

# **Лабораторная работа №10.**

## **Программирование в командном процессоре ОС UNIX**

---

### **1. Титульный лист**

---

- Номер лабораторной работы: **10**
  - ФИО студента: Шилов Александр Ильич
- 

### **2. Формулировка цели работы**

---

Цель работы: Изучение принципов программирования в командном процессоре Unix, создание скриптов для автоматизации задач и обработки данных.

---

### **3. Описание результатов выполнения задания**

---

#### **Задание 1: Скрипт для резервного копирования**

- Код скрипта:

```
#!/bin/bash

# Скрипт для резервного копирования файлов

# Проверка наличия аргументов
if [ $# -lt 2 ]; then
    echo "Использование: $0 <исходный_каталог> <каталог_назначения>"
    exit 1
fi

# Получение аргументов
SOURCE_DIR=$1
DEST_DIR=$2
BACKUP_DATE=$(date +%Y%m%d_%H%M%S)
ARCHIVE_NAME=
```

```

"backup_{BACKUP_DATE}.tar.gz"

# Проверка существования исходного каталога
if [ ! -d "$SOURCE_DIR" ]; then
    echo "Ошибка: Исходный каталог '$SOURCE_DIR' не существует."
    exit 1
fi

# Создание каталога назначения, если он не существует
if [ ! -d "$DEST_DIR" ]; then
    mkdir -p "$DEST_DIR"
    if [ $? -ne 0 ]; then
        echo "Ошибка: Не удалось создать каталог назначения '$DEST_DIR'."
        exit 1
    fi
    echo "Создан каталог назначения: $DEST_DIR"
fi

# Создание архива
echo "Создание резервной копии из '$SOURCE_DIR' в '$DEST_DIR/$ARCHIVE_NAME'."
tar -czf "$DEST_DIR/$ARCHIVE_NAME" -C "$SOURCE_DIR" .

# Проверка успешности создания архива
if [ $? -eq 0 ]; then
    echo "Резервное копирование успешно завершено."
    echo "Архив сохранен как: $DEST_DIR/$ARCHIVE_NAME"
    echo "Размер архива: $(du -h "$DEST_DIR/$ARCHIVE_NAME" | cut -f1)"
else
    echo "Ошибка: Не удалось создать архив."
    exit 1
fi

exit 0

```

• Результат выполнения:

```
$ ./backup.sh test_source test_backup
```

```
Создание резервной копии из 'test_source' в 'test_backup/backup_20250506'
Резервное копирование успешно завершено.
Архив сохранен как: test_backup/backup_20250506_052140.tar.gz
Размер архива: 4.0K
```

## Задание 2: Командный файл для обработки аргументов

- Код скрипта:

```
#!/bin/bash

# Скрипт для обработки аргументов командной строки

echo "Общее количество аргументов: $#"
echo "Имя скрипта: $0"

# Вывод всех аргументов
echo "Все аргументы: $@"

# Вывод аргументов по отдельности
echo "Аргументы по отдельности:"
count=1
for arg in "$@"; do
    echo "Аргумент $count: $arg"
    count=$((count + 1))
done

# Вывод аргументов в обратном порядке
echo "Аргументы в обратном порядке:"
for (( i=$#; i>0; i-- )); do
    echo "Аргумент $i: ${!i}"
done

exit 0
```

- Пример вывода:

```
$ ./args.sh
one two three four
Общее количество аргументов: 4
Имя скрипта: ./args.sh
Все аргументы: one two three four
Аргументы по отдельности:
Аргумент 1: one
Аргумент 2: two
Аргумент 3: three
Аргумент 4: four
Аргументы в обратном порядке:
Аргумент 4: four
Аргумент 3: three
Аргумент 2: two
Аргумент 1: one
```

