

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ



**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ОБРАЗОВАНИЯ «ЛИПЕЦКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Институт

компьютерных наук

Кафедра

автоматизированных систем управления

ЛАБОРАТОРНАЯ РАБОТА №4

По дисциплине "Операционные системы Linux"

На тему "Создание и использование сценариев (скриптов) в Linux"

Студент

ПИ-22-1

подпись, дата

Кистерёв В.А.

Руководитель

канд.техн.наук, доцент

ученая степень, ученое звание

подпись, дата

Кургасов В.В.

Липецк, 2024 г.

Оглавление

| | |
|---------------------------------|-----------|
| Цель работы | 3 |
| Ход работы | 4 |
| 1. Часть I..... | 4 |
| 2. Часть II | 7 |
| Контрольные вопросы..... | 16 |
| Вывод | 22 |

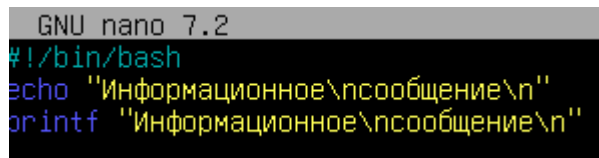
Цель работы

Изучить основные возможности языка программирования высокого уровня Shell, получить навыки написания и использования скриптов.

Ход работы

1. Часть I

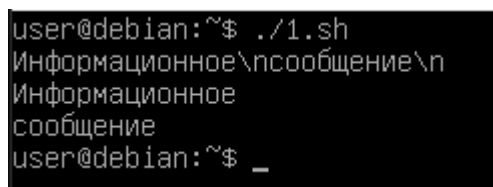
1.1. Используя команды ECHO, PRINTF, вывести информационные сообщения на экран. Код скрипта представлен на рисунке 1.



```
GNU nano 7.2
#!/bin/bash
echo "Информационное\nсообщение\n"
printf "Информационное\нсообщение\n"
```

Рисунок 1 – Код скрипта 1.1

Результат работы скрипта 1.1 представлен на рисунке 2.



```
user@debian:~$ ./1.sh
Информационное\nсообщение\n
Информационное
сообщение
user@debian:~$ _
```

Рисунок 2 – Результат работы скрипта 1.1

ECHO используется для простого вывода текста и автоматически добавляет новую строку, но его поведение может отличаться (например, с флагами -e и -n).

PRINTF предоставляет более гибкое и предсказуемое форматирование, аналогичное функции printf в C, требует явного указания новой строки (\n), поддерживает форматные строки и плейсхолдеры (например, %s, %d), что делает его более подходящим для сложного вывода и кроссплатформенных скриптов.

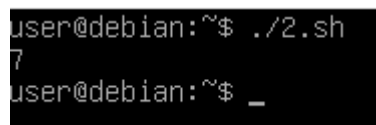
1.2. Присвоить переменной A целочисленное значение. Посмотреть значение переменной A. Код скрипта представлен на рисунке 3.



```
GNU nano 7.2
#!/bin/bash
A=7
echo $A
```

Рисунок 3 – Код скрипта 1.2

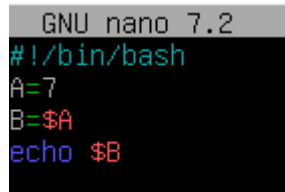
Результат работы скрипта 1.2 представлен на рисунке 4.



```
user@debian:~$ ./2.sh
7
user@debian:~$ _
```

Рисунок 4 – Результат работы скрипта 1.2

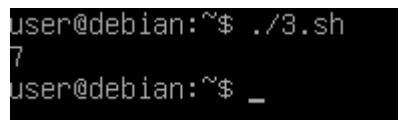
1.3. Присвоить переменной В значение переменной А. Посмотреть значение переменной В. Код скрипта представлен на рисунке 5.



```
GNU nano 7.2
#!/bin/bash
A=7
B=$A
echo $B
```

Рисунок 5 – Код скрипта 1.3

Результат работы скрипта 1.3 представлен на рисунке 6.



```
user@debian:~$ ./3.sh
7
user@debian:~$ _
```

