

# Лабораторная работа №10

## Дисциплина: Операционные системы

Королев Федор Константинович

### Содержание

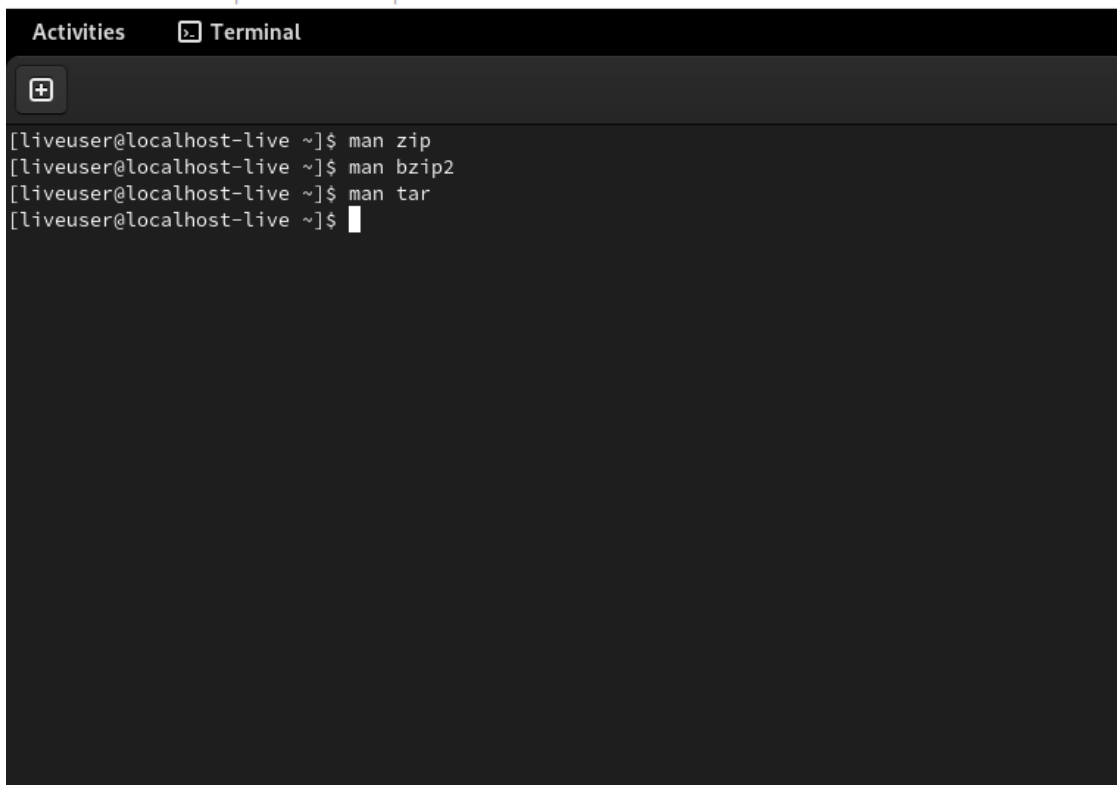
Цель работы .....	1
Ход работы.....	1
Контрольные вопросы .....	7
Вывод.....	10

### Цель работы

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

### Ход работы

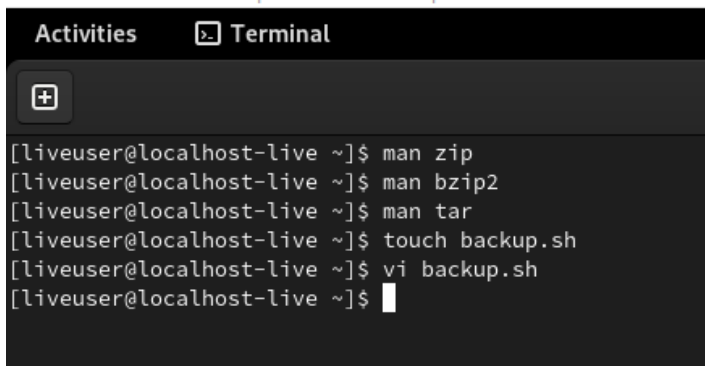
1.1. Изучил архиваторы zip, bzip2 или tar(Рис. 1):



```
Activities  Terminal
[liveuser@localhost-live ~]$ man zip
[liveuser@localhost-live ~]$ man bzip2
[liveuser@localhost-live ~]$ man tar
[liveuser@localhost-live ~]$
```

