ЛАБОРАТОРНАЯ РАБОТА №10 РАЗРАБОТКА СЦЕНАРИЕВ BASH

Цель работы – практическое знакомство с методами создания и использования сценариев ОС Linux.

1 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Необходимость использования сценариев командной оболочки

Одна из причин применения сценариев командной оболочки — возможность быстрого и простого программирования. Командная оболочка очень удобна для небольших утилит, выполняющих относительно простую задачу, для которой производительность менее важна, чем простота настройки, сопровождения и переносимость. Оболочка может использоваться для управления процессами, обеспечивая выполнение команд в заданном порядке, зависящем от успешного завершения каждого этапа выполнения.

Хотя внешне командная оболочка очень похожа на режим командной строки в ОС Windows, она гораздо мощнее и способна выполнять самостоятельно очень сложные программы. Командная оболочка выполняет программы оболочки, часто называемые сценариями или скриптами, которые интерпретируются во время выполнения. Такой подход облегчает отладку, потому что можно выполнять программу построчно и не тратить время на перекомпиляцию. Но для задач, которым важно время выполнения или необходимо интенсивное использование процессора, командная оболочка оказывается неподходящей средой.

1.2 Командная оболочка

Командная оболочка — это программа, которая действует как интерфейс между пользователем и ОС Linux, позволяя вводить команды, которые должна выполнить операционная система. В ОС Linux вполне может сосуществовать несколько установленных командных оболочек, и разные пользователи могут выбрать ту, которая им больше нравится. Поскольку ОС Linux — модульная система, можно вставить и применять одну из множества различных стандартных командных оболочек. В Lmux стандартная командная оболочка, всегда устанавливаемая как /bin/sh и входящая в комплект средств проекта GNU, называется bash (GNU Bourne-Again SHell). В данной работе используется оболочка bash версии 3, ее функциональные возможности являются общими для всех командных оболочек, удовлетворяющих требованиям стандарта POSIX.

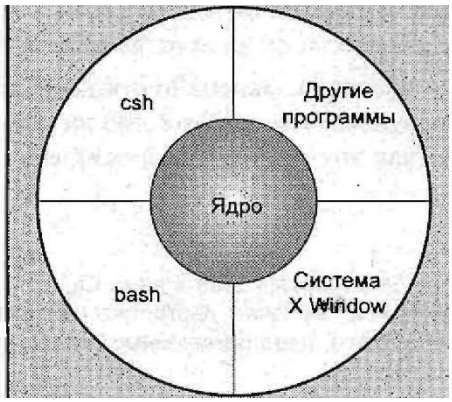

Рис. 1. Укрупненная архитектура ОС Linux

Каналы и перенаправление

Прежде чем заняться подробностями программ командной оболочки, необходимо сказать несколько слов о возможностях перенаправления ввода и вывода программ (не только программ командной оболочки) в ОС Linux.

Перенаправление вывода

Ранее были рассмотрены некоторые виды перенаправления, например, такие как:

$ ls -l > lsoutput.txt

сохраняющие вывод команды ls в файле с именем lsoutput.txt.

Однако перенаправление позволяет сделать гораздо больше, чем демонстрирует этот простой пример. Сейчас нужно знать только то, что дескриптор файла 0 соответствует стандартному вводу программы, дескриптор файла 1 — стандартному выводу, а дескриптор файла 2 — стандартному потоку ошибок. Каждый из этих файлов можно перенаправлять независимо друг от друга. На.самом деле можно перенаправлять и другие дескрипторы файлов, но, как правило, нет нужды перенаправлять любые другие дескрипторы, кроме стандартных: 0, 1 и 2.

В предыдущем примере стандартный вывод перенаправлен в файл с помощью оператора >. По умолчанию, если файл с заданным именем уже есть, он будет перезаписан. Для дозаписи в конец файла используйте оператор >>. Например, команда

$ ps >> lsoutput.txt

добавит вывод команды ps в конец заданного файла. В этом примере и далее знак $ перед командой – приглашение ОС Linux.

Для перенаправления стандартного потока ошибок перед оператором > вставьте номер дескриптора файла, который хотите перенаправить. Поскольку у стандартного потока ошибок дескриптор файла 2, укажите оператор 2>. Часто бывает полезно скрывать стандартный поток ошибок, запрещая вывод его на экран.

Предположим, что вы хотите применить команду kill для завершения процесса из сценария. Всегда существует небольшой риск, что процесс закончится до того, как выполнится команда kill. Если это произойдет, команда kill выведет сообщение об ошибке в стандартный поток ошибок, который по умолчанию появится на экране. Перенаправив стандартный вывод команды и ошибку, вы сможете помешать команде kill выводить какой бы то ни было текст на экран.

Команда $ kill -HUP 1234 > killout.txt 2>killer.txt

поместит вывод и информацию об ошибке в разные файлы.

Если вы предпочитаете собрать оба набора выводимых данных в одном файле, можно применить оператор >2 для соединения двух выводных потоков. Таким образом, команда

$ kill -1 1234 > killerr.txt 2>41

поместит свой вывод и стандартный поток ошибок в один и тот же файл. Обратите внимание на порядок следования операторов. Приведенный пример читается как "перенаправить стандартный вывод в файл killerr.txt, а затем перенаправить стандартный поток ошибок туда же, куда и стандартный вывод". Если вы нарушите порядок, перенаправление выполнится не так, как вы ожидаете.

Поскольку обнаружить результат выполнения команды kill можно с помощью кода завершения, часто не потребуется сохранять какой бы то ни было стандартный вывод или стандартный поток ошибок. Для того чтобы полностью отбросить любой вывод, можно использовать универсальную "мусорную корзину" Linux, /dev/null, следующим образом:

$ kill -1 1234 >/dev/null 2>fil

Перенаправление ввода

Также как вывод можно перенаправить ввод. Например $ more < killout.txt

Каналы

Процессы можно соединять с помощью оператора канала | . Как пример, можно применить команду sort для сортировки вывода команды ps.