Рисунок 6 – Результат работы скрипта 1.3

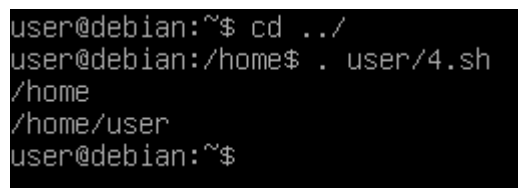
1.4. Присвоить переменной С значение "путь до своего каталога". Перейти в этот каталог с использованием переменной. Код скрипта представлен на рисунке 7.



```
GNU nano 7.2
#!/bin/bash
C="$HOME"
echo $(pwd)
cd "$C"
echo $(pwd)
```

Рисунок 7 – Код скрипта 1.4

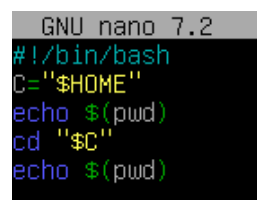
Результат работы скрипта 1.4 представлен на рисунке 8.



```
user@debian:~$ cd ../
user@debian:/home$ . user/4.sh
/home
/home/user
user@debian:~$
```

Рисунок 8 – Результат работы скрипта 1.4

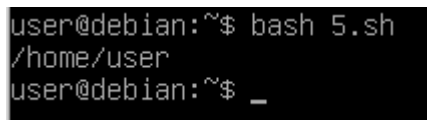
1.5. Присвоить переменной D значение "имя команды". Выполнить эту команду, используя значение переменной. Код скрипта представлен на рисунке 9.



```
GNU nano 7.2
#!/bin/bash
C="$HOME"
echo $(pwd)
cd "$C"
echo $(pwd)
```

Рисунок 9 – Код скрипта 1.5

Результат работы скрипта 1.5 представлен на рисунке 10.



```
user@debian:~$ bash 5.sh
/home/user
user@debian:~$ _
```

Рисунок 10 – Результат работы скрипта 1.5

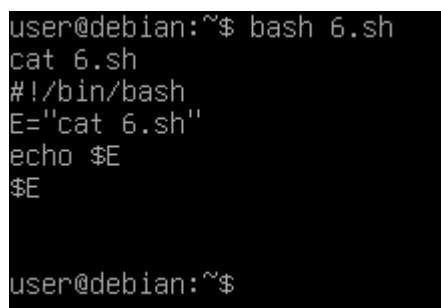
1.6. Присвоить переменной E значение "имя команды", а именно, команды просмотра содержимого файла, посмотреть содержимое переменной. Выполнить эту команду, используя значение переменной. Код скрипта представлен на рисунке 11.



```
GNU nano 7.2
#!/bin/bash
E="cat 6.sh"
echo E
$E
```

Рисунок 11 – Код скрипта 1.6

Результат работы скрипта 1.6 представлен на рисунке 12.



```
user@debian:~$ bash 6.sh
cat 6.sh
#!/bin/bash
E="cat 6.sh"
echo $E
$E

user@debian:~$
```

Рисунок 12 – Результат работы скрипта 1.6

1.7. Присвоить переменной F значение "имя команды", а именно, сортировки содержимого файла. Выполнить эту команду, используя значение переменной. Код скрипта представлен на рисунке 13.



```
GNU nano 7.2
#!/bin/bash
F="sort text.txt"
$F
```

Рисунок 13 – Код скрипта 1.7

Результат работы скрипта 1.7 представлен на рисунке 14.

```

user@debian:~$ cat text.txt
3
89
54
3
5
user@debian:~$ bash 7.sh
3
3
5
54
89
user@debian:~$

```

Рисунок 14 – Результат работы скрипта 1.7

2. Часть II

2.1. Программа запрашивает значение переменной, а затем выводит значение этой переменной. Код программы представлен на рисунке 15.

```

GNU nano 7.2
#!/bin/bash
printf "Значение переменной: "
read var
echo "$var"

```

Рисунок 15 – Код программы 2.1

Результат работы программы 2.1 представлен на рисунке 16.

```

user@debian:~$ bash 2-1.sh
Значение переменной: 555
555
user@debian:~$ _

```

Рисунок 16 – Результат работы программы 2.1

2.2. Программа запрашивает имя пользователя, затем здоровается с ним, используя значение введённой переменной. Код программы представлен на рисунке 17.

```

GNU nano 7.2
#!/bin/bash
printf "Введите ваше имя: "
read name
echo "Привет, $name!"

