествует метасимвол ., «mv file{mark}» переместит файл afile из текущего каталога в каталог с абсолютным полным именем /usr/andy/bin. Оболочка bash позволяет работать с массивами. Для создания массива используется команда setc флагом -A. За флагом следует имя переменной, а затем список значений, разделённых пробелами. Например, «set -A states Delaware Michigan “New Jersey”». Далее можно сделать добавление в массив, например, states[49]=Alaska. Индексация массивов начинается с нулевого элемента.

4). Оболочка bash поддерживает встроенные арифметические функции. Команда let является показателем того, что последующие аргументы представляют

собой выражение, подлежащее вычислению. Простейшее выражение – это единственный терм (term), обычно целочисленный. Команда let берет два операнда и присваивает их переменной. Команда read позволяет читать значения переменных со стандартного ввода: «echo “Please enter Month and Day of Birth ?”» «read mon day trash». В переменные mon и day будут считаны соответствующие значения, введенные с клавиатуры, а переменная trash нужна для того, чтобы отобрать всю избыточно введенную информацию и игнорировать её.

5). В языке программирования bash можно применять такие арифметические операции как сложение (+), вычитание (-), умножение (*), целочисленное деление (/) и целочисленный остаток от деления (%).

6). В (()) можно записывать условия оболочки bash, а также внутри двойных скобок можно вычислять арифметические выражения и возвращать результат.

7). Стандартные переменные: 1. PATH: значением данной переменной является список каталогов, в которых командный процессор осуществляет поиск программы или команды, указанной в командной строке, в том случае, если указанное имя программы или команды не содержит ни одного символа /. Если имя команды содержит хотя бы один символ /, то последовательность поиска, предписываемая значением переменной PATH, нарушается. В этом случае в зависимости от того, является имя команды абсолютным или относительным, поиск начинается соответственно от корневого или текущего каталога.

2. PS1 и PS2: эти переменные предназначены для отображения промптера командного процессора. PS1 – это промптер командного процессора, по умолчанию его значение равно символу \$ или #. Если какая-то интерактивная программа, запущенная командным процессором, требует ввода, то используется промптер PS2. Он по умолчанию имеет значение символа >.

3. HOME: имя домашнего каталога пользователя. Если команда cd вводится без аргументов, то происходит переход в каталог, указанный в этой переменной.

4. IFS: последовательность символов, являющихся разделителями в команд-

ной строке, например, пробел, табуляция и перевод строки (newline).

5. MAIL:командный процессор каждый раз перед выводом на экран промптера проверяет содержимое файла, имя которого указано в этой переменной, и если содержимое этого файла изменилось с момента последнего ввода из него, то перед тем как вывести на терминал промптер, командный процессор выводит на терминал сообщение Youhavemail(у Вас есть почта).

6. TERM: тип используемого терминала.

7. LOGNAME: содержит регистрационное имя пользователя, которое устанавливается автоматически при входе в систему.

8). Такие символы, как ' < > * ? | " &, являются метасимволами и имеют для командного процессора специальный смысл.

9). Снятие специального смысла с метасимвола называется экранированием мета символа. Экранирование может быть осуществлено с помощью предшествующего мета символу символа , который, в свою очередь, является мета символом. Для экранирования группы метасимволов нужно заключить её в одинарные кавычки. Строка, заключённая в двойные кавычки, экранирует все метасимволы, кроме \$, ' , , ". Например, `-echo*` выведет на экран символ , `-echoab'|'cd` выведет на экран строку `ab|*cd`.

10). Последовательность команд может быть помещена в текстовый файл. Такой файл называется командным. Далее этот файл можно выполнить по команде: `«bash командный_файл [аргументы]»`. Чтобы не вводить каждый раз последовательности символов `bash`, необходимо изменить код защиты этого командного файла, обеспечив доступ к этому файлу по выполнению. Это может быть сделано с помощью команды `«chmod +x имя_файла»`. Теперь можно вызывать свой командный файл на выполнение, просто вводя его имя с терминала так, как будто он является выполняемой программой. Командный процессор распознает, что в Вашем файле на самом деле хранится не выполняемая программа, а про-

грамма, написанная на языке программирования оболочки, и осуществить её интерпретацию.