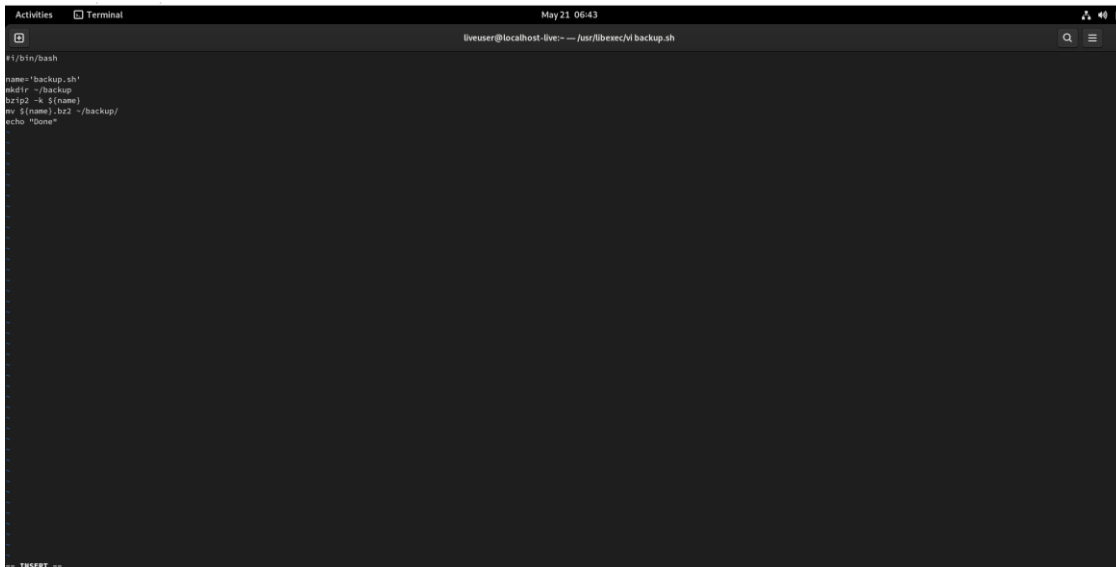
*Рис. 1 man zip, bzip2 или tar*

1.2. Написал скрипт, который при запуске будет делать резервную копию самого себя (т.е. файла, в котором содержится его исходный код) в другую директорию backup в вашем домашнем каталоге. Использовал bzip2 (Рис. 2 и 3):



```
Activities  Terminal
[liveuser@localhost-live ~]$ man zip
[liveuser@localhost-live ~]$ man bzip2
[liveuser@localhost-live ~]$ man tar
[liveuser@localhost-live ~]$ touch backup.sh
[liveuser@localhost-live ~]$ vi backup.sh
[liveuser@localhost-live ~]$
```

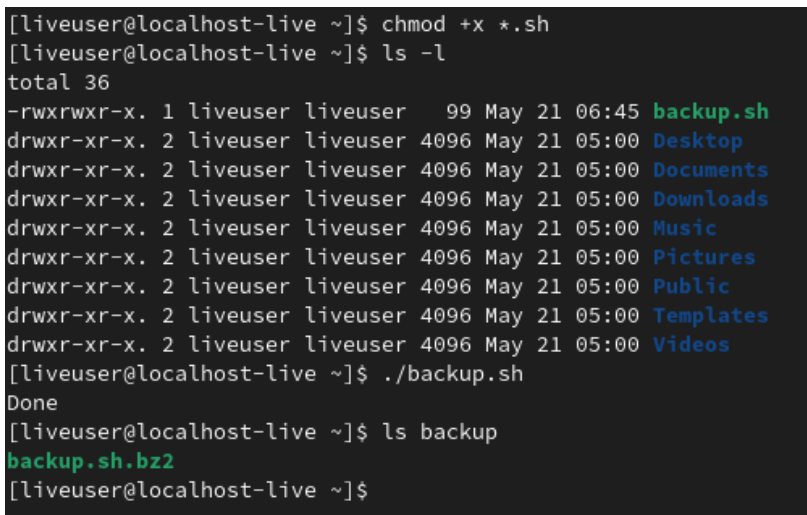
*Рис. 2 запуск vi*



```
#!/bin/bash
name="backup.sh"
mkdir ~/backup
for i in $(ls); do
  cp $i $(pwd)/backup/
done
```

