Linux™

Shell Scripting

Shell Programming

- Shell is the primary means of interacting with a Unix system
- A shell is a program that provides an environment for entering commands and shell scripts automate some complex tasks of day today life
- At login a process is started running the shell program and attached to your “terminal”.
- most common available shells are-
  - sh - Bourne shell
  - csh - C shell (C like syntax)
  - ksh -K shell (powerful high level programming language)
  - bash - Bourne Again shell

- The default shell on GNU/Linux and Mac OS X.

- Brainchild of Brian Fox

- Replacement for Bourne shell (sh), hence Bourne-again shell.

Shell Script

- A regular text file that contains executable shell commands.
  - Anything you can run normally on the command line can be put into a script.
  - Anything you can put into a script can also be run normally on the command line and it will do exactly the same thing.
- Creating scripts **#!<interpreter>** (a.k.a. Sha-bang/hashbang)
- Running scripts: it execute permission
  - $ chmod +x <script_name>
  - $ chmod 755 <script_name>
- Each of the shells has a scripting capability, combining commands into a single unit. The details, syntax and command options, depend on which shell is being used.
- Shell scripts can be run from the command line in several ways

  sh < script bash script ./script . script

First Script

- echo prints the rest of the line to the screen (standard output).
- This is useful for providing output or updates in a script.

```
$ cat> first.sh
  #!/bin/bash
  # My first script
  echo "Hello World!"

$ chmod +x first.sh
$ ./first.sh
Hello World
```

Shells – Variables

- No need to declare, no type (int, char...), case sensitive.
- Like most programming languages, the shell allows items to be stored in variables.
- Created when assigned a value.
- variable=value (no space in between)
- To read their values, precede them by a dollar sign ($).
- Reading an uninitialized variable?
- Local variables
  - Used only by the current shell, most frequently in scripts.
  - Do not get passed to child process
  - Usually lower case
  - Set value

    var = “value”
  - Access value

    echo $var

    echo ${var}

Shells – local Variables

Any name that starts with a letter can be used as a variable name.

For instance: v, v1, v1_1, v_filename_4

To add a string immediately after a variable name can be confusing.

The situation is solved by putting the variable name inside braces.

```
$ v=im1

$ echo $v_new

$ echo ${v}_new

 im1_new
```

Shells – Environment Variables

Environment variables

- Maintained and used by the shell but are also passed to any child processes created
- Child process cannot affect the parents environment variables
- Usually upper case

  e.g. $SHELL, $PATH, $HOME, $USER, $PS1
- Set value
  - VAR = “value” ; export VAR
  - export VAR = “value”
- Access value
  - echo $VAR - echo ${VAR}

Caution: Do not modify System variables this can sometimes create problems.

Shells - Variables

$$ is the process ID (PID) of the script itself.

$echo $$

24003

x is local variable and y is a environment variable

$x=5

$echo $x

5

$export Y='Hello'

$echo $Y

Hello

child shell can not see x but can see inherited y

$bash

$echo $x

$echo $Y

Hello

Shells – Common Environment Variables

- Common environment variables
  - PATH - Used to set the list of directories to search when a program is executed
    - export PATH=$PATH:/usr/local/bin

      Appends /usr/local/bin to the search path
    - export PATH=/usr/bin:/usr/local/bin

      Assigns path to only /usr/bin and /usr/local/bin
  - HOME - Set to your home directory path
  - LD_LIBRARY_PATH - Used by Linux and Solaris to list directories to search for shared libraries
  - Use env, printenv and set commands to display environment variables

Shells – Double Quotes

Double Quote (“ ”) - a.k.a. weak quoting

Preserves the literal meanings of all characters within it except $, ` , \ and itself.

