Тема 9. Основы программирования на языке shell (bash scripting)

На СЛАЙДЕ 1 – обсуждаемые вопросы, а мы начнем с традиционного для студентов программистских специальностей (особенно младших курсов) вопроса.

9.1 Ну и зачем нам это нужно?

Знание языка командной оболочки является залогом успешного решения задач администрирования системы. Даже если мы не мечтаем заниматься написанием своих сценариев, мы должны помнить, что во время загрузки Linux выполняется целый ряд сценариев из */etc/rc.d*. Они настраивают конфигурацию ОС и запускают различные сервисы, поэтому очень важно четко понимать эти сценарии (далее – скрипты) и иметь достаточно знаний, чтобы вносить в них какие-либо изменения. СЛАЙД 2.

Язык сценариев легок в изучении, в нем не так много специфических операторов и конструкций. Синтаксис языка достаточно прост и прямолинеен, хотя иногда бывает загадочен. Он очень напоминает команды, которые приходится вводить в командной строке. Короткие скрипты практически не нуждаются в отладке, и даже отладка больших скриптов отнимает весьма незначительное время.

Shell-скрипты очень хорошо подходят для быстрого создания прототипов сложных приложений, даже несмотря на ограниченный набор языковых конструкций и определенную «медлительность». Это позволяет детально проработать структуру будущего приложения, обнаружить возможные «ловушки» и лишь затем приступить к кодированию, например, с использованием C/C++ или Perl. СЛАЙД 3.

Скрипты возвращают нас к классической философии *NIX – «разделяй и властвуй», т.е. разделение сложного проекта на ряд простых подзадач. Многие считают такой подход наилучшим или, по меньшей мере, наиболее удачным способом решения возникающих проблем, нежели использование нового поколения «языков-комбайнов», таких как Perl.

Для следующих классов задач скрипты неприменимы (СЛАЙД 4):

- для ресурсоемких задач, особенно когда важна скорость исполнения (поиск, сортировка и т.п.);
- для задач, связанных с выполнением математических вычислений, особенно это касается вычислений с плавающей запятой, вычислений с повышенной точностью, комплексных чисел (для таких задач лучше использовать C/C++ или FORTRAN);
- для кросс-платформенного системного программирования (для этого лучше подходит язык C);

- для сложных приложений, когда структурирование является жизненной необходимостью (контроль за типами переменных, прототипами функций и т.п.);
- когда во главу угла поставлена безопасность системы, когда необходимо обеспечить целостность системы и защитить ее от вторжения, взлома и вандализма;
- для проектов, содержащих компоненты, очень тесно взаимодействующие между собой;
- для задач, выполняющих огромный объем работы с файлами;
- для задач, работающих с многомерными массивами;
- когда необходимо работать со структурами данных, такими как связанные списки или деревья;
- когда необходимо предоставить графический интерфейс с пользователем (GUI);
- когда необходим прямой доступ к аппаратуре компьютера;
- когда необходимо выполнять обмен через порты ввода-вывода или сокеты;
- когда необходимо использовать внешние библиотеки;
- для проприетарных программ (т.к. скрипты представляют собой исходные тексты программ, доступные для всеобщего обозрения).

9.2 Что такое BASH?

Название BASH – это аббревиатура от "Bourne-Again Shell" и игра слов от ставшего уже классикой "Bourne Shell" Стивена Бурна (Stephen Bourne). В последние годы BASH достиг такой популярности, что стал стандартной командной оболочкой de facto для многих UNIX-подобных систем. Большинство принципов программирования на BASH одинаково хорошо применимы и в других командных оболочках, таких как Korn Shell (ksh и его потомок zsh), от которой Bash позаимствовал некоторые особенности, и C Shell и его производных. СЛАЙД 5.

Bash позволяет запускать другие программы, для этого он определяет их тип. Обычные программы – это системные команды, которые существуют в скомпилированном виде в ОС. Когда такая программа запускается, создается новый процесс, потому что оболочка создает полную свою копию. У этого дочернего процесса то же самое окружение, что и у родителя, отличаются только идентификаторы процессов (PID). А сама только что описанная процедура носит название **порождения процесса**, или **форка** (от англ. *fork*).

После порождения процесса адресное пространство дочернего процесса заполняется новыми данными процесса. Это делается с помощью системного вызова *call*. СЛАЙД 6.

Механизм *fork-and-exec*, таким образом, переключает старую команду на новую, тогда как окружение, в котором выполняется новая программа, остается тем же самым, включая конфигурацию устройств ввода-вывода, переменные окружения и значение приоритета. С помощью этого механизма создаются все процессы в ОС *NIX, так работает и Linux. Более того, даже самый первый процесс *init*, идентификатор которого 1 (PID == 1), порождается при **начальной загрузке** ОС (*bootstrapping*).

Внутри оболочки содержатся так называемые встроенные команды. Когда они запускаются, то новый процесс не создается. Встроенные команды необходимы для обеспечения функциональности, которую невозможно или неудобно получать с помощью отдельных утилит.

Bash поддерживает 3 типа встроенных команд (СЛАЙД 7):

- Встроенные команды, унаследованные от Bourne shell (:, ., *break*, *cd*, *continue*, *eval*, *exec*, *exit*, *export*, *getopts*, *hash*, *pwd*, *readonly*, *return*, *set*, *shift*, *test*, [, *times*, *trap*, *umask* и *unset*).
- Встроенные команды Bash (*alias*, *bind*, *builtin*, *command*, *declare*, *echo*, *enable*, *help*, *let*, *local*, *logout*, *printf*, *read*, *shopt*, *type*, *typeset*, *ulimit* и *unalias*).
- Специальные встроенные команды POSIX-режима (:, ., *break*, *continue*, *eval*, *exec*, *exit*, *export*, *readonly*, *return*, *set*, *shift*, *trap* и *unset*). Когда Bash выполняется в режиме

POSIX, специальные встроенные команды будут другими и их отличие будет в следующем:

- Во время поиска команды, команда сначала ищется среди специальных встроенных команд, а затем — среди функций командной оболочки.
- Если специальная встроенная команда возвращает состояние ошибки, то в неинтерактивном режиме происходит выход из командной оболочки.
- Инструкции присваивания, предшествующие команде, оказывают эффект на среду оболочки только после завершения команды.

Когда исполняемая программа – это shell-скрипт, bash создает новый процесс (*fork*). тем самым создается копия оболочки (subshell), которая пошагово читает строки скрипта и выполняет так, как если бы они работали будучи введенными напрямую с клавиатуры.

Пока подоболочка обрабатывает каждую строку скрипта, родительская оболочка ожидает завершения дочернего процесса. Когда все строки прочитаны и выполнены, подоболочка заканчивает работу, а родитель просыпается и отображает новую подсказку командной строки.

Если вход не содержит комментарии, то оболочка читает его, делит на слова, и используя правила цитирования, определяет значение каждого входного символа. Затем эти слова и операторы транслируются в команды и другие конструкции, которые возвращают статус завершения, доступный для проверки или обработки. Описанная выше схема *fork-and-exec* используется только после того, как оболочка проанализировала вход следующим способом (СЛАЙД 8):

- Оболочка считает своим входом файл, строку или пользовательский терминал.
- Вход разбивается на слова и операторы с использованием правил цитирования. Эти лексемы разделяются метасимволами. Выполняется расширение псевдонимов.
- Оболочка разбирает лексемы на простые и составные команды.
- Bash выполняет расширение оболочки, составляя из расширенных лексем списки имен файлов, команд и аргументов.
- Если необходимо, выполняется перенаправление. Операторы перенаправления и их операнды удаляются из списка аргументов.
- Выполняются команды.
- В некоторых случаях Bash ожидает завершения команды с сохранением ее кода завершения.

