

Отчёт по лабораторной работе №10

Дисциплина: Операционные системы

Батова Ирина Сергеевна, НММбд-01-22

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	7
4	Выводы	16
5	Контрольные вопросы	17

Список иллюстраций

3.1	Справка о zip	7
3.2	Справка о bzip2	8
3.3	Справка о tar	8
3.4	Создание файла 'file1.sh'	9
3.5	Программа 1, архивация файла	9
3.6	Проверка корректности программы 1	10
3.7	Создание файла 'file2.sh'	10
3.8	Программа 2, вывод аргументов	11
3.9	Проверка корректности программы 2	11
3.10	Создание файла 'file3.sh'	12
3.11	Программа 3, аналог команды 'ls'	13
3.12	Проверка корректности программы 3	14
3.13	Создание файла 'file4.sh'	14
3.14	Программа 4, вычисление количества файлов с указанным форматом	15
3.15	Проверка корректности программы 4	15

Список таблиц

1 Цель работы

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

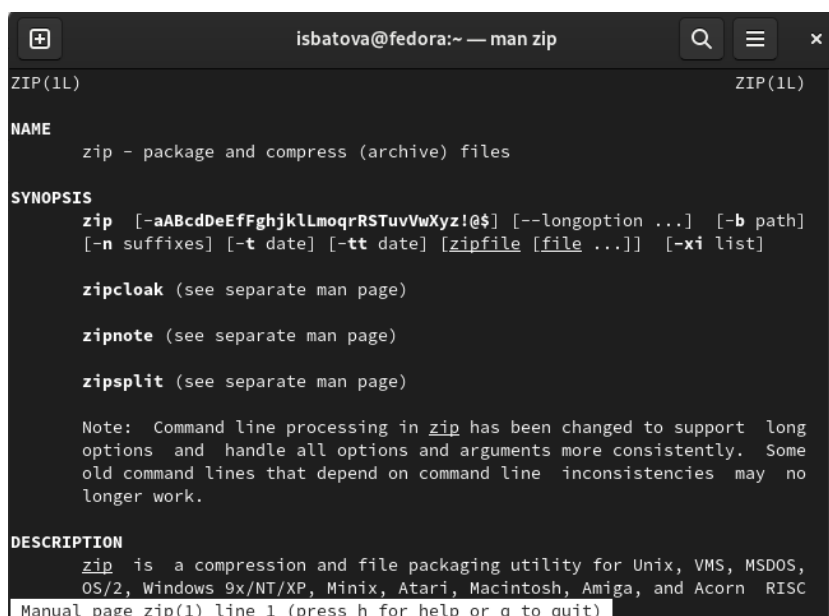
2 Задание

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 Выполнение лабораторной работы

1. Для начала работы изучим справку по командам архивации.

Чтобы изучить команду 'zip', вводим 'man zip' (рис. 3.1).



```
isbatova@fedora:~ — man zip
ZIP(1L)
NAME
    zip - package and compress (archive) files
SYNOPSIS
    zip [-aABcdDeEfFghjklLmoqrRSTuvVwXyz!@&] [--longoption ...] [-b path]
    [-n suffixes] [-t date] [-tt date] [zipfile [file ...]] [-xi list]

    zipcloak (see separate man page)
    zipnote (see separate man page)
    zipsplit (see separate man page)

    Note: Command line processing in zip has been changed to support long
    options and handle all options and arguments more consistently. Some
    old command lines that depend on command line inconsistencies may no
    longer work.
DESCRIPTION
    zip is a compression and file packaging utility for Unix, VMS, MSDOS,
    OS/2, Windows 9x/NT/XP, Minix, Atari, Macintosh, Amiga, and Acorn RISC
Manual page zip(1) line 1 (press h for help or q to quit)
```

