Shell Scripting

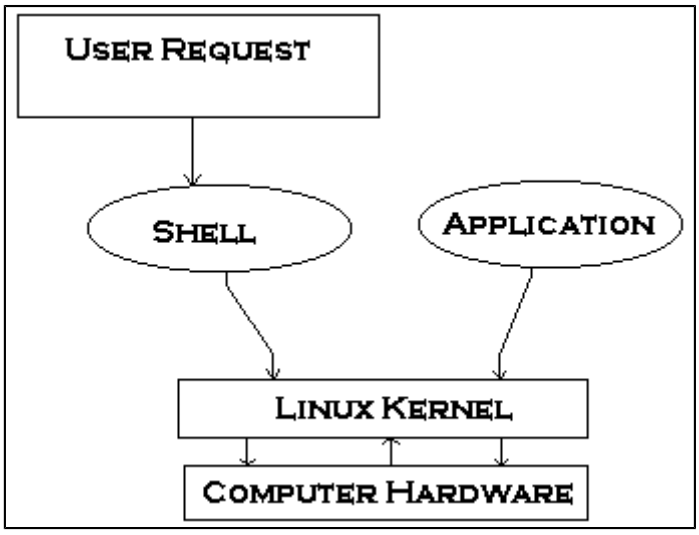
What is the Kernel?

Kernel is heart of Linux Os.

It manages resource of Linux Os. Resources are the facilities available in Linux. For e.g. Facility to store data, print data on printer, memory, file management etc .

Kernel decides who will use this resource, for how long and when. It runs your programs (or set up to execute binary files).

The kernel acts as an intermediary between the computer hardware and various programs/application/shell.



It's Memory resident portion of Linux. It performance following task :-

- •I/O management
- •Process management
- •Device management
- •File management
- •Memory management

What is Linux Shell ?

Computer understand the language of 0's and 1's called binary language.

In early days of computing, instructions were provided using binary language, which is quite difficult for all of us, to read and write. Shell is a special program called that accepts your instruction or commands in English (mostly) and if it is a valid

command, it is passed to kernel.

Shell is a user program or it's an environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Several shell available with Linux including:

Shell Name	Developed by	Where	Remark
BASH (Bourne-Again SHell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C SHell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (Korn SHell)	David Korn	AT & T Bell Labs	--
TCSH	See the man page. Type $ man tcsh	--	TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH).

Tip: To find all available shells in your system type following command:
$ cat /etc/shells

Note that each shell does the same job, but each understand a different command syntax and provides different built-in functions.

In MS-DOS, Shell name is COMMAND.COM which is also used for the same purpose, but it's not as powerful as our Linux Shells are!

Any of the above shells reads commands from user (via Keyboard or Mouse) and tells Linux Os what the users want. If we are giving commands from keyboard it is called command line interface (Usually in-front of $ prompt, This prompt depends upon your shell and Environment that you set or by your System Administrator, therefore you may get different prompt).

Tip: To find your current shell type following command
$ echo $SHELL

What is a Shell Script ?

Normally shells are interactive. It means shells accept commands from you (via keyboard) and execute them. But if you use commands one by one (sequence of 'n' number of commands) , the you can store this sequence of commands to a text file and tell the shell to execute this text file instead of entering the commands. This is know as ***shell script***.

"*Shell Script is **series of commands** written **in plain text file**. Shell script is similar to the batch file in MS-DOS, but is more flexible and powerful than a DOS batch file.*"

Why write Shell Scripts?

- •Shell script can take input from user, file and output them on the screen.
- •Useful to create our own commands.
- •Save time.
- •To automate routine tasks
- •Espcially useful in system administration where tasks like routine backups need to be automated.

Writing a shell script

Following steps are required to write shell script:

(1) Use any editor like vi or gedit to write shell script.

(2) After writing shell script set execute permissions for your script as follows
syntax:
chmod permission your-script-name

Examples:
$ chmod +x your-script-name (symbolic mode)
$ chmod 755 your-script-name (numeric mode)

Note: This will set read write execute(7) permission for owner, for group and other permission is read and execute only(5).

(3) Execute your script as
syntax:
bash your-script-name
sh your-script-name

./your-script-name

Examples:
$ bash bar
$ sh bar
$ ./bar

NOTE In the last syntax ./ means current directory, But only . (dot) means execute given command file in current shell without starting the new copy of shell, The syntax for . (dot) command is as follows
Syntax:
. command-name

Example:
$. foo

Using "source" or "." causes the commands to run in the current process. Running the script as an executable gives it its own process.

