

Лабораторная работа №12

Операционные системы

Машков Илья Евгеньевич

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	7
3.1	Скрипт на резервное копирование и архивацию	7
3.2	Скрипт на последовательный вывод заданных аргументов	8
3.3	Скрипт на команду ls -l	9
3.4	Скрипт на команду find	11
4	Выводы	12
5	Контрольные вопросы	13
	Список литературы	23

Список иллюстраций

3.1	Скрипт на резервное копирование и архивацию.	7
3.2	Запуск скрипта.	7
3.3	Проверка выполнения.	8
3.4	Скрипт на последовательный вывод.	8
3.5	Результат выполнения.	9
3.6	Скрипт на ls -l.	10
3.7	Запуск программы.	10
3.8	Скрипт на поиск файлов с определённым расширением.	11
3.9	Результат работы скрипта.	11

Список таблиц

1 Цель работы

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

2 Задание

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

3.1 Скрипт на резервное копирование и архивацию

Создаю файл **task1.sh**, в котором прописываю скрипт на резервное копирование и архивацию этой копии в папку **backup**, которую я предварительно создал в домашнем каталоге (рис. [3.1]).

```
#!/bin/bash  
tar -cvf ~/backup/backup task1.sh
```

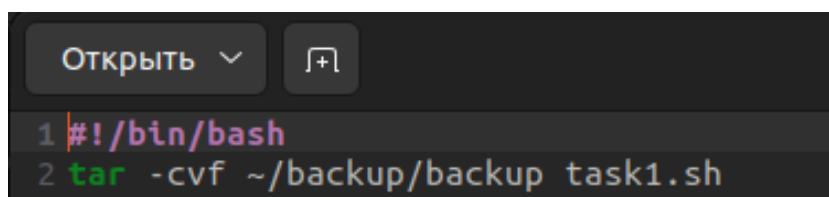


Рис. 3.1: Скрипт на резервное копирование и архивацию.

Затем запускаю данный скрипт (рис. [3.2]).

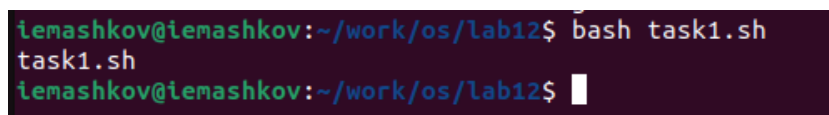


Рис. 3.2: Запуск скрипта.

Перехожу в папку **backup**, чтобы проверить правильность выполнения программы (рис. [3.3]).



Рис. 3.3: Проверка выполнения.

3.2 Скрипт на последовательный вывод заданных аргументов

Создаю файл **task2.sh**, в котором прописываю скрипт на последовательную печать введённых аргументов (рис. [3.4]).

```
#!/bin/bash
for A in $*
do echo $A $A
done
```

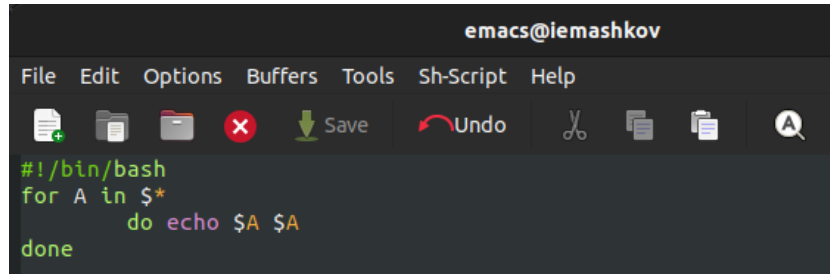


Рис. 3.4: Скрипт на последовательный вывод.

Затем запускаю программу (рис. [3.5]).


```

iemashkov@iemashkov:~/work/os/lab12$ touch task2.sh
iemashkov@iemashkov:~/work/os/lab12$ bash task2.sh 1 2 3 4 5
1 1
2 2
3 3
4 4
5 5
iemashkov@iemashkov:~/work/os/lab12$

```

Рис. 3.5: Результат выполнения.

