

Лабораторная работа 13

Программирование в командном процессоре ОС UNIX. Ветвления и циклы

Головина Мария Игоревна

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	7
4	Выполнение лабораторной работы	10
5	Ответы на контрольные вопросы	19
6	Выводы	24
	Список литературы	25

Список иллюстраций

4.1	Скрипт №1	10
4.2	Запуск	11
4.3	Проверка	11
4.4	Скрипт №2	12
4.5	Скрипт №2	12
4.6	Запуск	13
4.7	Скрипт №3	14
4.8	Запуск	14
4.9	Запуск	15
4.10	Скрипт №4	15
4.11	Запуск	16
4.12	Модифицированный скрипт №4	17
4.13	Запуск	18

Список таблиц

1 Цель работы

Изучить основы программирования в оболочке ОС UNIX. Научится писать более сложные командные файлы с использованием логических управляющих конструкций и циклов.

2 Задание

1. Используя команды `getopts` `grep`, написать командный файл, который анализирует командную строку, а затем ищет в указанном файле нужные строки.
2. Написать на языке Си программу, которая вводит число и определяет, является ли оно больше нуля, меньше нуля или равно нулю. Затем программа завершается с помощью функции `exit(n)`, передавая информацию о коде завершения в оболочку.
3. Написать командный файл, создающий указанное число файлов, пронумерованных последовательно от 1 до N.
4. Написать командный файл, который с помощью команды `tar` запаковывает в архив все файлы в указанной директории. Модифицировать его так, чтобы запаковывались только те файлы, которые были изменены менее недели тому назад.
5. Ответить на контрольные вопросы.

3 Теоретическое введение

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; • C-оболочка (или csh) — надстройка на оболочке Борна, использующая C-подобный синтаксис команд с возможностью сохранения истории выполнения команд; • оболочка Корна (или ksh) — напоминает оболочку C, но операторы управления программой совместимы с операторами оболочки Борна; • BASH — сокращение от Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек C и Корна (разработка компании Free Software Foundation). 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.

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.

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.

Целые числа можно записывать как последовательность цифр или в любом базовом формате типа `radix#number`, где `radix` (основание системы счисления) — любое число не более 26. Для большинства команд используются следующие основания систем исчисления: 2 (двоичная), 8 (восьмеричная) и 16 (шестнадцатеричная). Простейшими математическими выражениями являются сложение (+), вычитание (-), умножение (*), целочисленное деление (/) и целочисленный остаток от деления (%).

Команда `let` берет два операнда и присваивает их переменной. Положительным моментом команды `let` можно считать то, что для идентификации переменной ей не нужен знак доллара.

Последовательность команд может быть помещена в текстовый файл. Такой файл называется командным.

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.

Весьма необходимой при программировании является команда `getopts`, которая осуществляет синтаксический анализ командной строки, выделяя флаги, и используется для объявления переменных.

Флаги — это опции командной строки, обычно помеченные знаком минус.

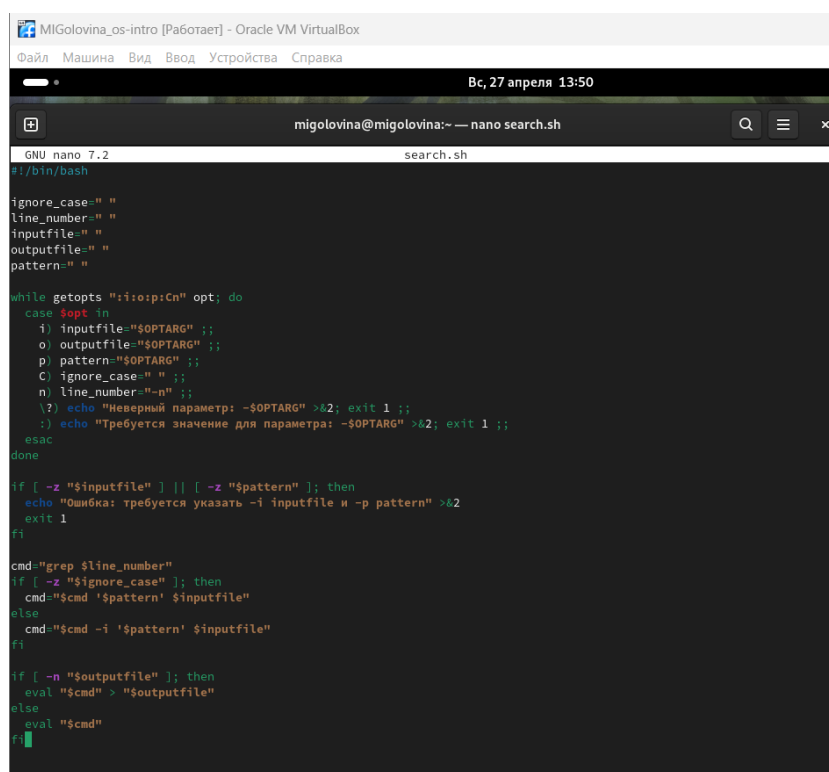
Строка опций `option-string` — это список возможных букв и чисел соответствующего флага. Если ожидается, что некоторый флаг будет сопровождаться некоторым аргументом, то за символом, обозначающим этот флаг, должно следовать двоеточие.