This is useful when a script is to be used in another script. Sourcing a file (dot-command) imports code into the script, appending to the script (same effect as the #include directive in a C program). The net result is the same as if the "sourced" lines of code were physically present in the body of the script. This is useful in situations when multiple scripts use a common data file or function librar

How Shell locates the executables

The locations /bin, /usr/bin are set in the environment variabel $PATH. Shell looks searches for a file in these paths. To add a path to the variable:

```
export PATH=$PATH:/path/to/dir1:/path/to/dir2
OR
set path = ($path /path/to/dir1 /path/to/dir2)
```

Shell Built in Variables

Shell Built in Variables	Meaning
$#	Number of command line arguments. Useful to test no. of command line args in shell script.
$*	All arguments to shell

$@	Same as above
$-	Option supplied to shell
$$	PID of shell
$!	PID of last started background process (started with &)

Variables in Shell

In Linux (Shell), there are two types of variables:
(1)**System variables** - Created and maintained by Linux itself. This type of variables are defined in CAPITAL LETTERS.
(2) **User defined variables (UDV)** - Created and maintained by user. This type of variables are defined in lowercase letters.

You can get the values of the system variables by giving command like **$ set**, some of the important System variables are:

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our Os type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME	User name who is currently login to this PC

To print the values :
$ echo $USERNAME
$ echo $HOME

How to define user defined variables (UDV)

To define UDV use following syntax
Syntax:
variable name=value

'**value**' is assigned to given '**variable name**' and Value must be on right side = sign.

Leave no spaces on either side of the '=' operator. (Causes the interepreter to parse as a string)

Variables are case sensitive.

To defnite a null variable, definte it as an empty string, i.e. temp=””

Variable names can start with _ or alphabets and can be alphanumeric.

To acess value of UDV use following syntax
Syntax:
$variablename

Echo command to display text or value of variable.

echo [options] [string, variables...]
Displays text or variables value on screen.
Options
-n Do not output the trailing new line.
-e Enable interpretation of the following backslash escaped characters in the strings:
\a alert (bell)
\b backspace
\c suppress trailing new line
\n new line
\r carriage return
\t horizontal tab
\\ backslash

Shell Arithmetic

Use to perform arithmetic operations.

Syntax:
expr op1 math-operator op2

Examples:

$ expr 1 + 3
$ expr 2 - 1
$ expr 10 / 2
$ expr 20 % 3
$ expr 10 * 3
$ echo `expr 6 + 3`

Note:
expr 20 %3 - Remainder read as 20 mod 3 and remainder is 2.
expr 10 * 3 – For multiplication use * and not *

Parameter substitution.

Now consider following command
$(**$ echo 'expr 6 + 3'**)

The command (**$ echo 'expr 6 + 3'**) is know as **Parameter substitution.**When a command is eclosed in backquotes, the command get executed and we will get output. Mostly this is used in conjunction with other commands. For e.g.

```
$pwd
$cp /mnt/cdrom/lsoft/samba*.rmp `pwd`
```

More about Quotes

There are three types of quotes

Quotes	Name	Meaning
"	Double Quotes	"Double Quotes" - Anything enclosed in double quotes removes meaning of that characters (except \ and $). globbing – allows expansion of filenames and matching characters from regular expressions and variables ($)
'	Single quotes	'Single quotes' - Enclosed in single quotes remains unchanged, strictly parsed as a text string
`	Back quote	`Back quote` - To execute command

Exit Status

By default in Linux if particular command/shell script is executed, it returns a value which can be used to check whether command or shell script executed successfully or not.

Conventionally,

(1) If return *value is zero* (0), command executed successfully.
(2) If return *value is nonzero*, some sort of error encountered while executing command/shell script.

This value is know as ***Exit Status***.

To determine this exit Status you can use **$?** special variable of shell.

This can be used to link up scripts. The read Statement

Use to get input (data from user) from keyboard and store (data) to variable.
Syntax:
read variable1, variable2,...variableN

The read Statement

Use to get input (data from user) from keyboard and store (data) to variable.
Syntax:
read variable1, variable2,...variableN

Wild cards (Filename Shorthand or meta Characters)

Wild card /Shorthand	Meaning	Examples	
*	Matches any string or group of characters.	**$ ls ***	will show all files
		$ ls a*	will show all files whose first name is starting with letter 'a'
		$ ls *.c	will show all files having extension .c
		$ ls ut*.c	will show all files having extension .c but file name must begin with 'ut'.
?	Matches any single character.	**$ ls ?**	will show all files whose names are 1 character long
		$ ls fo?	will show all files whose names are 3

			character long and file name begin with fo
[...]	Matches any one of the enclosed characters	**$ ls [abc]***	will show all files beginning with letters a,b,c

Note:

[..-..] A pair of characters separated by a minus sign denotes a range.But

$ ls /bin/[!a-o]

$ ls /bin/[^a-o]

If the first character following the [is a ! or a ^ ,then any character not enclosed is matched i.e. all files that have file names that beginning with a,b,c,e...o will not be displayed.