```

Рисунок 17 – Код программы 2.2

Результат работы программы 2.2 представлен на рисунке 18.

```
user@debian:~$ bash 2-2.sh
Введите ваше имя: Виктор
Привет, Виктор!
user@debian:~$
```

Рисунок 18 – Результат работы программы 2.2

2.3. Программа запрашивает значение двух переменных, вычисляет сумму (разность, произведение, деление) этих переменных. Результат выводится на экран. Код программы представлен на рисунке 19.

```
GNU nano 7.2
#!/bin/bash
read a
read b
echo "Сумма чисел: $(expr $a + $b)"
echo "Разность чисел: $(expr $a - $b)"
echo "Произведение чисел: $(expr $a \* $b)"
echo "Деление чисел: $(expr $a / $b)"
```

Рисунок 19 – Код программы 2.3

Результат работы программы 2.3 представлен на рисунке 20.

```
user@debian:~$ bash 2-3.sh
20
5
Сумма чисел: 25
Разность чисел: 15
Произведение чисел: 100
Деление чисел: 4
user@debian:~$
```

Рисунок 20 – Результат работы программы 2.3

2.4. Вычислить объём цилиндра. Исходные данные запрашиваются программой. Результат выводится на экран. Код программы представлен на рисунке 21.

```
GNU nano 7.2
#!/bin/bash
read r
read h
echo "Объём цилиндра: $(echo "3.14 * $r * $r * $h" | bc)"
```

Рисунок 21 – Код программы 2.4

Результат работы программы 2.4 представлен на рисунке 22.


```

user@debian:~$ bash 2-4.sh
5
10
Объём цилиндра: 785.00
user@debian:~$ _

```

Рисунок 22 – Результат работы программы 2.4

2.5. Используя позиционные параметры, отобразить имя программы, количество аргументов командной строки, значение каждого аргумента командной строки. Код программы представлен на рисунке 23.

```

GNU nano 7.2
#!/bin/bash
echo "Имя: $0"
echo "Кол-во аргументов командной строки: $# "
echo "Значение каждого аргумента командной строки: "
for arg in "$@"
do
    echo "$arg"
done

```

Рисунок 23 – Код программы 2.5

Результат работы программы 2.5 представлен на рисунке 24.

```

user@debian:~$ bash 2-5.sh 555 666 777
Имя: 2-5.sh
Кол-во аргументов командной строки: 3
Значение каждого аргумента командной строки:
555
666
777
user@debian:~$ _

```

Рисунок 24 – Результат работы программы 2.5

2.6. Используя позиционный параметр, отобразить содержимое текстового файла, указанного в качестве аргумента командной строки. Код программы представлен на рисунке 25.

```

GNU nano 7.2
#!/bin/bash
A="cat $1"
$A

```

Рисунок 25 – Код программы 2.6

Результат работы программы 2.6 представлен на рисунке 26.

```

user@debian:~$ bash 2-6.sh text.txt
3
89
54
3
5
user@debian:~$

```

Рисунок 26 – Результат работы программы 2.6

2.7. Используя оператор FOR, отобразить содержимое текстовых файлов текущего каталога поэкранно. Код программы представлен на рисунке 27.

```

GNU nano 7.2
#!/bin/bash
for file in *.txt
do
    if [ -f "$file" ]; then
        less "$file"
    fi
done

```

Рисунок 27 – Код программы 2.7

2.8. Программой запрашивается ввод числа, значение которого затем сравнивается с допустимым значением. В результате этого сравнения на экран выдаются соответствующие сообщения. Код программы представлен на рисунке 28.

```

GNU nano 7.2
#!/bin/bash
read num
if [ $num -lt 100 ] && [ $num -gt 90 ]; then
    echo "Число $num находится в диапазоне от 90 до 100"
else
    echo "Число $num вне диапазона"
fi

```

Рисунок 28 – Код программы 2.8

Результат работы программы 2.8 представлен на рисунке 29.

```

user@debian:~$ bash 2-8.sh
67
Число 67 вне диапазона
user@debian:~$ bash 2-8.sh
92
Число 92 находится в диапазоне от 90 до 100
user@debian:~$ _

```

Рисунок 29 – Результат работы программы 2.8

2.9. Программой запрашивается год, определяется, високосный ли он.