Если не применять каналы, придется использовать несколько шагов, подобных следующим:

$ ps > psout.txt

$ sort psout.txt > pasoirt.out

Соединение процессов каналом даст более элегантное решение:

$ ps | sort > pssort.out

При желании увидеть на экране вывод, разделенный на страницы, можно подсоединить третий процесс, more:

$ ps | sort | more

Предположим, что необходимо видеть все имена выполняющихся процессов, за исключением командных оболочек. Можно использовать следующую командную строку:

```
$ ps -xo comm | sort | uniq | grep -v sh | more
```

В ней берется вывод команды ps, сортируется в алфавитном порядке, из него извлекаются процессы с помощью команды uniq, применяется утилита grep -v sh для удаления процесса с именем sh и в завершение полученный список постранично выводится на экран. Это более элегантное решение, чем строка из отдельных команд, каждая со своими временными файлами.

1.3 Командная оболочка как средство программирования

Есть два способа написания программ оболочки. Вы можете ввести последовательность команд и разрешить командной оболочке выполнить их в интерактивном режиме или сохранить эти команды в файле и затем запускать его как программу.

Интерактивные программы

Легкий и очень полезный во время обучения или тестирования способ проверить работу небольших фрагментов кода— просто набрать с клавиатуры в командной строке сценарий командной оболочки.

Предположим, что у вас большое количество файлов на языке C, и вы хотите проверить наличие в них строки posix. Вместо того чтобы искать в файлах строку с помощью команды grep и затем выводить на экран отдельно каждый файл, можно выполнить всю операцию в интерактивном сценарии:

```
$ for file in *
> do
> if grep -l POSIX $file
> then
> more $file
> fi
> done posix
This is a file with POSIX in it - treat it well
$
```

Обратите внимание на то, как меняется знак $, стандартное приглашение командной оболочки, на символ >, когда оболочка ожидает очередной ввод. Вы можете продолжить набор, дав оболочке понять, когда закончите, и сценарий немедленно выполнится.

В этом примере команда grep выводит на экран найденные ею имена файлов, содержащих строку posix, а затем команда more отображает на экране содержимое файла. В конце на экран возвращается приглашение

командной оболочки. Обратите внимание также на то, что вы ввели переменную командной оболочки, которая обрабатывает каждый файл для самодокументирования сценария. С таким же успехом можно использовать переменную i, но имя file более информативно с точки зрения пользователей.

Командная оболочка также обрабатывает групповые символы или метасимволы (часто называемые знаками подстановки). Например, символ * - знак подстановки, соответствующий строке символов, односимвольный знак подстановки ? соответствует одиночному символу. Подстановочный шаблон из фигурных скобок {} позволяет формировать множество из произвольных строк, которое командная оболочка раскроет. Например, команда

```
$  ls my_{finger, toe}s
```

будет проверять файлы с именами my_figers и my_toes в текущем каталоге.

Каждый раз вводить последовательность команд утомительно. Можно сохранить команды в файле, который принято называть **сценарием** или **скриптом** командной оболочки, а затем выполнять эти файлы.

Создание сценария

Создать файл, содержащий команды, можно помощью любого текстового редактора. В данной работе рекомендуется использовать встроенный в mc редактор. Для создания нового файла в mc используйте комбинацию клавиш Shift+F4. Создайте файл с именем first с таким содержимым:

```
#!/bin/sh
# first
# Этот файл просматривает все файлы в текущем каталоге для поиска строки
# POSIX, а затем выводит имена найденных файлов в стандартный вывод.
for file in *
do
  if grep -q POSIX $file
  then
   echo $file
  fi
done
exit 0
```

Комментарий начинается со знака # и продолжается до конца строки. Принято знак # ставить в первой символьной позиции строки. Первая строка #! /bin/sh — это особая форма комментария; символы #! сообщают системе о том, что следующий за ними аргумент — программа, применяемая для выполнения данного файла. В данном случае программа

/bin/sh — командная оболочка, применяемая по умолчанию.

Команда exit гарантирует, что сценарий вернет осмысленный код завершения. Он редко проверяется при интерактивном выполнении программ, но если вы хотите запускать данный сценарий из другого сценария и проверять, успешно ли он завершился, возврат соответствующего кода завершения очень важен. Даже если вы не намерены разрешать вашему сценарию запускаться из другого сценария, все равно следует завершать его с подходящим кодом.

В программировании средствами командной оболочки ноль означает успех. Поскольку представленный вариант сценария не может обнаружить какие-либо ошибки, он всегда возвращает код успешного завершения.

В сценарии не используются никакие расширения и суффиксы имен файлов; ОС Linux и UNIX, как правило, редко применяют при именовании файлов расширения для указания типа файла.

Превращение сценария в исполняемый файл

Файл сценария можно выполнить двумя способами. Более простой путь — запустить оболочку с именем файла сценария как параметром:

```
$ /bin/sh first
```

Этот вариант будет работать, но лучше запускать сценарий, введя его имя и тем самым присвоив ему статус других команд Linux. Сделать это можно с помощью команды chmod, изменив режим файла (file mode) и сделав его исполняемым для всех пользователей:

```
$ chmod +x first
```

После этого можно выполнять файл с помощью команды $ first

При этом может появиться сообщение об ошибке, говорящее о том, что команда не найдена. Исправить ошибку можно введя с клавиатуры в командной строке ./first в каталоге, содержащем сценарий, чтобы задать командной оболочке полный относительный путь к файлу.

Указание пути, начинающегося с символов ./, дает еще одно преимущество: в этом случае вы случайно не сможете выполнить другую команду с тем же именем, что и у вашего файла сценария.

После того как вы убедитесь в корректной работе вашего сценария, можете переместить его в более подходящее место, чем текущий каталог. Если команда предназначена только для собственных нужд, можете создать каталог bin в своем исходном каталоге и добавить его в свой путь. Если вы хотите, чтобы сценарий выполняли другие пользователи, можно использовать каталог /usr/local/bin или другой системный каталог как удобное хранилище для вновь созданных программ.

1.4 Синтаксис языка командной оболочки

Переменные

В командной оболочке переменные перед применением **обычно не объявляются**. Вместо этого они создаются (например, когда им присваивается начальное значение). По умолчанию все переменные считаются **строками и хранятся как строки**, даже когда им присваиваются числовые значения. Командная оболочка и некоторые утилиты преобразуют строки, содержащие числа, в числовые значения, когда с переменными нужно выполнить арифметические операции. Командная оболочка считает foo и Foo двумя разными переменными, отличающимися от третьей переменной FOO.

