

Отчёт по лабораторной работе №12

**Программирование в командном процессоре ОС UNIX. Расширенное
программирование**

Хусаинова Динара Айратовна

Содержание

| | | |
|---|------------------------|----|
| 1 | Цель работы | 5 |
| 2 | Теоретическое введение | 6 |
| 3 | Ход работы | 8 |
| 4 | Контрольные вопросы | 13 |
| 5 | Вывод | 18 |

Список иллюстраций

| | | |
|-----|--|----|
| 3.1 | Содержимое файла | 8 |
| 3.2 | Наша проверка | 9 |
| 3.3 | Содержимое файла | 10 |
| 3.4 | Предоставление прав доступа и проверка | 10 |
| 3.5 | Вывод информации о вводимой команде | 11 |
| 3.6 | Ввод команды, которой нет | 11 |
| 3.7 | Содержимое файла | 12 |
| 3.8 | Вывод случайных букв | 12 |

List of Tables

1 Цель работы

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

2 Теоретическое введение

Командные процессоры или оболочки – это программы, позволяющие пользователю взаимодействовать с компьютером. Их можно рассматривать как настоящие интерпретируемые языки, которые воспринимают команды пользователя и обрабатывают их. Поэтому командные процессоры также называют интерпретаторами команд. На языках оболочек можно писать программы и выполнять их подобно любым другим программам. UNIX обладает большим количеством оболочек. Наиболее популярными являются следующие четыре оболочки:

- оболочка Борна (Bourne) – первоначальная командная оболочка UNIX: базовый, но полный набор функций;
- C-оболочка – добавка университета Беркли к коллекции оболочек: она надстраивается над оболочкой Борна, используя C-подобный синтаксис команд, и сохраняет историю выполненных команд;
- оболочка Корна – 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).

Стандарт POSIX

POSIX (Portable Operating System Interface for Computer Environments) – интерфейс переносимой операционной системы для компьютерных сред. Представляет собой набор стандартов, подготовленных институтом инженеров по электронике и радиотехнике (IEEE), который определяет различные аспекты построения опе-

рационной системы. POSIX включает такие темы, как программный интерфейс, безопасность, работа с сетями и графический интерфейс. POSIX-совместимые оболочки являются будущим поколением оболочек UNIX и других ОС. Windows NT рекламируется как система, удовлетворяющая POSIX-стандартам.

POSIX-совместимые оболочки разработаны на базе оболочки Корна; фонд бесплатного программного обеспечения (Free Software Foundation) работает над тем, чтобы и оболочку BASH сделать POSIX-совместимой.

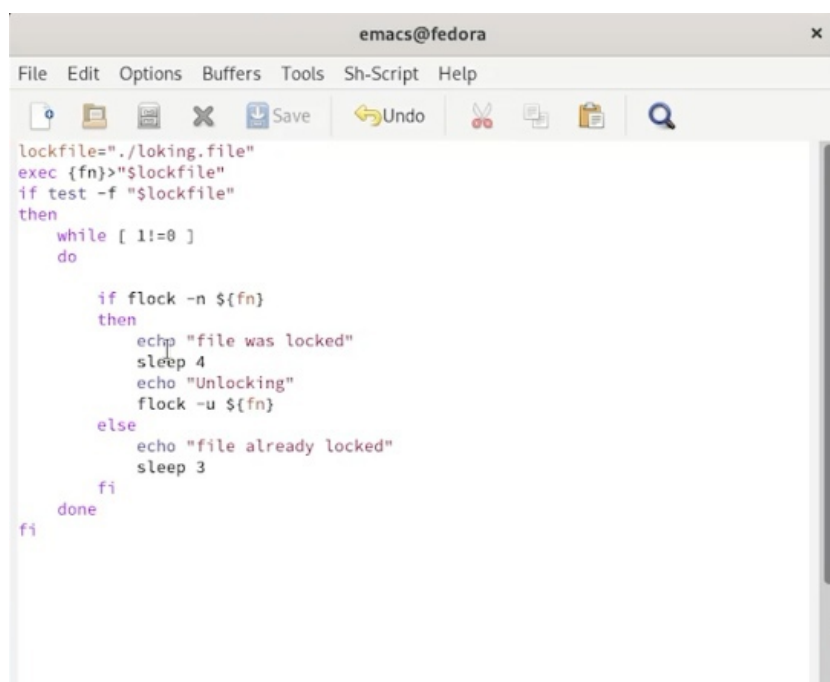
Рассмотрим основные элементы программирования в оболочке bash. В других оболочках большинство команд будет совпадать с описанными ниже.

