National University of Computer and Emerging Sciences

Operating System
Lab - 03
Shell Scripting

Contents

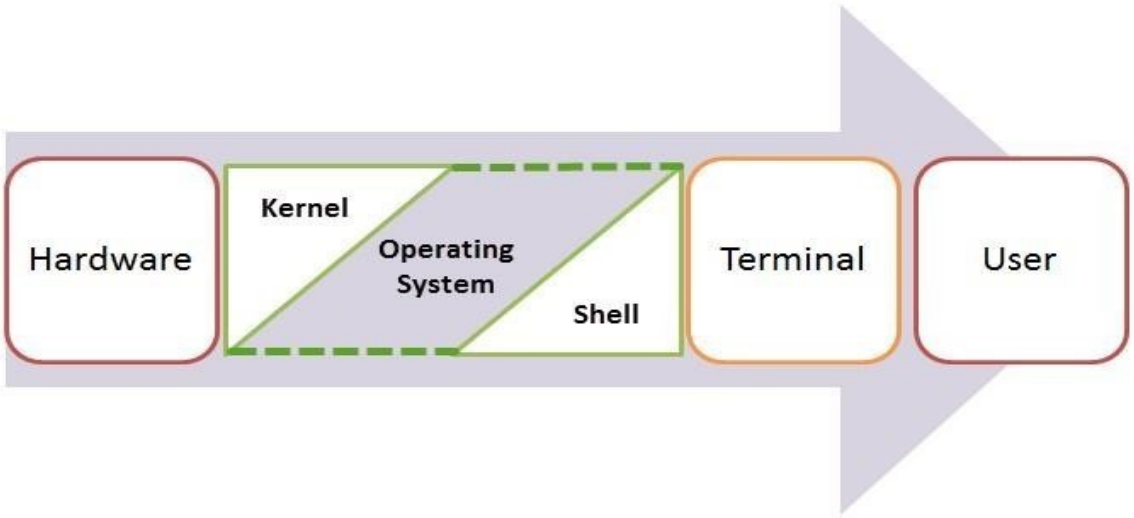
Objective

The lab examines the use of shell scripts as a way to write programs that accomplish various forms of processing in a Linux environment.

Shell Scripting

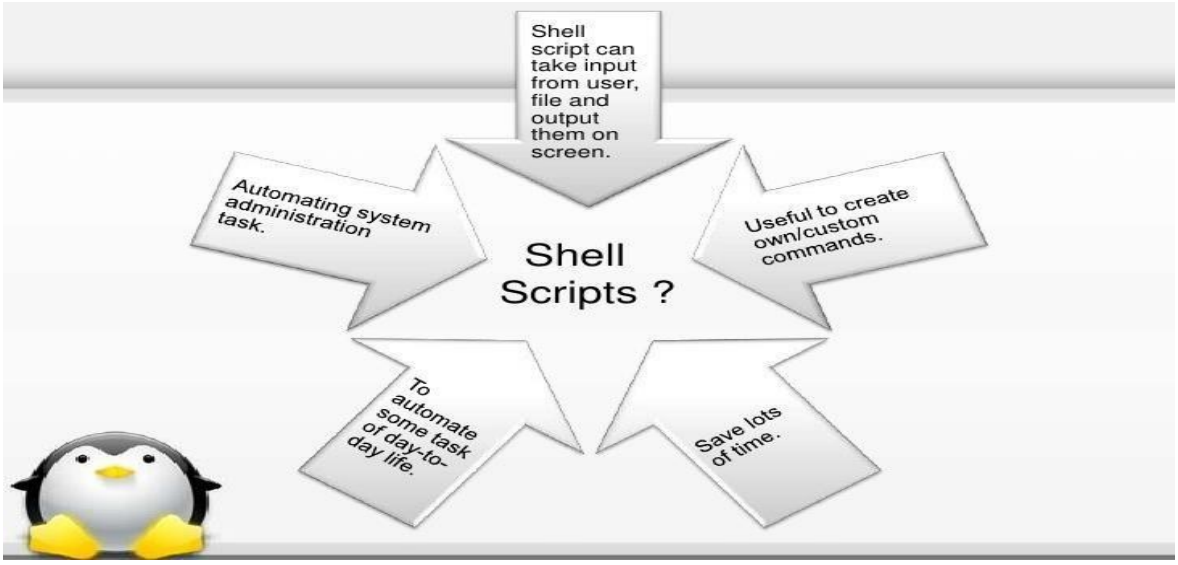
We have already talk about shell programming/scripting in *Lab Manual 02 – Introduction to Shell Scripting*. A shell script can be viewed as a high level program that is created with a simple text editor. Once created, a user may execute a shell script by simply invoking the filename of the shell script**. It is unnecessary to compile the shell script first. Rather, each time the shell script is invoked, a shell interprets or compiles the shell script as it executes it.**