Результат выводится на экран. Код программы представлен на рисунке 30.

```
GNU nano 7.2
#!/bin/bash
read year
if (( year % 4 == 0 && year % 100 != 0 )) || (( year % 400 == 0 )); then
    echo "$year високосный год"
else
    echo "$year не високосный год"
fi
```

Рисунок 30 – Код программы 2.9

Результат работы программы 2.9 представлен на рисунке 31.

```
user@debian:~$ bash 2-9.sh
2025
2025 не високосный год
user@debian:~$ _
```

Рисунок 31 – Результат работы программы 2.9

2.10. Вводятся целочисленные значения двух переменных. Вводится диапазон данных. Пока значения переменных находятся в указанном диапазоне, их значения инкрементируются. Код программы представлен на рисунке 32.

```
GNU nano 7.2
#!/bin/bash
read var1
read var2
read max
while [[ $var1 -le $max && $var2 -le $max ]]; do
    var1=$((var1+1))
    var2=$((var2+1))
    echo "$var1, $var2"
done
```

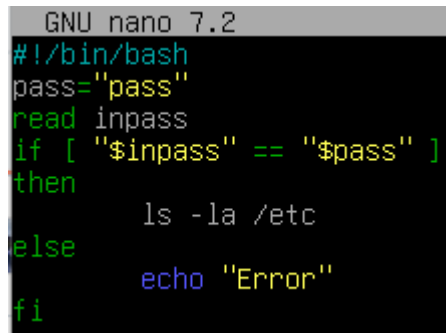
Рисунок 32 – Код программы 2.10

Результат работы программы 2.10 представлен на рисунке 33.

```
user@debian:~$ bash 2-10.sh
9
11
15
10, 12
11, 13
12, 14
13, 15
14, 16
user@debian:~$ _
```

Рисунок 33 – Результат работы программы 2.10

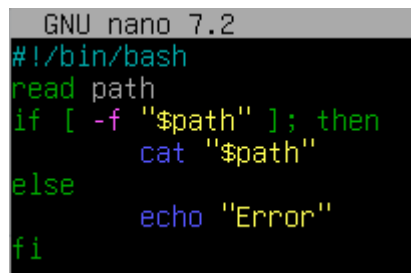
2.11. В качестве аргумента командной строки указывается пароль. Если пароль введён верно, постранично отображается содержимое каталога /etc. Код программы представлен на рисунке 34.



```
GNU nano 7.2
#!/bin/bash
pass="pass"
read inpass
if [ "$inpass" == "$pass" ]
then
    ls -la /etc
else
    echo "Error"
fi
```

Рисунок 34 – Код программы 2.11

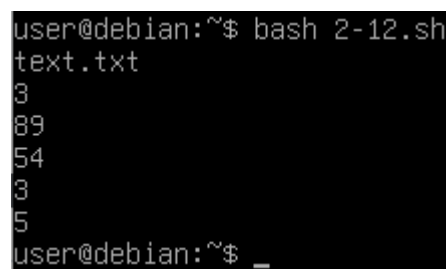
2.12. Проверить, существует ли файл. Если да, выводится на экран его содержимое, если нет - выдаётся соответствующее сообщение. Код программы представлен на рисунке 35.



```
GNU nano 7.2
#!/bin/bash
read path
if [ -f "$path" ]; then
    cat "$path"
else
    echo "Error"
fi
```

Рисунок 35 – Код программы 2.12

Результат работы программы 2.12 представлен на рисунке 36.



```
user@debian:~$ bash 2-12.sh
text.txt
3
89
54
3
5
user@debian:~$ _
```

