

Лабораторная работа №10

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

Тулеуов Мадн

Содержание

1	Цель работы	5
2	Задачи	6
3	Ход работы	7
4	Вывод	12
5	Контрольные вопросы.	13

Список таблиц

Список иллюстраций

3.1	Код 1 скрипта	7
3.2	Работа скрипта	7
3.3	Проверка работы скрипта	8
3.4	Код 2 скрипта	8
3.5	Работа скрипта	9
3.6	Код 3 скрипта	9
3.7	Проверка 3 скрипта	10
3.8	Код 4 скрипта	10
3.9	Проверка	11

1 Цель работы

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

2 Задачи

1. Написать скрипт, который при запуске будет делать резервную копию самого себя (то есть файла, 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. Способ использования команд архивации необходимо узнать, изучив справку.
2. Написать пример командного файла, обрабатывающего любое произвольное число аргументов командной строки, в том числе превышающее десять. Например, скрипт может последовательно распечатывать значения всех переданных аргументов.
3. Написать командный файл — аналог команды ls (без использования самой этой команды и команды dir). Требуется, чтобы он выдавал информацию о нужном каталоге и выводил информацию о возможностях доступа к файлам этого каталога.
4. Написать 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.

3 Ход работы

1. Открыл в emacs файл *script1* и написал код, который создает архив, содержащий сам файл, и перемещает его в папку *backup*. (рис. 3.1)

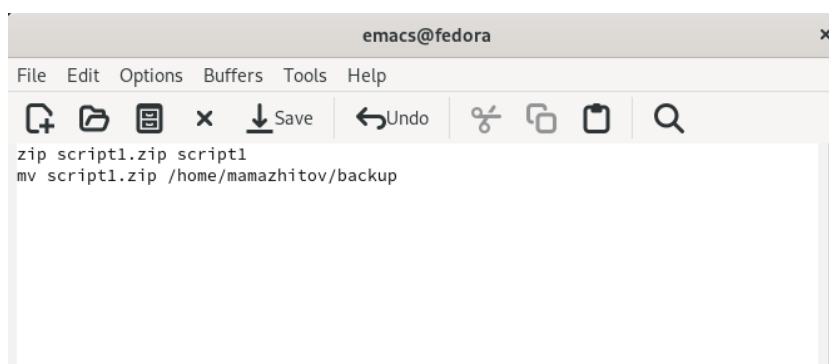


Рис. 3.1: Код 1 скрипта

Запустил скрипт. (рис. 3.2)

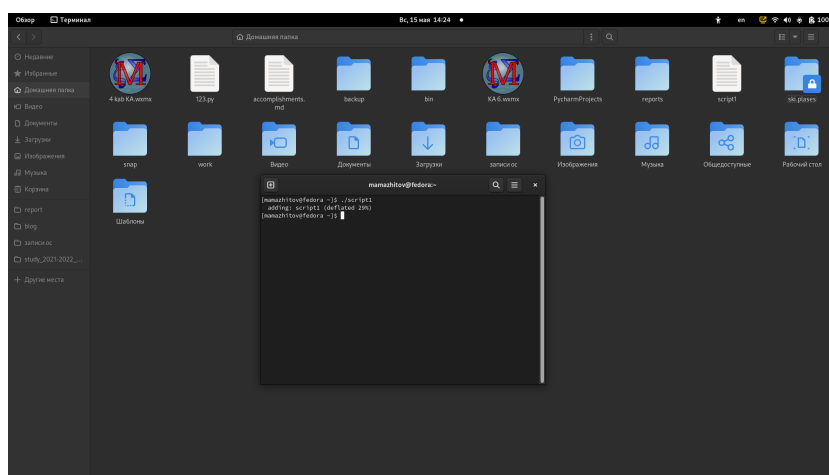


Рис. 3.2: Работа скрипта

Проверка. (рис. 3.3)