Часто бывает необходимо обеспечить проведение каких-либо действий циклически и управление дальнейшими действиями в зависимости от результатов проверки некоторого условия. Для решения подобных задач язык программирования `bash` предоставляет возможность использовать такие управляющие конструкции, как `for`, `case`, `if` и `while`.

Более подробно о Linux см. в [1-7]

4 Выполнение лабораторной работы

1. Используя команды `getopts` `grep`, написала командный файл, который анализирует командную строку, а затем ищет в указанном файле нужные строки (рис. 4.1).



```
GNU nano 7.2 search.sh
#!/bin/bash

ignore_case=" "
line_number=" "
inputfile=" "
outputfile=" "
pattern=" "

while getopts "i:o:p:C:n" opt; do
  case $opt in
    i) inputfile="$OPTARG" ;;
    o) outputfile="$OPTARG" ;;
    p) pattern="$OPTARG" ;;
    C) ignore_case=" " ;;
    n) line_number="n" ;;
    ?) echo "Неверный параметр: -$OPTARG" >&2; exit 1 ;;
    :) echo "Требуется значение для параметра: -$OPTARG" >&2; exit 1 ;;
  esac
done

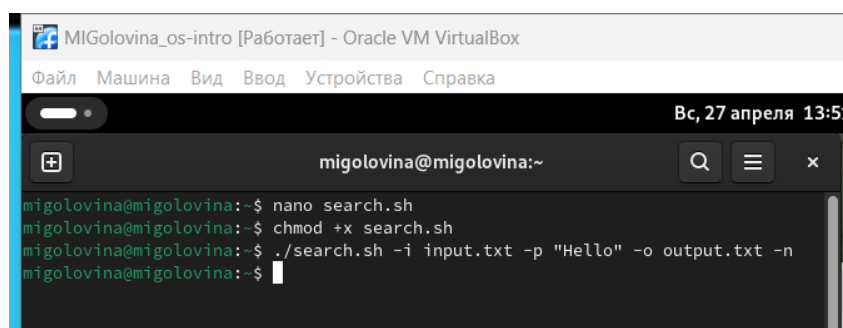
if [ -z "$inputfile" ] || [ -z "$pattern" ]; then
  echo "Ошибка: требуется указать -i inputfile и -p pattern" >&2
  exit 1
fi

cmd="grep $line_number"
if [ -z "$ignore_case" ]; then
  cmd="$cmd '$pattern' $inputfile"
else
  cmd="$cmd -i '$pattern' $inputfile"
fi

if [ -n "$outputfile" ]; then
  eval "$cmd" > "$outputfile"
else
  eval "$cmd"
fi
```

Рис. 4.1: Скрипт №1

2. Запустила скрипт №1 (рис. 4.2).



The screenshot shows a terminal window titled "MIGolovina_os-intro [Работает] - Oracle VM VirtualBox". The terminal prompt is "migolovina@migolovina:~". The user has entered the following commands: `nano search.sh`, `chmod +x search.sh`, and `./search.sh -i input.txt -p "Hello" -o output.txt -n`. The output of the script is not yet visible.

Рис. 4.2: Запуск

3. Проверила правильность работы скрипта №1 (рис. 4.3).

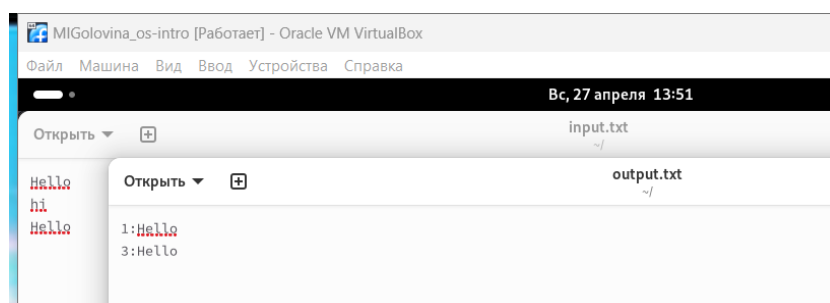


Рис. 4.3: Проверка

4. Написала на языке Си программу, которая вводит число и определяет, является ли оно больше нуля, меньше нуля или равно нулю (рис. 4.4).