To Sum up we can say in other words,
"Shell programming is a group of commands grouped together under single filename. The



shell can be used either interactively - enter commands at the command prompt, or as an interpreter to execute a shell script. Shell scripts are dynamically interpreted, NOT compiled."

A shell script invokes commands that you can issue through a command-line. In addition, a shell script also allows for more sophisticated processing, such as decision making and repetition for the work it invokes. In this manner, a shell script can be written to accomplish a series of tasks more easily and quickly than if the user had to type the commands for the tasks every time. Shell scripts are also a common way to schedule jobs to be automatically invoked by the system at specific times.



To create a shell script file, create a new file with any name and with extension '.sh'. Note you can also create the shell script file without the extension specified but then the file editor e.g. gedit's content will no longer be content-aware. The demonstration of creating the file is shown below:

```
student@oslab-vm:~/Desktop$ nano shelly.sh
student@oslab-vm:~/Desktop$ cat shelly.sh
echo "hellow from shelly shell script";
student@oslab-vm:~/Desktop$
```

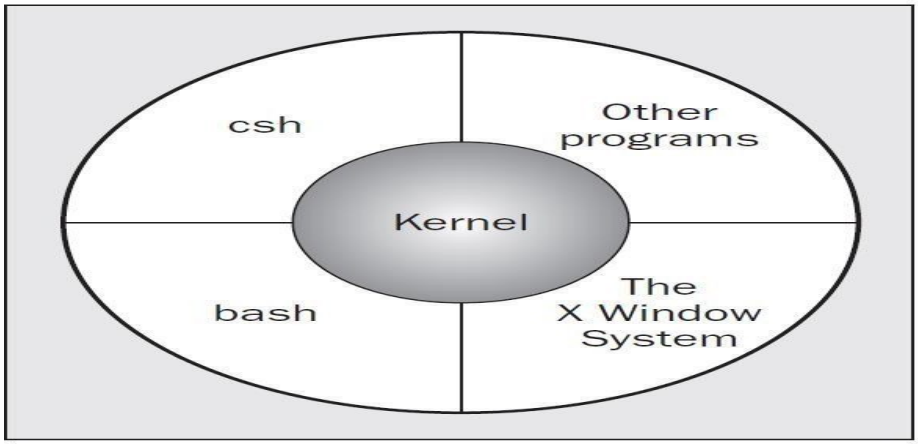
To invoke a shell script simply type in your terminal: './' followed by the filename with the extension '.sh'. For example, if I want to execute shell script having filename: 'shelly.sh' I would execute it as shown in the screenshot below.