Рис. 3.1: Справка о zip

Аналогично изучаем команды 'bzip2' (рис. 3.2) и 'tar' (рис. 3.3).

```
isbatova@fedora:~ — man bzip2
bzip2(1)                                General Commands Manual                                bzip2(1)

NAME
  bzip2, bunzip2 - a block-sorting file compressor, v1.0.8
  bzip2recover - recovers data from damaged bzip2 files

SYNOPSIS
  bzip2 [ -cdfkqstvl123456789 ] [ filenames ... ]
  bunzip2 [ -fkvsVL ] [ filenames ... ]
  bzip2recover filename

DESCRIPTION
  bzip2 compresses files using the Burrows-Wheeler block sorting text
  compression algorithm, and Huffman coding. Compression is generally
  considerably better than that achieved by more conventional
  LZ77/LZ78-based compressors, and approaches the performance of the PPM
  family of statistical compressors.

  The command-line options are deliberately very similar to those of GNU
  gzip, but they are not identical.

Manual page bzip2(1) line 1 (press h for help or q to quit)
```

Рис. 3.2: Справка о bzip2

```
isbatova@fedora:~ — man tar
TAR(1)                                GNU TAR Manual                                TAR(1)

NAME
  tar - an archiving utility

SYNOPSIS
  Traditional usage
  tar {A|c|d|r|t|u|x}[GnSkUW0mpsMBiajJzZhPlRvwo] [ARG...]

  UNIX-style usage
  tar -A [OPTIONS] ARCHIVE ARCHIVE

  tar -c [-f ARCHIVE] [OPTIONS] [FILE...]
  tar -d [-f ARCHIVE] [OPTIONS] [FILE...]
  tar -t [-f ARCHIVE] [OPTIONS] [MEMBER...]
  tar -r [-f ARCHIVE] [OPTIONS] [FILE...]
  tar -u [-f ARCHIVE] [OPTIONS] [FILE...]
  tar -x [-f ARCHIVE] [OPTIONS] [MEMBER...]

Manual page tar(1) line 1 (press h for help or q to quit)
```

Рис. 3.3: Справка о tar

Далее создаем файл для написания скрипта и открываем его (рис. 3.4).


```
[isbatova@fedora ~]$ touch file1.sh  
[isbatova@fedora ~]$ emacs &
```

Рис. 3.4: Создание файла 'file1.sh'

Нам необходимо 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. В качестве архиватора я выбрала bzip2.

Вводим скрипт в наш файл (рис. 3.5).

```
#!/bin/bash  
  
archive='file1.sh'  
mkdir ~/backup  
bzip2 -k ${archive}  
mv ${archive}.bz2 ~/backup/  
echo "Сделано"
```

Рис. 3.5: Программа 1, архивация файла

В данном скрипте мы сначала сохраняем в переменную archive сам файл, далее создаем каталог 'backup', архивируем наш скрипт и перемещаем его в созданный каталог. После выполнения на экран выводится команда "Сделано".

Далее мы добавляем право на выполнение файла командой 'chmod +x *.sh' и выполняем скрипт командой './file1.sh'. Для проверки корректности выполнения переходим в создавшийся каталог (команда cd), проверяем, что там есть архивированный файл (команда ls) и просматриваем содержимое архива (команда 'bunzip2 -c'). Программа работает корректно (рис. 3.6).

```

[isbatova@fedora ~]$ chmod +x *.sh
[isbatova@fedora ~]$ ./file1.sh
Сделано
[isbatova@fedora ~]$ cd backup/
[isbatova@fedora backup]$ ls
file1.sh.bz2
[isbatova@fedora backup]$ bunzip2 -c file1.sh.bz2
#!/bin/bash

archive='file1.sh'
mkdir ~/backup
bzip2 -k ${archive}
mv ${archive}.bz2 ~/backup/
echo "Сделано"

```

Рис. 3.6: Проверка корректности программы 1

2. Для начала работы создаем файл для написания скрипта и открываем его (рис. 3.7).

```

[isbatova@fedora ~]$ touch file2.sh
[isbatova@fedora ~]$ emacs &

```

Рис. 3.7: Создание файла 'file2.sh'

Нам необходимо написать командный файл, обрабатывающий любое произвольное число аргументов командной строки, в том числе превышающее десять. Сделаем так, чтобы скрипт последовательно распечатывал значения всех переданных аргументов.

