

Лабораторная работа №10

**Программирование в командном процессоре ОС UNIX. Командные
файлы**

Желдакова Виктория Алексеевна

Содержание

1	Цель работы	5
2	Теоретическое введение	6
3	Выполнение лабораторной работы	7
4	Выводы	11
5	Контрольные вопросы	12

Список иллюстраций

3.1	Листинг первого скрипта	7
3.2	Результат выполнения первого скрипта	7
3.3	Листинг второго скрипта	8
3.4	Результат выполнения второго скрипта	8
3.5	Листинг третьего скрипта	9
3.6	Результат выполнения третьего скрипта	9
3.7	Листинг четвёртого скрипта	9
3.8	Результат выполнения четвёртого скрипта	10

Список таблиц

1 Цель работы

Изучить основы программирования в оболочке ОС UNIX/Linux. Научиться писать небольшие командные файлы.

2 Теоретическое введение

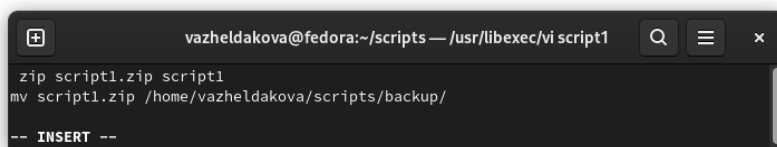
Bash - это оболочка Unix и командный язык, написанный Брайаном Фоксом для проекта GNU в качестве замены свободного программного обеспечения для оболочки Bourne. Впервые выпущенный в 1989 году, он использовался в качестве оболочки входа по умолчанию для большинства дистрибутивов Linux. Bash была одной из первых программ, портированных Линусом Торвальдсом на Linux, наряду с GCC. Версия также доступна для Windows 10 через подсистему Windows для Linux. Это также пользовательская оболочка по умолчанию в Solaris 11. Bash также была оболочкой по умолчанию во всех версиях Apple macOS до выхода в 2019 году macOS Catalina, которая изменила оболочку по умолчанию на zsh, хотя Bash остается доступной в качестве альтернативной оболочки.

Bash - это командный процессор, который обычно работает в текстовом окне, где пользователь вводит команды, вызывающие действия. Bash также может читать и выполнять команды из файла, называемого сценарием оболочки. Как и большинство Unix-оболочек, он поддерживает глобализацию имен файлов (сопоставление подстановочных знаков), конвейер, документы here, подстановку команд, переменные и структуры управления для тестирования условий и итерации. Ключевые слова, синтаксис, динамически изменяемые переменные и другие основные функции языка скопированы из sh. Другие функции, например, история, копируются из csh и ksh. Bash - это POSIX-совместимая оболочка, но с рядом расширений.

Название оболочки - это аббревиатура от Bourne Again Shell, каламбур на название оболочки Bourne, которую она заменяет, и понятие “рождение свыше”.

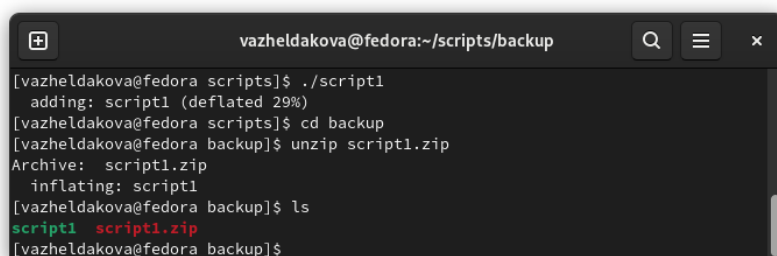
3 Выполнение лабораторной работы

Для того, чтобы написать скрипт, который при запуске будет делать резервную копию самого себя в другую директорию backup в домашнем каталоге, мы открыли в редакторе vi новый файл. Первым делом мы использовали команду zip для того, чтобы архивировать наш файл, а затем переместили полученный архив в заранее созданную директорию backup с помощью команды mv (рис. 3.1). Сохранили скрипт, дали ему права на выполнение и запустили (рис. 3.2).