Variable – Yes

Wildcards – No

Command substitution - yes

- **Allows to print the value of variables.**

```
rekha@rekha-ThinkPad-P50:~$ echo $SHELL
/bin/bash
rekha@rekha-ThinkPad-P50:~$ echo "$SHELL"
/bin/bash
```

- **Disables the meaning of wildcards.**

```
rekha@rekha-ThinkPad-P50:~$ echo "/etc/*.conf"
/etc/*.conf
```

- **Allows wild char in command substitution.**

```
rekha@rekha-ThinkPad-P50:~$ echo "$(ls  -l /etc/adduser.conf)"
-rw-r--r-- 1 root root 3028 Feb 16  2017 /etc/adduser.conf
rekha@rekha-ThinkPad-P50:~$ echo "$(ls -l /etc/*.conf|head -2)"
-rw-r--r-- 1 root root  3028 Feb 16  2017 /etc/adduser.conf
-rw-r--r-- 1 root root   112 Jan 10  2014 /etc/apg.conf
```

Shells – Single Quotes

Single Quote (“ ”) - a.k.a. strong quoting

It is used to turn off the special meaning of all characters.

Variable – No

Wildcards – No

Command substitution - No

- **Does not allow to print the value of variables.**

```
rekha@rekha-ThinkPad-P50:~$ echo $SHELL
/bin/bash
rekha@rekha-ThinkPad-P50:~$ echo '$SHELL'
$SHELL
```

- **Disables the meaning of wildcards.**

```
rekha@rekha-ThinkPad-P50:~$ echo '/etc/*.conf'
/etc/*.conf
```

- **Does not allow command substitution.**

```
rekha@rekha-ThinkPad-P50:~$ echo '$(ls  -l /etc/adduser.conf)'
$(ls  -l /etc/adduser.conf)
```

Shells – The Backtick

The backtick `…` is actually called command substitution. The purpose of command substitution is to evaluate the command which is placed inside the backtick and provide its result as an argument to the actual command.

The command substitution can be done in two ways one is using $(…) and the other is `…`. Both work same, but the $(…) form is the modern way and has more clarity and readability.

```
rekha@rekha-ThinkPad-P50:~$ echo $`date`
$Mon Aug 6 20:54:12 IST 2018
```

Shells – Console I/O - read

- Read input from the keyboard and assign it to a variable

```
rekha@rekha-ThinkPad-P50:~$ read text
rekha
rekha@rekha-ThinkPad-P50:~$ echo "hello $text"
hello rekha
```

- Read with timeout option

```
rekha@rekha-ThinkPad-P50:~$ read -t 3 response
hello
rekha@rekha-ThinkPad-P50:~$ echo $?
0
rekha@rekha-ThinkPad-P50:~$ read -t 3 response
rekha@rekha-ThinkPad-P50:~$ echo $?
142
```

- Read with -s option to hide typed text like password

```
rekha@rekha-ThinkPad-P50:~$ read -s passwd
rekha@rekha-ThinkPad-P50:~$ echo $passwd
hello
```

Shells – Console I/O - reply

- Read with -p option to add the prompt

```
rekha@rekha-ThinkPad-P50:~$ read -p "Please enter your name: " name
Please enter your name: rekha
rekha@rekha-ThinkPad-P50:~$ echo $name
Rekha
```

- If there are no variables, text of the line read is assigned to variable $REPLY

```
rekha@rekha-ThinkPad-P50:~$ read -p "Please enter your name: "
Please enter your name: rekha
rekha-ThinkPad-P50:~$ echo $REPLY
Rekha
```

- Read with -a option to read input in an array

```
rekha@rekha@rekha-ThinkPad-P50:~$ read -a arr
a good life example for 100 years!
rekha@rekha-ThinkPad-P50:~$ echo ${arr[*]}
a good life example for 100 years!
```

Shells – Arguments

- Passing arguments to our scripts is via positional parameters(a.k.a. command-line arguments).
- Are predefined buffers in the shell script $ 1 through $ 9
- During execution, the shell puts the first argument as $ 1, the second as $ 2 and so on.
- Other Special parametres/variables:
  - Name of the script ($ 0)
  - Number of arguments ($ #)
  - All parameters ($ * and $ @)
  - Exit status ($?)