Простая команда оболочки, например:

```
$ touch file1
```

состоит из самой команды и последующих аргументов, отделенных пробелами. Более сложные команды составляются из простых, и делаться это может разными способами: с помощью так называемых конвейеров, когда выход одной команды становится входом другой команды; с помощью циклических и условных операторов; с помощью группирования и т.д. Например:

```
ls | more
```

или

```
gunzip file.tar.gz | tar xvf -
```

Одним из способов группирования команд являются функции языка оболочки в том же смысле, что и функции в компилируемых языках программирования. Они выполняются как обычные команды. Когда имя функции используется как простая команда, выполняется список команд, ассоциированных с этой функцией.

Версия 0.9pre-release от 22.03.2013. Возможны незначительные изменения.

Функция оболочки выполняются в текущем контексте, т.е. новый процесс не создается.

У команд и функций могут быть параметры. **Параметр** – это сущность, которая хранит значение. Это может быть имя, число или специальное значение. Для оболочки при решении задач переменная является параметром, который хранит имя. У переменной может быть значение и 0 или более атрибутов. Переменные могут создаваться с помощью встроенной команды объявления *declare*.

Если не задано значение, то переменной присваивается пустая строка. Переменные можно только удалять, для этого используется команда *unset*.

Расширения (подстановки) оболочки выполняются после того, как каждая строка команд разбивается на лексемы. Выполняются следующие подстановки (СЛАЙД 9):

- Подстановки фигурными скобками.
- Подстановки тильдой.
- Подстановки параметра и переменной.
- Подстановки команды.
- Арифметические подстановки.
- Разбиение на слова.
- Подстановки в имени файла.

Перед выполнением команды ее вход или выход могут быть перенаправлены с использованием специальной нотации, интерпретируемой оболочкой. Перенаправление можно также использовать для открытия и закрытия файлов в текущем окружении исполняющей оболочки.

Когда выполняется команда, для дальнейшего использования сохраняются слова, которые были отмечены как присваивание переменных или перенаправления. Остальные слова подставляются (расширяются); первое слово, оставшееся после расширения, трактуется как имя команды, а остальные – ее аргументы. Потом выполняются перенаправления, далее подставляются (расширяются) строки. Если в результате не получилось имя команды, то переменные окажут влияние на текущее окружение командной оболочки.

Важной функцией оболочки является поиск команд. Bash делает это в следующем порядке (СЛАЙД 10):

- Проверяет, содержит ли команда символы косой черты. Если нет, то первым делом проверяется список функций, чтобы проверить, есть ли там команда с именем, равным разыскиваемому.
- Если команда - не функция, то проверяется список встроенных команд.
- Если команда – это ни функция, ни встроенная команда, анализируются каталоги из переменной окружения PATH. Bash использует хэш-таблицу (структуру данных в основной памяти) для запоминания полных имен исполняемых модулей, так что экстенсивный поиск по PATH на самом деле не выполняется.
- Если поиск завершился неудачей, то Bash печатает сообщение об ошибке и возвращает код 127.
- Если же поиск успешен, или команда содержит символы косой черты, то оболочка запускает команду на выполнение в отдельном рабочем окружении.
- Если выполнение происходит с ошибкой из-за того, что файл не является исполняемым или каталогом, то он трактуется как скрипт оболочки.
- Если команда не запускалась асинхронно, оболочка ожидает завершения команды и получения кода завершения.

9.3 Наши первые скрипты

Для первого примера мы напишем скрипт, который выводит на экран сообщение

"Hello World".

В текстовом редакторе создадим скрипт *first.sh* с содержимым, приведенным на СЛАЙДЕ 11.

Первая строка указывает операционной системе, что скрипт должен выполняться программой */bin/sh*. Это стандартное размещение интерпретатора команд Bash в *nix-системах. В Linux-системах */bin/sh* как правило символьная ссылка на *bash*.

Вторая линия также начинается с символа диеза (#). Это помечает строку как комментарий и полностью игнорируется интерпретатором языка командной оболочки.

Единственное исключение из этого мы видели выше – самая первая строка в файле начинается с двух символов (#!). Это специальная директива, которую Linux трактует особо. Это означает, что даже если мы используем *csh*, *ksh* или что-то еще, все равно остальной код должен интерпретироваться оболочкой Bash.

Аналогично, кстати, Perl-скрипты могут начинаться со строки *#!/usr/bin/perl*, и это указывает оболочке, что программный код должен выполняться *perl*-транслятором. Для Bash-программирования, мы будем использовать *#!/bin/sh*.

Третья строка запускает команду *echo* с двумя строковыми аргументами: первый это "Hello", второй – "World".

Команда *echo* автоматически запишет один пробел между своими параметрами.

А дальше идет символ # и текст комментария, который игнорируется интерпретатором языка командной оболочки.

Теперь запустим:

```
chmod u+x first.sh
```

и этим сделаем текстовый файл исполняемым, а затем выполним его:

```
./first.sh
```

И мы скорее всего получим вполне ожидаемый результат.

Теперь сделаем несколько изменений. Для начала вспомним, что *echo* записывает ОДИН пробел между параметрами. Поместим несколько пробелов между "Hello" и "World". Что будет выведено? А если разместить один символ табуляции между ними?

Как это и делается в *shell*-программировании, попробуем это запустить и увидим результаты.

Выведено будет то же самое! Мы всегда вызываем *echo* с двумя аргументами. Теперь модифицируем код (*first1.sh*).

На этот раз сработало. Мы этого и ждали, т.к. у нас есть некоторый опыт программирования на других языках. Но ключ к пониманию того, что происходит с более сложными командами и скриптами – это попробовать понять и быть способным объяснить, ПОЧЕМУ это происходит.

echo в этот раз был запущен с одним аргументом – строкой "Hello World". Он печатает ее буквально.

Причина в том, что оболочка разбирает аргументы перед отправкой их программе, которая вызывается, а не во время вызова. В этом случае он удаляет кавычки, но передает строку как единственный аргумент.

И как последний пример, попробуем еще один скрипт (*first2.sh*), сначала предсказав результат, а потом запустив его (СЛАЙД 12).

Все ли получилось так, как мы ожидали? Если нет, не будем нервничать! Просто некоторые вещи мы еще не изучили, а кроме того, нужно понимать, что существуют более мощные команды, нежели *echo*!

9.4 Переменные

Почти в каждом языке программирования есть концепция переменных, как символьных имен ячеек памяти, которым можно присваивать значения, читать их и манипулировать содержимым. Bash не является исключением, и в данном параграфе мы покажем их использование.

Вернемся к нашему скрипту *first.sh*. То же самое можно сделать с использованием переменных (хотя наш пример получится синтетическим, т.к. большой необходимости в переменных здесь нет!). См. СЛАЙД 13.

Обратим внимание, что не должно быть пробелов ни слева, ни справа от знака "=": *VAR=value* сработает, а *VAR = value*, к сожалению, нет. В первом случае оболочка видит символ "=" и трактует команду как присваивание переменной. А во втором случае оболочка воспринимает *VAR* как имя команды и пытается ее выполнить.

Введем следующий код в файл *var.sh*.

Он присваивает строку "Hello World" переменной *MY_MESSAGE*, а затем выводит значение этой переменной.

Очевидно, что нам необходимы кавычки вокруг «*Hello World*». В противном случае мы возвращаемся к команде

```
echo Hello World
```

поскольку *echo* принимает любое число параметров, а переменная может содержать только одно значение, поэтому строка с пробелами должна быть закавыченной, чтобы оболочка трактовала ее как единое целое. В противном случае оболочка попытается выполнить команду *World* после присваивания *MY_MESSAGE=Hello*.

Оболочку совершенно не волнует тип переменных, они могут хранить строки, целые числа, вещественные числа, словом, все, что нам захочется. Однако если быть до конца честными, то все они хранятся как строки, но программы, которые ожидают числа, могут трактовать их именно так.