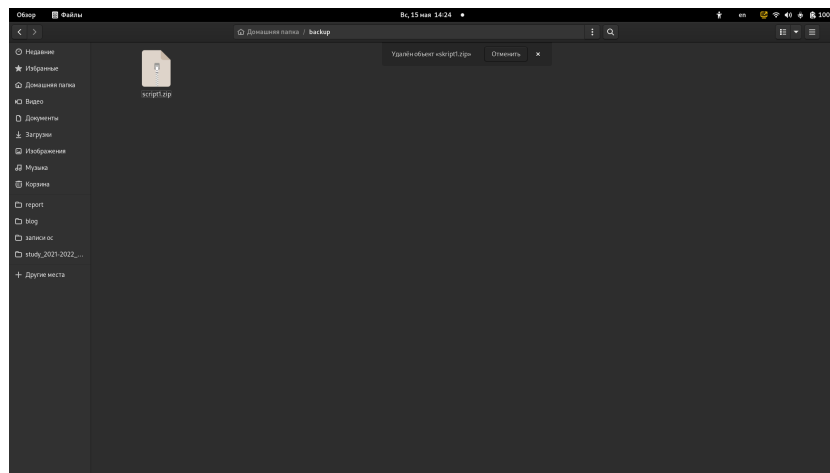


Рис. 3.3: Проверка работы скрипта

2. Открыл в emacs файл *script2*, написал цикл *for*, который последовательно выводит значения всех переданных аргументов.(рис. 3.4)

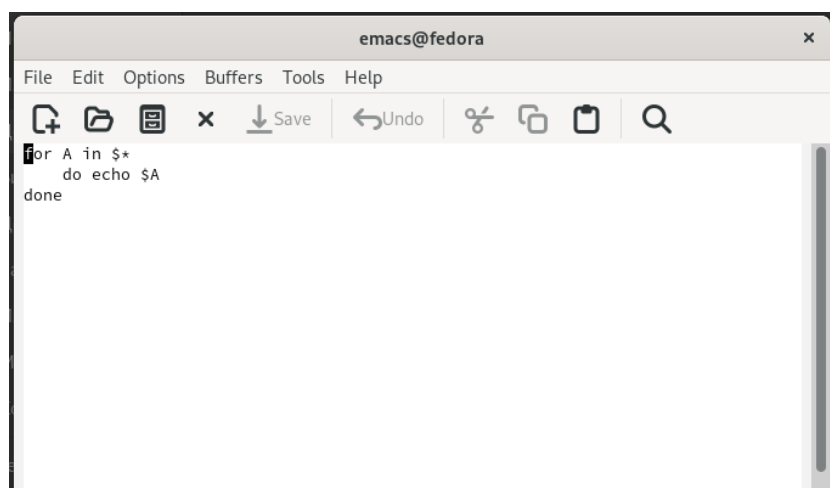
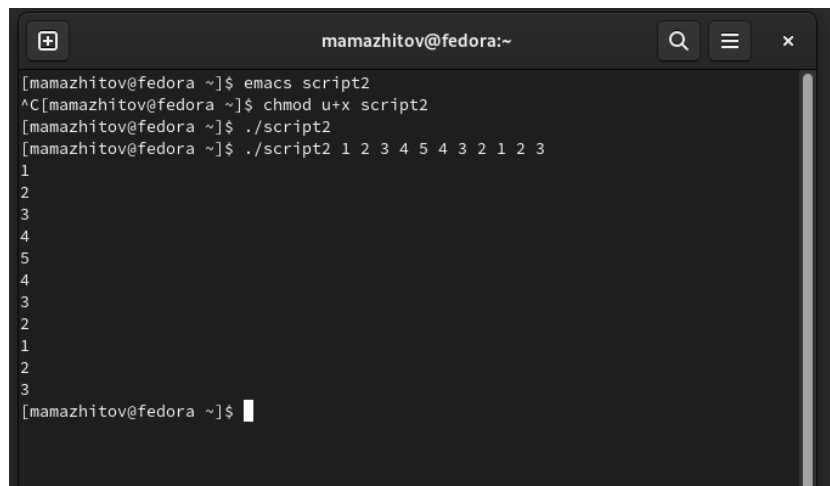


Рис. 3.4: Код 2 скрипта

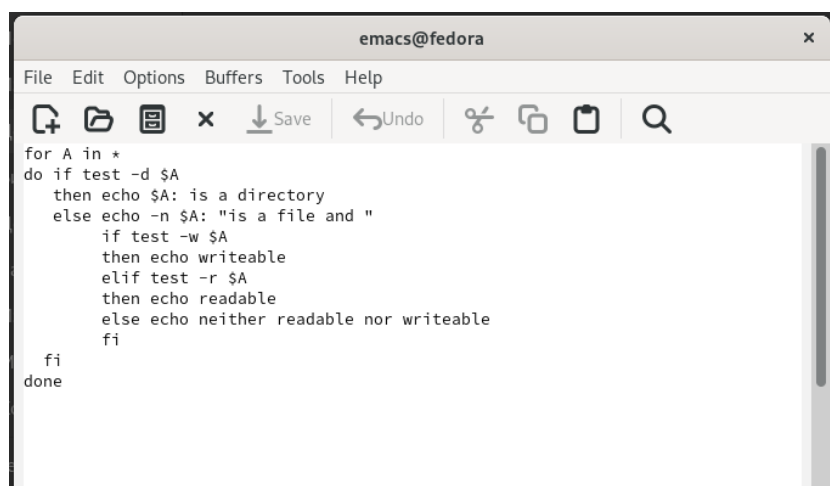
Запустил скрипт.(рис. 3.5)