3.3 Скрипт на команду `ls -l`

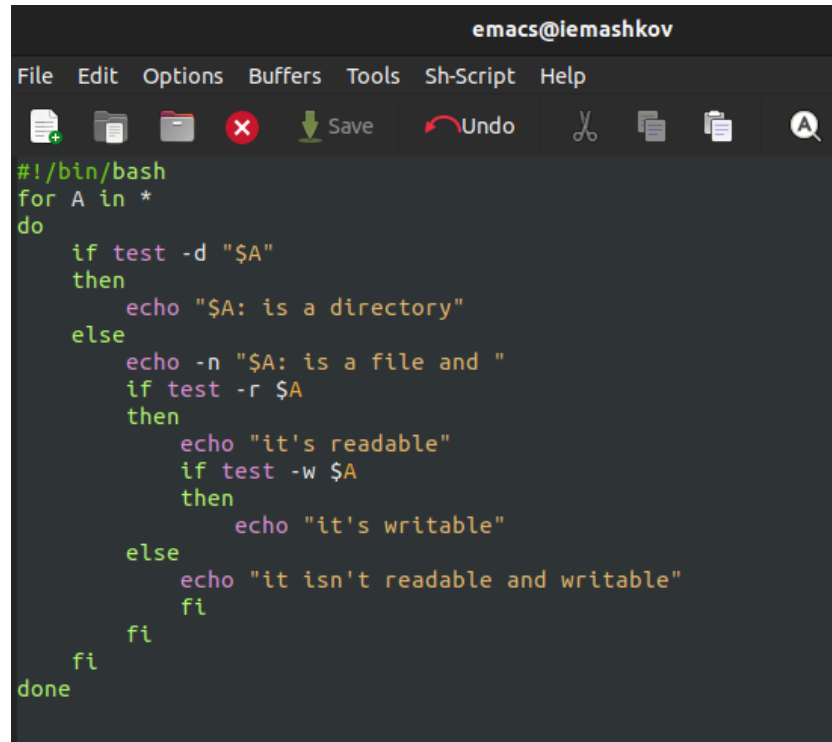
Создаю файл **task3.sh**, в котором прописываю скрипт, заменяющий команду **ls -l** (рис. [3.6]).

```

#!/bin/bash
for A in *
do
    if test -d "$A"
    then
        echo "$A: is a directory"
    else
        echo -n "$A: is a file and "
        if test -r $A
        then
            echo "it's readable"
            if test -w $A
            then
                echo "it's writable"
            else
                echo "it isn't readable and writable"
            fi
        fi
    fi
done

```

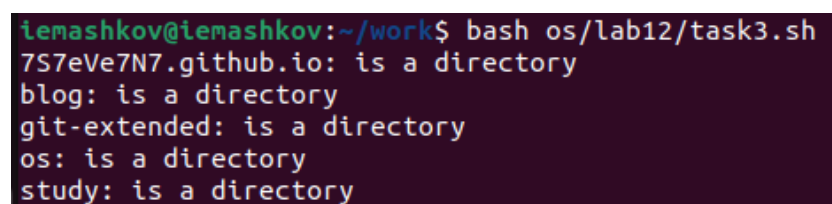
```
fi
fi
done
```

The image shows a screenshot of an Emacs editor window titled 'emacs@iemashkov'. The menu bar includes '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 with the following content:

```
#!/bin/bash
for A in *
do
  if test -d "$A"
  then
    echo "$A: is a directory"
  else
    echo -n "$A: is a file and "
    if test -r $A
    then
      echo "it's readable"
      if test -w $A
      then
        echo "it's writable"
      else
        echo "it isn't readable and writable"
      fi
    fi
  fi
fi
done
```

Рис. 3.6: Скрипт на ls -l.

Запускаю программу и проверяю вывод (рис. [3.7]).

The image shows a screenshot of a terminal window. The prompt is 'iemashkov@iemashkov:~/work\$'. The command executed is 'bash os/lab12/task3.sh'. The output of the script is as follows:

```
7S7eVe7N7.github.io: is a directory
blog: is a directory
git-extended: is a directory
os: is a directory
study: is a directory
```

Рис. 3.7: Запуск программы.