Если мы попытаемся присвоить строковое значение переменной, а затем прибавить к нему 1, то нам это не удастся (СЛАЙД 13, нижняя часть).

```
$ x="hello"
$ y=`expr $x + 1`
expr: non-numeric argument
$
```

Внешняя программа *expr* ожидает только числа, но нет синтаксических различий между следующими объявлениями (СЛАЙД 14, верхняя часть).

Важно отметить, что некоторые символы должны экранироваться символом обратной косой черты, чтобы правильно восприниматься оболочкой.

Мы можем вводить значения переменных с помощью команды *read*; поэтому следующий скрипт (*var2.sh*) попросит нас ввести имя, чтобы обратиться к нам лично (СЛАЙД 14, нижняя часть).

Встроенная shell-команда *read* читает строку из стандартного ввода и сохраняет значение в соответствующую переменную. Даже если мы зададим полное имя и не будем использовать двойные кавычки вокруг команды *echo*, она будет выдавать правильный результат. Как это сделано, ведь с переменной MY_MESSAGE мы поступили именно так, заключив ее в кавычки?

Здесь это получилось потому, что команда *read* автоматически добавляет кавычки вокруг ввода, так что пробелы трактуются правильно. При этом для вывода могут, конечно, понадобиться кавычки, например:

```
echo "$MY_MESSAGE"
```

Версия 0.9pre-release от 22.03.2013. Возможны незначительные изменения.

Переменные в Bash объявлять необязательно, как это делается, например, в языке C. Но что произойдет, если мы попытаемся прочитать необъявленную переменную? Результатом будет пустая строка, и нам не будет выдано ни предупреждений, ни сообщений об ошибках. Это на первых порах может приводить к весьма неожиданным результатам. Например:

```
MY_OBFUSCATED_VARIABLE=Hello
echo $MY_OSFUCATED_VARIABLE
```

не напечатает ничего, т.к. в параметре команды *echo* две опечатки.

Существует команда *export*, которая создает фундаментальный эффект над областями видимости переменных. Для того чтобы действительно узнать, что на самом деле происходит с нашими переменными, нам нужно понять, как использовать эту команду.

Создадим маленький скрипт (*myvar2.sh*). См. СЛАЙД 15.

Запустим его:

```
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

Переменная *MYVAR* не получила ни одного значения, поэтому она пуста. Затем мы присваиваем ей значение и получаем вполне ожидаемый результат. Теперь запустим его несколько иначе:

```
$ MYVAR=hello
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

И к сожалению, получаем тот же результат. Что же происходит? А то: когда мы вызываем скрипт *myvar2.sh* из оболочки, создается ее копия, чтобы запустить сценарий. Во многом это потому, что мы в первой строке скрипта указали *#!/bin/sh*, см комментарий выше.

Нам же нужно экспортировать переменную для того, чтобы унаследовать ее от другой программы, включая shell-скрипты. Вот что нам нужно было сделать:

```
$ export MYVAR
$ ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
```

Теперь взглянем на третью строку скрипта: это изменение значения *MYVAR*. Однако нет способа получить это измененное значение обратно в оболочку. Попытаемся прочесть значение *MYVAR*:

```
$ echo $MYVAR
hello
$
```

Когда shell-скрипт завершается, его окружение уничтожается, но *MYVAR* хранит свое значение (*hello*) внутри оболочки.

Чтобы принять изменение окружения обратно в скрипт, мы должны запустить его в нашем экземпляре оболочки вместо создания новой копии для запуска.

Для этого мы используем экзотическую на первый взгляд команду (".", символ точки):

```
$ MYVAR=hello
$ echo $MYVAR
hello
$ . ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
$ echo $MYVAR
hi there
```

Изменения делаются в нашей оболочке. Так, например, работают *.profile* или *.bash_profile*. И в этом случае нам не нужно экспортировать *MYVAR*.

Еще один момент, касающийся переменных, демонстрирует следующий скрипт (*user0.sh*).

Если кто-то полагает, что результат будет ожидаем, то он «жестоко ошибается». Например, если мы введем "alejandro" в *USER_NAME*, то создаст ли наш скрипт файл с именем *alejandro_file*?

Ничего подобного! Это будет приводить к ошибке, пока не появится переменная *USER_NAME_file*. Скрипт не знает, где переменная заканчивается, и начинается все остальное. Как мы ему можем в этом помочь?

А достаточно просто, если мы заключим переменную в фигурные скобки, как показано в скрипте *user.sh*.

Оболочка теперь знает, как сослаться на переменную *USER_NAME*, и видит, что мы хотим добавить суффикс *_file*. Это может сконфузить начинающих shell-программистов, т.к. источник проблемы трудно воспроизвести.

Видно, что и здесь мы использовали кавычки ("${USER_NAME}_file"). Если пользователь введет "*Dima Medvedev*" (с пробелом в середине), забудет указать кавычки, то аргументы посылаемые команде *touch* будут *Dima* и *Medvedev_file*, т.е. вместо создания *Dima Medvedev_file*, будут созданы два файла. Кавычки позволяют избежать этого.

9.5 Условный оператор *if*

Используется практически каждым shell-скриптом. В основе лежит команда *test,* но она обычно не используется напрямую, а создается ссылка на нее с помощью квадратных скобок, и это делает программы на Bash более читабельными. *If* – тоже обычная встроенная команда *shell*.

```
$ type [
[ is a shell builtin
$ whereis [
/usr/bin/[
$ ls -l /usr/bin/[
lrwxrwxrwx 1 root root 4 Mar 21 2013 /usr/bin/[ -> test
```

Это значит, что '[' на самом деле программа, такая же, как *ls* и другие. Так что и она должна быть окружена пробелами. Иначе говоря:

```
if [$foo == "bar" ]
```

не будет работать.

Он интерпретируется как

```
test$foo == "bar" ]
```

Здесь мы получили ']' без открывающей скобки '['. Разместим пробелы вокруг операторов. Покажем обязательные пробелы с помощью слова *'SPACE'* (в программах придется заменить *'SPACE'* на реальные пробелы). Если пробелов там не будет, код откажется работать:

```
if SPACE [ SPACE "$foo" SPACE == SPACE "bar" SPACE ]
```

test – это простая и мощная утилита. Для уточнения деталей нужно прочитать руководство по этой утилите в конкретной ОС (*man test*). Однако наиболее распространенные случаи рассмотрены далее.

Обычно *test* запускается не напрямую, а операторами *if* и *while*. Мы можем увидеть еще один источник недоразумений, если назовем нашу программу *test* и попытаемся запустить ее. Вместо нее будет вызваться встроенная команда оболочки!

У команды *if-then-else* простой синтаксис:

```
if [ ... ]
then
  # if-code
else
  # else-code
fi
```

Слово *fi* здесь закрывает оператор *if*. Кроме того, "*if* [...]" и команды ветки "*then*" должны располагаться в разных строках. В качестве альтернативы можно использовать точку с запятой ";", которая является разделителем:

```
if [ ... ]; then
  # do something
fi
```

Так же можно использовать *elif*, как здесь:

```
if  [ something ]; then
 echo "Something"
 elif [ something_else ]; then
   echo "Something else"
 else
   echo "None of the above"
fi
```

Этот код будет выводить "*Something*", если проверка [*something*] успешна, в противном случае будет проверяться [*something_else*] и выводиться "*Something else*", если она успешна. Если все *elif* неуспешны, то выводится "*None of the above*".

Рекомендуется самостоятельно попробовать следующую заготовку кода, задавая переменой *X* различные значение (1, 0, 1, *hello*, *bye* и другие). Это может выглядеть так:

```
$ X=5
$ export X
$ ./test.sh
  ... output of test.sh ...
$ X=hello
$ ./test.sh
  ... output of test.sh ...
$ X=test.sh
$ ./test.sh
  ... output of test.sh ...
```

Затем можно попытаться присвоить переменной *X* имя существующего файла, например, */etc/hosts*.