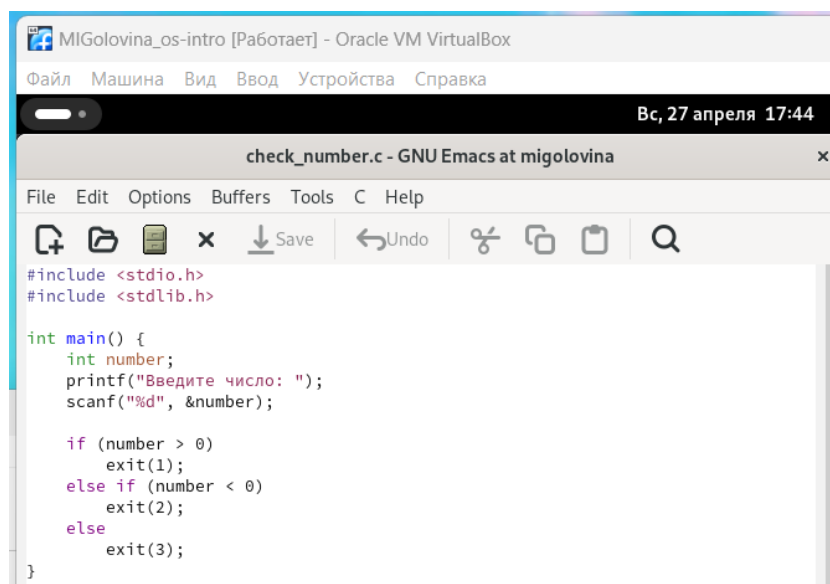
A screenshot of a GNU Emacs editor window titled 'check_number.c - GNU Emacs at migolovina'. The window shows a C program that includes `<stdio.h>` and `<stdlib.h>`. The `main` function prompts the user to enter a number using `printf` and `scanf`. It then uses `if` and `else if` statements to check if the number is greater than, less than, or equal to zero, and calls `exit(1)`, `exit(2)`, or `exit(3)` accordingly. The Emacs interface includes a menu bar with 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C', and 'Help', and a toolbar with icons for file operations and editing. The status bar at the top right shows 'Вс, 27 апреля 17:44'.

Рис. 4.4: Скрипт №2

5. Затем написала программу, которая завершается с помощью функции `exit(n)`, передавая информацию в о коде завершения в оболочку (рис. 4.5).

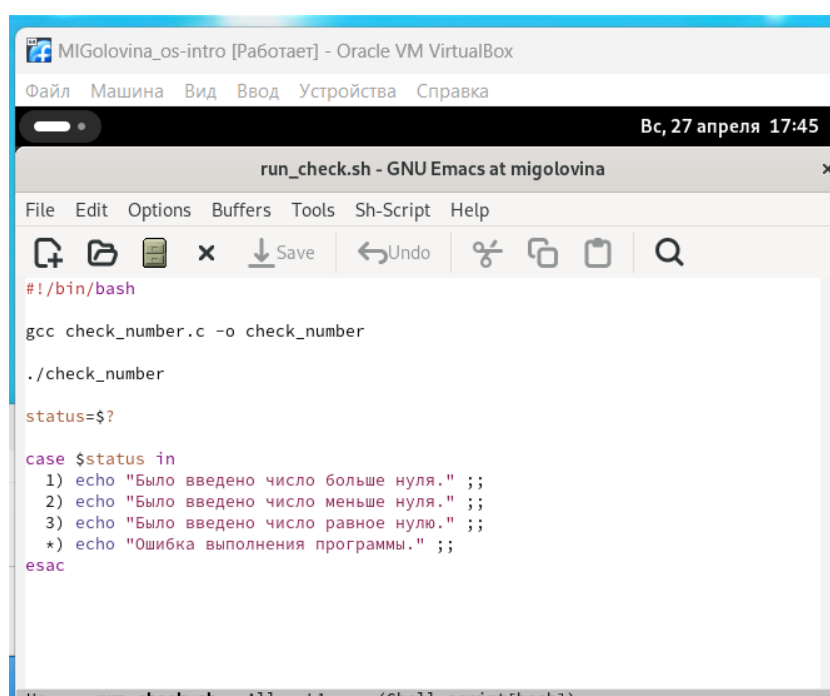
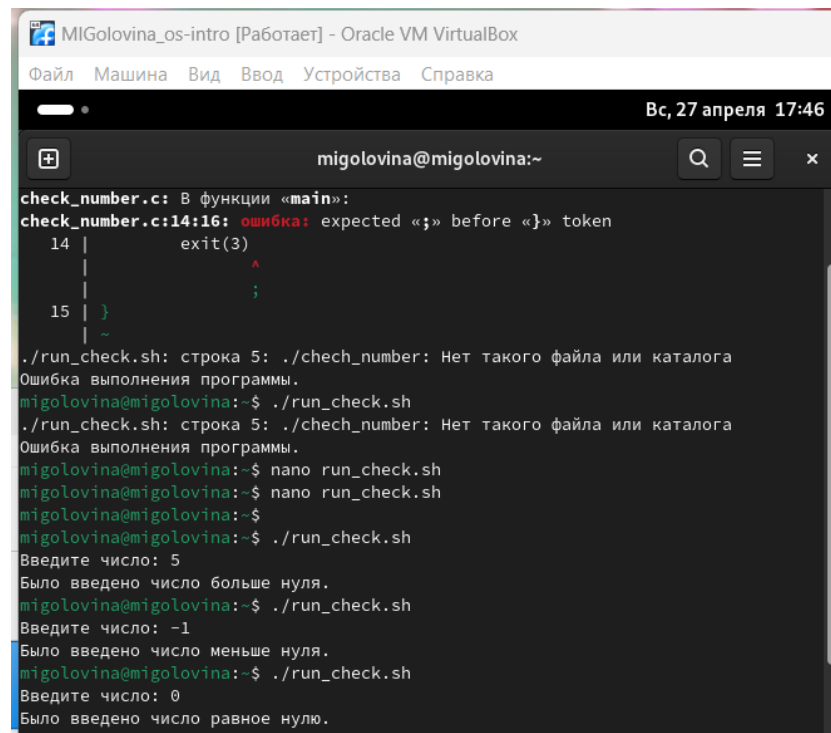
A screenshot of a GNU Emacs editor window titled 'run_check.sh - GNU Emacs at migolovina'. The window shows a shell script that compiles the C program from the previous figure using `gcc` and runs it using `./check_number`. It then checks the exit status using `status=$?` and uses a `case` statement to print messages for each status: 1) 'Было введено число больше нуля.', 2) 'Было введено число меньше нуля.', 3) 'Было введено число равное нулю.', and *) 'Ошибка выполнения программы.'. The script ends with `esac`. The Emacs interface is similar to the previous one, but the menu bar includes 'Sh-Script' instead of 'C'. The status bar at the top right shows 'Вс, 27 апреля 17:45'.

