Operating System

Lab Manual#03

7 SHELL SCRIPTING

If the shell offers the facility to read its commands from a text file ,then its syntax and features may be thought of as a programming language, and such files may be thought of as scripts. So a shell is actually two things:

1. An interface between user and OS.
2. A programming language.

Shell Script is series of commands written in plain text file the shell reads the commands from the file just as it would have typed them into a terminal. The basic advantage of shell scripting includes:

1. Shell script can take input from user, file and output them on screen.
2. Useful to create our own commands.
3. Save lots of time.
4. To automate some task of day today life.
5. System Administration part can be also automated.

EXAMPLES

```
# !/bin/bash
#Firstshellscript
clear
echo "Hello World"
```

```
#!/bin/bash
#Script to print userinformation who currently login, current date and time
clear
echo "Hello $USER"
#echo induce next line at the end
#\c restrict output on same line and used with flag e
echo -e "Today is \c"; date
#print the line count
echo "Number of lines, total words and characters: "; ls -l | wc
echo "Calendar"
cal
exit 0
```

8 WRITE AND EXECUTE SHELL SCRIPT

Use any editor (gedit etc) to write shell script. Then save the shell script to a directory and name it intro. Shell scripts don't need a special file extension, so leave the extension blank (or you can add the extension .sh).

FILE MODES AND PERMISSIONS

Every Linux file has a set of permissions that determine whether you can read, write, or run the file. Running ls -l displays the permissions. The file's mode represents the file's permissions and some extra information. There are four parts to the mode, as illustrated in Figure1 .

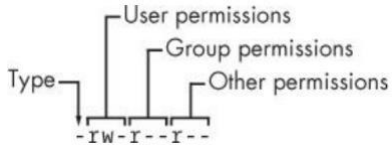

Figure 1: File Mode

The first character of the mode is the file type. A dash (-) in this position, as in the example, denotes a regular file, meaning that there's nothing special about the file. This is by far the most common kind of file. Directories are also common and are indicated by a d in the file type slot. The rest of a file's mode contains the permissions, which break down into three sets: user, group, and other, in that order. For example, the rw-characters in the example are the user permissions, the r– characters that follow are the group permissions, and the final r– characters are the other permissions.

File Mode	Explanation
r	Means that the file is readable.
w	Means that the file is writable.
x	Means that the file is executable (you can run it as a program).
-	Means nothing

The user permissions (the first set) pertain to the user who owns the file. The second set, group permissions, are for the file's group (somegroup in the example). Any user in that group can take advantage of these permissions. Everyone else on the system has access according to the third set, the other permissions.

MODIFYING PERMISSIONS

To execute a script first we will make it executable. To change permissions, use the chmod command. Only the owner of a file can change the permissions. There are two ways to write the chmod command:

1. chmod {a,u,g,o}{+,-}{r,w,x}

{all, user, group, or other}, {read, write, and execute} '+' means add permission and '-' means remove permission.
For example, to add group (g) and others (o) read (r) permissions to file, you could run these two commands:

$ chmod g+r file

$ chmod o+r file

Or you could do it all in one shot:

$ chmod go+r file

To remove these permissions, use go-r instead of go+r.

2. chmod h3DigitNumberi

   Every digit sets permission for the owner(user), group and others as shown in Table 2. To set the permission decipher the number first and then execute the command as

   $ chmod 700 file

User			Group			Other			3 Digit No.	Description
r	w	x	r	w	x	r	w	x		
1	0	0	0	0	0	0	0	0	400	user: read
0	0	0	1	0	0	0	0	0	040	group: read
1	1	1	0	0	1	0	0	1	711	user: read/write/execute; group, other: execute
1	1	0	1	0	0	1	0	0	644	user: read/write; group, other: read
1	1	0	0	0	0	0	0	0	600	user: read/write; group, other: none

Table 1: Examples of chmod with 3 Digit numbers

RUNNING SHELL SCRIPTS

Execute your script as

./script-name

9 IMPORTANT POINTS

Following are some important points while writing a script:

1. # is used to write comments in your script

2. Whenever a semicolon (;) is placed between two commands, shell will treat them as separate commands. So if you want to write all the commands in your shell script in one line place a semicolon in between them.
3. A script may start with the line #!/bin/bash to tell your interactive shell that the program which follows should be executed by bash.

10 VARIABLES IN SHELL

In Linux shell there are three types of variables:

1. User defined or shell variables
2. System or Environment variables
3. Parametric variables

USER DEFINED OR SHELL VARIABLES

The shell can store temporary variables, called shell variables, containing the values of text strings. Shell variables are very useful for keeping track of values in scripts. To assign a value to a shell variable, use the equal sign (=). Here's a simple example:

$ STUFF=blah

The preceding example sets the value of the variable named STUFF to blah. To access this variable, use

$STUFF (for example, try running echo $STUFF).