Мы можем использовать точки с запятыми (;), чтобы объединить две строки. Это обычно делается для экономии места в простых операторах *if*. Обратная косая черта просто сообщает оболочке, что это еще не конец строки, и две или более строки должны трактоваться как одна. Это полезно для читабельности и обычно для отступа следующей строки.

Как видно, *test* может выполнять проверки чисел, строк, имен файлов.

Существует и более простой способ записи операторов *if*. Команды && и || позволяют коду выполняться, если результат истинен (файл *test0.sh*). См. СЛАЙД 21.

Такая запись возможна по причине, которую мы выше описали, однако повсеместно ему следовать не стоит, он может приводить к нечитаемому коду.

Структура *if...then...else...* намного более читабельна, а использование конструкции [...] рекомендуется для цикла *while* и простых проверок, которые не слишком утомляют читателя.

Когда мы устанавливаем *X* в нечисловое значение, то первые сравнения приводят к сообщениями типа:

```
test.sh: [: integer expression expected before -lt
test.sh: [: integer expression expected before -gt
test.sh: [: integer expression expected before -le
test.sh: [: integer expression expected before -ge
```

Это потому что сравнения *-lt, -gt, -le, -ge* спроектированы только для целых чисел и не работают со строками. Строковые сравнения навроде (!=) будут воспринимать "5" как строку, не еще никто не придумал универсальной трактовки строки "*Hello*" как целого числа, так что целочисленные сравнения поступают выше правильно.

Если нам нужно, чтобы поведение shell-скрипта было более дружелюбно, придется проверить содержимое переменной прежде, чем тестировать ее (*test1.sh*). СЛАЙД 22.

9.6 Подстановка команды

Она позволяет выходу команды заменить команду саму по себе. Такая подстановка выполняется, когда команда заключается в круглые скобки (СЛАЙД 23):

```
$(command)
```

или используются обратные апострофы

```
`command`
```

Bash выполняет подстановку путем выполнения команды *COMMAND* и размещением результатов не в стандартном выводе, а в строке, из которой удалены все окружающие символы новой строки. Внедренные символы не удаляются, но их можно устранить при лексическом анализе.

```
$ echo `date`
Thu Mar 21 10:06:20 KrasTime 2013
```

Когда используется старомодная форма подстановки, обратная косая черта принимает свое непосредственное значение, за исключением случаев, когда за ней следуют символы “$”, “`”, or “\”. Первый обратный апостроф, не предваренный обратной косой чертов заканчивает подстановку команды. Когда используется форма “$ (COMMAND)”, то все символы внутри скобок составляют команду, и ни один из них не трактуется особым образом.

Подстановки команд могут быть вложенными. Если используется форма с кавычками, то для вложения необходимо экранировать внутренний обратный апостроф символом обратной косой черты.

Если подстановка появляется внутри двойных кавычек, то разбор слов и расширение имени файла в результате не появятся.

9.7 Циклы

Многие языки поддерживают концепцию повторяемого кода в виде циклов. Если мы хотим повторить какую-то задачу 12 раз, не будем же мы повторять ее код точно такое же количество раз.

В результате в оболочке Bash нам предлагается использовать циклы *for*, *while*, *until*, а также оператор *select*. Эти возможно и хуже, чем в других языках программирования, но никто и не утверждал, что Bash равен по мощности, скажем, языку C.

Цикл *for*

Синтаксис цикла *for* таков:

```
for NAME [in LIST]; do COMMANDS; done
```

Если фрагмент [*in LIST*] отсутствует, он заменяется содержимым псевдопеременной $@ (см. далее), и *for* выполняет *COMMANDS* один раз для каждого позиционального параметра из набора.

Возвращаемым значением является статус завершения последней выполненной команды. Если ни одна команда не выполнялась из-за того, что список *LIST* пуст, то возвращаемым значением является нуль.

Имя переменной *NAME* может быть любым, хотя обычно используется *i*. Список *LIST* может содержать слова, строки или числа, которые могут быть заданы литералами или сгенерированы командами. Команды *COMMANDS* выполняются, если они представляют собой любую команду ОС, скрипт, программу или оператор языка оболочки. После входа в цикл, *NAME* присваивается значение первого элемента списка *LIST*. На второй итерации его значением становится второй элемент списка. Цикл прерывается, когда *NAME* принял все возможные значения из *LIST*, и в нем больше не осталось элементов.

Первый пример (*loop-for.sh*) приведен в нижней части СЛАЙДА 24. Достаточно запустить скрипт и посмотреть результаты. Еще раз отметим, что элементы списка могут быть разного типа.

Попробуем еще один код (*loop-for2.sh*), который приведен на СЛАЙДЕ 25. Главное, убедиться в понимании картины происходящего. Попробуем запустить то же самое без символа снежинки. Можно попробовать скрипт на разных каталогах: со снежинкой, окруженной кавычками; с использованием символа экранирования (*).

Во всяком случае, мы должны видеть, что цикл *for* просматривает все, что ему задано, пока входная информация не закончится.

Цикл *while*

Цикл *while* выглядит еще более многообещающим. Он позволяет повторять выполнение списка команд до тех пор, пока соблюдается условие выполнения цикла, то есть статус завершения равен нулю. Синтаксис команды следующий (СЛАЙД 26):

```
while CONTROL-COMMAND; do CONSEQUENT-COMMANDS; done
```

Условие выполнения цикла (*CONTROL-COMMAND*) может выглядеть как одна или более команд, которые могут завершаться с успешным или неуспешным статусом. Последовательность команд, образующих тело цикла (*CONSEQUENT-COMMANDS*), составляют программы, скрипты или конструкции языка командной оболочки.

Как только условие выполнения цикла *CONTROL-COMMAND* перестает быть истинным, цикл завершается. В скрипте начинает выполняться команда, следующая за ключевым словом *done*.

Возвращаемым значением будет статус завершения последней команды из блока *CONSEQUENT-COMMANDS*, или нуль, если не было ни одной итерации цикла.

Пример кода приведен в нижней части СЛАЙДА 26. Очевидно, что цикл с вводом и выводом значений будет повторяться до тех пор, пока пользователь не введет строку *bye*.

Двоеточие (:) всегда истинно; несмотря на то, что ее использование может быть необходимым, предпочтительными все же остаются реальные коды завершения команд. Сравним выход из цикла на СЛАЙДЕ 26 с новым кодом (СЛАЙД 27). Видно, какой из них более элегантен. С другой стороны, всегда нужно осмысливать ситуации, когда один из этих вариантов оказывается лучше другого.

Еще один полезный «трюк» – цикл *while read f*. Приведенный на СЛАЙДЕ 28 код

использует оператор *case*, смысл которого мы поясним чуть позже. Там файл *myfile* читается построчно, и для каждой строки скрипт сообщает о языке, который вероятнее всего используется. Каждая строка должна завершаться символом новой строки (LF). Так если команда *cat myfile* не заканчивается пустой строкой, то последняя строка не обрабатывается.

Цикл *until*

Этот оператор очень похож на цикл *while*, но с одним отличием: он выполняется до тех пор, пока не станет истинным условие в *TEST-COMMAND*. Если это условие ложно, то цикл продолжается. Синтаксис (СЛАЙД 29) так же похож на цикл *while*:

```
until TEST-COMMAND; do CONSEQUENT-COMMANDS; done
```

Возвращаемым значением является статус завершения последней выполненной команды из списка *CONSEQUENT-COMMANDS*, или нуль, если ни одна из команд не выполнилась. И опять-таки в качестве *TEST-COMMAND* можно использовать любую команду, которая может завершаться с успешным или неуспешным статусом. Элементом *CONSEQUENT-COMMANDS* может быть команда ОС, скрипт либо конструкция языка оболочки.

Пример кода приведен на СЛАЙДЕ 29.

Символ точки с запятой (;) можно заменять одним или несколькими символам конца строки, где бы он не появлялся.

