

Отчет по лабораторной работе 10

Операционные системы

Никулина Ксения Ильинична

Содержание

1	Цель работы	5
2	Задани	6
3	Выполнение лабораторной работы	7
4	Выводы	13
5	Ответы на вопросы	14

Список иллюстраций

3.1	Создание файла	7
3.2	Создание файла	7
3.3	Результат работы программы	8
3.4	Создание файла	8
3.5	Создание файла	8
3.6	Результат работы программы	9
3.7	Создание файла	9
3.8	Создание файла	10
3.9	Результат работы программы	10
3.10	Создание файла	11
3.11	Создание файла	11
3.12	Результат работы программы	12

Список таблиц

1 Цель работы

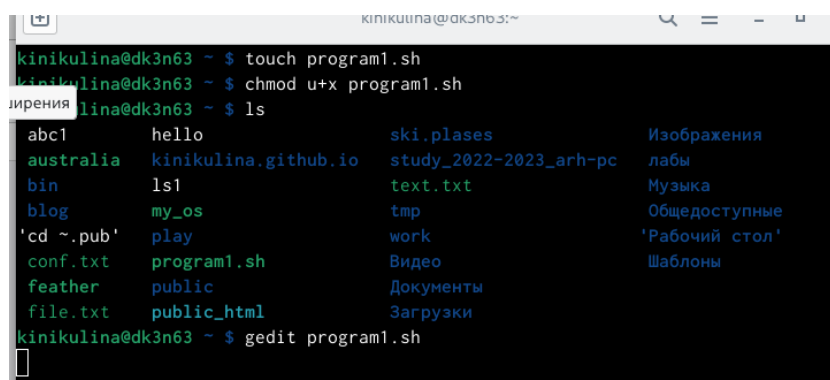
Изучить основы программирования в оболочке ОС UNIX/Linux. Научиться писать небольшие командные файлы

2 Задани

1. Написать скрипт, который при запуске будет делать резервную копию самого себя (то есть файла, в котором содержится его исходный код) в другую директорию backup в вашем домашнем каталоге. При этом файл должен архивироваться одним из архиваторов на выбор zip, bzip2 или tar. Способ использования команд архивации необходимо узнать, изучив справку
2. Написать пример командного файла, обрабатывающего любое произвольное число аргументов командной строки, в том числе превышающее десять. Например, скрипт может последовательно распечатывать значения всех переданных аргументов.
3. Написать командный файл — аналог команды ls (без использования самой этой команды и команды dir). Требуется, чтобы он выдавал информацию о нужном каталоге и выводил информацию о возможностях доступа к файлам этого каталога.
4. Написать командный файл, который получает в качестве аргумента командной строки формат файла (.txt, .doc, .jpg, .pdf и т.д.) и вычисляет количество таких файлов в указанной директории. Путь к директории также передаётся в виде аргумента командной строки.

3 Выполнение лабораторной работы

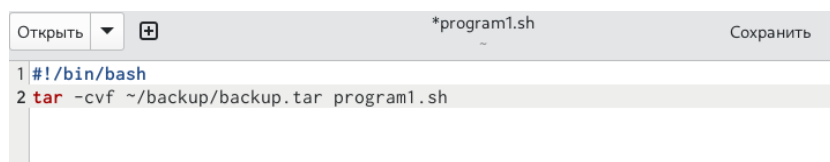
1. Создала файл для программы 1 (рис. 3.1).



```
kinikulina@edk3n63 ~ $ touch program1.sh
kinikulina@edk3n63 ~ $ chmod u+x program1.sh
kinikulina@edk3n63 ~ $ ls
abc1      hello          ski.plases     Изображения
australia kinikulina.github.io study_2022-2023_arh-pc лабы
bin       ls1            text.txt       Музыка
blog     my_os         tmp            Общедоступные
'cd ~/.pub' play          work          'Рабочий стол'
conf.txt program1.sh   Видео        Шаблоны
feather  public        Документы
file.txt public_html   Загрузки
kinikulina@edk3n63 ~ $ gedit program1.sh
```

Рис. 3.1: Создание файла

2. Написала текст программы 1 (рис. 3.2).



```
Открыть ▼ + *program1.sh Сохранить ≡
1 #!/bin/bash
2 tar -cvf ~/backup/backup.tar program1.sh
```