В командной оболочке можно получить доступ к содержимому переменной, если перед ее именем ввести знак $. Каждый раз, когда вы извлекаете содержимое переменной, вы должны перед ее именем добавить знак $. Когда вы присваиваете переменной значение, просто используйте имя переменной, которая при необходимости будет создана динамически. Вы можете увидеть это в действии, если в командной строке будете задавать и проверять разные значения переменной salut:

```
$ salut=Hello
$ echo $salut
Hello
$ salut="Yes Dear"
$ echo $salut
Yes Dear
$ salut=7+5
$ echo $salut
7+5
```

Примечания:

1. При наличии пробелов в содержимом переменной ее заключают в кавычки. Кроме того, не может быть пробелов справа и слева от знака равенства.

2. Для выполнения арифметических операций над целыми числами или целочисленными переменными следует использовать двойные круглые скобки:

```
$ echo $((7+5))
12
$ a=5
$ b=4
$ echo $(((a+b)*(a-b)))
9
```

С помощью команды **read** можно присвоить переменной пользовательский ввод. Команда принимает один параметр — имя переменной, в которую будут считываться данные и затем ждет, пока пользователь введет какой-либо текст. Команда read обычно завершается после нажатия пользователем клавиши <Enter>. При чтении переменной с терминала, как правило, заключать ее значения в кавычки не требуется:

```
$ read salut
Wie geht's?
$ echo $salut
Wie geht's?
```

Правила использования кавычек

Обычно параметры в сценариях отделяются неотображаемыми символами или знаками форматирования (например, пробелом, знаком табуляции или символом перехода на новую строку). Если вы хотите, чтобы параметр содержал один или несколько неотображаемых символов, его следует заключить в кавычки.

Поведение переменных, таких как $foo, заключенных в кавычки, зависит от **вида** используемых кавычек. Если вы заключаете в *двойные кавычки* $-представление переменной, оно во время выполнения командной строки заменяется значением переменной. Если вы заключаете его в *одинарные кавычки* или апострофы, никакой замены не происходит.

Пример 1.

В этом примере показано, как кавычки влияют на вывод переменной:

```
#!/bin/sh
myvar="Hi there"
echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar
echo Enter some text
read myvar
echo '$myvar' $myvar
exit 0
```

Данный сценарий ведет себя следующим образом:

```
$ ./variable
Hi there
Hi there
$myvar
$myvar
Enter some text
Hello world
$myvar Hello World
```

В сценарии создается переменная myvar, и ей присваивается строка Hi there. Содержимое переменной выводится на экран с помощью команды echo, демонстрирующей, как символ $ раскрывает содержимое переменной. Применение двойных кавычек не влияет на раскрытие содержимого переменной, а одинарные кавычки и обратный слэш влияют. Показано использование команды read для получения строки от пользователя.

Переменные окружения

При старте сценария командной оболочки некоторым переменным присваиваются начальные значения из окружения или рабочей среды. Обычно такие переменные обозначают прописными буквами, чтобы отличать их в сценариях от определенных пользователем переменных (командной оболочки), которые принято обозначать строчными буквами. Например:

$HOME Исходный каталог текущего пользователя

$PATH Разделенный двоеточиями список каталогов для поиска команд

$PS1 Подсказка или приглашение командной строки. Часто это знак $, но в оболочке bash можно применять и более сложные варианты. Например, строка [\u@\h \w]$ — популярный стандарт, сообщающий в подсказке пользователю имя компьютера и текущий каталог, а также знак $.

$PS2 Дополнительная подсказка или приглашение, применяемое как приглашение для дополнительного ввода; обычно знак >

$# Количество передаваемых параметров.

Переменные-параметры

Если сценарий вызывается с параметрами, создается несколько дополнительных переменных. Если параметры не передаются, переменная окружения $# равна 0.

Переменные-параметры перечислены в (табл.1).

Таблица 1.

Переменные-параметры

Переменная-параметр	Описание
$1, $2, ...	Параметры, передаваемые сценарию
$*	Список всех параметров в единственной переменной, разделенных первым символом из переменной окружения ifs.

Условия

Основа всех языков программирования — средства проверки условий и выполнение различных действий с учетом результатов этой проверки. Рассмотрим условные конструкции, которые можно применять в сценариях командной оболочки, а затем познакомимся с использующими их управляющими структурами.

Сценарий командной оболочки может проверить код завершения любой команды, вызванной из командной строки, включая сценарии, написанные пользователями.

Команда test или [

На практике в большинстве сценариев широко используется команда [или test -логическая проверка командной оболочки. В некоторых системах команды [и test - синонимы, за исключением того, что при использовании команды [для удобочитаемости применяется и завершающая часть]. В программном коде команда [упрощает синтаксис и делает его более похожим на другие языки программирования.

Поскольку команда test не часто применяется за пределами сценариев командной оболочки, многие пользователи ОС Linux, никогда раньше не писавшие сценариев пытаются создавать простые программы и называют их test. Если такая программа не работает, вероятно, она конфликтует с командой оболочки test.

Представим команду test на примере одного простейшего условия: проверки наличия файла. Для нее понадобится следующая команда: test -f <имя_файла>, поэтому в сценарии можно написать

```
if test -f fred.c then
fi
```

То же самое можно записать следующим образом:

```
if [ -f fred.c ] then
fi
```

Код завершения команды test (выполнено ли условие) определяет, будет ли выполняться условный программный код.

Необходимо вставлять пробелы между квадратной скобкой [и проверяемым условием. Это легко усвоить, если запомнить, что вставить символ [— это все равно, что написать test, а после имени команды всегда нужно вставлять пробел.

Если слово then записано в той же строке, что и if, нужно добавить точку с запятой для отделения команды test от then:

```
if  [  -f fred.c  ];  then fi
```

Варианты условий, которые используются в команде test, делятся на три типа:

- строковые сравнения,
- числовые сравнения,

- проверка файловых флагов.

Эти условия описаны в (табл. 2).

Таблица 2.