Рис. 4.5: Скрипт №2

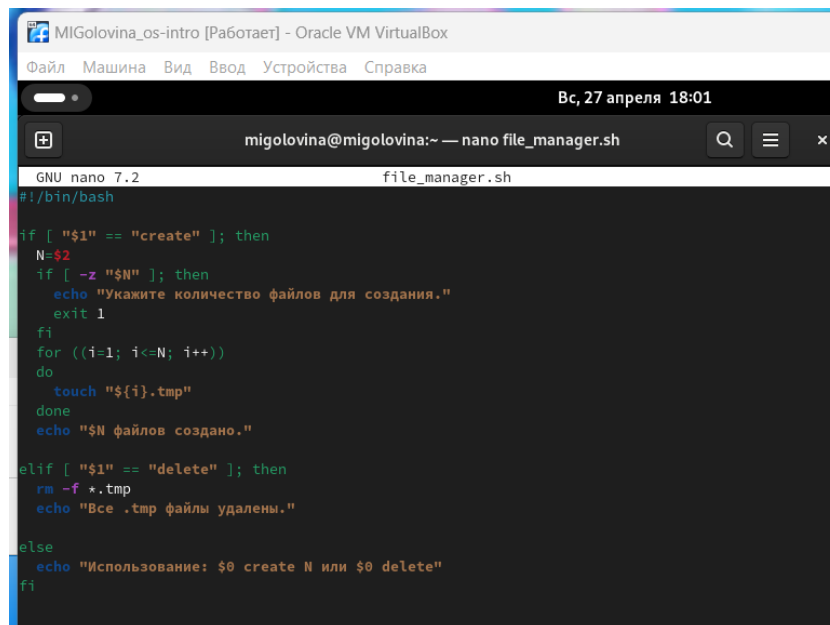
6. Запустила скрипт №2 (рис. 4.6).



```
MIGolovina_os-intro [Работает] - Oracle VM VirtualBox
Файл  Машина  Вид  Ввод  Устройства  Справка
Вс, 27 апреля 17:46
migolovina@migolovina:~
check_number.c: В функции «main»:
check_number.c:14:16: ошибка: expected «;» before «}» token
14 |         exit(3)
    |                ^
15 |     }
    |     ~
./run_check.sh: строка 5: ./check_number: Нет такого файла или каталога
Ошибка выполнения программы.
migolovina@migolovina:~$ ./run_check.sh
./run_check.sh: строка 5: ./check_number: Нет такого файла или каталога
Ошибка выполнения программы.
migolovina@migolovina:~$ nano run_check.sh
migolovina@migolovina:~$ nano run_check.sh
migolovina@migolovina:~$
migolovina@migolovina:~$ ./run_check.sh
Введите число: 5
Было введено число больше нуля.
migolovina@migolovina:~$ ./run_check.sh
Введите число: -1
Было введено число меньше нуля.
migolovina@migolovina:~$ ./run_check.sh
Введите число: 0
Было введено число равно нулю.
```