Использование арифметических вычислений. Операторы let и read

Оболочка bash поддерживает встроенные арифметические функции. Команда let является показателем того, что последующие аргументы представляют собой выражение, это единичный терм (term), обычно подлежащее вычислению. Простейшее выражение целочисленный. Целые числа можно записывать как последовательность цифр или в любом базовом формате. Этот формат – radix#number, где radix (основание системы счисления) любое число не более 26. Для большинства команд основания систем счисления это – 2 (двоичная), 8 (восьмеричная) и 16 (шестнадцатеричная). Простейшими математическими выражениями являются сложение (+), вычитание (-), умножение (*), целочисленное деление (/) и целочисленный остаток (%). Команда let берет два операнда и присваивает их переменной.


3 Ход работы

1. Напишем командный файл, реализующий упрощённый механизм семафоров. Командный файл должен в течение некоторого времени t_1 дожидаться освобождения ресурса, выдавая об этом сообщение, а дождавшись его освобождения, использовать его в течение некоторого времени $t_2 < t_1$, также выдавая информацию о том, что ресурс используется соответствующим командным файлом (процессом) (рис. 3.1, 3.2).

The image shows a screenshot of an Emacs editor window titled 'emacs@fedora'. The window has a menu bar with 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Sh-Script', and 'Help'. Below the menu bar is a toolbar with icons for file operations and editing. The main text area contains a shell script for a semaphore mechanism. The script defines a lockfile path, uses 'exec' to replace the current process, and then checks if the lockfile exists. If it does, it enters a loop where it tries to acquire the lock using 'flock'. If successful, it prints 'file was locked', sleeps for 4 seconds, prints 'Unlocking', and releases the lock. If it fails, it prints 'file already locked' and sleeps for 3 seconds. The script ends with 'done' and 'fi' to close the loops and the 'exec' command.

```
lockfile="./locking.file"
exec {fn}>"$lockfile"
if test -f "$lockfile"
then
  while [ 1!=0 ]
  do
    if flock -n ${fn}
    then
      echo "file was locked"
      sleep 4
      echo "Unlocking"
      flock -u ${fn}
    else
      echo "file already locked"
      sleep 3
    fi
  done
fi
```