11). Группу команд можно объединить в функцию. Для этого существует ключевое слово `function`, после которого следует имя функции и список команд, заключённых в фигурные скобки. Удалить функцию можно с помощью команды `unsetc`флагом `-f`.

12). Чтобы выяснить, является ли файл каталогом или обычным файлом, необходимо воспользоваться командами «`test -f [путь до файла]`» (для проверки, является ли обычным файлом) и «`test -d[путь до файла]`» (для проверки, является ли каталогом).

13). Команду «`set`» можно использовать для вывода списка переменных окружения. В системах Ubuntu и Debian команда «`set`» также выведет список функций командной оболочки после списка переменных командной оболочки. Поэтому для ознакомления со всеми элементами списка переменных окружения при работе с данными системами рекомендуется использовать команду «`set| more`». Команда «`typeset`» предназначена для наложения ограничений на переменные. Команду «`unset`» следует использовать для удаления переменной из окружения командной оболочки.

14). При вызове командного файла на выполнение параметры ему могут быть переданы точно таким же образом, как и выполняемой программе. С точки зрения командного файла эти параметры являются позиционными. Символ `$` является метасимволом командного процессора. Он используется, в частности, для ссылки на параметры, точнее, для получения их значений в командном файле. В командный файл можно передать до девяти параметров. При использовании где-либо в командном файле комбинации символов `$i`, где $0 < i < 10$, вместо неё будет осуществлена подстановка значения параметра с порядковым номером i , т.е. аргумента командного файла с порядковым номером i . Использование комбинации символов `$0` приводит к подстановке вместо неё имени данного командного файла.

15). Специальные переменные: 1. `$*` –отображается вся командная строка или параметры оболочки; 2. `$?` –код завершения последней выполненной команды; 3. `$$` –уникальный идентификатор процесса, в рамках которого выполняется командный процессор; 4. `$!` –номер процесса, в рамках которого выполняется последняя вызванная на выполнение в командном режиме команда; 5. `--` –значение флагов командного процессора; 6. `${#}` –*возвращает целое число –количество слов, которые были результатом \$*; 7. `${#name}` –возвращает целое значение длины строки в переменной `name`; 8. `${name[n]}` –обращение к `n`-му элементу массива; 9. `${name[*]}` –перечисляет все элементы массива, разделённые пробелом; 10. `${name[@]}` –то же самое, но позволяет учитывать символы пробелы в самих переменных; 11. `${name:-value}` –если значение переменной `name` не определено, то оно будет заменено на указанное `value`; 12. `${name:value}` –проверяется факт существования переменной; 13. `${name=value}` –если `name` не определено, то ему присваивается значение `value`; 14. `${name?value}` –останавливает выполнение, если имя переменной не определено, и выводит `value` как сообщение об ошибке; 15. `${name+value}` –это выражение работает противоположно `${name-value}`. Если переменная определена, то подставляется `value`; 16. `${name#pattern}` –представляет значение переменной `name` с удалённым самым коротким левым образцом (`pattern`); 17. `${#name[*]}` и `${#name[@]}` –эти выражения возвращают количество элементов в массиве `name`.

4 Библиография

1. Программное обеспечение GNU/Linux. Лекция 3. FHS и процессы (Г. Курячий, МГУ);
2. Программное обеспечение GNU/Linux. Лекция 4. Права доступа (Е. Алёхова, МГУ);
3. Электронный ресурс: [https://ru.wikibooks.org/wiki %D0%92%D0%B2%D0%B5%D0%B4%D0%](https://ru.wikibooks.org/wiki/%D0%92%D0%B2%D0%B5%D0%B4%D0%)
4. Электронный ресурс: <http://fedoseev.net/materials/courses/admin/ch02.html>

5 Выводы

В ходе выполнения данной лабораторной работы я изучил основы программирования в оболочке ОС UNIX/Linux и научился писать небольшие командные файлы.