Rules for defining variables:

1. Variable name must begin with Alphanumeric character or underscore character, followed by one or more Alphanumeric character.
2. Don't put spaces on either side of the equal sign when assigning value to variable.
3. Variables are case-sensitive, just like filename in Linux.
4. You can define NULL variable as

   var=

   var=""

5. Do not use ?,* etc, to name your variable names.

```
# !/bin/bash
#
# variables in shell script
#
myvar=He ll o
echo $myvar
myvar= " Yes dear "
echo $myvar
myvar=7+5
echo $myvar
```

Notice in last assignment, myvar is assigned the string "7+5" and not the result i.e 12.

SYSTEM OR ENVIRONMENT VARIABLES

An environment variable is like a shell variable, but it's not specific to the shell. All processes on Unix systems have environment variable storage. The main difference be-tween environment and shell variables is that the operating system passes all of your shell's environment variables to programs that the shell runs, whereas shell variables cannot be accessed in the commands that you run. Assign an environment variable with the shell's export command. For example, if you'd like to make the $STUFF shell variable into an environment variable, use the following:

$ STUFF=blah

$ export STUFF

Environment variables are useful because many programs read them for configuration and options. Listed below are some common environment variables:

1. $PATH
   PATH is a special environment variable that contains the command path (or path for short). A command path is a list of system directories that the shell searches when trying to locate a command. For example, when you run ls, the shell searches the directories listed in PATH for the ls program. If programs with the same name appear in several directories in the path, the shell runs the first match-ing program. If you run

   $ echo $PATH

   you'll see that the path components are separated by colons (:). For example:

   $ echo $PATH

   will output /usr/local/bin:/usr/bin:/bin
   To tell the shell to look in more places for programs, change the PATH environ-ment variable. For example, by using this command, you can add a directory dir to the beginning of the path so that the shell looks in dir before looking in any of the other PATH directories.

$ PATH=dir:$PATH

Or you can append a directory name to the end of the PATH variable, causing the shell to look in dir last:

$ PATH=$PATH:dir

2. $HOME
   The home directory of the current user.

3. $IFS
   An input field separator; a list of characters that are used to separate words when the shell is reading input, usually space, tab and newline characters.

PARAMETRIC VARIABLES

These variables keep the values of the command line arguments passed to the Scripts. Most shell scripts understand command-line parameters and interact with the commands that they run. These parametric variable passed to the script make the code more flexible. These variables are like any other shell variable like Environment and Shell Variables, except that you cannot change the values of certain ones. Following are some parametric variables and a set of environment variables used to handle paramet-ric variables.

1. **Individual Arguments: $1, $2, ...**
   $1, $2, and all variables named as positive nonzero integers contain the values of the script parameters, or arguments. For example, say the name of the following script is pshow:

```
# ! / b i n / b a s h
echo  F i r s t  argument :    $1
echo  Third    argument :    $3
```

   Try running the script as follows to see how it prints the arguments:

$ ./pshow one two three

   Output looks like:
   First argument: one
   Third argument: three

   The built-in shell command shift can be used with argument variables to remove the first argument ($1) and advance the rest of the arguments forward. Specifically, $2 becomes $1, $3 becomes $2, and so on. For example, assume that the name of the following script is shiftex:

```
# !/bin/bash
echo Argument :   $1
shift
echo Argument :   $1
shift
echo Argument :  $1
```

Run it like this to see it work:
$ ./shiftex one two
three Argument: one
Argument: two
Argument: three
As you can see, shiftex prints all three arguments by printing the first, shifting the remaining arguments,and repeating.

The shell maintains a variable called $# that contains the number of items on the command line in addition to the name of the command ($0).

```
# !/bin/bash
if [ $#   gt 0 ]; then
    echo "Your command line contains $# arguments"
    echo "All arguments displayed using \$ positional parameter"
else
    echo "Your command line contains no arguments"
fi
```

2. **Number of Arguments: $#**
   The $# variable holds the number of arguments passed to a script and is especially important when running shift in a loop to pick through arguments. When $# is 0, no arguments remain, so $1 is empty.

3. **Script Name: $0**
   The $0 variable holds the name of the script, and it is useful for generating diagnostic messages. For example, say your script needs to report an invalid argument that is stored in the $errormsg variable. You can print the diagnostic message with the following line so that the script name appears in the error message:

   echo $0: $errormsg

4. **Process ID: $$**
   The $$ variable holds the process ID of the shell.

11 INPUT VALUES

The read command is used to get a line of input into a variable.

```
# !/bin/bash
#
```

```
# Script to read your name from key board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

Rules to input variables:

1. Each argument must be a variable name without the leading "$".
2. The built in command reads a line of input and separates the line into individual words using the "IFS" inter field separator.
3. Each word in the line is stored in a variable from left to right.
4. The first word is stored in the first variable, the second word to the second variable and so on.
5. If there are fewer variables than words, then all remaining words are then assigned to the last variable.
6. If you have more variables than words defined, then any excess variables are set to null.