Рисунок 36 – Результат работы программы 2.12

2.13. Если файл есть каталог и этот каталог можно читать, просматривается содержимое этого каталога. Если каталог отсутствует, он создаётся. Если файл не есть каталог, просматривается содержимое файла. Код программы представлен на рисунке 37.

```

GNU nano 7.2
#!/bin/bash
read path

if [ -d "$path" ] && [ -r "$path" ]; then
    for file in "$path"/*
    do
        echo "$file"
    done
elif [ -f "$path" ]; then
    cat "$path"
else
    mkdir -p "$path"
fi

```

Рисунок 37 – Код программы 2.13

Результат работы программы 2.13 представлен на рисунке 38.

```

user@debian:~/testdir$ ls -l
итого 0
user@debian:~/testdir$ bash ../2-13.sh
test
user@debian:~/testdir$ ls -l
итого 4
drwxr-xr-x 2 user user 4096 ноя 10 11:25 test

```

Рисунок 38 – Результат работы программы 2.13

2.14. Анализируются атрибуты файла. Если первый файл существует и используется для чтения, а второй файл существует и используется для записи, то содержимое первого файла перенаправляется во второй файл. Код программы представлен на рисунке 39.

```

GNU nano 7.2
#!/bin/bash
read f1
read f2
if [[ ! -r "$f1" ]]; then
    echo "error"
    exit 1
fi
if [[ ! -w "$f2" ]]; then
    echo "error"
    exit 1
fi
cat "$f1" > "$f2"
echo "f1 > f2"

```

Рисунок 39 – Код программы 2.14

Результат работы программы 2.14 представлен на рисунке 40.

```

user@debian:~/testdir$ ls -a
.  ..  1  2  2-14.sh
user@debian:~/testdir$ cat 1
Текст
user@debian:~/testdir$ bash 2-14.sh
1
2
f1 > f2
user@debian:~/testdir$ cat 2
Текст
user@debian:~/testdir$ _

```

Рисунок 40 – Результат работы программы 2.14

2.15. Если файл запуска программы найден, программа запускается. Код программы представлен на рисунке 41.

```

GNU nano 7.2
#!/bin/bash
if [ -e "1.sh" ]; then
    echo "Запуск 1.sh"
    ./1.sh
else
    echo "Файл не найден"
fi

```

Рисунок 41 – Код программы 2.15

Результат работы программы 2.15 представлен на рисунке 42.

```

user@debian:~$ bash 2-15.sh
Запуск 1.sh
Информационное\посообщение\п
Информационное
сообщение
user@debian:~$

```

Рисунок 42 – Результат работы программы 2.15

2.16. В качестве позиционного параметра задаётся файл, проанализируйте его размер. Если размер файла больше нуля, содержимое файла сортируется по первому столбцу по возрастанию, отсортированная информация помещается в другой файл. Код программы представлен на рисунке 43.

```

GNU nano 7.2
#!/bin/bash
read f
if [[ -s $f ]]; then
    sort -k1 $f > sort.txt
    cat sort.txt
else
    echo "Пусто"
fi

```

Рисунок 43 – Код программы 2.16

Результат работы программы 2.16 представлен на рисунке 44.

```
user@debian:~$ cat text.txt
3
89
54
3
5
user@debian:~$ bash 2-16.sh
text.txt
3
3
5
54
89
user@debian:~$
```

Рисунок 44 – Результат работы программы 2.16

Контрольные вопросы

1. В чём отличие пользовательских переменных от переменных среды?

Пользовательские переменные — это переменные, которые задаются в рамках текущей сессии командной оболочки (Shell) и не передаются автоматически другим процессам или скриптам. Они доступны только в текущем Shell или в процессе, где они были созданы.

Переменные среды — это переменные, которые задаются в операционной системе и могут быть доступны для всех запущенных процессов и дочерних процессов. Они часто задаются глобально (например, PATH, HOME, USER) и передаются при запуске команд или скриптов. Чтобы пользовательская переменная стала переменной среды, её нужно "экспортировать" командой `export`.

2. Математические операции в SHELL.