Make sure the file i.e. in my case 'shelly.sh' is executable. In case the file is not executable you need to add access permission using 'chmod' command which we discussed in *lab manual 02*.

```
student@oslab-vm:~$ chmod +x Desktop/shelly.sh
student@oslab-vm:~$ ls -l Desktop/shelly.sh
-rwxrwxr-x 1 student student 40 Aug 27 01:33 Desktop/shelly.sh
student@oslab-vm:~$ ./Desktop/shelly.sh
hellow from shelly shell script
student@oslab-vm:~$ ./shelly.sh
bash: ./shelly.sh: No such file or directory
student@oslab-vm:~$ cd Desktop
student@oslab-vm:~/Desktop$ ./shelly.sh
hellow from shelly shell script
student@oslab-vm:~/Desktop$
```

Notice the path when I invoke shell script. The first one is: ‘./Desktop/shelly.sh’ which specified that in folder Desktop there is an executable filename ‘shelly.sh’ which it simply executed. Now when I tried ‘./shelly.sh’ it gave me the error of no such file or directory because the parent directory of Desktop does not contain that file. After changing directory to Desktop, I retried the command and it executed. Hence in invoking a shell script you need to specify the path where the filename is. Otherwise the system won’t be able to find it and/or execute it.

Shell Name	A Bit of History
sh (Bourne)	The original shell from early versions of UNIX.
csh, tcsh, zsh	The C shell, and its derivatives, originally created by Bill Joy of Berkeley UNIX fame. The C shell is probably the third most popular type of shell after bash and the Korn shell.
ksh, pdksh	The Korn shell and its public domain cousin. Written by David Korn, this is the default shell on many commercial UNIX versions.
bash	The Linux staple shell from the GNU project. bash, or Bourne Again SHell, has the advantage that the source code is freely available, and even if it’s not currently running on your UNIX system, it has probably been ported to it. bash has many similarities to the Korn shell.



```
student@oslab-vm:~/Desktop$ /bin/sh
$ date
Sun Aug 27 01:59:54 EDT 2017
$ date; pwd
Sun Aug 27 01:59:59 EDT 2017
/home/student/Desktop
$ echo 'hellow from alishah'; ./shelly.sh
hellow from alishah
hellow from shelly shell script
$ exit
student@oslab-vm:~/Desktop$ /bin/sh
$ message="This is operating system lab 03";
$ echo $message;
This is operating system lab 03
$ pwd; date; echo $message; ./shelly.sh;
/home/student/Desktop
Sun Aug 27 02:11:30 EDT 2017
This is operating system lab 03
hellow from shelly shell script
$
```

Variables

Like most other programming languages, a shell script can use variables to store data for future retrieval. The names for shell script variable may consist of alphabetic letters, underscores, or digits (but not in the beginning). Letters are case sensitive. There are two types of variables used in shell programming.

1. System Variables
2. User Defined Variables

System variables are created and maintained by Linux. These types of variables will be denoted in upper case letters. To get the details of available system variables, issue the command $set. Users variables are defined by users. They are usually defined in lower case letters.

To store a value in a variable within a shell script, specify the assignment as shown code example.

In the figure above, the value 2 is stored in x, the value 100.25 in y, and the string value ‘hello world’ in message. When a variable appears initially in a script in an assignment statement, the script automatically declares the variable with the appropriate data type to hold the assigned value. To obtain the value stored in the variable, specify the $ symbol followed by the variable name. As shown below:

The description of each instruction is written in comment with #.

```
# this is comment
# there must not be a space in assignment
x=2 #variable initialize
y=110.2 # all data types are considered as string
msg="hello world"

#assignment of values of other variables

z=msg
#msg will be the value of z

a=$msg
# hello world will be the value of a

#print the variable values
echo "we are printing";
echo "the value of x = $x"
echo  "the value of y = $y"
echo "the value of msg = $msg"
echo "the value of z = $z"
echo "the value of a = $a"
```

Notice that the example uses double quotes to enclose the text and variable name. You may omit the double quotes, but you cannot use single quotes. The use of single quotes in this example will actually instruct echo to write the enclosed contents exactly as they appear.
Run the above program and display output to your teacher.

#!/bin/sh is a special form of comment;
The #! characters tell the system that the argument that follows on the line is the program to be used to execute this file. In this case, /bin/sh is the default shell program.