Mulitple commands on one command line

Syntax:

command1;command2

To run two command with one command line.

Examples:

$ date;who

Splitting a command to span multiple command lines

Use the \ operator

Command Line arguments

Positional Parameters.

Telling the command/utility which option to use.

Informing the utility/command which file or group of files to process (reading/writing of files).

$0 – absolute path of the script

$1- $9 – positional parameters

Also note that you ***can't assign a new value to command line arguments i.e positional parameters***. So following all statements in shell script are invalid:

$1 = 5

$2 = "My Name"

$* or $@ refer to an all the parameters passed (excluding $0)

$# holds the number of parameters passed to the script (excluding $0)

I/O Redirection and file descriptors

> - output redirection

>> - output and append

< - input redirection

eg. cat > file.

Input the contents and Ctrl+d

Standard File	File Descriptors number	Use	Example
stdin	0	as Standard input	Keyboard
stdout	1	as Standard output	Screen
stderr	2	as Standard error	Screen

You can redirect the output from a file descriptor directly to file with following syntax
Syntax:
file-descriptor-number>filename

2>/dev/null – to redirect errors so that they are not displayed on stderr

1>&2 directs the standard output (stdout) to standard error (stderr) device.

Functions

Syntax:

```
function-name ( )
{
      command1
      command2
      .....
      ...
      commandN
```

```
        return #OR return value
}
```

To call the function, simply use function-name.

Functions can be stored in the in the .bash_profile, so that they are available in every session.

If we store the functions in a file called *function_library* then you must include this file in your script using dot command as follows:

$. *function_library #sourcing the file into existing script*

Parameters can be passed to the function by position. i.e. function_test arg1 arg2

Filter

If a Linux command accepts its input from the standard input and produces its output on standard output is know as a filter. A filter performs some kind of process on the input and gives output. For e.g.. Suppose you have file called 'hotel.txt' with 100 lines data, And from 'hotel.txt' you would like to print contains from line number 20 to line number 30 and store this result to file called 'hlist' then give command:
$ tail +20 < hotel.txt | head -n30 >hlist

Here **head** command is filter which takes its input from tail command (tail command start selecting from line number 20 of given file i.e. hotel.txt) and passes this lines as input to head, whose output is redirected to 'hlist' file.

Operators

Arithmetic : + - / * %

Relational : -lt -le -gt -ge –eq –ne

Logical : -a -o !

String comparison

String Comparison	True if
string1 == string2	strings are equal

string1 != string2	strings are not equal
-n string	string not null
-z string	string is null

Testing files

test -r filenm	user has read permissions
-w filenm	user has write permissions
-x filenm	user has execute permissions
-f filenm	filenm is regular file
-d filenm	filenm is directory
-c filenm	filenm is character special file
-b filenm	filenm block special file

if condition

if condition which is used for decision making in shell script, If given condition is true then command1 is executed.

Syntax:

```
if condition
then
        command1 if condition is true or if exit status
        of condition is 0 (zero)
        ...
        ...
fi
```

For compression you can use test or [expr] statements or even exit status can be also used.

test command or [expr]

test command or [expr] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero for false.

Syntax:

test expression OR [expression]

if...else...fi

If given condition is true then command1 is executed otherwise command2 is executed.

Syntax:

```
if condition
then
            condition is zero (true - 0)
            execute all commands up to else statement

else
            if condition is not true then
            execute all commands up to fi
fi
```

Multilevel if-then-else

Syntax:

```
if condition
then
            condition is zero (true - 0)
            execute all commands up to elif statement
elif condition1
then
            condition1 is zero (true - 0)
            execute all commands up to elif statement
elif condition2
then
            condition2 is zero (true - 0)
            execute all commands up to elif statement
else
            None of the above condtion,condtion1,condtion2 are true
(i.e.
            all of the above nonzero or false)
            execute all commands up to fi
fi
```

Loops in Shell Scripts

Bash supports:

- for loop
- while loop

Note that in each and every loop,

(a) First, the variable used in loop condition must be initialized, then execution of the loop begins.

(b) A test (condition) is made at the beginning of each iteration.

(c) The body of loop ends with a statement that modifies the value of the test

(condition) variable.

for Loop

Syntax:

for { variable name } in { list }
do
execute one for each item in the list until the list is
not finished (And repeat all statement between do and done)
done

Syntax:

```
for (( expr1; expr2; expr3 ))
do
      .....
                    ...
      repeat all statements between do and
      done until expr2 is TRUE
Done
```

`eg.` for ((i = 0 ; i <= 5; i++))

Note : for i in {1..100} is also a valid loop declaration.

while loop

Syntax:

```
while [ condition ]
do
      command1
      command2
      command3
      ..
      ....
 done
```

The case Statement

The case statement is good alternative to Multilevel if-then-else-fi statement. It enables you to match several values against one variable. Its easier to read and write.