Оператор *select*

Этот оператор позволяет сравнительно легко создавать систему меню для организации пользовательского интерфейса. Синтаксис (СЛАЙД 30) похож на цикл *for*:

```
select WORD [in LIST]; do RESPECTIVE-COMMANDS; done
```

Здесь *LIST* заменяется генерированием списка элементов меню. Замена выводится в *STDERR*; каждый пункт упорядочен по номеру. Если не представлен *in LIST*, печатаются позиционные параметры, как если бы использовалась предопределенная переменная оболочки $@. Список *LIST* печатается только один раз.

После вывода всех пунктов, показывается подсказка из другой предопределенной переменной (*PS3*), и из *STDIN* считывается одна строка. Если она содержит число, соответствующее одному из пунктов, то значением слова *WORD* становится имя этого пункта. Если строка пустая, то снова выводятся пункты меню и подсказка из переменной PS3. Если прочитан символ конца файла (EOF), то цикл завершается. Так как большинство пользователей не имеют понятия о комбинациях клавиш, которые используются для последовательности EOF, то хорошим стилем считается наличие команды *break* в одном из пунктов. Любое другое значение в прочитанной строке присвоит *WORD* пустое значение.

Прочитанная строка хранится в предопределенной переменной *REPLY*.

Команды из *RESPECTIVE-COMMANDS* будут выполняться после каждого выбора до тех пор, пока не будет прочитан такое значение, которое соответствует пункту меню, содержащему оператор *break*. Это и завершит цикл.

Два примера кода приведены на СЛАЙДАХ 30 и 31.

Любой оператор внутри конструкции *select* может содержать вложенный цикл *select*. Это позволяет создавать подменю внутри меню.

По умолчанию переменная *PS3* не меняется, когда осуществляется вход во вложенный *select*. Если нужно вывести другую подсказку для конкретного подменю, то необходимо присвоить этой переменной желаемое значение в соответствующий момент

времени.

Конструкции *break* и *contiue*

Для управления ходом выполнения цикла служат команды *break* и *continue*. Они точно соответствуют своим аналогам в других языках программирования, например, в C. Команда *break* прерывает исполнение цикла, в то время как *continue* передает управление в начало цикла, минуя все последующие команды в его теле.

Пример кода приведен на СЛАЙДАХ 32 и 33.

Команде *break* может быть передан необязательный параметр. Команда *break* без параметра прерывает тот цикл, в который она вставлена. Команда *break N* прерывает цикл, стоящий на *N* уровней выше (причем 1-й уровень – это уровень текущего цикла). Пример кода на СЛАЙДЕ 34.

Команда *continue*, как и команда *break*, может иметь необязательный параметр. В простейшем случае команда *continue* передает управление в начало текущего цикла. Команда *continue N* прерывает исполнение текущего цикла и передает управление в начало внешнего цикла, отстоящего от текущего на *N* уровней (1-й уровень – это уровень текущего цикла). Пример кода на СЛАЙДЕ 35.

9.8 Оператор *case*

Конструкция *case* эквивалентна оператору *switch* в языках C/C++. Она позволяет выполнять тот или иной участок кода, в зависимости от результатов проверки условий. Она является, своего рода, краткой формой записи большого количества операторов *if/then/elif/else* и может быть неплохим инструментом при создании разного рода меню, как и оператор *select*, описанный выше. Синтаксис (СЛАЙД 36):

```
case EXPRESSION in
CASE1) COMMAND-LIST;;
CASE2) COMMAND-LIST;;
 ...
CASEN) COMMAND-LIST;;
esac
```

Каждый *caseX* – это выражение, соответствующее некоторому шаблону. Будут выполнены команды из списка *COMMAND-LIST* для первого найденного соответствия. Символ вертикальной черты «|» используется для разделения множества шаблонов, а операция, обозначенная символом закрывающей круглой скобки «)», завершает список шаблонов. Каждый *caseX* и соответствующие ему команды называются **предложениями**. Каждое предложение должно заканчиваться парой символов точки с запятой (;;). И наконец, каждый оператор *case* заканчивается оператором *esac*.

Пример кода (файл *talk.sh*) приведен на СЛАЙДЕ 37.

Этот оператор очень часто используют для обработки параметров, переданных скрипту.

9.9 Предопределенные переменные

Имеется набор переменных, которые уже получили свои значения и которые в большинстве случаев не могут быть переопределены.

Они содержат полезную информацию, которую можно использовать в скриптах, чтобы узнать об окружении, в котором они запущены (СЛАЙД 38).

Первый набор переменных, на который мы бросим свой взгляд, это $0 .. $9 и $#.

Переменная $0 – это базовое имя программы, по которому она была вызвана.

Переменные $1 .. $9 – это первые 9 дополнительных параметров, которые были переданы скрипту при вызове.

Переменная $@ – это список всех параметров с $1 и до «бесконечности». Переменная $* похожа на нее, но не сохраняет пробельные символы и расставляет

кавычки так, что "*File with spaces*" становится "*File*" "*with*" "*spaces*". Это похоже на работу утилиты *echo*. В подавляющем большинстве случаев рекомендуется избегать $*, а вместо нее использовать $@.

Переменная $# – это количество параметров, переданных скрипту.

Теперь взглянем на пример кода (СЛАЙД 39):

Значение переменной $0 изменяется в зависимости от того, как вызван скрипт. Внешняя утилита *basename* может помочь сделать это более аккуратно (Нижняя часть СЛАЙДА 39):

```
echo "My name is `basename $0`"
```

Значения переменных $# и $1 .. $9 устанавливаются оболочкой автоматически.

Мы можем обеспечить прием скриптом более 9 параметров с использованием команды *shift*; взглянем на скрипт на СЛАЙДЕ 40.

Выполнение этого скрипта продолжается до тех пор, пока $# не обнулится, и в этой точке список параметров опустеет.

Еще одна специальная переменная – это $?. Она хранит код завершения последней выполненной команды. Таким образом, код на СЛАЙДЕ 41, попытается запустить */usr/local/bin/my-command*, которая должна завершаться со значением 0, если все прошло благополучно, или ненулевое значение в случае ошибки. И мы можем управлять этим проверкой значения $? после вызова команды. Это помогает создавать устойчивые и более-менее интеллектуальные скрипты.

Хорошо спроектированные программы должны возвращать 0 в случае успеха.

Еще две переменные, значения которых устанавливаются оболочкой – это $$ и $!. Обе они представляют собой идентификаторы процессов.

Переменная $$ – это PID (*Process IDentifier*) процесса текущей запущенной оболочки. Это значение можно использовать для создания временных файлов с именами */tmp/my-script.$$*, которые могут оказаться полезными, если большое количество экземпляров одного и того же скрипта запускаются одновременно, и все они нуждаются во временных файлах.

Переменная $! – это PID последнего запущенного фонового процесса. А это может оказаться полезным для отслеживания того, что процесс получает во время своей работы.

Следующей полезной переменной является IFS (*Internal Field Separator*). Его значение по умолчанию – пробел, табуляция, новая строка, но если мы изменим ее, настоятельно рекомендуется сделать копию, как показано на СЛАЙДЕ 42.

Очень важно при работе в частности с IFS (но вообще нельзя сказать, что все переменные под нашим контролем), что она может содержать пробелы и другие неуправляемые символы. Значит, хорошей мыслью будет использование кавычек вокруг нее, например:

```
old_IFS="$IFS"
```

вместо

```
old_IFS=$IFS
```

Ранее мы уже обсуждали роль фигурных скобок во избежание путаницы. См. СЛАЙД 43.

```
foo=sun
echo $fooshine     # $fooshine is undefined
echo ${foo}shine   # displays the word "sunshine"
```

Это еще не все, поскольку эти замечательные скобки играют другую, возможно более мощную роль. Мы можем столкнуться с проблемами неопределенных или пустых

переменных (в *Bash* нет большой разницы между неопределенными и пустыми значениями).