A terminal window titled 'vazheldakova@fedora:~/scripts — /usr/libexec/vi script1'. The content of the script is displayed as follows:

```
zip script1.zip script1
mv script1.zip /home/vazheldakova/scripts/backup/
-- INSERT --
```

Рис. 3.1: Листинг первого скрипта

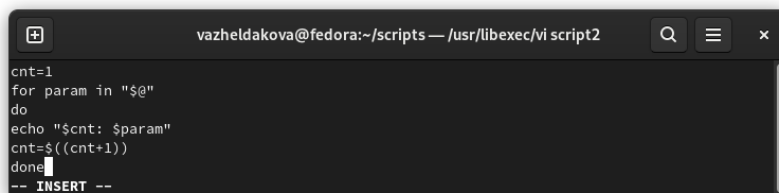
A terminal window titled 'vazheldakova@fedora:~/scripts/backup'. The execution of the script is shown with the following output:

```
[vazheldakova@fedora scripts]$ ./script1
adding: script1 (deflated 29%)
[vazheldakova@fedora scripts]$ cd backup
[vazheldakova@fedora backup]$ unzip script1.zip
Archive:  script1.zip
  inflating: script1
[vazheldakova@fedora backup]$ ls
script1  script1.zip
[vazheldakova@fedora backup]$
```

Рис. 3.2: Результат выполнения первого скрипта

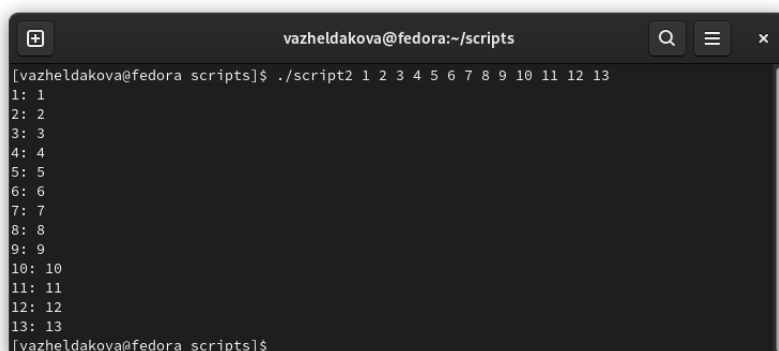
Перешли к созданию второго скрипта, который распечатывает значения всех переданных аргументов. Сначала создали переменную cnt равную 1 для отсчёта введённых значений. Оформили цикл for, который проходит по неопределённому

числу переданных аргументов, в каждой итерации распечатывает номер введённого значения и само значение и увеличивает счётчик на единицу (рис. 3.3 и рис. 3.4).



```
cnt=1
for param in "$@"
do
echo "$cnt: $param"
cnt=$((cnt+1))
done
-- INSERT --
```

Рис. 3.3: Листинг второго скрипта



```
[vazheldakova@fedora scripts]$ ./script2 1 2 3 4 5 6 7 8 9 10 11 12 13
1: 1
2: 2
3: 3
4: 4
5: 5
6: 6
7: 7
8: 8
9: 9
10: 10
11: 11
12: 12
13: 13
[vazheldakova@fedora scripts]$
```

Рис. 3.4: Результат выполнения второго скрипта

Перешли к созданию третьего скрипта - аналога команды `ls`, для начала создали переменную `path`, принимающую значение переданного аргумента. Создали цикл `for`, проходящийся по всем файлам в каталоге `path`, оформили `if-else` конструкцию для проверки типа файла (директория/файл). В случае, если у файла были права на чтение или запись, то выводили соответствующую информацию (рис. 3.5 и рис. 3.6).


```

vazheldakova@fedora:~/scripts — /usr/libexec/vi script3
path=$1
for A in "$1"*
do if test -d $A
then echo $A: is a directory
else echo -n $A: "is a file and "
if test -w $A
then echo writeable
elif test -r $A
then echo readable
else echo neither readable not writeable
fi
fi
done
-- INSERT --