‘echo’ & ‘read’ Statements

To send text, number or other information to the screen. You can use ‘echo’ command. It is the simplest form, you provide information you want as output as an argument to echo. Similarly you can use read command to take input for variable or taking multiple variable values from user. There are different switches which creates difference in taking input and separating input into different line to display.

```
# Taking input from user
read x
echo "The value of x = $x"

# Taking input in two variables
read x y

# use space while giving input,
# pressing enter will skip the input for secon variable
echo "The values of x= $x and y = $y"

# lets skip the frequent use of echo
# use -p switch with command to print a statement before input
read -p "enter values of x and y" x y
echo "The values of x= $x and y = $y"
```

Implement the above script and display the output to your teacher.

The different switch with echo and read can be explored with command **man.** With the specification of display with command. For instance as above -p switch with read command let the read command to include a print statement.

Command line Arguments

You can send arguments when you invoke a shell script, the arguments can be in any number. The shell store those arguments passed in variable $n where n = {1,2,3…9} , 0 is reserved for the file name. The below shell script and terminal shows the execution of a shell script which is using command line arguments.

```
# Reading command line argument
echo "The first command line argument is File name $0"
echo "The rest of the command line arguments are: First=$1 Second=$2 and Third=$3"
# Assignning the cla(commandl line argument) to variable in program
x=$1
y=$2
z=$y
echo "The values of x=$x y=$y and z=$z"
```

Execute the above script with following commands.

./filename 1 2 3

Other arguments available in shell script and their usage/description is provided in the table below.

S.NO	Variable	Description
1	$0	The filename of the current script.
2	$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).

3	$#	The number of arguments supplied to a script.
4	$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
5	$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
6	$?	The exit status of the last command executed.
7	$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
8	$!	The process number of the last background command.

Arithmetic Operations

In shell script, you can perform arithmetic operations, in calculations or within the formula. You must enclose the formula or calculation with a set of double parentheses (( )). There should be no space between the two right and left parentheses. The example below shows how to use arithmetic operations. You can use any number directly in expression such as a+2.

```
a=50
#taking user input
read -p "enter b value" b
#command line argument
c=$1
# Simple Arithmetic operation
#expression should be enclosed in
#brackets and $ with variable is not compulsory
((result=a+b+c))
echo "result=$result"

#combined expression
((result1=((a*a))-((b*c))/a+c))
echo "result1=$result1"
```

Implement the above example and check output. Show it to your teahcer

Some arithmetic operators which are commonly used is given below in the table:

S.NO	Name	Operator	Example
1	Assignment	'='	X=100, y=2.45
2	Addition	'+'	((x=$x + 100))
3	Subtraction	'-'	((z=1+1+100))
4	Multiplication	'*'	((abc=2*200))
5	Division	'/'	((abc=10/5))
6	Increment	'++'	((x++)), ((++x))
7	Decrement	'--'	((x--)), ((--x))

Conditional Statements

Shell script can use conditions and branches to specify under what circumstances one or more statements should execute. One form of conditions with branches involves if statement structure. A decision structure requires one or more conditions to indicate when a branch should be executed. A condition in aa shell script can involve numeric values or text values. A condition is typically enclosed within a set of double brackets [[ ]]. T***here must be a space before and***

after each double bracket. Otherwise the script may not function properly. If equivalent does not provide correct output use operator as switches such as -gt instead >

```
a=50
#taking user input
read -p "enter b value" b
#command line argument
c=$1
#Checking ondition
#expression should be enclosed in
#brackets [[ ]]and $ with variable is not compulsory
if [[ a > b ]];then
#keep space before closing brackets
echo "$a is greater"
#close condition with reversed if
fi

#nested if
if [[ a -gt b ]];then
if [[ a -gt c ]];then
echo " a is greater"
else
echo "c is greater"
#close inner if
fi
else
echo "b is greater"
#close outer if
fi
```

implement the above script and check the output and show it to your teacher.
Following are the other tables.