Рис. 4.6: Запуск

7. Написала командный файл, создающий указанное число файлов, пронумерованных последовательно от 1 до N (рис. 4.7).



```
GNU nano 7.2 file_manager.sh
#!/bin/bash

if [ "$1" == "create" ]; then
    N=$2
    if [ -z "$N" ]; then
        echo "Укажите количество файлов для создания."
        exit 1
    fi
    for ((i=1; i<=N; i++))
    do
        touch "${i}.tmp"
    done
    echo "$N файлов создано."
elif [ "$1" == "delete" ]; then
    rm -f *.tmp
    echo "Все .tmp файлы удалены."
else
    echo "Использование: $0 create N или $0 delete"
fi
```

Рис. 4.7: Скрипт №3

8. Запустила скрипт №3 с созданием файлов (рис. 4.8).

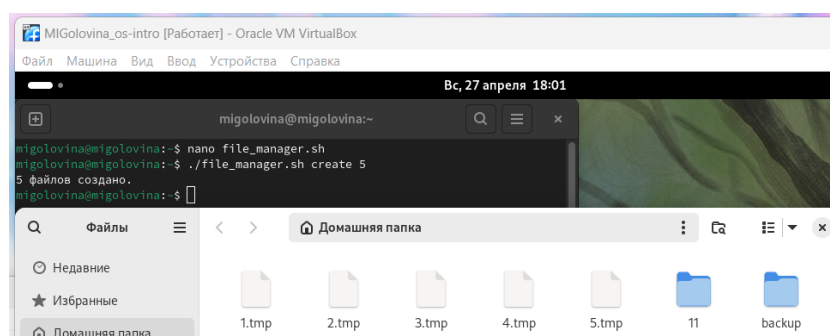


Рис. 4.8: Запуск

9. Запустила скрипт №3 с удалением файлов (рис. 4.9).

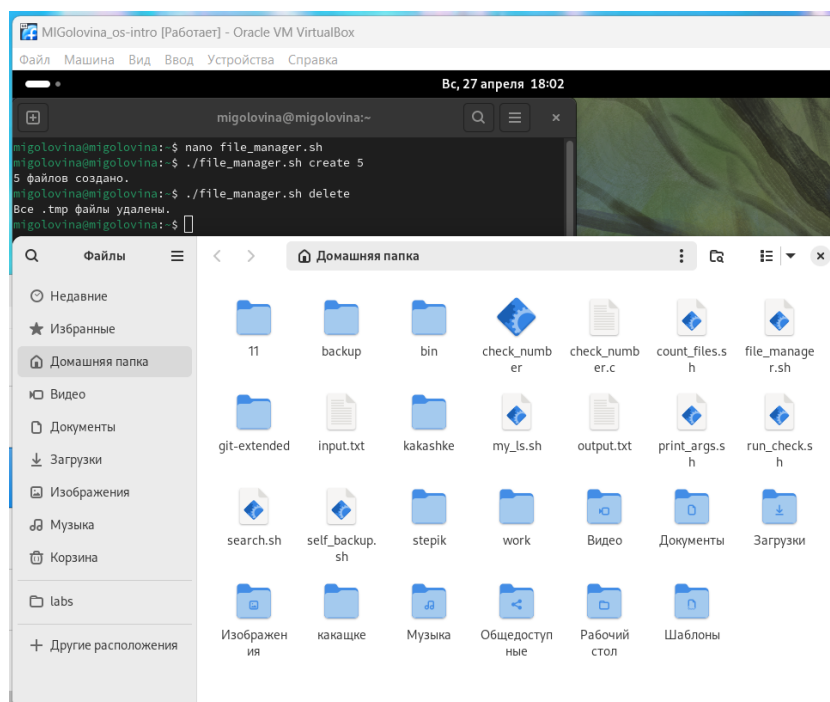


Рис. 4.9: Запуск

10. Написала командный файл, который с помощью команды tar запаковывает в архив все файлы в указанной директории (рис. 4.10).