Рис. 3 код программы в vi

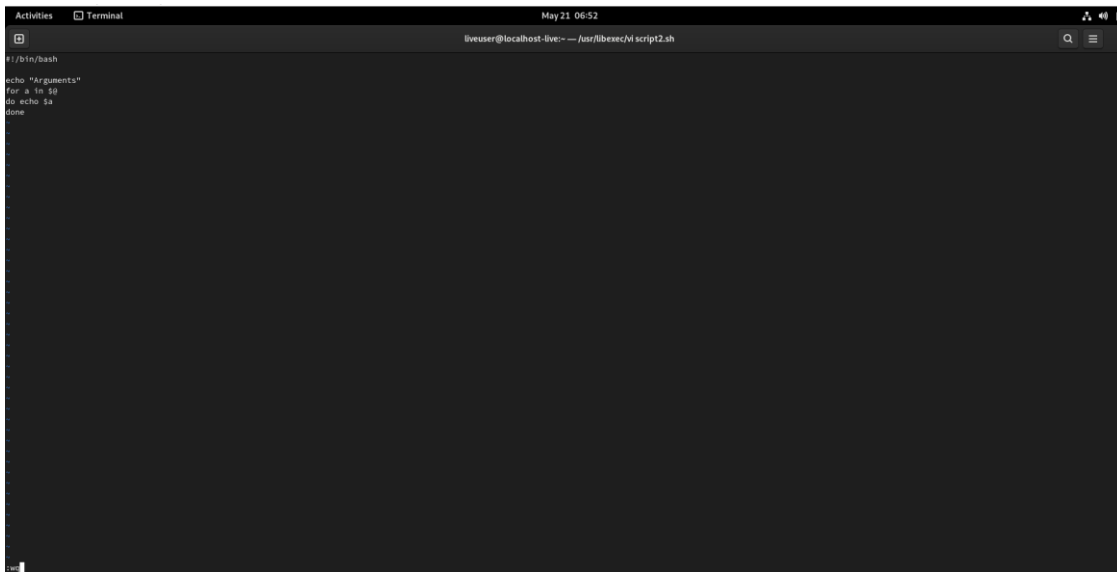
Проверил работу скрипта, предварительно добавив права на исполнение(Рис. 4):



```
[liveuser@localhost-live ~]$ chmod +x *.sh
[liveuser@localhost-live ~]$ ls -l
total 36
-rwxrwxr-x. 1 liveuser liveuser 99 May 21 06:45 backup.sh
drwxr-xr-x. 2 liveuser liveuser 4096 May 21 05:00 Desktop
drwxr-xr-x. 2 liveuser liveuser 4096 May 21 05:00 Documents
drwxr-xr-x. 2 liveuser liveuser 4096 May 21 05:00 Downloads
drwxr-xr-x. 2 liveuser liveuser 4096 May 21 05:00 Music
drwxr-xr-x. 2 liveuser liveuser 4096 May 21 05:00 Pictures
drwxr-xr-x. 2 liveuser liveuser 4096 May 21 05:00 Public
drwxr-xr-x. 2 liveuser liveuser 4096 May 21 05:00 Templates
drwxr-xr-x. 2 liveuser liveuser 4096 May 21 05:00 Videos
[liveuser@localhost-live ~]$ ./backup.sh
Done
[liveuser@localhost-live ~]$ ls backup
backup.sh.bzz2
[liveuser@localhost-live ~]$
```

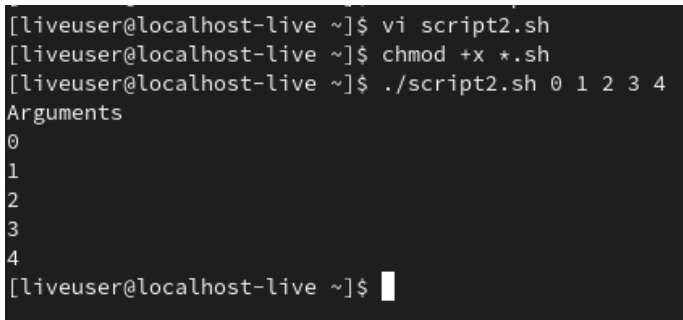
Рис. 4 работа скрипта

2. Написал пример командного файла, обрабатывающего любое произвольное число аргументов командной строки, в том числе превышающее 10(Рис. 5):