Условия

Варианты условий	Результат
Сравнения строк	
Строка1 = Строка 2	True (истина), если строки одинаковы
Строка1 != Строка2	True (истина), если строки разные
-n Строка	True (истина), если Строка не null
-z Строка	True (истина), если Строка null (пустая строка)
Сравнения чисел	
Выр1 -eq Выр2	True (истина), если выражения равны
Выр1 -ne Выр2	True (истина), если выражения не равны
Выр1 -gt Выр2	True (истина), если Выр1 больше, чем Выр2
Выр1 -ge Выр2	True (истина), если Выр1 не меньше Выр2
Выр1 -lt Выр2	True (истина), если Выр1 меньше, чем Выр2
Выр1 -le Выр2	True (истина), если Выр1 не больше Выр2
!Выражение	True (истина), если Выражение ложно, и наоборот
Файловые флаги	
-d файл	True (истина), если файл— каталог
-e файл	True (истина), если файл существует.
-f файл	True (истина), если файл— обычный файл
-r файл	True (истина), если файл доступен для чтения
-s файл	True (истина), если файл ненулевого размера
-w файл	True (истина), если файл доступен для записи
-x файл	True (истина), если файл— исполняемый файл

Пример 2: тестирования состояния файла /bin/bash.

```
#!/bin/sh
if [ -f /bin/bash ]
then
echo "file  /bin/bash exists"
fi

if [ -d /bin/bash  ]
then
  echo  "/bin/bash  is a directory"
else
    echo "/bin/bash is NOT a directory"
fi
```

Для того чтобы тест мог оказаться истинным, предварительно, для

проверки всех файловых флагов требуется наличие файла. Данный перечень включает только самые широко используемые опции команды test, полный список можно найти в интерактивном справочном руководстве.

Управляющие структуры

В командной оболочке есть ряд управляющих структур или конструкций, похожих на аналогичные структуры в других языках программирования.

В следующих разделах элемент синтаксической записи операторы— это последовательности команд, которые выполняются, когда или пока условие удовлетворяется или пока оно не удовлетворяется.

Оператор разветвления if

Оператор if очень прост: он проверяет результат выполнения команды и затем в зависимости от условия выполняет ту или иную группу операторов.

```
if условие then
операторы
else
операторы
fi
```

Наиболее часто оператор if применяется, когда задается вопрос, и решение принимается в зависимости от ответа.

Пример 3:

```
#!/bin/sh
echo "Сейчас утро? Ответьте yes или no"
read timeofday
if [ $timeofday = "yes" ]; then
  echo "Доброе утро"
else
  echo "Добрый вечер"
fi
exit 0
```

В результате будет получен следующий вывод на экран:

```
Сейчас утро? Ответьте yes или no yes
yes
Доброе утро
$
```

В этом сценарии для проверки содержимого переменной timeofday применяется команда [. Результат оценивается оператором if, который затем разрешает выполнять разные строки программного кода.

Дополнительные пробелы, используемые для формирования отступа

внутри оператора if нужны только для удобства читателя; командная оболочка их игнорирует.

Конструкция elif

К сожалению, с этим простым сценарием связано несколько проблем. Во-первых, он принимает в значении no (нет) любой ответ за исключением yes (да). Можно усовершенствовать сценарий, воспользовавшись конструкцией elif, которая позволяет добавить второе условие, проверяемое при выполнении части else оператора if (пример 4).

Можно откорректировать предыдущий сценарий так, чтобы он выводил сообщение об ошибке, если пользователь вводит что-либо отличное от yes или no. Для этого следует заменить ветку else веткой elif и добавить еще одно условие:

Пример 4:

```
#!/bin/sh
echo "Сейчас утро? Ответьте yes или no"
read timeofday
if  [ $timeofday = "yes" ]
then
echo  "Доброе утро"
elif [ $timeofday = "no" ]; then
  echo "Добрый вечер "
else
 echo "Извините, $timeofday не распознается. Ответьте yes или no "
exit 1
fi
exit 0
```

Пример 4 очень похож на предыдущий, но теперь, если первое условие не равно true, оператор командной оболочки elif проверяет переменную снова. Если обе проверки не удачны, выводится сообщение об ошибке, и сценарий завершается со значением 1, которое в вызывающей программе можно использовать для проверки успешного выполнения сценария.

Проблема, связанная со значением переменной

Данный сценарий исправляет наиболее очевидный дефект, а более тонкая проблема остается незамеченной. Запустите новый вариант сценария, но вместо ответа на вопрос просто нажмите клавишу <Enter> . Вы получите сообщение об ошибке:

[: = : unary operator expected

Что же не так? Проблема в первой ветви оператора if. Когда проверялась переменная timeofday, она состояла из пустой строки. Следовательно, ветвь оператора if выглядела следующим образом: if [= "yes"] и не представляла собой верное условие. Во избежание этого

следует заключить имя переменной в кавычки: if ["$timeofday" = "yes"]

Теперь проверка с пустой переменной будет корректной:

if ["" = "yes"]

Новый сценарий будет таким:

Пример 5:

```
#!/bin/sh
echo " Сейчас утро? Ответьте yes или no "
read timeofday
if  [ "$timeofday " = "yes" ]
then
  echo  "Доброе утро"
elif [ "$timeofday" = "no" ]; then
  echo "Добрый вечер "
else
 echo "Извините, $timeofday не распознается. Ответьте yes или no "
exit 1
fi
exit 0
```

Этот вариант безопасен, даже если пользователь в ответ на вопрос просто нажмет клавишу <Enter>.

Примечание. Если вы хотите, чтобы команда echo не переходила на новую строку, наиболее переносимый вариант— применить команду printf (см. раздел "printf" далее) вместо команды echo. В оболочке bash для запрета перехода на новую строку допускается команда echo –n. Поэтому можно написать:

echo -n " Сейчас утро? Ответьте yes или no: "

Нужно оставлять дополнительный пробел перед закрывающими кавычками для формирования зазора перед вводимым пользователем ответом.

Проверка выполнения нескольких условий (выполнение нескольких команд):

Иногда необходимо выполнить оператор, только если удовлетворяется несколько условий, например

```
if   [  -f this_file  ]; then
  if   [  -f that_file  ]; then
          if   [  -f other_file  ];   then
          echo  "All  files present"
          fi
  fi
fi
```

В другом случае может потребоваться, чтобы хотя бы одно условие из последовательности условий было истинным.

```
if  [ -f this_file  ];   then
```

```
  foo="True"
elif  [  -f that_file  ];  then
  foo="True"
elif  [  -f  the_other_file  ];  then
  foo="True"
else
  foo="False"
  fi
 if  [ "$foo" = "True"  ];  then
  echo  "One of the files exists"
 fi
```

Несмотря на то, что это можно реализовать с помощью нескольких операторов if, результаты получаются очень громоздкими. В командной оболочке есть пара специальных конструкций для работы со списками условий: И-список (AND list) и ИЛИ-список (OR list). Обе они часто применяются вместе, но мы рассмотрим синтаксическую запись каждой из них отдельно.