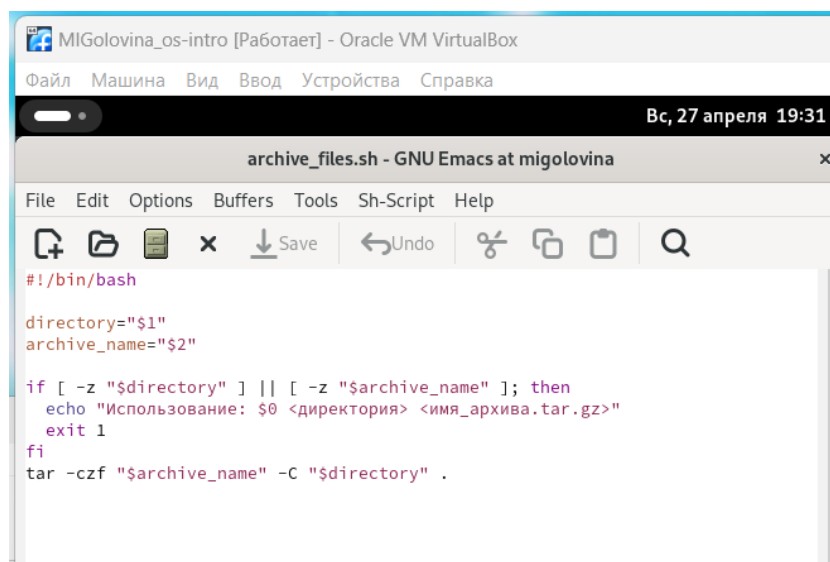


Рис. 4.10: Скрипт №4

11. Проверила правильность работы скрипта №4 (рис. 4.11).

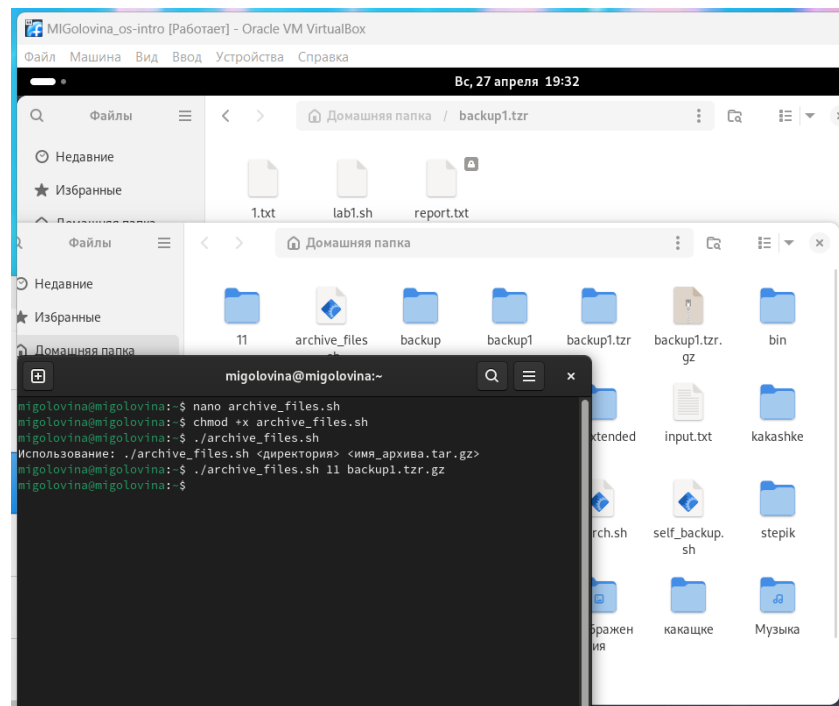
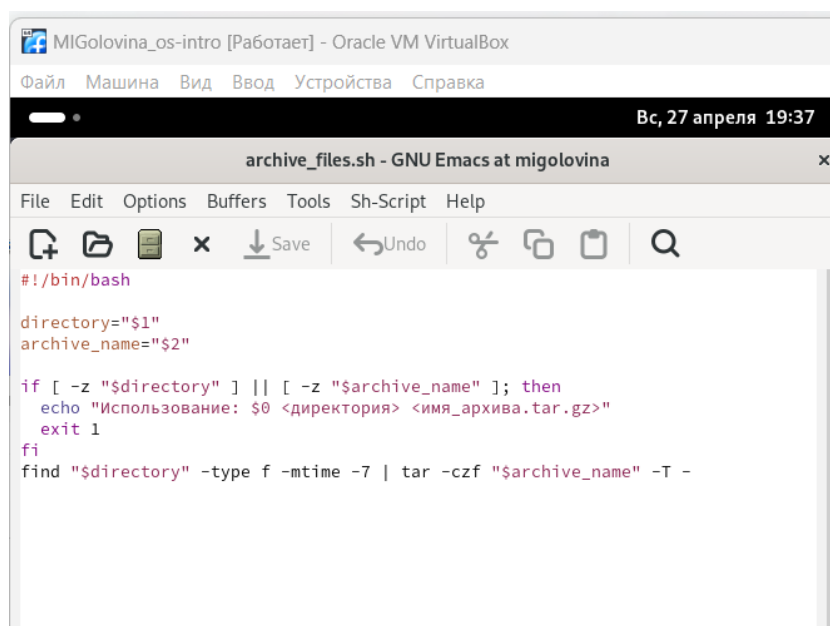


Рис. 4.11: Запуск

12. Модифицировала его так, чтобы запаковывались только те файлы, которые были изменены менее недели тому назад (рис. 4.12).



```
#!/bin/bash

directory="$1"
archive_name="$2"

if [ -z "$directory" ] || [ -z "$archive_name" ]; then
    echo "Использование: $0 <директория> <имя_архива.tar.gz>"
    exit 1
fi

find "$directory" -type f -mtime -7 | tar -czf "$archive_name" -T -
```

Рис. 4.12: Модифицированный скрипт №4

13. Проверила правильность работы модифицированного скрипта №4 (рис. 4.13).