В Shell (обычно Bash) для математических операций можно использовать встроенные механизмы, например:

- Арифметические операции: выполняются с помощью команды `expr`, встроенной арифметики `$((...))`, или команды `let`.

```
result=$((5 + 3))
```

- Модуль `bc`: для работы с числами с плавающей точкой и более сложных операций можно использовать команду `bc`.

```
echo "scale=2; 5 / 3" | bc
```

3. Условные операторы в SHELL.

В Shell есть несколько условных операторов:

- `if ... then ... fi`: основной оператор для выполнения условий.

```
if [ "$a" -gt "$b" ]; then
```

```
    echo "a больше b"
```

```
fi
```

- `if ... elif ... else ... fi`: многоуровневое ветвление.

```
if [ "$a" -gt "$b" ]; then
```

```
    echo "a больше b"
```



```
elif [ "$a" -lt "$b" ]; then
```

```
    echo "a меньше b"
```

```
else
```

```
    echo "a равно b"
```

```
fi
```

- case ... in ... esac: используется для проверки нескольких значений переменной.

```
case "$value" in
```

```
    "1") echo "Значение один" ;;
```

```
    "2") echo "Значение два" ;;
```

```
    *) echo "Неизвестное значение" ;;
```

```
esac
```

4. Принципы построения простых и составных условий.

Простые условия: проверка одного условия, как в `if ["$a" -eq 1]`;

Составные условия: используются логические операторы `&&` (и) и `||` (или).

```
if [ "$a" -eq 1 ] && [ "$b" -eq 2 ]; then
```

```
    echo "Оба условия верны"
```

```
fi
```

Скобки `[[...]]` позволяют использовать более сложные выражения и упрощают работу с составными условиями.

5. Циклы в SHELL.

Основные типы циклов:

- for: проходит по списку значений.

```
for i in 1 2 3; do
```

```
    echo "$i"
```

```
done
```

- while: выполняется, пока условие истинно.

```
while [ "$a" -lt 5 ]; do
```

```
    echo "$a"
```

```
    a=$((a + 1))
```

done

- until: выполняется, пока условие ложно.

```
until [ "$a" -ge 5 ]; do
```

```
    echo "$a"
```

```
    a=$((a + 1))
```

```
done
```

6. Массивы и модули в SHELL.

Массивы можно объявить и затем заполнить его элементами.

```
array=(one two three)
```

```
echo ${array[0]} # выводит "one"
```

Доступ к элементам массива осуществляется с использованием индекса.

Все элементы массива выводятся с помощью `${array[@]}`.

Модули: в Shell можно подключать другие скрипты с помощью `source` (или `.`) для повторного использования кода.

```
source ./script.sh
```

7. Чтение параметров командной строки.

Для работы с параметрами командной строки используются специальные переменные:

- \$1, \$2, ... — это параметры, переданные скрипту. Например, \$1 — первый параметр.

- \$# — количество параметров.

- \$@ — все параметры в виде списка.

8. Как различать ключи и параметры?

Обычно ключи начинаются с `-` или `--`, чтобы их можно было отличить от других параметров. Например, `-f` — ключ, а `file.txt` — параметр.

9. Чтение данных из файлов.

Для чтения данных из файла можно использовать следующие методы:

- Команда `cat`: выведет содержимое файла.

```
cat filename.txt
```

- Перенаправление ввода: через `while` можно построчно читать файл.

```
while read line; do
```

```
    echo "$line"
```

```
done < filename.txt
```

10. Стандартные дескрипторы файлов.

В Unix-подобных системах существуют три стандартных дескриптора:

1. STDIN (стандартный ввод), дескриптор 0 – для ввода данных.

2. STDOUT (стандартный вывод), дескриптор 1 – для вывода данных.

3. STDERR (стандартный вывод ошибок), дескриптор 2 – для вывода сообщений об ошибках.

Пример перенаправления:

- `command > output.txt` — перенаправление вывода STDOUT в файл.

- `command 2> error.txt` — перенаправление вывода ошибок в файл.

- `command > output.txt 2>&1` — перенаправление и STDOUT, и STDERR в один файл.

11. Перенаправление вывода.