Рассмотрим следующую заготовку кода, которая печатает подсказку для пользовательского ввода, но принимает значения, заданные по умолчанию (СЛАЙД 44). Опция "-en" команды *echo* позволяет подавить вывод символа новой строки.

Это можно сделать более изящно с использованием особенностей переменных языка командной оболочки. Фигурные скобки и два символа ":-" позволят нам задать значение по умолчанию, если переменная не установлена (СЛАЙД 45).

Еще один вариант синтаксиса – использование символов ":=". В этом случае переменной присваивается значение по умолчанию, если она не была определена (СЛАЙД 46).

Этот означает, что любой последующий доступ к переменной $*myname* будет выдавать значение, введенное пользователем, или "*Dima Medvedev*" в противном случае.

9.9 Константы

В Bash константы создаются путем указания переменным атрибута *readonly* с помощью одноименной встроенной команды. Она помечает каждую специфицированную переменную как неизменяемую. Синтаксис (СЛАЙД 47):

```
readonly OPTION VARIABLE(s)
```

Значения этих переменных во всех дальнейших операциях не могут быть изменены.

9.10 Массивы

Массив – это переменная, содержащая несколько значений. Любая переменная может использоваться в качестве массива. Нет ограничений ни на размер массива, ни на то, чтобы элементы массива индексировались или присваивались непрерывно. Массивы всегда 0-*based*, т.е. у первого элемента индекс всегда 0.

Непрямое объявление можно сделать, используя следующий синтаксис для объявления переменных (СЛАЙД 48):

```
ARRAY[INDEXNR]=value
```

Здесь *INDEXNR* трактуется как арифметическое выражение, которое должно выдавать в результате положительное число.

Явное объявление массива делается с использованием встроенной команды *declare*:

```
declare -a ARRAYNAME
```

Объявление с числовым индексом также принимается, однако сам индекс будет проигнорирован. Атрибуты массива могут специфицироваться командами *declare* и *readonly*. Атрибуты распространяются на все элементы массива, т.е. у нас не получится объявить смешанные массивы.

«Массивовые» переменные также можно создать, используя составное присваивание в таком формате:

```
ARRAY=(value1 value2 ... valueN)
```

Каждое значение становится доступным в форме *[indexnumber=]string*. Номер индекса не обязателен, но если он указан, то значение присваивается элементу с указанным индексом, в противном случае используется номер последнего индекса плюс один. Этот формат можно использовать и с *declare*. Если не были указаны номера индексов, то индексирование начинается с 0.

Добавление пропущенных или дополнительных элементов массива делается использованием следующего синтаксиса:

```
ARRAYNAME[indexnumber]=value
```

Важно отметить, что команда *read* имеет опцию *-a*, которая позволяет читать и присваивать значения элементам массивов.

Для того чтобы сослаться на содержимое элемента массива, нужно использовать фигурные скобки. Это необходимо, как видно из следующего примера, чтобы обойти интерпретацию оболочкой операторов подстановки. Если номер индекса это @ или *, то таким образом ссылаются на все элементы массива.

Демо:

```
$ ARRAY=(one two three)

$echo ${ARRAY[*]}
one two three

$echo $ARRAY[*]
one[*]

$echo ${ARRAY[2]}
three

$ARRAY[3]=four

$echo ${ARRAY[*]}
one two three four
```

Ссылка на содержимое элемента массива без указания номер индекса – то же самое, что и ссылка на содержимое первого элемента, который «адресуется» индексом нулевого элемента.

Команду *unset* используют как для уничтожения обычных переменных и массивов, так и их элементов.

Демо:

```
$unset ARRAY[1]

$echo ${ARRAY[*]}
one three four

$unset ARRAY
$echo ${ARRAY[*]}
<--no output-->
```

9.11 Функции

Что такое функции

Shell-функции – это способ группирования команд для дальнейшего выполнения, при этом будет использоваться имя этой группы, которая еще называется **подпрограммой**. Имя подпрограммы должно быть уникальным для скрипта или всей оболочки. Все инструкции, которые составляют функцию, выполняются как обычные команды. Когда функция вызывается как простая команда, на самом деле начинает выполняться список команд ассоциированных с именем функции. Функция выполняется в контексте оболочки, в которой она объявлена, при этом для интерпретации команд не создается новый процесс.

Существуют две формы синтаксиса для определения функции (СЛАЙД 49, верхняя часть):

```
function FUNCTION { COMMANDS; }
```

или

```
FUNCTION () { COMMANDS; }
```

В обоих случаях определяется функция *FUNCTION*. Использование встроенной shell-команды *function* не обязательно. Однако если это слово пропущено, то необходимо указывать круглые скобки.

Команды, перечисленные внутри фигурных скобок, определяют тело функции. Они выполняются всякий раз, когда *FUNCTION* используется как имя команды. Статус завершения функции – это статус завершения последней команды в теле функции.

Часто начинающие *shell*-программисты допускают две следующие ошибки: 1) Фигурные скобки не отделены от тела функции пробельными символами; 2) Тело функции не заканчивается точкой с запятой или символом новой строки.

Пример кода приведен в нижней части СЛАЙДА 49.

Передача параметров функциям

Как видно, функции – это своего рода мини-скрипты. Они так же могут принимать параметры, они могут объявлять переменные, чья область видимости будет ограничена телом функции. Для этого используется встроенная команда *local*. Они могут возвращать значения вызвавшей их оболочке.

Любая функция также имеет систему интерпретации позиционных (позициональных) параметров. При этом следует обратить внимание, что параметры, посылаемые функции, это не то же самое, что параметры, передаваемые команде или скрипту.

Когда выполняется функция, аргументы становятся позиционными параметрами на время ее работы. Специальный параметр $# содержит количество параметров, и он обновляется в соответствии с происходящими изменениями. Позиционный параметр $0 неизменен. Переменная *FUNCNAME* оболочки устанавливается в имя функции на время ее выполнения.

Если в теле функции запускается встроенная команда *return*, то функция завершается, и начинает выполняться команда, следующая за вызовом функции. Когда функция завершается, позиционные параметры и специальный параметр $# заполняются значениями, которые они принимали до выполнения функции. Если команде *return* передавался числовой аргумент, то возвращается статус завершения.

Пример простого кода приведен на СЛАЙДЕ 50.

Программисты, использующие другие языки, могут поначалу быть удивлены правилами области видимости для *shell*-функций. По существу, за исключением локальных переменных, нет областей видимости ни для чего, кроме позиционных параметров – $1, $2, $@ и т.д.

Рассмотрим следующий код (СЛАЙД 51).

Параметры из списка $@ изменяются внутри функции, чтобы отразить процесс вызова функции. Переменная *x*, тем не менее, действительно глобальная, ведь *myfunc* ее изменяет, а это изменение останется в силе, когда управление перейдет в основную часть скрипта.

Функция будет вызвана в «подоболочке», если ее вывод конвейеризуется и передается на вход другой программы, например:

```
myfunc 1 2 3 | tee out.log
```

будет два раза сообщать, что "x is 1". Это происходит потому, что создается новый процесс оболочки, чтобы конвейеризовать функцию *myfunc()*. Это может сделать отладку очень сложной; в некоторых случаях скрипты могут неожиданно завершиться с ошибкой, когда добавляется конструкция "| tee". А как должно это выглядеть, не всегда очевидно. Команда *tee* должна быть запущена до функции слева от символа конвейера. Поясним на примере более простого кода:

```
ls | grep foo
```

Команда *grep* должна быть запущена первой, ее вход перенаправляется на выход команды *ls* , когда *ls* начинает работу. В скрипте оболочка должна быть уже создана, даже если мы знаем, что собираемся конвейеризовать выход на вход команды *tee*. То есть ОС должна запустить *tee*, затем стартовать новую оболочку, чтобы вызвать функцию *myfunc*(). Это неприятно, но нам стоит быть в курсе этого.