Рис. 3.2: Создание файла

3. Проверила работу написанной программы (рис. 3.3).

```

kinikulina@dk3n63 ~ $ mkdir backup
kinikulina@dk3n63 ~ $ bash program1.sh
program1.sh
kinikulina@dk3n63 ~ $ ls backup
backup.tar
kinikulina@dk3n63 ~ $ 

```

Рис. 3.3: Результат работы программы

4. Создала файл для программы 2 (рис. 3.4).

```

kinikulina@dk3n63 ~ $ touch program2.sh
kinikulina@dk3n63 ~ $ chmod u+x program2.sh
kinikulina@dk3n63 ~ $ ls
abc1      file.txt      public
australia hello         public_html
backup    kinikulina.github.io ski.places
bin       ls1           study_2022
blog     my_os        text.txt

```

Рис. 3.4: Создание файла

5. Написала текст программы 2 (рис. 3.5).

```

Открыть ▼ + program2.sh
~
1 #!/bin/bash
2 echo 'Введите числа: '
3 for a in $@
4 do echo $a
5 done

```

Рис. 3.5: Создание файла

6. Проверила работу написанной программы (рис. 3.6).

```
program1.sh
kinikulina@dk3n63 ~ $ ./program2.sh 3 4 6 7
Введите числа:
3 4 6 7
3 4 6 7
3 4 6 7
3 4 6 7
kinikulina@dk3n63 ~ $
```

Рис. 3.6: Результат работы программы

7. Создала файл для программы 3 (рис. 3.7).

```
kinikulina@dk3n63 ~ $ touch program3.sh
kinikulina@dk3n63 ~ $ chmod u+x program3.sh
kinikulina@dk3n63 ~ $ ls
abc1      hello          public_html    Изображения
australia kinikulina.github.io ski.plases     лабы
backup    ls1            study_2022-2023_arh-pc Музыка
bin       my_os         text.txt       Общедоступные
blog      play          tmp            'Рабочий стол'
'cd ~/.pub' program1.sh    work          Шаблоны
conf.txt  program2.sh   Видео
feather   program3.sh   Документы
file.txt  public        Загрузки
```

Рис. 3.7: Создание файла

8. Написала текст программы 3 (рис. 3.8).

```

1 #!/bin/bash
2 for A in *
3 do
4   if test -d $A
5   then
6     echo -n $A: is a directory
7   else
8     echo -n $A: is a file and
9     if test -w $A
10    then
11      echo writeable
12      if test -r $A
13      then
14        echo readable
15      else
16        echo neither readable nor writeable
17      fi
18    fi
19  fi
20 done

```

Рис. 3.8: Создание файла

9. Проверила работу написанной программы (рис. 3.9).

```

kinikulina@dk3n63 ~ $ ./program3.sh
abc1: is a file andwriteable
readable
australia: is a file andwriteable
readable
backup: is a directorybin: is a directoryblog: is a directory./program3.sh
ка 4: test: cd: ожидается бинарный оператор
cd ~/.pub: is a file and./program3.sh: строка 9: test: cd: ожидается бинарн
ратоп
conf.txt: is a file andwriteable
readable
feather: is a file andwriteable
readable
file.txt: is a file andwriteable
readable
hello: is a file andwriteable

```

Рис. 3.9: Результат работы программы

10. Создала файл для программы 4 (рис. 3.10).

```

...directoryЗагрузки: is a directoryИзображения: is a directoryлабы: is a directo
...ка: is a directoryОбщедоступные: is a directory./program3.sh: строка 4: te
...бочий: ожидается бинарный оператор
Рабочий стол: is a file and./program3.sh: строка 9: test: Рабочий: ожидае
...арный оператор
Шаблоны: is a directorykinikulina@dk3n63 ~ $
kinikulina@dk3n63 ~ $ touch program4.sh
kinikulina@dk3n63 ~ $ chmod u+x program4.sh
kinikulina@dk3n63 ~ $ ls
abc1      hello      public      Загрузки
australia kinikulina.github.io public_html  Изображения
backup    ls1        ski.places  лабы
bin       my_os     study_2022-2023_arh-pc Музыка
blog      play      text.txt    Общедоступны
'cd ~/.pub' program1.sh tmp          'Рабочий стол'
conf.txt  program2.sh work        Шаблоны