A terminal window titled 'mamazhitov@fedora:~' with search, menu, and close icons. It shows a sequence of commands: 'emacs script2', '^C[mamazhitov@fedora ~]\$ chmod u+x script2', '[mamazhitov@fedora ~]\$./script2', and '[mamazhitov@fedora ~]\$./script2 1 2 3 4 5 4 3 2 1 2 3'. The output of the script is a list of numbers: 1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3. The prompt '[mamazhitov@fedora ~]\$' is visible at the bottom.

```
[mamazhitov@fedora ~]$ emacs script2
^C[mamazhitov@fedora ~]$ chmod u+x script2
[mamazhitov@fedora ~]$ ./script2
[mamazhitov@fedora ~]$ ./script2 1 2 3 4 5 4 3 2 1 2 3
1
2
3
4
5
4
3
2
1
2
3
[mamazhitov@fedora ~]$
```

Рис. 3.5: Работа скрипта

3. Открыл в emacs файл *script3* и скопировал программу из теории к лабораторной работы.(рис. 3.6)

An emacs editor window titled 'emacs@fedora' with a menu bar (File, Edit, Options, Buffers, Tools, Help) and a toolbar with icons for file operations, undo, redo, and search. The code displayed is a shell script for checking file permissions.

```
for A in *
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 nor writeable
fi
fi
done
```

Рис. 3.6: Код 3 скрипта

Проверил его работу.(рис. 3.7)

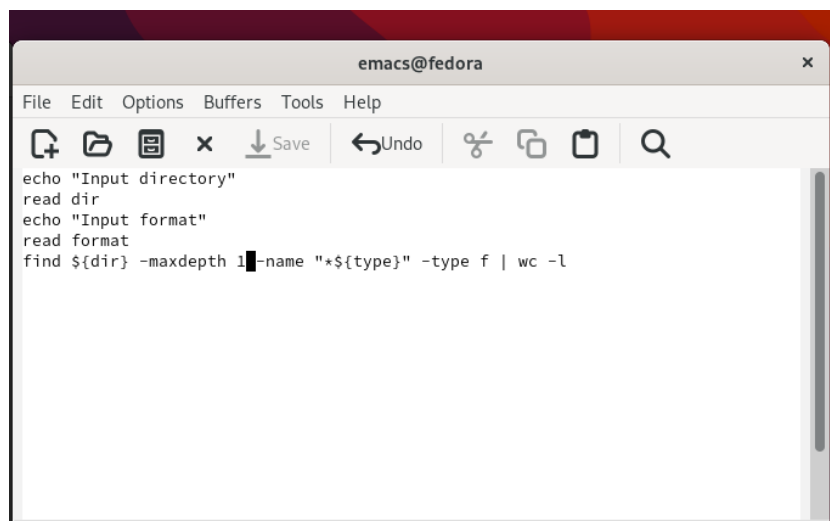
```

[mamazhitov@fedora ~]$ emacs script3
[mamazhitov@fedora ~]$ chmod u+x script3
[mamazhitov@fedora ~]$ ./script3
123.py: is a file and writeable
./script3: строка 2: test: слишком много аргументов
4 kab KA.wmx: is a file and ./script3: строка 5: test: слишком много аргументов
./script3: строка 7: test: слишком много аргументов
neither readable nor writeable
accomplishments.md: is a file and writeable
backup: is a directory
bin: is a directory
./script3: строка 2: test: KA: ожидается бинарный оператор
KA 6.wmx: is a file and ./script3: строка 5: test: KA: ожидается бинарный опера
тор
./script3: строка 7: test: KA: ожидается бинарный оператор
neither readable nor writeable
lab07.sh~: is a file and writeable
PycharmProjects: is a directory
reports: is a directory
script1: is a file and writeable
script1~: is a file and writeable
script2: is a file and writeable
script3: is a file and writeable

```

Рис. 3.7: Проверка 3 скрипта

4. Открыл в emacs файл *script4*. Написал программу, которая просит ввести путь к директории и формат файла, а затем выводит количество файлов с данным форматом в данной директории.(рис. 3.8)