И-список

Эта конструкция позволяет выполнять последовательность команд, причем каждая последующая выполняется **только** при успешном завершении предыдущей. Синтаксическая запись такова: *оператор1 && оператор2 && оператор3 && ...*

Выполнение операторов начинается с самого левого, если он возвращает значение true (истина), выполняется оператор, расположенный справа от первого оператора. Выполнение продолжается до тех пор, пока очередной оператор не вернет значение false (ложь), после чего никакие операторы списка не выполняются. Операция && проверяет условие предшествующей команды.

Каждый оператор выполняется независимо, позволяя соединять в одном списке множество разных команд, как показано в приведенном далее сценарии. И-список успешно обрабатывается, если все команды выполнены успешно, в противном случае его обработка заканчивается неудачно.

Пример 6: И-список

В следующем сценарии выполняется обращение к файлу file_one (для проверки его наличия, и если файл не существует, он создается) и затем удаляется файл file_two. Далее И-список проверяет наличие каждого файла и между делом выводит на экран кое-какой текст.

```
#!/bin/sh
touch file_one
rm -f file_two
if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo "there"
then
```

```
  echo "in if"
else
  echo "in else"
fi
exit 0
```

Результат выполнения сценария

```
hello
in else
```

В примере 6 команды touch и rm гарантируют, что файл file_one в текущем каталоге существует, а файл file_two отсутствует. И-список выполняет команду [-f file_one], которая возвращает значение true, потому что файл существует. Поскольку предыдущий оператор завершился успешно, теперь выполняется команда echo. Она тоже завершается успешно (echo всегда возвращает true). Затем выполняется третья проверка [-f fiie_two]. Она возвращает значение false, т. к. файл не существует. Поскольку последняя команда вернула false, заключительная команда echo не выполняется. В результате И-список возвращает значение false, поэтому в операторе if выполняется вариант else.

ИЛИ-список

Эта конструкция позволяет выполнять последовательность команд до тех пор, пока одна из них не вернет значение true, и далее не выполняется ничего более. У нее следующая синтаксическая запись:

оператор1 || оператор2 || оператор3 || ...

Операторы выполняются слева направо. Если очередной оператор возвращает значение false, выполняется следующий за ним оператор. Это продолжается до тех пор, пока очередной оператор не вернет значение true, после этого никакие операторы уже не выполняются.

ИЛИ-список очень похож на И-список, за исключением того, что правило для выполнения следующего оператора - выполнение предыдущего оператора со значением false.

Пример 7:

```
#!/bin/sh
rm –f file_one
if [ -f file_one ] || echo "hello" || echo "there"
then
  echo "in if"
else
  echo "in else"
fi
exit 0
```

В результате выполнения данного сценария будет получен следующий вывод:

```
hello
in if
```

В первых двух строках просто задаются файлы для остальной части сценария. Первая команда списка [-f file one] возвращает значение false, потому что файла в каталоге нет. Далее выполняется команда echo, она возвращает значение true, и больше в ИЛИ-списке не выполняются никакие команды. Оператор if получает из списка значение true, поскольку одна из команд ИЛИ-списка (команда echo) вернула это значение.

Результат, возвращаемый обоими этими списками, — это результат последней выполненной команды списка.

Описанные конструкции списков выполняются так же, как аналогичные конструкции в языке C, когда проверяются множественные условия. Для определения результата выполняется минимальное количество операторов. Операторы, не влияющие на конечный результат, не выполняются. Обычно этот подход называют оптимизацией вычислений (short circuit evaluation).

Попробуйте проанализировать следующий список:

```
[  -f file_one ] &&  команда  в  случае  true  ||   команда в  случае false
```

В нем будет выполняться первая команда в случае истинности проверки и вторая команда в противном случае. Всегда лучше всего поэкспериментировать с этими довольно необычными списками, и, как правило, придется использовать скобки для изменения порядка вычислений.

Операторные блоки

Если необходимо использовать несколько операторов в том месте программного кода, где разрешен только один, например в ИЛИ-списке или И-списке, то можете сделать это, заключив операторы в фигурные скобки {} и создав тем самым **операторный блок**. Например:

```
get_confirm &&   {
grep -v "$cdcatnum"  $tracks_file >  $temp_file
cat $temp_file > $tracks_file
echo
add_record_tracks
                        }
```

Оператор выбора case

Оператор case немного сложнее уже рассмотренных ранее операторов. Формат записи оператора следующий:

```
case переменная in
образец  [ | образец]   ...)   операторы;;
образец  [ | образец]   ...)   операторы;;
```

esac

Конструкция оператора case позволяет сопоставлять содержимое переменной с образцами и затем выполнять разные операторы в зависимости от того, с каким образцом найдено соответствие. Это гораздо проще, чем проверять несколько условий, применяемых во множественных операторах if, elif и else.

Каждая ветвь с образцами завершается удвоенным символом точка с запятой (;;). В каждой ветви оператора case можно поместить несколько операторов, поэтому сдвоенная точка с запятой необходима для отметки завершения очередного оператора и начала следующей ветви с новым образцом в операторе case.

Оператор цикла for

Цикл for предназначен для обработки в цикле ряда значений, которые могут представлять собой любое множество строк. Строки могут быть просто перечислены в программе или, что бывает чаще, представлять собой результат выполненной командной оболочкой подстановки имен файлов. Синтаксис оператора цикла:

```
for переменная in значения
do
  операторы
done
```

Пример 8:

```
#!/bin/sh
for foo in bar dog 413 do
 echo $foo
done
exit 0
```

В результате будет получен следующий вывод:

```
bar
dog
413
```

Что произойдет, если изменить первую строку с for foo in bar dog 413 на for foo in "bar dog 413"? Напоминаем, что вставка кавычек заставляет командную оболочку считать все, что находится между ними, единой строкой. Это один из способов сохранения пробелов в переменной.

Как работает цикл

В данном примере создается переменная foo и ей в каждом проходе цикла for присваиваются разные значения. Поскольку оболочка считает по умолчанию все переменные строковыми, применять строку 413 так же допустимо, как и строку dog.