### Задание 3: Аналог команды ls

- Код скрипта:

```
#!/bin/bash

# Скрипт-аналог команды ls

# Функция для вывода справки
show_help() {
    echo "Использование: $0 [опции] [каталог]"
    echo "Опции:"
    echo "  -a      Показать все файлы, включая скрытые"
    echo "  -l      Подробный формат вывода"
    echo "  -h      Показать эту справку"
    exit 0
}

# Инициализация переменных
show_hidden=0
long_format=0
directory=". "
```

```
# Обработка опций
while getopts "alh" opt; do
    case $opt in
        a) show_hidden=1 ;;
        l) long_format=1 ;;
        h) show_help ;;
        \?) echo "Неизвестная опция: -$OPTARG" >&2; exit 1 ;;
    esac
done

# Сдвиг обработанных опций
shift $((OPTIND - 1))

# Если указан каталог, используем его
if [ $# -gt 0 ]; then
    directory="$1"
fi

# Проверка существования каталога
if [ ! -d "$directory" ]; then
    echo "Ошибка: Каталог '$directory' не существует."
    exit 1
fi

# Получение списка файлов
if [ $show_hidden -eq 1 ]; then
    files=($(ls -a "$directory"))
else
    files=($(ls "$directory"))
fi

# Вывод результатов
echo "Содержимое каталога: $directory"
echo "-----"

for file in "${files[@]}"; do
    # Пропускаем . и .. если не показываем скрытые файлы
    if [ $show_hidden -eq 0 ] && [[ "$file" == "." || "$file" == ".." ]]
        continue
    fi
    if [ $long_format -eq 1 ]; then
        echo -e "\t$file"
    else
        echo $file
    fi
done
```

```

fi

full_path="$directory/$file"

if [ $long_format -eq 1 ]; then
    # Тип файла
    if [ -d "$full_path" ]; then
        type="d"
    elif [ -L "$full_path" ]; then
        type="l"
    else
        type="-"
    fi

    # Права доступа
    permissions=""
    if [ -r "$full_path" ]; then permissions="${permissions}r"; fi
    if [ -w "$full_path" ]; then permissions="${permissions}w"; fi
    if [ -x "$full_path" ]; then permissions="${permissions}x"; fi

    # Размер и дата модификации
    size=$(du -h "$full_path" 2>/dev/null | cut -f1)
    mod_date=$(date -r "$full_path" "+%Y-%m-%d %H:%M")

    echo "$type$permissions $size $mod_date $file"
else
    echo "$file"
fi

done

exit 0

```

## Задание 4: Подсчет файлов по форматам

- Код скрипта:

```
#!/bin/bash
```

```
# Скрипт для подсчета файлов по форматам

# Проверка наличия аргумента
if [ $# -lt 1 ]; then
    echo "Использование: $0 <каталог>"
    exit 1
fi

# Получение аргумента
directory="$1"

# Проверка существования каталога
if [ ! -d "$directory" ]; then
    echo "Ошибка: Каталог '$directory' не существует."
    exit 1
fi

echo "Подсчет файлов по форматам в каталоге: $directory"
echo "-----"

# Получение списка всех файлов рекурсивно
files=$(find "$directory" -type f | sort)

# Инициализация ассоциативного массива для подсчета
declare -A formats_count

# Подсчет файлов по расширениям
for file in $files; do
    # Получение расширения файла
    extension="${file##*.}"

    # Если файл не имеет расширения, считаем его как "без расширения"
    if [ "$extension" = "$file" ]; then
        extension="без расширения"
    else
        extension=".${extension}"
    fi

    # Увеличиваем счетчик для данного расширения
    ((formats_count[$extension]++))
done

# Выводим результат
for format in "${!formats_count[@]}"; do
    echo "$format: ${formats_count[$format]}"
done
```

```

if [ -z "${formats_count[$extension]}" ]; then
    formats_count[$extension]=1
else
    formats_count[$extension]=$((${formats_count[$extension]} + 1))
fi
done

# Вывод результатов
echo "Формат | Количество файлов"
echo "-----"
for format in "${!formats_count[@]}"; do
    count=${formats_count[$format]}

    # Правильное склонение слова "файл"
    if [ $count -eq 1 ]; then
        word="файл"
    elif [ $count -ge 2 ] && [ $count -le 4 ]; then
        word="файла"
    else
        word="файлов"
    fi

    echo "$format: ${formats_count[$format]} $word"
done

# Общее количество файлов
total=0
for count in "${formats_count[@]}"; do
    total=$((total + count))
done

echo "-----"
echo "Всего: $total файлов"

exit 0

```

- Пример вывода:

```
$ ./countfiles.sh .  
Подсчет файлов по форматам в каталоге: .
```

---

#### Формат | Количество файлов

---

```
.txt: 3 файла  
.sh: 4 файла  
.md: 1 файл  
.gz: 1 файл
```

---

```
Всего: 9 файлов
```

---

## 4. Выводы

---

1. Я освоил основы создания скриптов на Bash, включая работу с переменными, условными операторами и циклами.
2. Научился работать с аргументами командной строки и обрабатывать их в скриптах.
3. Изучил механизм архивации файлов в Unix с помощью команды tar.
4. Освоил работу с файловой системой: проверку существования файлов и каталогов, получение информации о файлах.
5. Научился использовать ассоциативные массивы для подсчета и группировки данных.
6. Понял принципы создания пользовательских аналогов стандартных команд Unix.

---

## 5. Ответы на контрольные вопросы

---

### 1. Что такое командная оболочка?

Командная оболочка (shell) - это программа, которая предоставляет интерфейс между пользователем и операционной системой. Она интерпретирует команды пользователя и передает их ядру ОС для выполнения. Оболочка также предоставляет язык программирования для написания скриптов.