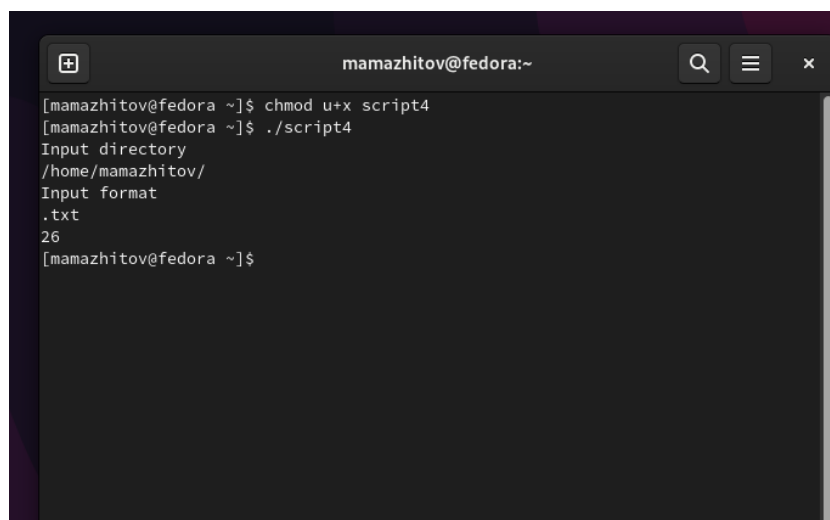
```

echo "Input directory"
read dir
echo "Input format"
read format
find ${dir} -maxdepth 1 -name "${type}" -type f | wc -l

```

Рис. 3.8: Код 4 скрипта

Проверил работу скрипта. (рис. 3.9)

A terminal window titled 'mamazhitov@fedora:~' with search, menu, and close icons. It shows the execution of a script named 'script4'. The script prompts for an input directory, which is '/home/mamazhitov/'. It then prompts for an input format, which is '.txt'. Finally, it outputs the number '26' and returns to the shell prompt.

```
[mamazhitov@fedora ~]$ chmod u+x script4
[mamazhitov@fedora ~]$ ./script4
Input directory
/home/mamazhitov/
Input format
.txt
26
[mamazhitov@fedora ~]$
```

Рис. 3.9: Проверка

4 Вывод

Мы научились писать небольшие командные файлы.

5 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:
 - оболочка Борна (Bourne shell или sh) — стандартная командная оболочка UNIX/Linux, содержащая базовый, но при этом полный набор функций;
 - С-оболочка (или csh) — надстройка на оболочкой Борна, использующая С-подобный синтаксис команд с возможностью сохранения истории выполнения команд;
 - оболочка Корна (или ksh) — напоминает оболочку С, но операторы управления программой совместимы с операторами оболочки Борна;
 - BASH — сокращение от Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек С и Корна (разработка компании Free Software Foundation).
2. POSIX (Portable Operating System Interface for Computer Environments) — набор стандартов описания интерфейсов взаимодействия операционной системы и прикладных программ.
3. `mark=/usr/andy/bin`

Данная команда присваивает значение строки символов `/usr/andy/bin` переменной `mark` типа строка-символов.

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"
```

4. Команда `let` является показателем того, что последующие аргументы представляют собой выражение, подлежащее вычислению. Команда `read` позволяет читать значения переменных со стандартного ввода
5. Простейшими математическими выражениями являются сложение (+), вычитание (-), умножение (*), целочисленное деление (/) и целочисленный остаток от деления (%).
6. Для облегчения программирования можно записывать условия оболочки `bash` в двойные скобки — (()).
7. Переменные `PS1` и `PS2` 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 `>`. Другие стандартные переменные:
 - `HOME` — имя домашнего каталога пользователя. Если команда `cd` вводится без аргументов, то происходит переход в каталог, указанный в этой переменной.
 - `IFS` — последовательность символов, являющихся разделителями в командной строке, например, пробел, табуляция и перевод строки (new line).
 - `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 You have mail (у Вас есть почта).
- TERM — тип используемого терминала.
 - 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 метасимвола. Экранирование может быть осуществлено с помощью предшествующего метасимволу символа , который, в свою очередь, является метасимволом.
 10. Командный файл можно создать с помощью какого-либо редактора, затем сделать его исполняемым и запустить его из терминала, введя “./название файла”.
 11. помощью ключевого слова *function*.
 12. Вводим команду *ls -lrt* и если первым в правах доступа стоит *d* то это каталог. Иначе это файл.
 13. Для создания массива используется команда *set* с флагом *-A*. Если использовать *typeset -i* для объявления и присвоения переменной, то при последующем её применении она станет целой. Изъять переменную из программы можно с помощью команды *unset*.
 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.

15.

- `$*` — отображается вся командная строка или параметры оболочки;
- `$?` — 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`.