Пример 9:Применение цикла for с метасимволами

Цикл for обычно используется в командной оболочке вместе с метасимволами или знаками подстановки для имен файлов. Это означает применение метасимвола для строковых значений и предоставление оболочке возможности подставлять все значения на этапе выполнения.

Этот прием использован в первом примере first. В сценарии применялись средства подстановки командной оболочки — символ * для подстановки имен всех файлов из текущего каталога. Каждое из этих имен по очереди используется в качестве значения переменной $file внутри цикла for.

Рассмотрим еще один пример подстановки с помощью метасимвола. Допустим, нужно вывести на экран все имена файлов сценариев в текущем каталоге, начинающиеся с буквы "f" , и имена всех сценариев заканчиваются символами .sh. Это можно сделать следующим образом:

```
#!/bin/sh
for file in $(ls f*.sh); do
 echo $file
done
echo $file
exit 0
```

В примере показано применение синтаксической конструкции $(команда). Обычно список параметров для цикла for задается выводом команды, включенной в конструкцию $(). Командная оболочка раскрывает f*.sh, подставляя имена всех файлов, соответствующих данному шаблону.

Все подстановки переменных в сценариях командной оболочки делаются во время выполнения сценария, а не в процессе их написания, поэтому все синтаксические ошибки в объявлениях переменных обнаруживаются только на этапе выполнения, как было показано ранее, когда мы заключали в кавычки пустые переменные.

Цикл while

Поскольку по умолчанию командная оболочка считает все значения строками, оператор for хорош для циклической обработки наборов строк, но не слишком удобен, если не известно заранее, сколько раз придется его выполнить.

Если нужно повторить выполнение последовательности команд, но заранее не известно, сколько раз следует их выполнить, применяется цикл while со следующей синтаксической записью:

```
while условие do
операторы
done
```

Пример 10: Программа проверки паролей.

```
#!/bin/sh
```

```
echo "Enter password"
read trythis
while  [  "$trythis" ! = "secret"  ];  do
echo  "Sorry, try again"
read trythis
done
exit 0
```

Следующие строки - пример вывода данного сценария:

```
Enter password
password
Sorry,  try again
secret
$
```

Это небезопасный способ выяснения пароля, но он вполне подходит для демонстрации применения цикла while. Операторы, находящиеся между операторами do и done, выполняются бесконечное число раз до тех пор, пока условие остается истинным (true). В данном случае проверяется, равно ли значение переменной trythis строке secret. Цикл будет выполняться, пока $trythis не равно secret. Затем выполнение сценария продолжится с оператора, следующего сразу за оператором done.

Цикл until

У цикла until следующая синтаксическая запись:

```
until условие
do
операторы
done
```

Запись очень похожа на синтаксическую запись цикла while, но с обратным проверяемым условием. Другими словами, цикл продолжает выполняться, пока условие не станет истинным (true).

Пример 11: Вычисление суммы целых чисел, вводимых с клавиатуры. Признак окончания ввода – число 0.

```
#!/bin/sh
sum=0
read num
until [ $num –eq 0 ]; do
 sum=$((sum+num))
 read num
done
echo $sum
exit 0
```

Как правило, если нужно выполнить цикл хотя бы один раз,

применяют цикл while; если такой необходимости нет, используют цикл until.

Функции

В командной оболочке можно определять функции, используемые для структурирования кода. Как альтернативу можно использовать разбиение большого сценария на много маленьких, каждый из которых выполняет небольшую задачу. У этого подхода есть несколько недостатков: выполнение вложенного в сценарий другого сценария будет гораздо медленнее, чем выполнение функции и значительно труднее возвращать результаты.

Для определения функции в командной оболочке просто введите ее имя и следом за ним пустые круглые скобки, а операторы тела функции заключите в фигурные скобки.

Имя_функции ()
{
операторы
}

Пример 12. Простая функция

Начнем с действительно простой функции.

```
#!/bin/sh
foo ()   {
  echo "Function foo is executing"
          }
echo "script starting"
foo
echo "script ended"
exit 0
```

Сценарий выведет на экран следующий текст:

```
script starting
Function foo is executing
script ended
```

Данный сценарий начинает выполняться с первой строки. Когда он находит конструкцию foo () {, он знает, что здесь дается определение функции, названной foo. Он запоминает ссылку на функцию и foo продолжает выполнение после обнаружения скобки }. Когда выполняется строка с единственным именем foo, командная оболочка знает, что нужно выполнить предварительно определенную функцию. Когда функция завершится, выполнение сценария продолжится в строке, следующей за вызовом функции foo.

Всегда необходимо определить функцию прежде, чем ее запускать, что похоже на стиль, принятый в языке программирования Pascal, когда

вызову функции предшествует ее определение, за исключением того, что в командной оболочке нет опережающего объявления (forward) функции. Это ограничение не создает проблем, потому что все сценарии выполняются с первой строки, поэтому если просто поместить все определения функций перед первым вызовом любой функции, все функции окажутся определенными до того, как будут вызваны.

Когда функция вызывается, позиционные параметры сценария $*, $#, $1, $2 и т. д. заменяются параметрами функции. Именно так считываются параметры, передаваемые функции. Когда функция завершится, они восстановят свои прежние значения.

Функция возвращает числовые значения с помощью команды return. Обычный способ возврата функцией строковых значений — сохранение строки в переменной, которую можно использовать после завершения функции. Другой способ — вывести строку с помощью команды echo и перехватить результат, как показано далее.

```
foo () { echo JAY;}
…
result="$(foo)"
```

В функциях командной оболочки можно объявлять локальные переменные с помощью ключевого слова local. В этом случае переменная действительна только в пределах функции. В других случаях функция может обращаться к переменным командной оболочки, у которых глобальная область действия. Если у локальной переменной то же имя, что и у глобальной, в пределах функции локальная переменная перекрывает глобальную. Для того чтобы убедиться в этом на практике, можно выполнить следующий сценарий.

```
#!/bin/sh
sample_text="global variable"
 foo () {
  local  sample_text="local variable"
  echo  "Function  foo  is  executing"
  echo $sample_text
}
echo "script starting"
echo $sample_text
foo
echo "script ended"
echo $sample_text
exit 0
```

При отсутствии команды return, задающей возвращаемое значение, функция возвращает статус завершения последней выполненной команды.