```

Рис. 3.10: Создание файла

11. Написала текст программы 4 (рис. 3.11).

```

1 #!/bin/bash
2 b="$1"
3 shift
4 for a in $@
5 do
6     k=7
7     for i in ${b}/*.${a}
8     do
9         if test -f "$i"
10        then
11            let k=k+1
12        fi
13    done
14    echo "$k файлов в $b с расширением $a"
15 done

```

Рис. 3.11: Создание файла

12. Проверила работу написанной программы (рис. 3.12).

```
7 файлов в /afs/.dk.sci.pfu.edu.ru/home/k/i/kinikulina с расширением pdf
kinikulina@dk3n63 ~ $ ./program4.sh ~ txt sh pdf
10 файлов в /afs/.dk.sci.pfu.edu.ru/home/k/i/kinikulina с расширением txt
11 файлов в /afs/.dk.sci.pfu.edu.ru/home/k/i/kinikulina с расширением sh
7 файлов в /afs/.dk.sci.pfu.edu.ru/home/k/i/kinikulina с расширением pdf
kinikulina@dk3n63 ~ $
```

Рис. 3.12: Результат работы программы

4 Выводы

В процессе выполнения данной лабораторной работы я изучила основы программирования в оболочке ОС UNIX/Linux, научилась писать небольшие командные файлы

5 Ответы на 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:
 - оболочка Борна (Bourne shell или sh) – стандартная командная оболочка UNIX/Linux, содержащая базовый, но при этом полный набор функций;
 - С-оболочка (или csh) – надстройка на оболочкой Борна, использующая Сподобный синтаксис команд с возможностью сохранения истории выполнения команд;
 - оболочка Корна (или ksh) – напоминает оболочку С, но операторы управления программой совместимы с операторами оболочки Борна;
 - BASH – сокращение от Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек С 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/Linuxподобных 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 отобразить 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:

- `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, 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 (new line).
- `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 `You have mail` (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 метасимвола. Экранирование может быть осуществлено с помощью предшествующего метасимволу символа , который, в свою очередь, является метасимволом. 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 , – `echo ab'|'cd` выведет на экран строку `ab|*cd`. 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 осуществит её интерпретацию.
- 10) 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 `unset` с флагом `-f`.
- 11) Чтобы выяснить, является ли файл каталогом или обычным файлом, необ-

ходимо воспользоваться командами «test -f [путь до файла]» (для проверки, является ли обычным файлом) и «test -d [путь до файла]» (для проверки, является ли каталогом).

- 12) 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.
- 13) 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.
- 14) Специальные переменные:
 - \$* – отображается вся командная строка или параметры оболочки;
 - \$? – код завершения последней выполненной команды;
 - \$\$ – уникальный идентификатор процесса, в рамках которого выполняется командный процессор;

- `$_` – номер процесса, в рамках которого выполняется последняя вызванная на выполнение в командном режиме команда;
- `$-` – значение флагов командного процессора;
- `${#}` – возвращает целое число – количество слов, которые были результатом `$`;
- `${#name}` – возвращает целое значение длины строки в переменной `name`;
- `${name[n]}` – обращение к n-му элементу массива;
- `${name[*]}` – перечисляет все элементы массива, разделённые пробелом;
- `${name[@]}` – то же самое, но позволяет учитывать символы пробелы в самих переменных;
- `${name:-value}` – если значение переменной `name` не определено, то оно будет заменено на указанное `value`;
- `${name:value}` – проверяется факт существования переменной;
- `${name=value}` – если `name` не определено, то ему присваивается значение `value`;
- `${name?value}` – останавливает выполнение, если имя переменной не определено, и выводит `value` как сообщение об ошибке;
- `${name+value}` – это выражение работает противоположно `${name-value}`. Если переменная определена, то подставляется `value`;
- `${name#pattern}` – представляет значение переменной `name` с удалённым самым коротким левым образцом (`pattern`);
- `${#name[*]}` и `${#name[@]}` – эти выражения возвращают количество элементов в массиве `name`.