```

Рис. 3.5: Листинг третьего скрипта

```

vazheldakova@fedora:~/scripts
[vazheldakova@fedora scripts]$ ./script3
backup: is a directory
script1: is a file and writeable
script2: is a file and writeable
script3: is a file and writeable
script4: is a file and writeable
[vazheldakova@fedora scripts]$

```

Рис. 3.6: Результат выполнения третьего скрипта

Перешли к созданию четвертого скрипта, который выводит количество файлов заданного формата в указанной директории. Первым делом вывели надписи с просьбой ввести формат и директорию, записали введённые значения в переменные `form` и `dir`. Используя команду `find`, начали поиск с максимальной глубиной 1 файла оканчивающегося на `.form` (рис. 3.7 и рис. 3.8).

```

vazheldakova@fedora:~/scripts — /usr/libexec/vi script4
echo "Enter directory: "
read dir
echo "Enter format: "
read form
find $dir -maxdepth 1 -name ".*$form" -type f | wc -l
"script4" 5L, 120B

```

Рис. 3.7: Листинг четвёртого скрипта

```
vazheldakova@fedora:~/scripts
[vazheldakova@fedora scripts]$ ./script4
Enter directory:
/home/vazheldakova/
Enter format:
txt
3
[vazheldakova@fedora scripts]$ ls /home/vazheldakova/
1.sh      4.sh      heemv.sh~  lab07.sh~  Видео      Музыка
2.sh      bin       hu         scripts    Документы  Общедоступные
'#3.sh#'  file.txt  '#lab07.sh#' test.txt    Загрузки   'Рабочий стол'
3.sh      fil.txt   lab07.sh  work       Изображения  Шаблоны
[vazheldakova@fedora scripts]$
```

Рис. 3.8: Результат выполнения четвёртого скрипта

4 Выводы

Изучили основы программирования в оболочке ОС UNIX/Linux. Научились писать небольшие командные файлы.

5 Контрольные вопросы

1. Объясните понятие командной оболочки. Приведите примеры командных оболочек. Чем они отличаются?

Командные процессоры или оболочки - это программы, позволяющие пользователю взаимодействовать с компьютером. Их можно рассматривать как настоящие интерпретируемые языки, которые воспринимают команды пользователя и обрабатывают их. Поэтому командные процессоры также называют интерпретаторами команд. На языках оболочек можно писать программы и выполнять их подобно любым другим программам. UNIX обладает большим количеством оболочек.

Наиболее популярными являются следующие четыре оболочки:

- оболочка Борна (Bourne) - первоначальная командная оболочка UNIX: базовый, но полный набор функций;
- C-оболочка - добавка университета Беркли к коллекции оболочек: она надстраивается над оболочкой Борна, используя C-подобный синтаксис команд, и сохраняет историю выполненных команд;
- оболочка Корна - напоминает оболочку C, но операторы управления программой совместимы с операторами оболочки Борна;
- BASH - сокращение от Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек C и Корна (разработка компании Free Software Foundation).

2. Что такое POSIX?

POSIX (англ. portable operating system interface for Unix — переносимый интерфейс операционных систем Unix) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой, библиотеку языка C и набор приложений и их интерфейсов. Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода, но может быть использован и для не-Unix систем.

3. Как определяются переменные и массивы в языке программирования bash?

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 `$`

Для создания массива используется команда `set` с 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. Каково назначение операторов `let` и `read`?

Команда `let` является показателем того, что последующие аргументы представляют собой выражение, подлежащее вычислению.

Команда `read` позволяет читать значения переменных со стандартного ввода.

5. Какие арифметические операции можно применять в языке программирования bash?