В следующем сценарии, my_name, показано, как в функцию передаются, параметры и как функции могут вернуть логический результат

true или false. Этот сценарий можно вызвать с параметром, задающим имя, которое необходимо использовать в вопросе.

1.После заголовка командной оболочки определите функцию yes_or_no

```
 #!/bin/sh
yes_or_no () {
echo "Is your name $* ? "
while true do
echo -n "Enter yes or no: "
read x
case  "$x"  in
y | yes ) return 0;;
n | no ) return 1 ;;
*  ) echo "Answer yes or no"
esac
done
}
```

2. Далее начинается основная часть программы.

```
echo  "Original parameters  are  $*"
if yes_or_no  "$1"
then
  echo "Hi $1, nice name"
else
 echo "Never mind"
fi
exit 0
```

Типичный вывод этого сценария может выглядеть следующим образом:

```
$ ./my_name Rick Neil
Original parameters are Rick Neil
Is your name Rick ?
Enter yes or no: yes
Hi Rick, nice name
$
```

Как работает сценарий

Когда сценарий начинает выполняться, функция определена, но еще не выполняется. В операторе if сценарий вызывает функцию yes_or_no, передавая ей оставшуюся часть строки как параметры после замены $1 первым параметром исходного сценария строкой Rick. Функция использует эти параметры, в данный момент хранящиеся в позиционных параметрах $1, $2 и т. д., и возвращает значение в вызывающую программу. В зависимости от возвращенного функцией значения

конструкция if выполняет один из операторов.

Команды

В сценариях командной оболочки можно выполнять два сорта команд. Как уже упоминалось, существуют "обычные" команды, которые могут выполняться и из командной строки (называемые внешними командами), и встроенные команды (называемые внутренними командами). Внутренние команды реализованы внутри оболочки и не могут вызываться как внешние программы. Но большинство внутренних команд представлено и в виде автономных программ, это условие - часть требований стандарта POSIX. Обычно не важно, команда внешняя или внутренняя, за исключением того, что внутренние команды выполняются быстрее.

В этом разделе представлены основные команды, как внутренние, так и внешние, которые используются при написании сценариев.

Команда break

Команда break используется для выхода из циклов for, while и until до того, как будет удовлетворено управляющее условие.

Пример 13:

```
#!/bin/sh
rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4
for file in fred*
 do
  if [ -d "$file" ]; then
  break;
fi
done
echo first directory starting fred was  $file
rm -rf fred*
exit 0
```

Команда continue

Как и одноименный оператор языка C, эта команда заставляет охватывающий ее цикл for, while нли until начать новый проход или следующую итерацию. При этом переменная цикла принимает следующее значение в списке.

Пример 14:

```
#!/bin/sh
rm -rf fred*
```

```
echo > fred1
echo > fred2
mkdir fred3
echo > fred4
for file in fred*
do
  if [ -d "$file" ]; then
  echo "skipping directory $file"
  continue
  fi
  echo file is $file
done
rm -rf fred*
exit 0
```

Команда continue может принимать в качестве необязательного параметра номер прохода охватывающего цикла, с которого следует возобновить выполнение цикла.

Таким образом, можно иногда выскочить из вложенных циклов. Данный параметр редко применяется, т. к. он часто сильно затрудняет понимание сценариев. Например,

```
for x in  1 2 3
do
  echo before $x
  continue  1
  echo after $x
done
```

У приведенного фрагмента будет следующий вывод:

```
before 1
before 2
before 3
```

Команды echo и printf

При использовании команды **echo** возникает общая проблема - удаление символа перехода на новую строку. В ОС Linux общепринятый метод - echo -n "string to output"

printf

Команда printf есть только в современных командных оболочках. Группа X/Opeп полагает, что ее следует применять вместо команды echo для генерации форматированного вывода. У команды **printf** следующая синтаксическая запись.

printf "строка формата" параметр1 параметр2 ...

Строка формата очень похожа с некоторыми ограничениями на применяемую в языках программирования C и C++. Главным образом не

поддерживаются числа с плавающей точкой, поскольку все арифметические операции в командной оболочке выполняются над целыми числами. Строка формата состоит из произвольной комбинации литеральных символов, escape-последовательностей и спецификаторов преобразования. Все символы строки формата, отличающиеся от \ и %, отображаются на экране при выводе.

В (табл.3) приведены поддерживаемые командой escape-последовательности.

Таблица 3.

Некоторые escape-последовательности

Escape-последовательность	Описание
\"	Двойная кавычка
\\	Символ обратный слэш
\n	Символ перехода на новую строку
\r	Возврат каретки
\t	Символ табуляции
\v	Символ вертикальной табуляции
\ooo	Один символ с восьмеричным значением ooo
\xHH	Один символ с шестнадцатеричным значением HH

Спецификатор преобразования состоит из символа %, за которым следует символ преобразования. Основные варианты преобразований перечислены в (табл. 4).

Таблица 4.

Спецификаторы преобразования

Символ преобразования	Описание
d	Вывод десятичного числа
c	Вывод символа
s	Вывод строки
%	Вывод знака %

Строка формата используется для интерпретации остальных параметров команды и вывода результата, как показано в следующем примере:

```
$ printf     "%s\n" hello
hello
$ printf     "%s %d\t%s" "Hi There" 15 people
Hi There  15  people
```

Обратите внимание на то, что для превращения строки Hi There в

единый параметр ее нужно заключить в кавычки ("").

Команда return

Команда return служит для возврата значений из функций, как уже упоминалось ранее при обсуждении функций. Команда принимает один числовой параметр, который становится доступен в сценарии, вызывающем функцию. Если параметр не задан, команда return по умолчанию возвращает код завершения последней команды.

Команда shift

Команда shift сдвигает все переменные-параметры на одну позицию назад, так что параметр $2 становится параметром $1, параметр $3 - $2 и т. д. Предыдущее значение параметра $1 отбрасывается, а значение параметра $0 остается неизменным. Если в вызове команды shift задан числовой параметр, параметры сдвигаются на указанное количество позиций. Остальные переменные $* и $# также изменяются в связи с новой расстановкой переменных-параметров.

Команда shift часто полезна при поочередном просмотре параметров, переданных в сценарий, и если вашему сценарию требуется 10 и более параметров, вам понадобится команда shift для обращения к 10-му параметру и следующим за ним.