Функции не могут изменять значения, с которыми они должны быть вызваны, либо это должно быть сделано непосредственно над самими переменными, но ни в коем случае через параметры, передаваемыми скрипту. Пример кода приведен на СЛАЙДЕ 52.

Эта довольно нахальная функция *myfunc*() заменяет $*a*, так что сообщение «Hello World» становится «Goodbye Cruel World».

Еще один, вероятно, неприятный для начинающих *shell*-программистов момент. Код приведен на СЛАЙДЕ 53. Как видно, переменная объявляется внутри функции, но от этого она не перестает быть глобальной. Однако, до вызова функции переменная пуста, а после вызова ей присваивается соответствующее значение.

Локальные переменные

Переменные, объявленные как локальные, имеют ограниченную область видимости, и доступны только в пределах блока, в котором они были объявлены. Для функций это означает, что локальная переменная «видна» только в теле самой функции. Используется встроенная команда *local*. Синтаксис объявления локальных переменных таков (СЛАЙД 54):

```
local var=value
```

или

```
local varName
```

Команда *local* может использоваться только внутри функции. Она создает имя переменной, область видимости которой ограничена этой функцией и вложенными в нее элементами. См. код на СЛАЙДАХ 54 и 55.

Локальные переменные делают возможной рекурсию, однако она сопряжена с большими накладными расходами и не рекомендуется для использования в реальных сценариях, за исключением учебных примеров. Один из них приведен далее.

Возврат значений из функций

Функции могут возвращать значения одним из четырех способов (СЛАЙД 56):

- Изменение состояния одной или нескольких переменных.
- Использование команды *exit* в конце *shell*-скрипта.
- Использованием команды *return* для досрочного перехода в конец функции и возврата необходимого значения вызвавшей стороне.
- *echo*-печать в *stdout*, который будет перехвачен вызвавшей стороной. Например, так:

```
c=`expr $a + $b`
```

Функции могут возвращать значение в виде кода завершения. Код завершения может быть задан явно, с помощью команды *return*, в противном случае будет возвращен код завершения последней команды в функции (0 – в случае успеха, иначе – ненулевой код ошибки). Код завершения в сценарии может быть получен через переменную $?.

Синтаксис (СЛАЙД 57, верхняя часть):

```
exit [N]
```

Версия 0.9pre-release от 22.03.2013. Возможны незначительные изменения.

Команда *return* завершает исполнение функции и может иметь необязательный целочисленный аргумент, который возвращается в вызывающий сценарий как «код завершения» функции. Это значение так же записывается в переменную $?.

Синтаксис (СЛАЙД 57, верхняя часть):

```
return [N]
```

Код в нижней части СЛАЙДА 57 выводит вычисленные значения факториала, но печать значений нужна чаще всего только при отладке функций. От ненужной *echo*-печати легко избавиться. Модифицированный код приведен на СЛАЙДЕ 58.

Библиотеки функций

Тем не менее, когда написан пакет скриптов, очень часто рекомендуется оформить его в виде «библиотеки» полезных функций, и включить этот файл, используя команду *source*, в начале текста тех скриптов, которые используют функции.

Используется команда *source* (или «.»). Синтаксис: (СЛАЙД 59)

```
. ./library.sh
```

Это, повторимся, указывается в начале исходного текста *shell*-скрипта, и очень сильно напоминает директиву *#include* в программах на C/C++.

Здесь (СЛАЙД 60) мы видим два скрипта, *function1.sh* и *function2.sh*. Каждый из них включает содержимое общей библиотеки *common.lib*, а затем использует определенные в ней переменные и функции.

Это не что-то из ряда вон выходящее, а просто пример того, как может быть повторно использован код в shell-программировании.

9.12 Встроенная команда *shift*

Эта команда унаследована Bash от Bourne Shell (СЛАЙД 61). Она принимает один числовой аргумент. Позиционные параметры сдвигаются влево на это число (*N*). Позиционные параметры от *N*+1 до $# переименовываются и становятся списком от $1 до $# - *N*+1.

Скажем, есть у нас команда, которая принимает 10 аргументов, и *N* == 4, тогда $5 становится $1, $6 становится $2 и так далее. $10 становится $6. Прежние $1, $2, $3 и $4 отбрасываются. Аргумент $0 не затрагивается.

Если *N* == 0 или больше, чем $#, то позиционные параметры не меняются, т.е. команда не приводит ни к какому результату. Если *N* не представлена, она принимается равной 1. Возвращаемый статус равен нулю, если *N* больше, чем $# или меньше нуля; в противном случае он не равен 0.

Пример кода приведен на СЛАЙДЕ 62.

Оператор *shift* обычно используется, когда количество аргументов команды заранее неизвестно. Например, пользователи могут передавать столько аргументов, сколько им нравится. В подобных случаях аргументы обычно обрабатываются внутри цикла *while* с проверкой условия, использующего переменную ($#). Оно выполняется (истинно), если количество аргументов больше, чем нуль. Переменная $1 и оператор *shift* обрабатывают каждый аргумент. Количество аргументов сокращается всякий раз, когда выполняется *shift*, и в конце концов станет равным нулю, что будет соответствовать окончанию цикла.

В коде, приведенном на СЛАЙДЕ 62, оператор *shift* обрабатывает каждый файл из списка, генерируемого программой *find*.

Оператор *shift* может применяться и к входным переменным функции.

9.13 Ловушки

Как и в любой другой UNIX-подобной системе, в ОС Linux поддерживается целая

система сигналов, с помощью которых могут взаимодействовать процессы.

Сигналы

Если быть более точным, то они посылаются процессам или командам, чтобы уведомить их о наступлении некоторого события.

Например, при выполнении команды:

```
$ ls -R /
```

пользователь может нажать CTRL+C (или *Break*) чтобы прервать выполнение команды. В то время, когда нажимается сочетание CTRL+C, отправляются сигналы SIGINT (в числовом формате – 2), чтобы показать прерывание от клавиатуры. Когда SIGINT отправляется команде *ls*, Linux прерывает нормальный поток выполнения. В нашем случае команда *ls* будет досрочно завершена.

Тем не менее, мы можем зарегистрировать обработчик сигнала для сочетания клавиш CTRL+C и предпринять какие-то действия: игнорировать его либо выдать на экран сообщение. Для перехвата сигналов и обработки ошибок принято использовать команду *trap*.

Мы можем посылать различные сигналы командам и процессам. Например, для прерывания процесса на переднем плане нажимается комбинация Ctrl+C, а для прерывания фонового процесса используется команда *kill* и отправляется сигнал *SIGTERM*.

Все сигналы можно просмотреть (СЛАЙД 63) командой *kill*:

```
$ kill -l
```

Аналогично по умолчанию команда *kill* отправляет сигнал *TERM*. Например, этот сигнал отправим процессу с идентификатором 1234:

```
$ kill 1234
```

Самые распространенные сигналы:

```
Signal name  Signal value      Effect
SIGHUP       1                 Hangup
SIGINT       2                 Interrupt from keyboard
SIGKILL      9                 Kill signal
SIGTERM      15                Termination signal
SIGSTOP      17,19,23          Stop the process
```

Сигналы SIGKILL and SIGSTOP нельзя перехватить, заблокировать или игнорировать.

Оператор *trap*

При выполнении скрипта пользователь может нажать известное сочетание клавиш, чтобы прервать процесс. Он может приостановить процесс нажатием клавиш CTRL+Z. Может произойти сбой из-за бага в *shell*-скрипте, например, вследствие арифметического переполнения. Это может приводить к ошибочному или непредсказуемому выводу.

Как отмечалось выше, сигналы могут досрочно завершать скрипт. С помощью оператора *trap* можно перехватить такое прерывание. Он обеспечивает конструкции для перехвата и последующей очистки внутри скрипта.