Shells – Arguments

Contents of pos.sh

```
#!/bin/bash
echo "\$0: name of the script is " $0
echo "\$*: arguments are" $*
echo "\$#:Number of arguments are" $#
echo "\$n: first 3 args are " $1 $2 $3
echo "\$@:args in array are " $@
```

```
rekha@rekha-ThinkPad-P50:~/workingDir/shell$ ./pos.sh foo bar baz
$0: name of the script is  ./pos.sh
$*: arguments are foo bar baz
$#:Number of arguments are 3
$n: first 3 args are  foo bar baz
$@:args in array are  foo bar baz
```

Shells – Exit

- Commands return a value to the system when they terminate.
- The value (0-255) denotes success/failure of command's execution.
- The $? special variable stores the status of preceding command.
- 0 is success and any other value is failure
- Checkout the man page for more details on exit status

```
$ls -l /bin
#check the error status for previous command
$echo $?
$ls -l I ExistNot
#check the error status for previous command
$echo $?
```

Shell: wildchars (globbing)

Expanded by the shell first (this is known as Globbing).

? matches any single character

* matches 0+ number of characters (but '.' and '/')

[...] matches any element in the set.

$ls -l ????

$ls f*

$ls b?r

$ls [a-c]*

$rm -i out*

$cp [a-c]* dir1

Shell: Array

- Variables containing multiple values.
- No maximum limit to the size.
- Index-based access.
- Creating and dereferencing arrays

<ArrayName>[index]=value

<ArrayName>=(value1 value2 value3 ...)

${ArrayName[index]} # dereferencing

```
rekha@rekha-ThinkPad-P50:~$ ar=(1 2 foo)
rekha@rekha-ThinkPad-P50:~$ echo ${ar[2]}
foo
rekha@rekha-ThinkPad-P50:~$ echo ${ar[*]}
1 2 foo
rekha@rekha-ThinkPad-P50:~$ ar[1]=bar
rekha@rekha-ThinkPad-P50:~$ echo ${ar[*]}
1 bar foo
```

Shell: Sourcing

- **<command/script>** or **source <command/script>** is called sourcing. Script is run in environment of the current shell (i.e. no forking of subshell).

```
rekha@rekha-ThinkPad-P50:~/workingDir/shell$ echo $x
200
rekha@rekha-ThinkPad-P50:~/workingDir/shell$ cat > sourcing.sh
#!/bin/bash
echo $x
rekha@rekha-ThinkPad-P50:~/workingDir/shell$ . sourcing.sh
200
rekha@rekha-ThinkPad-P50:~/workingDir/shell$ source sourcing.sh
200
rekha@rekha-ThinkPad-P50:~/workingDir/shell$ bash sourcing.sh
```

Shell: hyphen

- Subtraction operator

- Represents OLDPWD

```
$cd -
$echo $ OLDPWD
```

- As filename for stdin/stdout

```
cat - > file1.txt
cat file1.txt - > file2.txt
grep ‘xyz’ file2.txt | diff file1.txt -
```

Shell: Summary

#	- comments	**$#this is a comment**
,	- sequence	**$echo xy{1,2,3}z** **$echo $((2 + 3 , $x + 5))**
.	- sourcing	**$. proj.sh**
..	- range	**$echo xy{1..3}z**
..	**- parent dir**	**$cd ..**
\	- escape to turn off the special meaning of metacharacters	**$echo \$x**
*	- match any	**$echo $(ls -l a*)**
?	- match one	**$echo $(ls l??)**
	- ternary operator	**(condition?true:false).**
~	- home	**$cd ~**

Shell: Summary

` xxx`	- command substitution	**$echo `ps -aux`**
$(xxx)	- command substitution	**$echo $(ps -aux)**
-	- pass stdin/stdout	**$ls -l h*|diff list -**
;	- separating multiple commands	**$ps;ls;finger**
“xxx”	- weak quote	**$echo “foo$x”**
‘xxx’	- strong quote	**$echo ‘foo$x’**
$x	- special variables	Positional arguments(**$1..$9**) Name of the script (**$0**) Number of arguments (**$#**) All parameters (**$*** and $@) Exit status (**$?**) Process ID (**$$**)