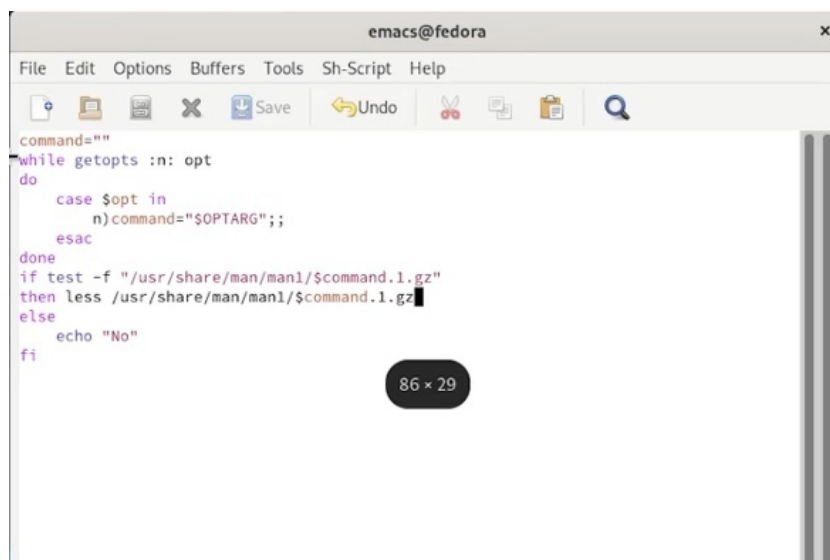
Рис. 3.1: Содержимое файла

A terminal window titled 'dakhusainova@fedora:~' with search, menu, and close icons. It shows a series of commands and outputs: 'chmod 777 lab1201.sh', './lab1201.sh', and then a loop of 'file was locked' followed by 'Unlocking' repeated five times. The prompt ends with '^C[dakhusainova@fedora ~]\$' and a cursor.

```
[dakhusainova@fedora ~]$ chmod 777 lab1201.sh
[dakhusainova@fedora ~]$ ./lab1201.sh
file was locked
Unlocking
file was locked
Unlocking
file was locked
Unlocking
file was locked
Unlocking
file was locked
^C[dakhusainova@fedora ~]$
```

Рис. 3.2: Наша проверка

2. Теперь мы реализуем команду `man` с помощью командного файла. Изучите содержимое каталога `/usr/share/man/man1`. В нем находятся архивы текстовых файлов, содержащих справку по большинству установленных в системе программ и команд. Каждый архив можно открыть командой `less` сразу же просмотрев содержимое справки. Командный файл должен получать в виде аргумента командной строки название команды и в виде результата выдавать справку об этой команде или сообщение об отсутствии справки, если соответствующего файла нет в каталоге(рис. 3.3,3.3,3.5,3.6).

The image shows the Emacs editor window titled 'emacs@fedora'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Sh-Script', and 'Help'. The toolbar contains icons for opening files, saving, undo, redo, and search. The main text area displays a shell script with the following content:

```
command=""
while getopts :n: opt
do
    case $opt in
        n) command="$OPTARG";;
        esac
    done
    if test -f "/usr/share/man/man1/$command.1.gz"
    then less /usr/share/man/man1/$command.1.gz
    else
        echo "No"
    fi
done
```

A status bar at the bottom center of the editor shows '86 x 29'.

Рис. 3.3: Содержимое файла

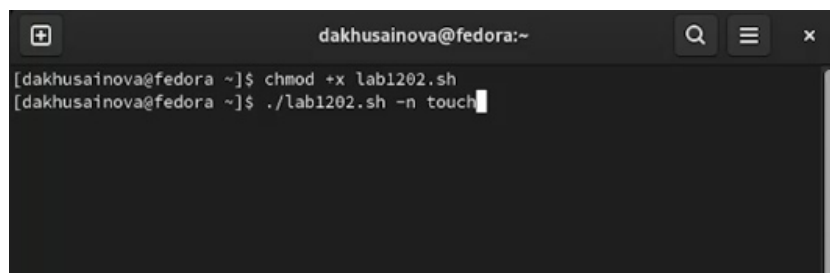
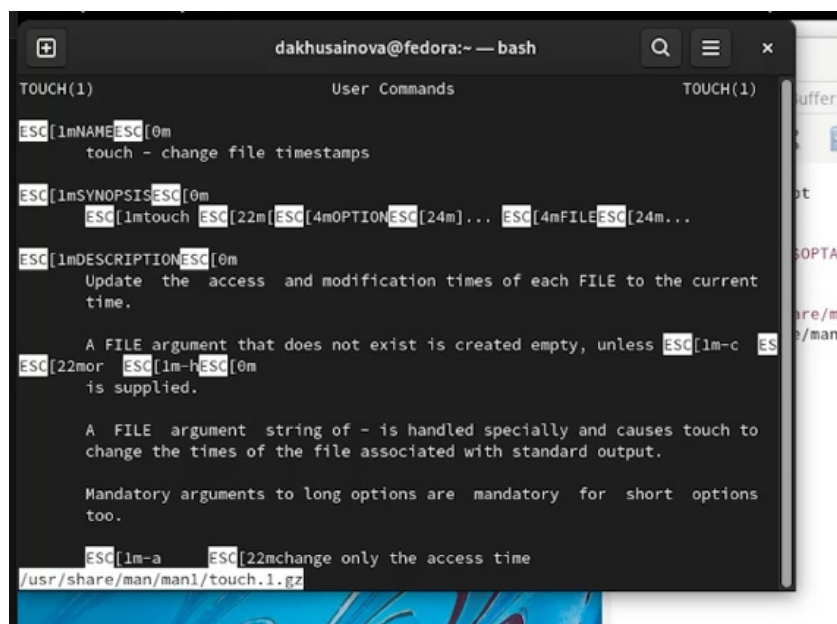
The image shows a terminal window titled 'dakhusainova@fedora:~'. The prompt is '[dakhusainova@fedora ~]'. The user has entered two commands: 'chmod +x lab1202.sh' and './lab1202.sh -n touch', both of which have been executed successfully. The prompt is now '[dakhusainova@fedora ~]\$'.