Оператор Синтаксис Результат

! *expr* Если *expr* равно 0, то возвращает 1; иначе 0

!= *expr1* != *expr2* Если *expr1* не равно *expr2*, то возвращает 1; иначе 0

% *expr1* % *expr2* Возвращает остаток от деления *expr1* на *expr2*

%= *var* %= *expr* Присваивает остаток от деления *var* на *expr* переменной *var*

& *expr1* & *expr2* Возвращает побитовое AND выражений *expr1* и *expr2*

&& *expr1* && *expr2* Если и *expr1* и *expr2* не равны нулю, то возвращает 1; иначе 0

&= *var* &= *expr* Присваивает переменной *var* побитовое AND *var* и *expr*

() *expr1* *expr2* Умножает *expr1* на *expr2*

= *var* = *expr* Умножает *expr* на значение переменной *var* и присваивает результат переменной *var*

(+) *expr1* + *expr2* Складывает *expr1* и *expr2*

+= *var* += *expr* Складывает *expr* со значением переменной *var* и результат присваивает переменной *var*

(-) -*expr* Операция отрицания *expr* (унарный минус)

(-) *expr1* - *expr2* Вычитает *expr2* из *expr1*

-- *var* -= *expr* Вычитает *expr* из значения переменной *var* и присваивает результат переменной *var*

/ *expr* / *expr2* Делит *expr1* на *expr2*

/= *var* /= *expr* Делит значение переменной *var* на *expr* и присваивает результат переменной *var*

(<) *expr1* < *expr2* Если *expr1* меньше, чем *expr2*, то возвращает 1, иначе возвращает 0

(«) *expr1* « *expr2* Сдвигает *expr1* влево на *expr2* бит

(«=) *var* «= *expr* Побитовый сдвиг влево значения переменной *var* на *expr*

(<=) *expr1* <= *expr2* Если *expr1* меньше или равно *expr2*, то возвращает 1; иначе возвращает 0

(=) *var* = *expr* Присваивает значение *expr* переменной *var*

(==) *expr1* == *expr2* Если *expr1* равно *expr2*, то возвращает 1; иначе возвращает 0

(>) $\text{exp1} > \text{exp2}$ 1, если exp1 больше, чем exp2 ; иначе 0

(>=) $\text{exp1} \geq \text{exp2}$ 1, если exp1 больше или равно exp2 ; иначе 0

(») $\text{exp} \gg \text{exp2}$ Сдвигает exp1 вправо на exp2 бит

(»=) $\text{var} \gg \text{exp}$ Побитовый сдвиг вправо значения переменной var на exp

$\wedge \text{exp1} \wedge \text{exp2}$ Исключающее OR выражений exp1 и exp2

$\wedge = \text{var} \wedge = \text{exp}$ Присваивает переменной var побитовое XOR var и exp

$\text{exp1} | \text{exp2}$ Побитовое OR выражений exp1 и exp2

$| = \text{var} | = \text{exp}$ Присваивает переменной var результат операции XOR var и exp

|| exp1 || exp2 1, если или exp1 или exp2 являются ненулевыми значениями; иначе 0

$\sim \text{exp}$ Побитовое дополнение до exp

6. Что означает операция (())?

Для облегчения программирования можно записывать условия оболочки `bash` в двойные скобки — (()).

7. Какие стандартные имена переменных Вам известны?

`PATH`, `PS1`, `PS2`, `HOME`, `IFS`, `MAIL`, `TERM`, `LOGNAME`.

8. Что такое метасимволы?

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

- `-` — соответствует произвольной, в том числе и пустой строке;
- `?` — соответствует любому одинарному символу;
- `[c1-c1]` — соответствует любому символу, лексикографически находящемуся между символами `c1` и `c2`.

9. Как экранировать метасимволы?

Экранирование может быть осуществлено с помощью предшествующего метасимволу символа , который, в свою очередь, является метасимволом.

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 *,
- `echo ab'|cd` выведет на экран строку `ab|cd`.

10. Как создавать и запускать командные файлы?

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 программа, написанная на языке программирования оболочки, и осуществит ее интерпретацию.

11. Как определяются функции в языке программирования bash?