## **2. Что такое POSIX?**

POSIX (Portable Operating System Interface) - это набор стандартов, определяющих интерфейс между операционной системой и прикладными программами. Эти стандарты обеспечивают совместимость между различными UNIX-подобными операционными системами, позволяя программам работать на разных платформах без изменений.

## **3. Каковы основные возможности командных оболочек?**

Основные возможности командных оболочек включают:

- Выполнение команд и программ
- Перенаправление ввода/вывода
- Конвейерная обработка данных (**pipes**)
- Управление заданиями (**jobs**)
- Подстановка имен файлов (**wildcards**)
- Программирование с использованием переменных, условий, циклов
- Обработка аргументов командной строки
- Автодополнение команд и имен файлов
- История команд

## **4. Что такое переменная окружения?**

Переменная окружения - это динамическая переменная в операционной системе, которая содержит информацию, используемую одной или несколькими программами. Переменные окружения определяют поведение системы и программ, например, **PATH** определяет каталоги для поиска исполняемых файлов, **HOME** указывает на домашний каталог пользователя.

## **5. Каково назначение команд **set** и **unset**?**

- **set** используется для установки или отображения переменных оболочки. Без аргументов выводит список всех переменных. С опциями может изменять поведение оболочки.
- **unset** используется для удаления переменных или функций из окружения оболочки.

## 6. Как определяются функции в языке программирования bash?

Функции в bash определяются следующим образом:

```
function_name() {  
    # команды  
    return value # необязательно  
}
```

или

```
function function_name {  
    # команды  
    return value # необязательно  
}
```

## 7. Каково назначение команд let и read?

- `let` выполняет арифметические операции над переменными, например:  
`let "a = 5 + 3".`
- `read` считывает строку из стандартного ввода и присваивает ее переменным, например: `read name` - считывает строку и сохраняет в переменной `name`.

## 8. Что такое позиционные параметры?

Позиционные параметры - это специальные переменные в скриптах, которые содержат аргументы командной строки. `$0` содержит имя скрипта, `$1` - первый аргумент, `$2` - второй и т.д. `$#` содержит количество аргументов, `$@` и `$*` содержат все аргументы.

## 9. Как осуществляется возврат значений из функций?

Функции в bash возвращают значения через:

- Код возврата (0-255) с помощью команды `return`
- Вывод результата в `stdout` с помощью `echo` или `printf`, который можно захватить при вызове функции
- Изменение глобальных переменных внутри функции

## 10. Что такое локальные переменные?

Локальные переменные - это переменные, объявленные с ключевым словом `local` внутри функции. Они видны только внутри функции, в которой объявлены, и не влияют на одноименные переменные вне функции.

## 11. Что такое рекурсия и как она реализована в языке программирования bash?

Рекурсия - это процесс, при котором функция вызывает сама себя. В bash рекурсия реализуется так же, как и в других языках - функция содержит вызов самой себя с измененными параметрами и условие выхода из рекурсии:

```
factorial() {  
    if [ $1 -le 1 ]; then  
        echo 1  
    else  
        local temp=$(factorial $(( $1 - 1 )))  
        echo $(( $1 * temp ))  
    fi  
}
```

## 12. Что такое командная подстановка?

Командная подстановка - это механизм, позволяющий использовать вывод одной команды как аргумент для другой команды или для присваивания переменной. Реализуется с помощью обратных кавычек `command` или синтаксиса `$(command)`.

## 13. Как создаются и используются массивы в языке программирования bash?

Массивы в bash создаются и используются следующим образом:

```
# Создание массива
array=("element1" "element2" "element3")

# Доступ к элементам
echo ${array[0]} # первый элемент
echo ${array[@]} # все элементы

# Добавление элемента
array+=("element4")

# Количество элементов
echo ${#array[@]}
```

Ассоциативные массивы (с версии Bash 4.0):

```
# Объявление ассоциативного массива
declare -A assoc_array

# Заполнение
assoc_array["key1"]="value1"
assoc_array["key2"]="value2"

# Доступ к элементам
echo ${assoc_array["key1"]}

# Все ключи
echo ${!assoc_array[@]}
```

## 14. Что такое регулярные выражения?

Регулярные выражения - это шаблоны, используемые для поиска и обработки текста. Они представляют собой последовательность символов, определяющую правила поиска. В Unix регулярные выражения используются во многих утилитах, таких как grep, sed, awk, и в языках программирования, включая bash.

## 15. Как осуществляется обработка ошибок в языке программирования bash?

Обработка ошибок в bash осуществляется следующими способами:

- Проверка кода возврата команд с помощью `$?`
- Использование условных операторов для проверки успешности выполнения команд
- Использование конструкции `set -e` для автоматического завершения скрипта при ошибке
- Использование конструкции `trap` для перехвата сигналов и выполнения действий при ошибках
- Перенаправление вывода ошибок с помощью `2>` или `2>&1`