Syntax:

case $variable-name in
pattern1) command
...
..

```
                command;;
    pattern2)   command
                ...
                ..
                command;;
    patternN)   command
                ...
                ..
                command;;
    *)          command         #default
                ...
                ..
                command;;
esac
```

Debugging shell scripts

Use -v and -x option with sh or bash command to debug the shell script. General syntax is as follows:

Syntax:

sh option { shell-script-name }

OR

bash option { shell-script-name }

Option can be

-v Print shell input lines as they are read.

-x After expanding each simple-command, bash displays the expanded value of PS4 system variable, followed by the command and its expanded arguments.

Note : set -v OR set -x can be used in the script itself to enable these flags and script can be executed with ./script.sh. set +x can be used to turn off the debugging flag.

/dev/null - Use to send unwanted output of program

This is special Linux file which is used to send any unwanted output from program/command.

Syntax:

command > /dev/null

Local and Global Shell variable (export command)

Normally all our variables are local. Local variable can be used in same shell, if you load another copy of shell (by typing the **/bin/bash** at the $ prompt) then new shell ignored all old shell's variable. For e.g. Consider following example

$ vech=Bus

$ echo $vech

Bus

$ /bin/bash

$ echo $vech

NOTE:-Empty line printed

$ vech=Car

$ echo $vech

Car

$ exit

$ echo $vech

Bus

Command	Meaning
$ vech=Bus	Create new local variable 'vech' with Bus as value in first shell
$ echo $vech	Print the contains of variable vech
$ /bin/bash	Now load second shell in memory (Which ignores all old shell's variable)
$ echo $vech	Print the contains of variable vech
$ vech=Car	Create new local variable 'vech' with Car as value in second shell
$ echo $vech	Print the contains of variable vech
$ exit	Exit from second shell return to first shell
$ echo $vech	Print the contains of variable vech (Now you can see first shells variable and its value)

Global shell defined as:

"*You can copy old shell's variable to new shell (i.e. first shells variable to seconds shell), such variable is know as Global Shell variable.*"

To set global varible you have to use export command.

Syntax:

export variable1, variable2,.....variableN

Examples:

$ vech=Bus

$ echo $vech

Bus

$ export vech

$ /bin/bash

$ echo $vech

Bus

$ exit

$ echo $vech

Bus

Command	Meaning
$ vech=Bus	Create new local variable 'vech' with Bus as value in first shell
$ echo $vech	Print the contains of variable vech
$ export vech	Export first shells variable to second shell i.e. global varible
$ /bin/bash	Now load second shell in memory (Old shell's variable is accessed from second shell, *if they are exported*)
$ echo $vech	Print the contains of variable vech
$ exit	Exit from second shell return to first shell
$ echo $vech	Print the contains of variable vech

Conditional execution i.e. && and ||

The control operators are && (read as AND) and || (read as OR). The syntax for AND list is as follows

Syntax:

command1 && command2

command2 is executed if, and only if, command1 returns an exit status of zero.

The syntax for OR list as follows

Syntax:

command1 || command2

command2 is executed if and only if command1 returns a non-zero exit status.

You can use both as follows

Syntax:
command1 && comamnd2 if exist status is zero || command3 if exit status is non-zero
if command1 is executed successfully then shell will run command2 and if command1 is not successful then command3 is executed.

Example:
$ rm myf && echo "File is removed successfully" || echo "File is not removed"

If file (myf) is removed successful (exist status is zero) then "*echo File is removed successfully*" statement is executed, otherwise "*echo File is not removed*" statement is executed (since exist status is non-zero)

The shift Command

The shift command moves the current values stored in the positional parameters (command line args) to the left one position. For example, if the values of the current positional parameters are:

$1 = -f $2 = foo $3 = bar
after executing the shift command, the resulting positional parameters would be as follows:

$ = foo $2 = bar

Multiple parameters can be shifted by specifying a number with the shift command. The following command would shift the positional parameters two places:

shift 2

Note: shift makes handling command line arguements easier.

Selecting portion of a file using cut utility

General Syntax of cut utility:
Syntax:
cut -f{field number} {file-name}

Use of Cut utility:
Selecting portion of a file.

Crontab -e to edit

1. 11,12 -> run command for 11 and 12

2.11-18 -> run command for 11, 12, 13...,18

3. for every minute, * * * * *

4. for every 10 min, */10 * * * *

5.

Table: Cron special keywords and its meaning (instead of the 5 fields)

Keyword	Equivalent
@yearly	0 0 1 1 *
@daily	0 0 * * *
@hourly	0 * * * *
@reboot	Run at startup.