Группу команд можно объединить в функцию. Для этого существует ключевое слово `function`, после которого следует имя функции и список команд, заключенных в фигурные скобки.

12. Каким образом можно выяснить, является файл каталогом или обычным файлом?

`ls -lrt` Если есть `d`, то файл является каталогом.

13. Каково назначение команд set, typeset и unset?

Команда set используется с флагом -A. За флагом следует имя переменной, а затем список значений, разделенных пробелом. Например, set -A states Delaware Michigan "New Jersey". Далее можно сделать добавление в массив, например, states[49]=Alaska. Индексация массивов начинается с нулевого элемента. В командном процессоре Си имеется еще несколько стандартных переменных. Значение всех переменных можно просмотреть с помощью команды set. Наиболее распространенным является сокращение, избавляющееся от слова let в программах оболочек. Если объявить переменные целыми значениями, любое присвоение автоматически трактуется как арифметическое.

typeset -i используется для объявления и присвоения переменной, и при последующем использовании она становится целой. Или можете использовать ключевое слово integer (псевдоним для typeset -l) и объявлять переменные целыми. Таким образом, выражения типа $x=y+z$ воспринимаются как арифметические. Группу команд можно объединить в функцию. Для этого существует ключевое слово function, после которого следует имя функции и список команд, заключенных в фигурные скобки. Удалить функцию можно с помощью команды unset с флагом -f.

14. Как передаются параметры в командные файлы?

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. Рассмотрим это на примере. Пусть к командному файлу `where` имеется доступ по выполнению и этот командный файл содержит следующий конвейер: `who | grep $1`. Если Вы введёте с терминала команду `where andy`, то в случае, если пользователь, зарегистрированный в ОС UNIX под именем `andy`, в данный момент работает в ОС UNIX, то на терминал будет выведена строка, содержащая номер терминала, используемого указанным пользователем. Если же в данный момент этот пользователь не работает в ОС UNIX, то на терминал ничего не будет выведено. Команда `grep` производит контекстный поиск в тексте, поступающем со стандартного ввода, для нахождения в этом тексте строк, содержащих последовательности символов, переданные ей в качестве аргументов, и выводит результаты своей работы на стандартный вывод. В примере команда `grep` используется как фильтр, обеспечивающий ввод со стандартного ввода и вывод всех строк, содержащих последовательность символов `andy`, на стандартный вывод. В ходе интерпретации файла командным процессором вместо комбинации символов `$1` осуществляется подстановка значения первого и единственного параметра `andy`.

15. Назовите специальные переменные языка `bash` и их назначение.

- `$*` — отображается вся командная строка или параметры оболочки;
- `$?` — код завершения последней выполненной команды;
- `$$` — уникальный идентификатор процесса, в рамках которого выполняется командный процессор;
- `$_` — номер процесса, в рамках которого выполняется последняя вызванная на выполнение в командном режиме команда;
- `$-` — значение флагов командного процессора;
- `${#}` — *возвращает целое число — количество слов, которые были результатом* `$`;
- `${#name}` — возвращает целое значение длины строки в переменной `name`;
- `${name[n]}` — обращение к `n`-му элементу массива;

- `${name[*]}` — перечисляет все элементы массива, разделённые пробелом;
- `${name[@]}` — то же самое, но позволяет учитывать символы пробелы в самих переменных;
- `${name:-value}` — если значение переменной `name` не определено, то оно будет заменено на указанное `value`;
- `${name:value}` — проверяется факт существования переменной;
- `${name=value}` — если `name` не определено, то ему присваивается значение `value`;
- `${name?value}` — останавливает выполнение, если имя переменной не определено, и выводит `value` как сообщение об ошибке;
- `${name+value}` — это выражение работает противоположно `${name-value}`. Если переменная определена, то подставляется `value`;
- `${name#pattern}` — представляет значение переменной `name` с удалённым самым коротким левым образцом (`pattern`);
- `${#name[*]}` и `${#name[@]}` — эти выражения возвращают количество элементов в массиве `name`.