3.4 Скрипт на команду find

Создаю файл **task4.sh**, в котором прописываю скрипт на команду **find** (рис. [3.8]).

```
#!/bin/bash
echo "Write the file extension: "
read extension
echo "Write the directory: "
read directory
count=$(find "$directory" -name ".*$extension" -type f | wc -l)
echo "There are $count $extension files in the directory '$directory'."
```

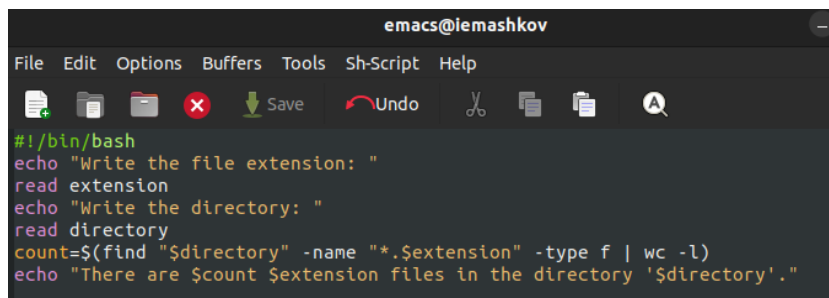


Рис. 3.8: Скрипт на поиск файлов с определённым расширением.

Затем запускаю программу, которая искала файлы с расширением **.png** в снимках экрана (рис. [3.9]).

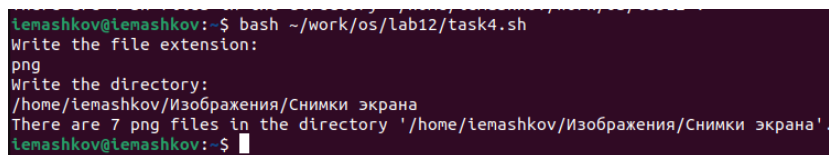


Рис. 3.9: Результат работы скрипта.

4 Выводы

В процессе выполнения лабораторной работы я изучил основы программирования в оболочке ОС UNIX/Linux, а также приобрёл навыки по созданию небольших командных файлов.

5 Контрольные вопросы

1. Объясните понятие командной оболочки. Приведите примеры командных оболочек. Чем они отличаются?

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; –С-оболочка - добавка университета Беркли к коллекции оболочек: она надстраивается над оболочкой Борна, используя С-подобный синтаксис команд, и сохраняет историю выполненных команд; –оболочка Корна - 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. Что такое 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 операционной 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.

3. Как определяются переменные и массивы в языке программирования bash?

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 `{имя переменной}` например, использование команд `b=/tmp/andy-ls -l myfile > blsls/tmp/andy - ls,ls - l >bls` приведет к подстановке в командную строку значения переменной `bls`. Если переменной `bls` не было предварительно присвоено никакого значения, то ее значением является символ пробел. Оболочка `bash` позволяет создание массивов. Для создания массива используется команда `set` с флагом `-A`. За флагом следует имя переменной, а затем список значений, разделенных пробелом. 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. Каково назначение операторов `let` и `read`?

****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. 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.****

5. Какие арифметические операции можно применять в языке программирования `bash`?

Оператор Синтаксис Результат ! !exp Если exp равно 0, возвращает 1; иначе 0 != exp1 !=exp2 Если exp1 не равно exp2, возвращает 1; иначе 0 % exp1%exp2 Возвращает остаток от деления exp1 на exp2 %= var=%exp Присваивает остаток от деления var на exp переменной var & exp1&exp2 Возвращает побитовое AND выражений exp1 и exp2 && exp1&&exp2 Если exp1 и exp2 не равны нулю, возвращает 1; иначе 0 &= var &= exp Присваивает var побитовое AND переменных var и выражения exp * exp1 * exp2 Умножает exp1 на exp2 = var = exp Умножает exp на значение var и присваивает результат переменной var + exp1 + exp2 Складывает exp1 и exp2 += var += exp Складывает exp со значением var и результат присваивает var - -exp Операция отрицания exp (называется унарный минус) - exp1 - exp2 Вычитает exp2 из exp1 -= var -= exp Вычитает exp из значения var и

присваивает результат `var / expr / expr2` Делит `expr1` на `expr2` `/= var /= expr`
 Делит `var` на `expr` и присваивает результат `var < expr1 < expr2`.