Рис. 3.4: Предоставление прав доступа и проверка



```
dakhusainova@fedora:~ — bash
TOUCH(1)                                User Commands                                TOUCH(1)

ESC[1mNAMEESC[0m
touch - change file timestamps

ESC[1mSYNOPSISESC[0m
ESC[1mtouch ESC[22mESC[4mOPTIONESC[24m)... ESC[4mFILEESC[24m...

ESC[1mDESCRIPTIONESC[0m
Update the access and modification times of each FILE to the current
time.

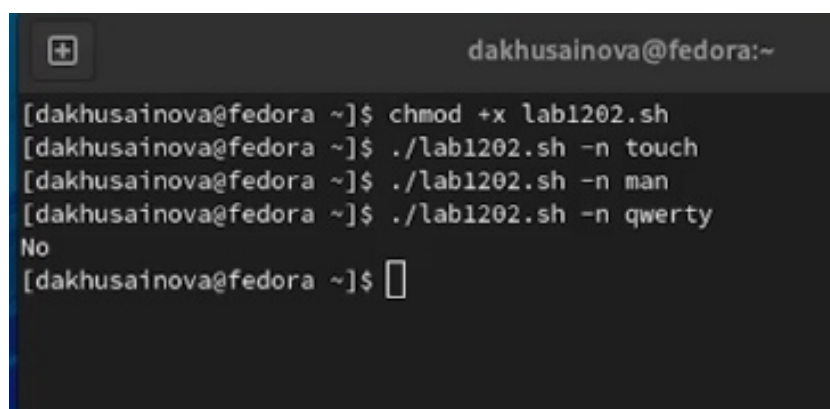
A FILE argument that does not exist is created empty, unless ESC[1m-c ESC[22mor ESC[1m-hESC[0m
is supplied.

A FILE argument string of - is handled specially and causes touch to
change the times of the file associated with standard output.

Mandatory arguments to long options are mandatory for short options
too.

ESC[1m-a ESC[22mchange only the access time
/usr/share/man/man1/touch.1.gz
```

Рис. 3.5: Вывод информации о вводимой команде



```
dakhusainova@fedora:~
[dakhusainova@fedora ~]$ chmod +x lab1202.sh
[dakhusainova@fedora ~]$ ./lab1202.sh -n touch
[dakhusainova@fedora ~]$ ./lab1202.sh -n man
[dakhusainova@fedora ~]$ ./lab1202.sh -n qwerty
No
[dakhusainova@fedora ~]$
```

Рис. 3.6: Ввод команды, которой нет

3. Используя встроенную переменную \$RANDOM, напомним командный файл, генерирующий случайную последовательность букв латинского алфавита(рис. 3.7,3.8).