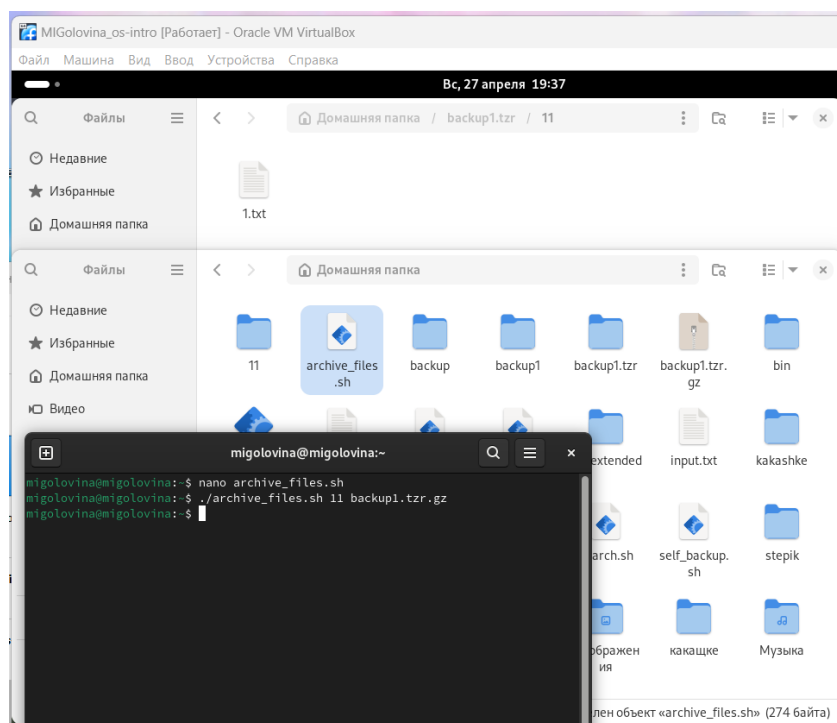


Рис. 4.13: Запуск

5 Ответы на контрольные вопросы

1. Каково предназначение команды `getopts`?

Команда `getopts` в Bash используется для разбора и обработки опций (аргументов) командной строки, переданных скрипту или функции. Она позволяет удобно управлять флагами и параметрами, которые пользователь может указать при запуске скрипта.

Вот основные аспекты предназначения и использования `getopts`:

1. Обработка опций `getopts` позволяет обрабатывать короткие (один символ) и длинные (более одного символа) опции, что делает интерфейс вашего скрипта более удобным и понятным.

2. Синтаксис

Основной синтаксис команды выглядит следующим образом:

`getopts "options" variable`

`options`: строка, содержащая допустимые опции. Если опция принимает аргумент, после нее указывается двоеточие (:).

`variable`: переменная, в которую будет записываться текущая опция.

3. Переменные

`$OPTARG`: содержит значение аргумента для опции, если опция требует аргумента.

`$OPTIND`: индекс следующего аргумента, который будет обработан. Полезно, если вы хотите продолжить обработку аргументов после завершения `getopts`.

4. Преимущества использования `getopts`

Упрощение кода: Позволяет легко обрабатывать множество опций без необходимости писать сложные конструкции.

Универсальность: Поддерживает как обязательные, так и необязательные аргументы для опций.

Стандартизация: Использование `getopts` делает скрипт более стандартным и понятным для пользователей.

2. Какое отношение метасимволы имеют к генерации имён файлов?

Метасимволы играют ключевую роль в генерации имен файлов в оболочке Unix/Linux, включая Bash. Они позволяют использовать шаблоны и упрощают работу с файлами и каталогами. Вот основные метасимволы и их влияние на генерацию имен файлов:

1. Звездочка (*)

Звездочка представляет собой любой набор символов, включая пустую строку. Она позволяет выбирать файлы и каталоги с любыми именами.

2. Вопросительный знак (?)

Вопросительный знак соответствует любому одному символу. Это полезно, когда нужно указать точное количество символов.

3. Квадратные скобки ([])

Квадратные скобки позволяют указать диапазон символов или конкретные символы, которые могут находиться на данном месте.