Если `expr1` меньше, чем `expr2`, возвращает 1, иначе возвращает 0 « `expr1` «
`expr2` Сдвигает `expr1` влево на `expr2` бит «= `var` «= `expr` Побитовый сдвиг влево
 значения `var` на `expr` `<= expr1 <= expr2` Если `expr1` меньше, или равно `expr2`,
 возвра- щает 1; иначе возвращает 0 = `var` = `expr` Присваивает значение
`expr` переменной `va` `== expr1==expr2` Если `expr1` равно `expr2`. Возвращает 1;
 иначе возвращает 0 > `expr1 > expr2` 1 если `expr1` больше, чем `expr2`; иначе 0
 >= `expr1 >= expr2` 1 если `expr1` больше, или равно `expr2`; иначе 0 » `expr` » `expr2`
 Сдвигает `expr1` вправо на `expr2` бит »= `var` »=`expr` Побитовый сдвиг вправо
 значения `var` на `expr` ^ `expr1` ^ `expr2` Исключающее OR выражений `expr1` и `expr2`
 ^= `var` ^= `expr` Присваивает `var` побитовое исключающее OR `var` и `expr` | `expr1`
 | `expr2` Побитовое OR выражений `expr1` и `expr2` |= `var` |= `expr` Присваивает `var`
 «исключающее OR» пе- ременной `var` и выражения `expr` || `expr1` || `expr2` 1 если
 или `expr1` или `expr2` являются нену- левыми значениями; иначе 0 ~ ~`expr`
 Побитовое дополнение до `expr`.

6. Что означает операция `(())`?

Условия оболочки `bash`.

7. Какие стандартные имена переменных Вам известны?

****Имя переменной (идентификатор)** — это строка символов, которая отличает эту переменную от других объектов программы (идентифицирует переменную в программе). При задании имен переменным нужно соблюдать следующие правила: § первым символом имени должна быть буква. Остальные символы — буквы и цифры (прописные и строчные буквы различаются). Можно использовать символ «_»; § в имени нельзя использовать символ «.»; § число символов в имени не должно превышать 255; § имя переменной не должно совпадать с зарезервированными (служебными) словами языка. `Var1`, `PATH`, `trash`, `mon`, `day`, `PS1`, `PS2`

Другие стандартные переменные: `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. Это символы 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. В командном процессоре Си имеется еще несколько стандартных переменных. Значение всех переменных можно просмотреть с помощью команды `set`.**

8. Что такое метасимволы?

Такие символы, как `' < > * ? | " &` являются метасимволами и имеют для командного процессора специальный смысл.

9. Как экранировать метасимволы?

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, ее нужно заключить в одинарные кавычки. Строка, заключенная в двойные кавычки, экранирует все метасимволы, кроме `$, ' , , " .` Например, `-echo` выведет на экран символ, `-echo ab'|cd` выдаст строку `"ab|cd"`.

10. Как создавать и запускать командные файлы?

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 ее интерпретацию.

11. Как определяются функции в языке программирования `bash`?

Группу команд можно объединить в функцию. Для этого существует ключевое слово `function`, после которого следует имя функции и список команд, заключенных в фигурные скобки. Удалить функцию можно с помощью команды `unset` с флагом `-f`. Команда `typeset` имеет четыре опции для работы с функциями: `-f` — перечисляет определенные на текущий момент функции; `--ft` — при последующем вызове функции иницирует ее трассировку; `--fx` — экспортирует все перечисленные функции в любые дочерние программы оболочек; `--fu` — обозначает указанные функции как автоматически загружаемые. Автоматически загружаемые функции хранятся в командных файлах, а при их вызове оболочка просматривает переменную `FPATH`, отыскивая файл с одноименными именами функций, загружает его и вызывает эти функции.

12. Каким образом можно выяснить, является файл каталогом или обычным файлом?

`ls -lrf` Если есть `d`, то является файл каталогом.

13. Каково назначение команд `set`, `typeset` и `unset`?