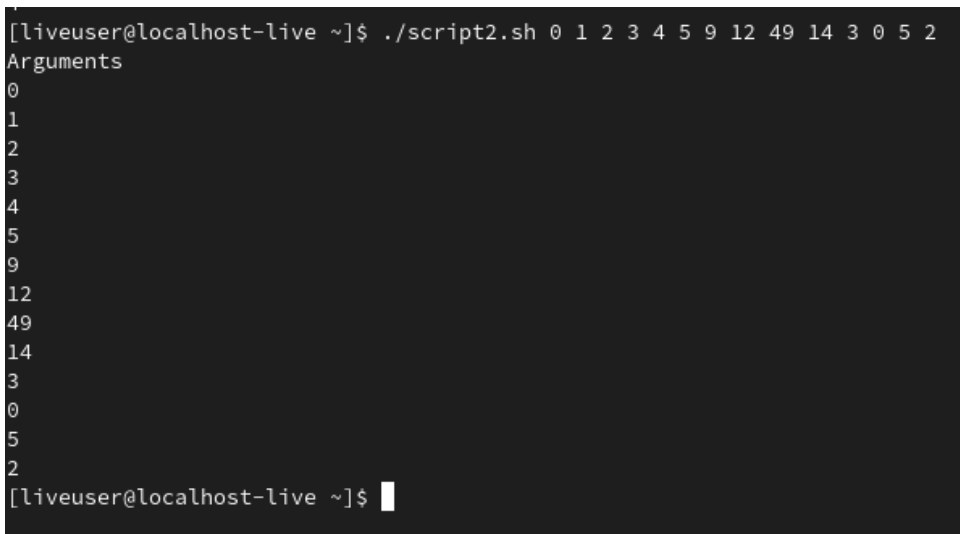


*Рис. 5 скрипт 2*

Дал права доступа и проверил работу(Рис. 6 и 7):

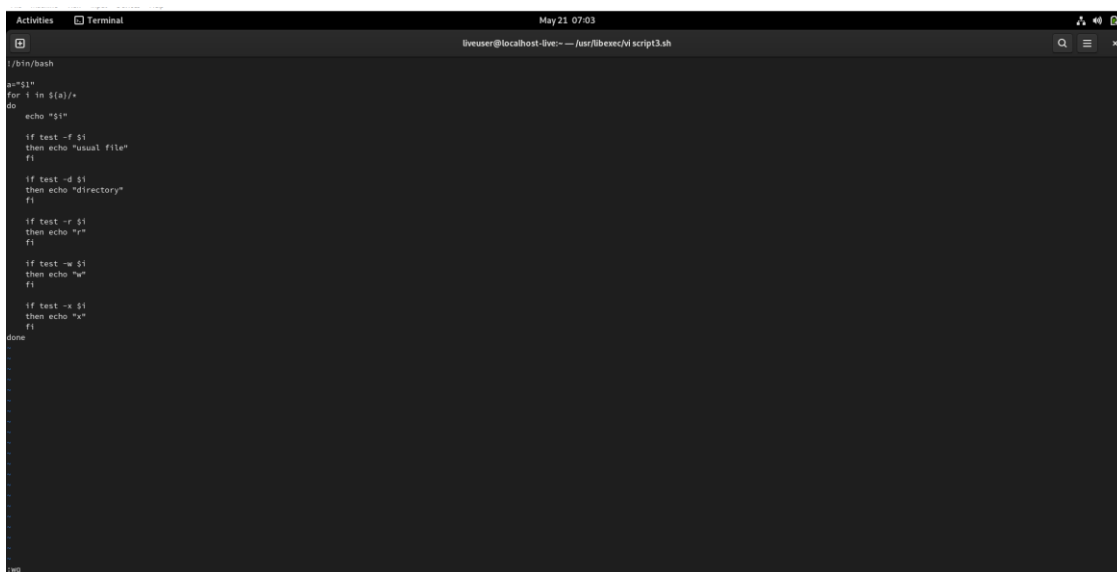


*Рис. 6 проверка работы скрипта*



*Рис. 7 числа больше 10*

3. Написал скрипт - аналог ls(Рис. 8):

A screenshot of a terminal window titled "Terminal" with a timestamp of "May 21 07:03". The window shows a shell script being executed. The script starts with a shebang line, followed by a loop that iterates over files in a directory. For each file, it checks if it's a regular file, a directory, readable, writable, or executable, and prints the result. The script ends with a "done" statement.

```
#!/bin/bash
p="$1"
for i in $(ls $p)
do
    echo "$i"
    if test -f $i
    then echo "usual file"
    fi
    if test -d $i
    then echo "directory"
    fi
    if test -r $i
    then echo "r"
    fi
    if test -w $i
    then echo "w"
    fi
    if test -x $i
    then echo "x"
    fi
done
```