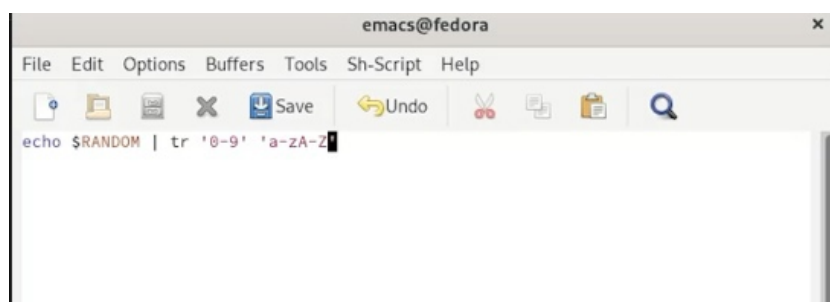


Рис. 3.7: Содержимое файла

```
[dakhusainova@fedora ~]$ chmod +x lab1203.sh
[dakhusainova@fedora ~]$ ./lab1203.sh
dcdff
[dakhusainova@fedora ~]$ ./lab1203.sh
baiee
[dakhusainova@fedora ~]$ ./lab1203.sh
cghhg
[dakhusainova@fedora ~]$ ./lab1203.sh
cfbaa
[dakhusainova@fedora ~]$ ./lab1203.sh
ifjff
[dakhusainova@fedora ~]$ ./lab1203.sh
d cgf
[dakhusainova@fedora ~]$ ./lab1203.sh
beedc
[dakhusainova@fedora ~]$ ./lab1203.sh
cgjjc
[dakhusainova@fedora ~]$
```

Рис. 3.8: Вывод случайных букв

4 Контрольные вопросы

1. Найдите синтаксическую ошибку в следующей строке:

`“while [$1 != “exit”]”` Вместо `exit` должна стоять цифра, а не это слово.

2. Как объединить (конкатенация) несколько строк в одну?

Самый простой способ объединить две или более строковые переменные – записать их одну за другой:

```
VAR1=“Hello,”
```

```
VAR2=“ World”
```

```
VAR3=“VAR1VAR2”
```

```
echo “$VAR3”
```

```
Hello, World
```

3. Найдите информацию об утилите `seq`. Какими иными способами можно реализовать её функционал при программировании на `bash`?

Эти утилиты выводят последовательность целых чисел с шагом, заданным пользователем. По-умолчанию, выводимые числа отделяются друг от друга символом перевода строки, однако, с помощью ключа `-s` может быть задан другой разделитель.

```
bash$ seq 5
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
bash$ seq -s : 5
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

Обе утилиты, и `jot`, и `seq`, очень удобно использовать для генерации списка аргументов в цикле `for`.

4. Какой результат даст вычисление выражения $\$(10/3)$?

```
3
```

5. Укажите кратко основные отличия командной оболочки `zsh` от `bash`.

Чтобы дать лучше понять набор отличительных черт Z Shell, вот список того, что вы получите, используя Z Shell вместо Bash: Встроенная команда `zmv` поможет вам массово переименовать файлы/директории, например, чтобы добавить `‘.txt’` к имени каждого файла, запустите `zmv -C ‘(*)(#q.)’ ‘$1.txt’`. Утилита `zcalc` — это замечательный калькулятор командной строки, это удобный способ считать быстро, не покидая терминал. Загрузите её через `autoload -Uz zcalc` и запустите командой `zcalc`. Команда `zparseopts` — это однострочник, который поможет вам разобрать сложные варианты, которые предоставляются вашему скрипту(?) Команда `autopushd` позволяет вам делать `popd` после того, как вы с помощью `cd`, чтобы вернуться в предыдущую директорию. Поддержка чисел с плавающей точкой (коей Bash, к удивлению, не содержит). Поддержка для структур данных “хэш”. Есть также ряд особенностей, которые присутствуют в Bash, но их нет почти во всех остальных командных оболочках. Вот также некоторые из них:

Опция командной строки `-porc`, которая позволяет пользователю иметь дело с инициализацией командной строки, не читая файл `.bashrc` Использование опции `-rcfile` с `bash` позволяет вам исполнять команды из определённого файла. Отличные возможности вызова (набор опций для командной строки) Может быть вызвана командой `sh` Bash можно запустить в определённом режиме

POSIX. Примените `set -o posix`, чтобы включить режим, или `--posix` при запуске. Вы можете управлять видом командной строки в Bash. Настройка переменной `PROMPT_COMMAND` с одним или более специальными символами настроит её за вас. Bash также можно включить в режиме ограниченной оболочки (с `rbash` или `-restricted`), это означает, что некоторые команды/действия больше не будут доступны: Настройка и удаление значений служебных переменных `SHELL`, `PATH`, `ENV`, `BASH_ENV` Перенаправление вывода с использованием операторов `>`, `>|`, `<>`, `>&`, `&>`, `>>` Разбор значений `SHELLOPTS` из окружения оболочки при запуске Использование встроенного оператора `exes`, чтобы заменить оболочку другой командой.

6. Проверьте, верен ли синтаксис данной конструкции `1 for ((a=1; a <= LIMIT; a++))`

Надо убрать дополнительные круглые скобки.

7. Сравните язык `bash` с какими-либо языками программирования. Какие преимущества у `bash` по сравнению с ними? Какие недостатки?

Использование Bash для написания Shell-сценариев Каждая популярная операционная система обычно предлагает интерфейс командной строки (CLI) через приложение терминала. Приложение терминала выполняет команды через определенный интерпретатор командной строки, такой как Bash, Z shell, C shell и KornShell. Определенные команды помещаются в файл и выполняются через выбранный интерпретатор командной строки. Другими словами, мы можем писать Shell-сценарии с помощью Bash, Z shell и т. д. Bash является широко используемым интерпретатором командной строки, поскольку он включен по умолчанию почти во все Unix или Unix-подобные операционные системы. Поэтому с помощью Bash можно писать переносимые POSIX-сценарии.

Bash позволяет писать Shell-сценарии с минимальной грамматикой. Если нужно выполнить несколько команд, достаточно поместить их в сценарий Bash построчно. Bash поддерживает основные концепции программирования:

операторы `if-else`; циклы; арифметические операции; функции; переменные.

Bash поддерживает процессы нативно. Иначе говоря, вы можете инициировать другие двоичные файлы в качестве команд. Например, если вам нужно выполнить двоичный ping, можете написать команду ping в своих сценариях Bash. Есть несколько способов отображения графического интерфейса с помощью сценариев Bash.

Bash — это командный язык, а не язык программирования общего назначения. Поэтому с усложнением логики вашего автоматизированного сценария он становится более запутанным и менее читаемым. Кроме того, Bash все и всегда воспринимает как команду, потому что это командный язык. См. следующий пример:

```
#!/bin/bash x=10 echo $x # prints 10 x = 10 # x: command not found echo $x
```

У Bash нет стандартного API, однако он поставляется с простыми встроенными функциями (например, со встроенной тестовой обработкой). Однако вам часто придется создавать процессы для обработки данных (помните sed?). Таким образом, Bash работает очень медленно по сравнению с другими языками, предназначенными для создания автоматизированных сценариев.

Использование Python для написания Shell-сценариев Python — популярная альтернатива Bash для написания сценариев настройки среды, сборки и выпуска. Я знаю, что проект Electron использует Python для сценариев нескольких утилит. В Python нельзя выполнять команды напрямую, поскольку это не командный язык. Но запуск команд и перехват вывода реализуются проще простого с помощью модуля subprocess. Взгляните на следующий пример:

```
#!/usr/bin/python3 import subprocess o = subprocess.check_output(['node', '-version'], text = True) print('You are using Node' + o)
```

Приведенный выше сценарий Python печатает текущую компьютерную версию Nodejs. Этот сценарий эквивалентен следующему сценарию Bash:

```
#!/usr/bin/bash o=$(node -version) echo You are using Node $o
```

Используя встроенные функции Python, можно писать современные сложные Shell-сценарии. Но, в отличие от Bash, интерпретатор Python изначально не поддерживает выполнение

процесса. Поэтому, если нужно упростить сценарий Python, чтобы он больше походил на Bash, используется такой инструмент, как Shellpy. Посмотрите на следующий сценарий Shellpy, который выполняет работу предыдущего сценария:

```
#!/usr/local/bin/shellpy3 o = 'node -version print('You are using Node' + o.stdout)
```

Python — язык, очень удобный для разработчиков. Кроме того, он поставляется со многими полезными встроенными библиотеками. Интерпретатор Python предустановлен почти во все Unix-подобные операционные системы. Поэтому Python также является хорошей альтернативой для написания переносимых автоматизированных сценариев.

Но в отличие от других языков, для Python характерно медленное время выполнения программы. Когда вы используете некоторые библиотеки, Python работает слишком медленно даже по сравнению с Bash. Такие инструменты, как Shellpy и Plumbum, предлагают удобные API для работы с процессами и командами. Тем не менее, вам придется настроить и написать дополнительный код.

5 Вывод

Мы изучили основы программирования в оболочке ОС UNIX и научились писать более сложные командные файлы с использованием логических управляющих конструкций и циклов.