Например, вы можете просмотреть все позиционные параметры:

```
#!/bin/sh
while [ "S1" != ""]; do
  echo "$1"
  shift
done
exit 0
```

Команда stat

Команда stat предназначена для получения информации об указанном файле и о свойствах файловой системы на носителе, на котором хранится указанный файл.

Примеры использования команды:

Команда

stat res

где res – имя файла, выводит на экран всю информацию о файле с именем res и о владельце этого файла.

Команда

stat –f res

где res – имя файла, выводит на экран всю информацию о файловой системе на диске, на котором хранится файл с именем res.

Для получения доступа к отдельным полям информации о файле или

файловой системе к приведенным выше командам добавляется ключ –c и параметр, определяющий поле. Например, для получения размера файла в байтах должен быть указан ключ %s и команда stat записывается в виде

```
stat  res –c  %s
```

Список ключей команды stat можно получить с помощью команды stat --help

Выполнение команд и получение результатов их выполнения

При написании сценариев часто требуется перехватить результат выполнения команды для использования его в сценарии командной оболочки; т. е. выполнить команду и поместить ее вывод в переменную. Сделать это можно с помощью синтаксической конструкции $(команда).

Результат выполнения конструкции $ (команда) — просто вывод команды. Имейте в виду, что это не статус возврата команды, а просто строковый вывод, показанный далее.

```
#!/bin/sh
echo The current directory is  $PWD
echo The current users  are  $(who)
exit 0
```

Поскольку текущий каталог — это переменная окружения командной оболочки, первая строка не нуждается в применении подстановки команды. Результат выполнения программы who, напротив, нуждается в ней, если он должен стать переменной в сценарии.

Если вы хотите поместить результат в переменную, то можете просто присвоить его обычным образом whoisthere=$(who)

Возможность поместить результат выполнения команды в переменную сценария - мощное средство, поскольку оно облегчает использование существующих команд в сценариях и перехват результата их выполнения. Если необходимо преобразовать набор параметров, представляющих собой вывод команды на стандартное устройство вывода, и передать их как аргументы в программу, команда xargs сможет это сделать.

Встроенные документы

Особый способ передачи из сценария командной оьолочки входных данных команде – использование *встроенного документа*. Такой документ позволяет команде выполняться так, как будто она читает данные из файла или с клавиатуты, в то время как на самом деле она получает их из сценария.

Встроенный документ начинается со служебных символов <<, за которыми следует специальная символьная последовательность, повторяющаяся и в конце документа. Символы << обозначают в командной оболочке перенаправление данных, которое в данном случае

заставляет вход команды превратиться во встроенный документ. Специальная последовательность символов действует как маркер, указывая оболочке, где завершается встроенный документ. Маркерная последовательность не должна включаться в строки, передаваемые команде.

Пример 15: Применение встроенного документа

```
#!/bin/sh
cat <<!BUILTdoc!
Это пример встроенного
документа для описания сценария
!BUILTdoc!
```

Пример 15 выводит на экран следующие строки

```
Это пример встроенного
документа для описания сценария
```

Отладка сценариев

При обноружении ошибки при выполнении сценария командная оболочка выводит на экран номер строки, содержащей ошибку. Если ошибка сразу не видна, нужно добавить несколько дополнительных команд echo для вывода значений переменных и протестировать фрагменты программного кода, вводя их в командной оболочке в интерактивном режиме. Основной способ отслеживания наиболее трудно выявляемых ошибок – использование отладочных опций командной оболочки. Отладочные опции командной строки приведены в (табл. 5).

Таблица 5.

Отладочные опции командной строки

Опция	Назначение
sh –n <сценарий>	Только проверяет синтаксические ошибки
sh –v <сценарий>	Выводит на экран команды перед их выполнением
sh –x <сценарий>	Выводит на экран команды после обработки командной строки
sh –u <сценарий>	Выдает сообщение об ошибке при использовании неопределенной переменной

2 МЕТОДИКА ВЫПОЛНЕНИЯ

1. Получить полный список ключей команды stat.
2. Вычислить факториал целого числа, вводимого с клавиатуры. Предусмотреть проверку правильности ввода аргумента.
3. Найти первые N чисел Фибоначчи, используя рекуррентное

соотношение

$$A_{i+1}=A_i+A_{i-1}$$

Значения первых двух чисел и необходимое количество чисел N ввести с клавиатуры.

4. Написать и выполнить сценарии для решения индивидуальных задач, номер задачи определяется номером бригады:
5. Найти суммарный объем выполняемых файлов в текущем каталоге.
6. В текущем каталоге найти выполняемый файл наибольшего размера.
7. Вывести имена файлов текущего каталога, начинающиеся на букву a или b, в которые можно записывать данные.
8. В текущем каталоге найти имя файла, который был изменен позже всех. На экран вывести дату изменения и имя файла.
9. Написать сценарий для проверки, имеются ли в двух подкаталогах, имена которых задаются первым и вторым параметрами сценария, файлы с одинаковыми именами. Количество файлов с одинаковыми именами и имена файлов вывести на экран.
10. Для каждого подкаталога текущего каталога найти количество файлов. Вывести имена подкаталогов и количество файлов в этом каталоге.
11. В текущем каталоге найти количество файлов, имеющих различные имена, но одинаковые размеры. Вывести на экран величину размера и имена файлов, имеющих данный размер.
12. В текущем каталоге и его подкаталогах найти файлы, созданные в течение последней недели.

Используя команду printf, написать сценарий для перевода введенного с клавиатуры целого положительного числа в восьмеричную и шестнадцатеричную системы счисления.

3 ОТЧЕТ О РАБОТЕ

Готовится в письменном виде один на бригаду. Содержание отчета:

1. Результаты выполнения заданий 1- 3 – тексты сценариев и результаты их выполнения
2. Результаты выполнения индивидуального задания для бригады - текст сценария и результаты его выполнения.

4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назначение, создание и выполнение сценариев.
2. Использование кавычек в командной строке.
3. Переменные в bash.
4. Перенаправление ввода-вывода и каналы

5. Превращение сценария в исполняемый файл
6. Команда test или [
7. Оператор разветвления if
8. Проверка выполнения нескольких условий (выполнение нескольких команд)
9. Оператор выбора case
10. Операторы цикла
11. Команды break и continue - назначение, примеры использования
12. Команда printf – назначение, отличия от языка C, примеры использования
13. Встроенные документы.
14. Отладка сценариев