S.NO	Operator	Equivalent	Description	Example
1	-eq	=	Equal to	[[$count -eq 10]]
2	-ne	!=	Not equal to	[[$total -ne 1000]]
3	-lt	<	Less than	[[$balance -lt 0]]
4	-le	<=	Less than or equal to	[[$C -le $B]]
5	-gt	>	Greater than	[[5 -gt 6]]
6	-ge	>=	Greater than or equal to	[[$total -ge $subtotal]]

Shell script supports logical operators such as OR and AND to specify a compound condition such as condition that contain multiple comparisons. You can also use logical not to negate a condition. Table below shows the logical operator with example of their use.

S.NO	Operator	Description	Example
1	||	OR	[[$x = 0 || $y = 0]]
2	&&	AND	[[$A > $B && $B > $C]]
3	!	NOT	[[! $sum = 0]]

For a situation that requires more than one condition and/or more than two branches. You can

incorporate elif statement into one of the previous decision structure. The example below shows the use of if-then-elif-then-else statement. The condition for elif statement is same as if statement.

```
#nested if
if [[ a -gt b ]];then
echo "a is greater"
elif [[ a -lt c ]];then
echo " a is lesser"
else
echo " cis greater"
#No if closing with elseif
fi
```

While all the previous examples with if statement and conditions involved numeric variables and values, you can also compare text variables and text values with the operator listed previously. Such text comparisons refer to the ASCII value of the individual text characters on left and right sides. Here the first character is compared, followed by the second then the third and so on. Until it can determine whether the given comparison is true or false. Additionally, keep in mind that text is considered case sensitive and capital letters have lower ASCII values than the lowercase counterparts.

Consequently, a comparison of two string is true when both sides have exactly same text and case. The example below shows how to use text variable in string comparison however, the example below may exist potential error.

```
if [[ $YN = 'YES' ]]; then
    echo "Yes is specified"

else

    echo "You did not specify yes"
fi
```

However, imagine that $YN has no value then the condition becomes [[= 'YES']], resulting in a runtime error where the shell script does not execute correctly. To fix this problem you need to treat both sides of the comparisons as a string with an added letter. To ensure that each side contains at least one letter. The fix is demonstrated in an example below.

```
if [[ "x$YN" = 'xYES']]; then
        echo "Yes is specified"

else
        echo "You did not specify yes"
fi
```

In the script above we have added the letter 'x' on both sides however, it could have been any

other letter. The reason of adding an extra letter on both sides is because how a shell script interprets the script contents as it executes the script content. In this case, when executing if statement using any variable the shell script uses its value rather than the variable. If the variable is empty, the side of that comparison is considered empty hence resulting in a runtime error generated by the shell executing that script.

‘case’ Structure

Alternative form of shell script branching involves case structure. It allows the specification of a controlling value by a case statement. Following the case statement is a series of values or patterns, each with a branch of one or more statements. Like switch-case structure in C/C++, the shell compares the controlling value against the values to find a match, starting with the first and continuing till the last. When a match is found, the shell executes the corresponding branch. You can also include a default branch that is executed if the controlling value does not mtch any of the values or pattern.

```
#switch case
#take option to check case
read -p "input the case choice" x
case $x in
#only small close bracket
# | can be used to include more than one choice

a|A|1)
echo "This is the first choice"
#double semi colon to break
;;

b|B|2)
echo "This is the second choice"
;;

# * can not be used in other option as
# it is used as defualt

*)
echo " This is no choice";;

#close case in reverse
esac
```

Execute the above program and show the output to teacher.

Cases can be used as group for numeric range we have 0-9 and alphabets a-z.

```
#switch case
#take option to check case
read -p "input the case choice" x
case $x in
#only small close bracket
# group is []

[0-9])
echo "This is the first choice"
#double semi colon to break
;;

[a-z]|[A-Z])
echo "This is the second choice"
;;

# * can not be used in other option as
# it is used as defualt

*)
echo " This is no choice";;

#close case in reverse
esac
```

Change the previous case into group range and execute the program again.