Используется команда `set` с флагом `-A`. За флагом следует имя переменной, а затем список значений, разделенных пробелом. Например, `set -A states Delaware Michigan "New Jersey"` Далее можно сделать добавление в массив, например, `states[49]=Alaska`. Индексация массивов начинается с нулевого элемента. В командном процессоре Си имеется еще несколько стандартных переменных. Значение всех переменных можно просмотреть с помощью команды `set`. Наиболее распространенным является сокращение, избавляющееся от слова `let` в программах оболочек. Если объявить переменные целыми значениями, любое присвоение автоматически трактуется как арифметическое. Используйте `typeset -i` для объявления и присвоения переменной, и при последующем использовании она становится целой. Или можете использовать ключевое слово `integer` (псевдоним для `typeset -l`) и объявлять переменные целыми. Таким образом, выражения типа `x=y+z` воспринимаются как арифметические. Группу команд можно объединить в функцию. Для этого существует ключевое слово `function`, после которого следует имя функции и список команд, заключенных в фигурные скобки. Удалить функцию можно с помощью команды `unset` с флагом `-f`. Команда `typeset` имеет четыре опции для работы с функциями: – `-f` — перечисляет определенные на текущий момент функции; – `-ft` — при последующем вызове функции иницирует ее трассировку; – `-fx` — экспортирует все перечисленные функции в любые дочерние программы оболочек; – `-fu` — обозначает указанные функции как автоматически загружаемые. Автоматически загружаемые функции хранятся в командных файлах, а при их вызове оболочка просматривает переменную `FPATH`, отыскивая файл с одноименными именами функций, загружает его и вызывает эти функции. В переменные `top` и `day` будут считаны соответствующие значения, введенные с клавиатуры, а переменная `trash` нужна для того, чтобы отобразить всю избыточно введенную информацию и игнорировать ее. Изъять переменную из программы можно с помощью команды

unset.

14. Как передаются параметры в командные файлы?

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 нее 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 нее имени данного командного файла. Примере: пусть к командному файлу **where** имеется доступ по выполнению и этот командный файл содержит следующий конвейер: **who | grep \$1** Если Вы введете с терминала команду: **where andy**, то в случае, если пользователь, зарегистрированный в ОС UNIX под именем **andy**, в данный момент работает в ОС UNIX, на терминал будет выведена строка, содержащая номер терминала, используемого указанным пользователем. Если же в данный момент этот пользователь не работает в ОС UNIX, то на терминал не будет выведено ничего. Команда **grep** производит контекстный поиск в тексте, поступающем со стандартного ввода, для нахождения в этом тексте строк, содержащих последовательности символов, переданные ей в качестве аргументов, и выводит результаты своей работы на стандартный вывод. В этом примере команда **grep** используется как фильтр, обеспечивающий ввод со стандартного ввода и вывод всех строк, содержащих последовательность символов **andy**, на стандартный вывод. В ходе интерпретации этого файла командным процессором вместо комбинации символов **\$1** осуществляется подстановка значения первого и единственного параметра **andy**. Если предположить, что пользователь, зарегистрированный в ОС UNIX под именем **andy**, в данный момент работает в ОС UNIX, то на терминале Вы увидите примерно следующее: **\$ where andy andy ttyG Jan 14**

09:12 \$ Определим функцию, которая изменяет каталог и печатает список файлов: `$ function clist { > cd $1 > ls > }`. Теперь при вызове команды `clist` каталог будет изменен каталог и выведено его содержимое.

15. Назовите специальные переменные языка `bash` и их назначение.

`$*` — 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`-ному элементу массива;

`${name[*]}` — перечисляет все элементы массива, разделенные 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;

это выражение работает противоположно `{name-value}`. Если переменная определена, то подставляется `value`;

`${name#pattern}` — представляет значение переменной `name` с удаленным самым коротким левым образцом (`pattern`);

`${#name[*]}` и `${#name[@]}` — эти выражения возвращают количество элементов в массиве `name`.

`$#` вместо нее будет осуществлена подстановка числа параметров, указанных в командной строке при вызове данного командного файла на выполнение.

Список литературы

Операционные системы