4. Фигурные скобки ({})

Фигурные скобки позволяют создавать наборы имен файлов, что удобно для генерации нескольких файлов с похожими именами.

5. Обратная косая черта ()

Обратная косая черта используется для экранирования метасимволов, позволяя использовать их как обычные символы.

3. Какие операторы управления действиями вы знаете?

В Bash и других языках программирования операторы управления действиями (или операторы управления потоком) позволяют изменять порядок выполнения инструкций в программе.

Основные операторы, которые вы можете использовать в Bash:

1. Условные операторы

if: Позволяет выполнять блок кода, если условие истинно.

case: Используется для проверки переменной на соответствие нескольким значениям.

2. Циклы

for: Позволяет выполнять блок кода для каждого элемента в списке.

while: Выполняет блок кода, пока условие истинно.

until: Выполняет блок кода, пока условие ложно.

3. Операторы перехода

break: Прерывает выполнение цикла.

continue: Пропускает текущую итерацию цикла и переходит к следующей.

4. Команды управления

exit: Завершает выполнение скрипта.

exit 0 # 0 - код завершения (успех)

return: Завершает выполнение функции и возвращает значение.

return 1 # возвращает 1 как код ошибки

4. Какие операторы используются для прерывания цикла?

В Bash для прерывания выполнения циклов используются два основных оператора: break и continue.

1. Оператор break

Оператор break используется для немедленного выхода из цикла. Это означает, что когда break выполняется, выполнение цикла прекращается, и управление передается на следующую инструкцию после цикла.

2. Оператор continue

Оператор `continue` используется для пропуска текущей итерации цикла и перехода к следующей. Это означает, что если условие для `continue` истинно, оставшиеся команды в текущей итерации будут пропущены, и выполнение перейдет к следующей итерации цикла.

5. Для чего нужны команды `false` и `true`?

Команды `true` и `false` в Unix/Linux — это простые утилиты, которые всегда возвращают определённый код завершения, что делает их полезными в различных сценариях.

1. Команда `true`

Команда `true` всегда завершает выполнение с кодом 0, что в Unix/Linux означает успешное выполнение. Это может быть полезно в ситуациях, когда требуется команда, которая всегда “успешна”.

2. Команда `false`

Команда `false`, напротив, всегда завершает выполнение с кодом 1, что означает неуспех. Это может быть полезно для тестирования условий и обработки ошибок.

6. Что означает строка `if test -f man1.$s`, встреченная в командном файле?

Введенная строка означает условие существования файла `man1/i.$s`

7. Объясните различия между конструкциями `while` и `until`. Конструкции `while` и `until` в Bash (и других оболочках Unix/Linux) используются для создания циклов, но они работают немного по-разному.

1. Цикл `while`

Цикл `while` выполняет блок команд, пока заданное условие истинно (возвращает код 0). Если условие ложно (возвращает код 1), выполнение цикла прекращается.

2. Цикл `until`

Цикл `until`, наоборот, выполняет блок команд, пока заданное условие ложно (возвращает код 1). Как только условие становится истинным (возвращает код 0), выполнение цикла прекращается.

`while` используется, когда нужно продолжать выполнение, пока условие выполняется.

`until` используется, когда нужно продолжать выполнение, пока условие не выполняется.

6 Выводы

Я изучила основы программирования в оболочке ОС UNIX. Научилась писать более сложные командные файлы с использованием логических управляющих конструкций и циклов.

Список литературы

1. Dash, P. Getting Started with Oracle VM VirtualBox / P. Dash. – Packt Publishing Ltd, 2013. – 86 сс.
2. Colvin, H. VirtualBox: An Ultimate Guide Book on Virtualization with VirtualBox. VirtualBox / H. Colvin. – CreateSpace Independent Publishing Platform, 2015. – 70 сс.
3. Vugt, S. van. Red Hat RHCSA/RHCE 7 cert guide : Red Hat Enterprise Linux 7 (EX200 and EX300) : Certification Guide. Red Hat RHCSA/RHCE 7 cert guide / S. van Vugt. – Pearson IT Certification, 2016. – 1008 сс.
4. Робачевский, А. Операционная система UNIX / А. Робачевский, С. Немнюгин, О. Стесик. – 2-е изд. – Санкт-Петербург : БХВ-Петербург, 2010. – 656 сс.
5. Немет, Э. Unix и Linux: руководство системного администратора. Unix и Linux / Э. Немет, Г. Снайдер, Т.Р. Хейн, Б. Уэйли. – 4-е изд. – Вильямс, 2014. – 1312 сс.
6. Колисниченко, Д.Н. Самоучитель системного администратора Linux : Системный администратор / Д.Н. Колисниченко. – Санкт-Петербург : БХВ-Петербург, 2011. – 544 сс.
7. Robbins, A. Bash Pocket Reference / A. Robbins. – O'Reilly Media, 2016. – 156 сс.