```
# !/bin/bash
#
# Script to read your name from key board
#
read first middle last
echo "Hello $first $middle $last"
```

12 CONDITIONAL STATEMENTS

The Bourne shell has special constructs for conditionals, such as

1. If statement
2. If-else statement
3. If-elif statement
4. Case Statement

IF STATEMENT

The basic syntax of if statements is:

```
# !/bin/bash
if [ conditional expression ]
then
   statement1
   statement2
fi
```

TESTING CONDITION

test command or [expr]

are used to see if an expression is true, and if it is true it return zero(0) otherwise returns nonzero for false.
The folowing script determine whether given number is equal to 100 or not.

```
#!/bin/bash
#
#Scripttoseewhetherargumentispositi
ve count =100
if [ $count       eq  100 ]
then
        echo    " Count is  100 "
fi
```

The folowing script determine whether given argument number is positive using test.

```
#!/bin/bash
#
# Scriptto seewhether        argumentis      positive
if test$1   gt 0
then
        echo "$1 number is  positive"
fi
```

1. **Mathematical Operators**
   Figure 2 shows different conditions that we can use to test our expressions. It's important to recognize that the equal sign (=) looks for string equality, not numeric equality. Therefore, [1 = 1] returns 0 (true), but [01 = 1] returns false. When working with numbers, use -eq instead of the equal sign: [01 -eq 1] returns true.

Mathematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if [5 -ne 6]
-lt	is less than	5 < 6	if test 5 -lt 6	if [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if [5 -ge 6]

Figure 2: Conditions for test and expressions

2. **String comparisons**
   Following script checks whether the script's first argument is "hi" where a string comparison is made.

```
#!/bin/bash
#

if [ $1 = hi ]; then
echo "The first argument was hi"
fi
```

   There is a slight problem with the condition in preceding example due to a very common mistake: $1 could be empty, because the user might not enter a parame-ter. Without a parameter, the test reads [= hi], and the command aborts with an error. You can fix this by enclosing the parameter in quotes::

   if ["$1" = hi]; then

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

Table 2: String Comparisons

   -z and -n work with the string enclosed within double quotes.

```
# !/bin/bash
# String Comparisons

string1=$1
if [ z "$string1" ] ; then
        echo "NULL STRING"
fi
if [ n "$string1" ] ; then
        echo "NOT NULL STRING"
fi
```

3. **Test for file and directory**
   Most file tests that are commonly used are unary operations because they require only one argument: the file to test.

```
#!/bin/bash
somefile=a.txt
if [ r $somefile ]; then
        content=$(cat $somefile)
        echo $content
elif [ f $somefile ]; then
        echo "The file 'somefile' exists but is not readable to the script."
```

```
else
        echo "The file 'somefile' does not exist."
fi
```

In the above script first we check if the file somefile is readable. If so, we read it into a variable. If not, we check if it actually exists. If that's true, we report that it exists but isn't readable otherwise it moves to the else condition.

Test	Meaning
-s file	Non empty file
-f file	Is file exist or normal file and not a directory
-d dir	Is directory exist and not a file
-w file	Is writable file
-r file	Is read-only file
-x file	Is executable file

Table 3: Test for File and Directory

4. **Logical Operators**
   You can invert a test by placing the ! operator before the test arguments. For example, [! -f file] returns true if file is not a regular file. Furthermore, the -a and -o flags are the logical and and or operators. For example, [-f file1 -a file2].

Operator	Meaning
! expression	Logical NOT
expression1 -a expression2	Logical AND
expression1 -o expression2	Logical OR

Table 4: Logical Operators

IF ELSE STATEMENT

The basic syntax of if-else statements is:

```
if condition
then
    execute all commands upto else
else
   execute all commands upto fi
fi
```

Examples using if-else

```
# !/bin/bash
echo "Is it morning? Please answer yes or no"
read timeofday
if [ $timeofday = "yes" ]; then
       echo "Good morning"
```

```
else
      echo "good afternoon"
fi
```

```
# !/bin/bash
count =99
if [ $count    eq 100  ]
then
   echo "Count is  100"
else
   echo "Count is  not 100"
fi
```

```
# !/bin/bash
#scriptto seewhether        argument is positive or     negative
if [ $#    eq 0 ]
then
      echo "$0 : You  must supply one integer"
      exit 1
fi
if test $1  gt 0
then
      echo "$1 is positive number"
else
      echo "$1 is negative"
fi
```

IF-ELIF STATEMENT

The basic syntax of if-elif statements is:

```
if condition
then
      .............
elif condition
then
      .............
else
      .............
fi
```

Example:

```
# !/bin/bash
echo "Is it morning ? Please answer  yes or no"
read  timeofday
if [  $timeofday = "yes" ]; then
      echo "Good morning"
elif [ $timeofday = "no" ]; then
      echo "good afternoon"
else
       echo  "Sorry, $timeofday not      recognized. Enter yes or no"
       exit 1
fi
exit 0
```