Вводим скрипт в наш файл (рис. 3.8).

```
#!/bin/bash

echo "arguments"
for a in $@
do echo $a
done
```

Рис. 3.8: Программа 2, вывод аргументов

Сначала мы выводим слово 'arguments', затем пишем цикл для прохода по всем введенным пользователем аргументам и выводим эти аргументы на экран.

Далее мы добавляем право на выполнение файла командой 'chmod +x *.sh' и выполняем скрипт командой './file2.sh (аргументы)'. Для проверки корректности выполнения скрипта я выполнила программу как для числа аргументов меньше 10, так и больше (рис. 3.9).

```
[isbatova@fedora ~]$ chmod +x *.sh
[isbatova@fedora ~]$ ./file2.sh 0 1 2 3
arguments
0
1
2
3
[isbatova@fedora ~]$ ./file2.sh 0 1 2 3 4 5 6 7 8 9 10 11 12
arguments
0
1
2
3
4
5
6
7
8
9
10
11
12
```

Рис. 3.9: Проверка корректности программы 2

3. Для начала работы создаем файл для написания скрипта и открываем его (рис. 3.10).

```
[isbatova@fedora ~]$ touch file3.sh  
[isbatova@fedora ~]$ emacs &
```

Рис. 3.10: Создание файла 'file3.sh'

Нам необходимо 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.

Вводим скрипт в наш файл (рис. 3.11).

```
#!/bin/bash

a="$1"
for b in ${a}/*
do
    echo "$b"

    if test -f $b
    then echo "Файл"
    fi

    if test -d $b
    then echo "Каталог"
    fi

    if test -r $b
    then echo "Разрешено чтение"
    fi

    if test -w $b
    then echo "Разрешена запись"
    fi

    if test -x $b
    then echo "Разрешено выполнение"
    fi
done
```

Рис. 3.11: Программа 3, аналог команды 'ls'

Сначала мы в переменную *a* записываем путь до данного каталога. Далее мы пишем цикл, проходящий по всем каталогам и файлам заданного каталога. Затем на экран выводится название заданного каталога, после чего мы последовательно с помощью *if* проверяем, являются ли файлы обычными файлами, каталогами, а также наличие разрешения на чтение, запись и выполнение. В соответствии с результатами цикла на экран выводится соответствующая надпись.

Далее мы добавляем право на выполнение файла командой '*chmod +x *.sh*' и выполняем скрипт командой '*./file3.sh ~*' (то есть проверяем файлы в домашнем каталоге). Видим, что программа выполняется корректно (рис. 3.12).

```

[isbatova@fedora ~]$ chmod +x *.sh
[isbatova@fedora ~]$ ./file3.sh ~
/home/isbatova/1lab9.txt
Файл
Разрешено чтение
Разрешена запись
/home/isbatova/2lab9.txt
Файл
Разрешено чтение
Разрешена запись
/home/isbatova/3lab9.txt
Файл
Разрешено чтение
Разрешена запись
/home/isbatova/4lab9.txt
Файл
Разрешено чтение
Разрешена запись
/home/isbatova/backup
Каталог
Разрешено чтение
Разрешена запись
Разрешено выполнение

```

Рис. 3.12: Проверка корректности программы 3

4. Для начала работы создаем файл для написания скрипта и открываем его (рис. 3.13).

```

[isbatova@fedora ~]$ touch file4.sh
[isbatova@fedora ~]$ emacs &

```

Рис. 3.13: Создание файла 'file4.sh'

Нам необходимо 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. При этом путь к директории также передаётся в виде аргумента командной строки.