Iterative Structure

To accomplish iteration in a shell script, you have a number of mechanisms available for that purpose. These involves while loops, for loops and a form of while loop called until. The syntax of these stated iterative loops is shown below.

Example of utilizing while loop, the below examples takes then number of directories to be created then using while loop it takes the names of those directories from user and tries to create them.

```
read -p "Enter the number of directories to be created: " numDir

while [[ $numDir > 0 ]]
#All iterative loops open with do and close with done
do
read -p "Enter the name of the directory: " dirName
mkdir $dirName
#CHECKING THE EXIT STATUS OF LAST COMMAND
if [[ $? = 1 ]] ; then
echo "Directory creation failed"
fi
#decrementing expression
((numDir--))
done
```

Implement the above loop and show the output to teacher.

The for loop can be used in many different ways. It is up to the program requirement.

First way to use for loop for simply reading the input values, Values can be command line or any. It will iterate according to number of input values.

```
#first types of for loop
#reading direct input
# i is iteration/counter variable
#after in number of inputs will be read by i
#separate each input with space
x=Pakistan2
for i in safia $1 1 $x

#open loop with do
do
echo "i = $i"

#close  loop
done
```

Task: Implementing the same above example with for loop instead while loop.

The another way is simply gives the range for number of iteration to loop.

```
#Second type of for loop
#iterating on loop number range
#.. used for range
for i in {0..20}
do
((result=i+2))
echo "$result"
done
echo " result=$result"
```

Number of iteration might be asked from user hence direct variable value will give error. Therefore seq command is used with providing starting range and variable as given in example below.

```
#Third type of for loop
#iterating cannot be done with providing a variable
read -p "Num of entries?" x
#for i in {0..$x}
#it will not work
# use seq command to for the requirement
for i in $(seq 1  $x)
#seq will run 1 to x value iteration
do
((result=i+2))
echo "$result"
done
echo " result=$result"
```

For loop as used in c language can also be used.

```
#Fourth type of for loop
#iterating cannot be done with providing a variable
read -p "Num of entries?" x
for ((i=0; i<x ; i++ ))
do
((result=i+2))
echo "$result"
done
echo " result=$result"
```

Now implement above all types of for loop and display the output to teacher.

Functions

To help modularize and reuse shell script code, you can use functions within a shell script like you can in other programming languages. As a short example shows a shell script that defines a function named 'Min' that prints the smaller of two arguments. The script asks the user to input two numbers and calls the Min function to report the smaller number from the two provided.

```
Min() {
        if [[ $1 -lt $2 ]]; then
                Smallest=$1

else
                Smallest=$2
        fi
}
read -p "Enter two whole numbers, Separated by space: " N1 N2
Min $N1 $N2
echo "The smallest is: " $Smallest
```

At first line of a function definition, it specifies the function name and an empty set of parentheses. Unlike many other programming languages, the parentheses at the first line of a function definition are always empty, whether or not you supply arguments to the function. The remainder of the function definition consists of the commands in the function body enclosed with { and } braces. To access any arguments with a function, use the shell script variable explained in 'Accessing Arguments' section.

Special Symbols

The following are the meaning of symbols used in shell scripting:

S.NO	Symbol	Description
1	#	Usually denotes the beginning of a comment
2	;	Separates two statements appearing in the same line
3	\	An escape sequence
4	$	Variable e.g. $PATH will give the value of the variable PATH

5	“” or ‘’	Strings
6	{}	Block of code
7	:	A null (no operator) statement
8	()	Group of commands to be executed by a sub shell
9	[[]]	Condition e.g. [[$count -eq 0]]
10	$(())	Evaluation of an arithmetic operations

Coursera

- The Unix Workbench (cover Week 3 for Lab 3)

https://www.coursera.org/learn/unix https://

seankross.com/the-unix-workbench/

Reference Tutorial

https://www.shellscript.sh