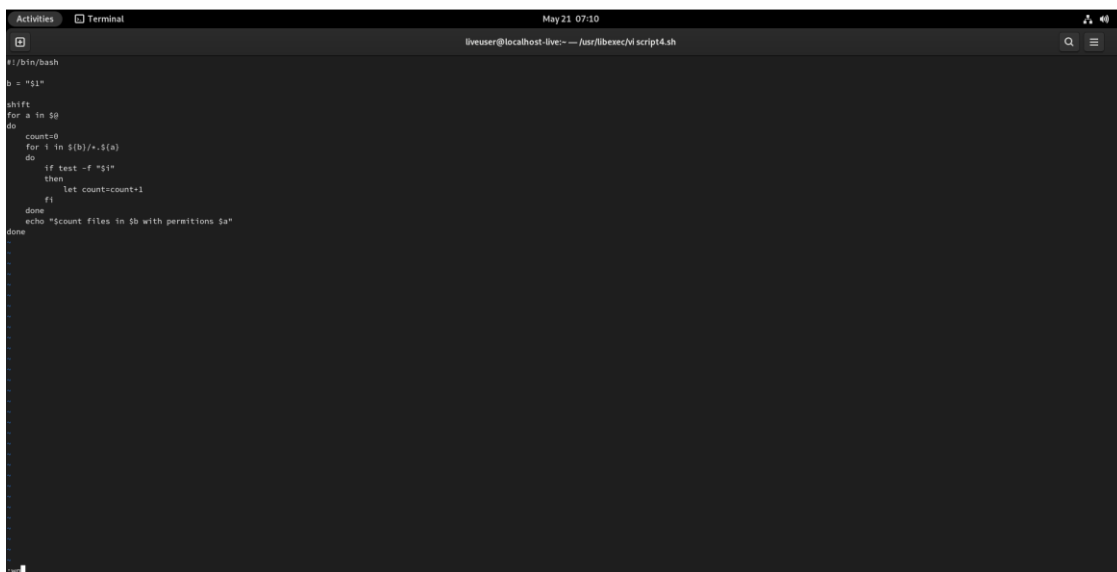
*Рис. 8 скрипт 3*

Проверка(Рис. 9):

```
[liveuser@localhost-live ~]$ chmod +x *.sh
[liveuser@localhost-live ~]$ ./script3.sh ~
./script3.sh: line 1: !/bin/bash: No such file or directory
/home/liveuser/backup
directory
r
w
x
/home/liveuser/backup.sh
usual file
r
w
x
/home/liveuser/Desktop
directory
r
w
x
/home/liveuser/Documents
directory
r
w
x
/home/liveuser/Downloads
directory
r
w
x
/home/liveuser/Music
directory
r
w
x
/home/liveuser/Pictures
directory
r
w
x
/home/liveuser/Public
directory
r
w
x
```

*Рис. 9 проверка скрипта*

4. Напишем скрипт, который получает в качестве аргумента путь и формат, а возвращает количество таких файлов(Рис. 10)



```
#!/bin/bash
0 = "s1"
shift
for a in $(
do
    count=0
    for i in $(ls | grep $a)
    do
        if test -f "$i"
        then
            let count=count+1
        fi
    done
    echo "Count files in $b with permissions $a"
done
```

Рис. 10 скрипт 4

Проверим скрипт, предварительно создав файлы для проверки(Рис. 11 и 12):

```
[liveuser@localhost-live ~]$ touch test1.pdf
[liveuser@localhost-live ~]$ touch test2.pdf test3.pdf test1.doc test2.doc
```

Рис. 11 создание файлов для проверки

```
[liveuser@localhost-live ~]$ ./script4.sh ~ doc pdf sh txt
2 files in /home/liveuser with permissions doc
3 files in /home/liveuser with permissions pdf
4 files in /home/liveuser with permissions sh
0 files in /home/liveuser with permissions txt
```

Рис. 12 выполнение скрипта

## 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 возможностью 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 Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек C 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. 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 `.`. Например, команда «`mv afile{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 `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). Оболочка `bash` поддерживает встроенные арифметические функции. Команда `let` является показателем того, что последующие аргументы представляют собой выражение, подлежащее вычислению. Простейшее выражение – это единичный 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, введённые с клавиатуры, а переменная `trash` нужна для того, чтобы отобрать всю избыточно введённую 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. `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 `>`. `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 (`newline`). `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). 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 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 программа, 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 unsetсфлагом -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*

–количествослов, 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.

## Вывод

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