Вводим скрипт в наш файл (рис. 3.14).

```
#!/bin/bash

a="$1"
shift
for b in $@
do
    c=0
    for d in ${a}/*.${b}
    do
        if test -f "$d"
        then
            let c=c+1
        fi
    done
    echo "В каталоге $a содержится $c файлов с расширением $b"
done
```

Рис. 3.14: Программа 4, вычисление количества файлов с указанным форматом

Сначала мы в переменную `a` записываем путь до данного каталога. Далее удаляем первый аргумент (сам каталог) и вводим цикл, проходящий по всем заданным аргументам. Далее пишем цикл, проходящий по файлам, имеющим расширение аргумента 1, и через `if` добавляем к счетчику +1, если путь указывает на файл. В конце выводим соответствующее сообщение на экран.

Далее мы добавляем право на выполнение файла командой `'chmod +x *.sh'` и выполняем скрипт командой `'./file4.sh ~ txt pdf doc sh'` (то есть проверяем файлы в домашнем каталоге в форматах `txt`, `pdf`, `doc`, `sh`). Видим, что программа выполняется корректно (рис. 3.15).

```
[isbatova@fedora ~]$ chmod +x *.sh
[isbatova@fedora ~]$ ./file4.sh ~ txt pdf doc sh
В каталоге /home/isbatova содержится 5 файлов с расширением txt
В каталоге /home/isbatova содержится 1 файлов с расширением pdf
В каталоге /home/isbatova содержится 1 файлов с расширением doc
В каталоге /home/isbatova содержится 5 файлов с расширением sh
```

Рис. 3.15: Проверка корректности программы 4

4 Выводы

В данной лабораторной работе мной были изучены основы программирования в оболочке ОС UNIX/Linux. Я также научилась писать небольшие командные файлы.

5 Контрольные вопросы

1. Командная оболочка – 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 Сподобный 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. 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. Значение, присвоенное некоторой переменной, может быть впоследствии использовано.

Оболочка `bash` также позволяет работать с массивами. Для создания массива используется команда `set` с флагом `-A`. За флагом следует имя переменной, а затем список значений, разделённых пробелами.

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

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

5. В языке программирования `bash` можно применять сложение, вычитание, умножение, целочисленное деление и целочисленный остаток от деления.
6. В `(())` можно записывать условия оболочки `bash`. Помимо этого, внутри двойных скобок можно вычислять арифметические выражения и возвращать результат.

7.

- `PATH` - значением данной переменной является список каталогов, в которых командный процессор осуществляет поиск программы или команды, указанной в командной строке.
- `PS1` и `PS2` - переменные предназначены для отображения промптера командного процессора.
- `HOME` - имя домашнего каталога пользователя.
- `IFS` - последовательность символов, являющихся разделителями в командной строке.
- `MAIL` - имя файла, указанного в этой переменной, проверяется командным процессором каждый раз перед выводом на экран промптера

- TERM - тип используемого терминала.
 - LOGNAME - переменная, содержащая регистрационное имя пользователя
8. Метасимволы - символы, имеющие для командного процессора определенный смысл. К ним относятся: ' < > * ? | " &.
 9. Экранирование осуществляется с помощью предшествующего метасимволу символа обратного слэша. Для экранирования группы метасимволов нужно заключить её в одинарные кавычки.
 10. Для создания мы создаем текстовый файл и помещаем в него последовательность команд. Чтобы не вводить каждый раз последовательности символов bash, нужно обеспечить доступ к выполнению этого файла командой "chmod +x имя_файла". Далее вызываем командный файл на выполнение, вводя его имя в терминале как программу.
 11. Группа команд объединяется объединить в функцию. Для этого существует ключевое слово function, после которого следует имя функции и список команд, заключённых в фигурные скобки.
 12. Необходимо воспользоваться командами "test -f [путь до файла]" (является ли обычным файлом) и "test -d [путь до файла]" (является ли каталогом).
 13. Команда "set" используется для вывода списка переменных окружения.

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. 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;
- `$-` – значение флагов командного процессора;
- `${#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`.