Shell: Summary

| - pipe

Technically this redirects standard output of one command to be the standard input of another. Error messages that are printed to standard error are not redirected with the pipe.

```
$ls |sort
```

> - Command output can be redirected to a file

```
$echo "smoothing=10mm" > settings.txt
```

>> - Command output can be appended to a file

```
$echo "No lowpass" >> settings.txt
```

(()) - assignment
let - assignment

```
$x=2
$y=3
$ ((x+=y))
5
$ echo $x
$let x=$x+$y
$echo $x
8
```

Shells – Expressions

- Sequence of operators and operands that reduces to a single value.

x=2	$x + $y	$x < $y
y=4	($x * $y) / $x - $y	$x != $y

- The expr command evaluates expressions. (requires space + escaping)

- Some operators:
  - Arithmetic operators
  - File operators
  - Comparison operators
  - Test operator
  - Logical operators
  - File operators
  - Comparison operators

```
rekha@rekha-ThinkPad-P50:~$x=2
rekha@rekha-ThinkPad-P50:~$x= 3
rekha@rekha-ThinkPad-P50:~$ expr $x \* $y
6
rekha@rekha-ThinkPad-P50:~$ expr $x + $y
5
rekha@rekha-ThinkPad-P50:~$ expr $x - $y
-1
rekha@rekha-ThinkPad-P50:~$ expr $x / $y
0
rekha@rekha-ThinkPad-P50:~$ expr $x \< $y
1
rekha@rekha-ThinkPad-P50:~$ expr $x \> $y
0
```

Shells – test operator

test expression

[expression]

- A shorthand version uses [and] around the arguments.
  WARNING: be very careful to put the spaces in correctly!
- Performs a variety of checks.
- Returns exit status of 0 if expression is true; 1 otherwise.

$[$a = my]

$[11 > 2]

Shells – Arithmatic Test

-eq Equal to

-ne Not equal to

-gt Greater than

-ge Greater than or equal to

-lt Less than

-le Less than or equal to

$[$a -eq 2]

$[11 -gt 2]

Shells – operators

String operators

- s1 == s2 Equal to
- s1 != s2 Not equal to
- -z str str is zero/null string
- -n str str is non-zero/not null

Logical operators

- expr1 AND expr2 → expr1 && expr2
- expr1 OR expr2 → expr1 || expr2
- NOT expr → !expr

```
$a=4
$ [ $a -eq 3 ] && echo howdy!
$ [ $a -eq 4 ] && echo howdy!
howdy!
$ [ $a -eq 4 ] || echo howdy!
$ [ $a -eq 3 ] || echo howdy!
howdy!
```

Shells – File Operators

File Unary operators

- -e file : file exists?
- -r file : file exists and readable?
- -w file : file exists and writable?
- -x file : file exists and executable?
- -L/-h file : file exists and a symbolic link?
- -f file : file exists and a regular file?
- -s file : if file is of 0 length
- -d file : file exists and a directory?

Binary Operators

- file1 -nt file2 :

  file1 newer than file2?

- file1 -ot file2 :

  file1 older than file2?

```
$ [ -e file1 ]
$ echo $?
1
$ [ -e list ]
$ echo $?
0
```

Control Flow: if-then-else

- The if command works like in most programming languages.
- It usually uses the result of the test command and its syntax is:

```
if [ EXPRESSION ] ; then
COMMANDS ;
 else
COMMANDS2 ;
fi
```

The else part is optional

```
$a=2
$ if [ $a = 2 ] ;
   then
      b="y-axis";
   fi
```

```
$ cat > test.sh
if who | grep $1 > /dev/null
then
   echo "$1 is logged in."
else
   echo "$1 is not logged in"
fi
$ . test.sh rekha
rekha is logged in.
$ . test.sh foo
foo is not logged in
```