Синтаксис и примеры (СЛАЙД 64):

```
trap [COMMANDS] [SIGNALS]

trap arg signal
trap command signal
trap 'action' signal1 signal2 signalN
trap 'action' SIGINT
```

```
trap 'action' SIGTERM SIGINT SIGFPE SIGSTP
trap 'action' 15 2 8 20
```

Здесь задается список *SIGNALS* сигналов для перехвата, его могут составлять имена сигналов с/без префикса *SIG* или номера сигналов. Если сигнал это 0 или *EXIT*, то выполняются команды из списка *COMMANDS* в конце выполнения *shell*-скрипта. Если один из сигналов – *DEBUG*,то список *COMMANDS* выполняется после каждой простой команды. Сигнал можно определить как *ERR*; и в этом случае список *COMMANDS* выполняется всякий раз, когда простая команда завершается с ненулевым статусом. Однако эти команды не будут выполнены, когда с ненулевым статусом завершилась часть оператора *if* или операторы цикла *while* или *until*. Они не будут выполнены ни при получении ненулевого кода завершения логических операций *AND* (&&) или *OR* (||), ни при получении статуса завершения, инвертированного оператором *NOT*(!).

Статус возврата команды *trap* как таковой равен нулю, если не обнаружено определение неправильного сигнала. У команды есть пара опций, которые документированы в справочной системе Bash-оболочки.

На СЛАЙДЕ 65 приведен очень простой пример кода, который перехватывает нажатие CTRL+C от пользователя и выводит сообщение об этом на экране. Если мы попытаемся завершить программу без указания сигнала KILL, то ничего не произойдет.

Для удаления ловушки используется следующий синтаксис (СЛАЙД 66):

```
trap - signal
trap - signal1 signal2
```

Например, установить ловушку для команды *rm*:

```
file=/tmp/test4563.txt
trap 'rm $file' 1 2 3 15
trap
```

Для очистки ловушки SIGINT вводится:

```
trap - SIGINT
trap
```

Для очистки всех ловушек можно ввести:

```
trap - 1 2 3 15
trap
```

В качестве более полного примера создадим скрипт *evenorodd.sh*, его текст приведен на СЛАЙДАХ 67-68.

Можно использовать оператор *trap* совместно с функциями по следующей схеме (СЛАЙД 69):

```
# define die()
die()
{
  echo "..."
}

# set trap and call die()
trap 'die' 1 2 3 15
....
...
```

Измененная версия скрипта со СЛАЙДОВ 67-68 приведена на СЛАЙДАХ 70-72.

9.14 Прочие полезности

1. $*RANDOM* – внутренняя функция Bash (не константа), которая возвращает **псевдослучайные** целые числа в диапазоне [0..32767]. Функция $*RANDOM* **не должна** использоваться для генерации ключей шифрования.

Использование этой переменной иллюстрируется кодом на СЛАЙДЕ 73, в котором генерируются 10 чисел в диапазоне от 200 до 500.

Сначала в этом скрипте вычисляется область значений в виде переменной *scope* с использованием оператора вычисления остатка для гарантии нахождения числа в заданном диапазоне. Затем к случайному числу прибавляется значение $MIN для того, чтобы гарантировать получение псевдослучайного числа в заданном диапазоне.

2. Предположим, что значение одной переменной – это имя второй переменной. Возможно ли получение значения второй переменной через обращение к первой? Например, Пусть *a=letter* и *letter=z*, тогда вопрос будет звучать так: «Можно ли получить значение *z*, обратившись к переменной *a*?». В действительности это возможно, и это называется **косвенной ссылкой** (*indirect reference*). Для этого необходимо прибегнуть к не совсем обычной (на первый взгляд) нотации

```
eval var1=\$$var2.
```

Как видно, использование косвенной ссылки в Bash многошаговый процесс. Сперва берется имя переменной (*var2*). Затем на нее ссылаются ($*var2*). Далее ссылаются на ссылку ($$*var2*). Далее первый $ экранируется обратной косой чертой. И в последнюю очередь производится вычисления выражения и присваивание результата (т.е. то, что и требовалось).

Пример кода представлен на СЛАЙДЕ 74.

Такой метод обращения к переменным имеет свои особенности. Если переменная, на которую делается ссылка, изменяет свое значение, то переменная-ссылка должна быть должным образом разыменована, т.е. должна быть выполнена операция получения ссылки.

Во многих версиях Bash имеется более удобный (как говорится, интуитивно понятный) способ использования косвенных ссылок.

Пример кода представлен на СЛАЙДЕ 75. Сравните с кодом со СЛАЙДА 74.

3. **Двойные круглые скобки** во многом похожи на инструкцию *let*. Внутри ((...)) вычисляются арифметические выражения, и возвращается их результат. В простейшем случае, конструкция *a*=$((5 + 3)) присвоит переменной "*a*" значение выражения "5 + 3", или 8. Но, кроме того, двойные круглые скобки позволяют работать с переменными в стиле языка C. См. код на СЛАЙДЕ 76, где демонстрируется использование тернарного оператора, а также операторов декремента и инкремента и др.

4. **Перенаправление ввода-вывода**.

По-умолчанию в системе всегда открыты три «файла» – *stdin* (клавиатура), *stdout* (экран) и *stderr* (вывод сообщений об ошибках). Эти, а также любые другие открытые файлы, могут быть перенаправлены. В данном случае, термин «перенаправление» означает «получить вывод из файла, команды, программы, сценария или даже отдельного блока в сценарии и передать его на вход в другой файл, команду, программу или сценарий».

Как известно, с каждым открытым файлом связан дескриптор файла. Дескрипторы файлов *stdin*, *stdout* и *stderr* – 0, 1 и 2, соответственно. При открытии дополнительных файлов, дескрипторы с 3 по 9 остаются незанятыми. Иногда дополнительные дескрипторы могут сослужить неплохую службу, временно сохраняя в себе ссылку на *stdin*, *stdout* или *stderr*. Это упрощает возврат дескрипторов в нормальное состояние после сложных

манипуляций с перенаправлением и перестановками. СЛАЙДЫ 77-80.

Операции перенаправления и/или конвейеры могут комбинироваться в одной командной строке (СЛАЙД 81, верхняя часть).

Допускается перенаправление нескольких потоков в один файл (СЛАЙД 81, нижняя часть).

Также имеются возможности по закрытию дескрипторов файлов. СЛАЙД 82.

Дочерние процессы наследуют дескрипторы открытых файлов. По этой причине и работают конвейеры. Чтобы предотвратить наследование дескрипторов, их следует закрыть перед запуском дочернего процесса. СЛАЙД 82.

Блоки кода, такие как циклы *while*, *until* и *for*, условный оператор *if-then*, также могут смешиваться с перенаправлением *stdin* и *stdout*. Даже функции могут использовать эту форму перенаправления. Оператор перенаправления < в таких случаях ставится в конце блока. См. пример кода для цикла *while* на СЛАЙДЕ 83.

Литература и дополнительные источники к Теме 9

1. Bash Guide for Beginners - http://tille.garrels.be/training/bash/
2. BUSYBOX - http://www.busybox.net/
3. GNU Bash - http://www.gnu.org/software/bash/
4. Home Page For The KornShell Command And Programming Language - http://www.kornshell.com/
5. Linux Shell Scripting Tutorial (LSST) v2.0 - http://bash.cyberciti.biz/guide/Main_Page
6. Tcsh Homepage - http://www.tcsh.org
7. Zsh - http://www.zsh.org/
8. Береснев, А. Администрирование GNU/Linux с нуля (+ CD-ROM). – СПб.: БХВ-Петербург, 2010. – 576 с.
9. Искусство программирования на языке сценариев командной оболочки- http://www.opennet.ru/docs/RUS/bash_scripting_guide/
10. Робачевский, А. Операционная система Unix, 2 изд./ А.Робачевский, С.Немнюгин, О.Стесик. – СПб.: БХВ-Петербург, 2010. – 656 с.