Перенаправление вывода позволяет направить результат команды в файл, другой процесс или переменную:

- `>` — перенаправляет стандартный вывод (STDOUT) в файл, перезаписывая его.

```
echo "Hello" > output.txt
```

- `>>` — перенаправляет стандартный вывод в файл, добавляя текст в конец файла.

```
echo "World" >> output.txt
```

- `2>` — перенаправляет стандартный вывод ошибок (STDERR) в файл.

```
ls non_existent_file 2> error.txt
```

- `&>` или `>&` — перенаправляет оба вывода (STDOUT и STDERR) в файл.

```
command &> output_and_errors.txt
```

- `<` — перенаправляет стандартный ввод (STDIN) из файла.

```
wc -l < input.txt
```

12. Подавление вывода.

Для того чтобы подавить вывод команды, можно перенаправить его в `null`, что в Unix-системах означает "черная дыра":

- Подавление стандартного вывода:

```
command > /dev/null
```

- Подавление стандартного вывода и ошибок:

```
command &> /dev/null
```

13. Отправка сигналов скриптам.

Для взаимодействия с процессами можно использовать сигналы:

- SIGINT (Ctrl+C) — прерывает выполнение процесса.
- SIGTERM — завершает процесс.
- SIGHUP — часто используется для перезапуска процессов.
- SIGKILL — завершает процесс немедленно.

Отправить сигнал можно с помощью команды `kill` или `pkill`:

```
kill -TERM <PID> # отправка SIGTERM процессу
```

```
pkill -HUP my_script.sh # отправка SIGHUP процессу с именем
```

В скриптах сигналы можно перехватывать через команду `trap`:

```
trap 'echo "Прервано!"' SIGINT
```

14. Использование функций.

Функции в Shell позволяют структурировать и переиспользовать код:

```
function greet() {  
    echo "Hello, $1!"  
}  
  
greet "User"
```

Функции позволяют передавать аргументы и возвращать значения.

Например:

```
function add() {  
    result=$(( $1 + $2 ))  
    echo $result  
}  
  
sum=$(add 3 5)
```

```
echo "Sum: $sum"
```

15. Обработка текстов (чтение, выбор, вставка, замена данных).

Для обработки текста используются команды:

- `cat` — отображает содержимое файла.
- `grep` — ищет строки, содержащие определенное выражение.

```
grep "pattern" file.txt
```

- `sed` — редактирует текст по шаблонам. Например, замена:

```
sed 's/old/new/g' file.txt
```

- `awk` — выбирает и обрабатывает текст по строкам и полям. Например,

вывод второго столбца:

```
awk '{print $2}' file.txt
```

- `cut` — выбирает столбцы или символы.

```
cut -d ':' -f 1 /etc/passwd
```

16. Отправка сообщений в терминал пользователя.

Команда `echo` может выводить сообщения в текущий терминал. Для отправки сообщения конкретному пользователю в Unix используют команду `write`:

```
write username < message.txt
```

17. BASH и SHELL – синонимы?

Shell — это общее название для интерфейса командной строки в Unix-подобных системах, где пользователи могут запускать команды и скрипты. Существует несколько различных оболочек (интерпретаторов команд), таких как:

Bash — одна из наиболее популярных оболочек, используемая в большинстве Linux-систем по умолчанию.

Zsh, Ksh, Tcsh и другие — альтернативные оболочки с различными возможностями.

Таким образом, Bash — это конкретная реализация Shell с расширенными возможностями, а Shell — это более общее понятие.

18. PowerShell в операционных системах семейства Windows: назначение и особенности.

PowerShell — это мощная оболочка командной строки и язык сценариев, разработанный Microsoft. Основные особенности PowerShell:

Объектно-ориентированность: в отличие от Bash, PowerShell оперирует не текстовыми строками, а объектами .NET, что позволяет передавать сложные данные между командами.

Кроссплатформенность: PowerShell Core работает не только на Windows, но и на Linux и macOS.

Модульная структура: PowerShell использует модули для расширения возможностей, например, модули для администрирования Active Directory, Azure и других систем.

Сильная интеграция с Windows: PowerShell позволяет управлять системными настройками, реестром и службами Windows.

Вывод

В ходе выполнения лабораторной работы были изучены основные возможности языка программирования высокого уровня Shell, получены навыки написания и использования скриптов.