Linux Shell and Shell Programming

UNIX shell and shell programming

Objectives

The objective of this lab is to introduce you UNIX shell and shell programming.

1. Introduction of UNIX shell and shell scripting
2. Variables and Arrays
3. Basic operators and conditional statements
4. Loops controls
5. Basic commands and external program executions

Description

UNIX shell is a command-line user interface, which allow user to communicate with the Operating System. The shell also supports a scripting language – an interactive command language. Current UNIX systems have different shells, and all have their own uniqueness and advantages (read Section 1.3 - Shells in the textbook).

What is shell script?

Shell scripts are a set of one or more commands which will be interpreted by the shell and executed by the shell.

How to create a shell script?

These executable scripts always start with #!/bin/<shell_name>, where shebang operator (#!) allow script to direct to the location of shell and then from second line you can start writing the command you want to perform.

How to write comment in shell scripts?

As we know comments are important part of any programming/scripting language. And all languages have their own way to comment. In a shell script, we use # symbol to start a comment and this comment extends till the end of the line.

Now let's learn more about shell by implementing simple shell script which lists directory content at current location.

1. Create a file called ListDir.sh and enter following lines:

```
!#/bin/bash
# The script lists content of the current directory
ls
```

2. Save the file, change the file permissions to have execute permissions for the user, and execute the script using the following commands:

```
$ chmod +x ListDir.sh
$ ./ListDir.sh
```

Types of variables:

As you know, variables are the building blocks of any programming language, similarly, the shell script also uses variables and these variables are used to store values which can be pre-defined or can be initialized during the execution of script.

Local variable is a variable that is available during current instance of the shell. These variables also known as user-defined variables. In a shell script, variables automatically define their datatypes based on the value assigned.

Environment variable is a variable that is available to any child process of the shell at any point of time; which means these variables can be access from anywhere in the system. These variables are already defined by UNIX system or any program that you have installed. For instance, $USER, $HOME, $PATH, $UID, $0 … $9, $#, $@, etc. You could print all the environment variables that is current set in your shell using the *printenv* command.

Shell variable is a variable that is available for all scripts as a global variable for the current shell instance (current running session).

Now let's learn more about shell by implementing simple shell script:

1. Use an editor to create a file called HelloWorld.sh and write following lines:

```
#!/bin/bash
Hello_MSG="Hello World!"
echo $Hello_MSG
echo "Login username is "$USER
echo "Enter your name"
read NameEntered
echo "Hi $NameEntered - hope you are well!"
```

2. Save the file, change mode to provide executable permission, and execute the script using the following commands:

```
$ chmod +x HelloWorld.sh
$ ./HelloWorld.sh
Hello World!
Login username is tr27p
Enter your name
student
Hi student - hope you are well!
```

Defining an array:

An array is variable that stores multiple scalar values in one variable. Shell also support array variables; it is defined as follows:

array_name=(value1 … valueN) or array_name[index]=value

Note that the indexing of shell arrays starts from 0, and if you insert a new value at wrong index then shell will add at the end of the array without an error or warning.

Accessing an array:

If you want to access a value at particular index, you can use *${array_name[index]}*, and if you want to access all values at the same time
then *${array_name[*]}* or *${array_name[@]}*.

Now let's learn more about shell by implementing simple shell script:

1. Use an editor to create a file called Arrays.sh and enter the following lines:

```
#!/bin/bash
Name=("Alice" "Bob" "Charlie")
Name[3]="dave"
echo "Names are: " ${Name[*]}
```

2. Save the file, change the mode to have execute permissions, and execute the program.

Performing operations:

A shell supports different types of operators, here are some examples:

1. Arithmetic operators: These operators perform arithmetic task such as addition, subtraction, multiplication, etc.

operator	Description	Example
+	Gives addition of two numerical values	`expr $a + $b`
-	Gives subtraction of two numerical values	`expr $a - $b`
*	Gives multiplication of two numerical values	`expr $a \* $b`
/	Gives division of two numerical values	`expr $a / $b`
%	It returns remainder after dividing two numerical values	`expr $a % $b`
=	This operator assigns value	a = $b
==	It compares two numerical values and returns true if equals	[$a == $b]
!=	It compares two numerical values and returns true if not equals	[$a != $b]

2. Relational operators: These operators give the relation between numerical values.

Operator	Description	Example
-eq	Checks two numerical values are equal or not; returns true if equals	[$a -eq $b]
-ne	Checks two numerical values are equal or not; returns true if not equals	[$a -ne $b]
-gt	Checks which numerical values is greater; returns true if left hand side value is greater	[$a -gt $b]

-lt	Checks which numerical values is smaller; returns true if left hand side value is less than right hand side	[$a -lt $b]
-ge	Checks which numerical values is greater; returns true if left hand side value is greater than or equal to left hand side	[$a –ge $b]
-le	Checks which numerical values is smaller; returns true if left hand side value is less than or equal to left hand side	[$a –le $b]

3. Boolean operators: These operators perform a simple Boolean task such as true/false, AND, OR

Operator	**Description**	**Example**
!	This perform invers operation on current condition	[! false]
-o	This is logical OR operation, returns true if at least one condition is true	[$a -ne $b –o $a –ne $c]
-a	This is logical AND operation, returns true if all conditions are true	[$a -ne $b –a $a –ne $c]

4. String operators: These operators are specific to string values, where you can find if two strings are the same or not, if a string is empty or not, etc.

Operator	**Description**	**Example**
=	Returns true if both strings are equal	[$a = $b]
!=	Returns true if both strings are not equal	[$a != $b]
-z	Returns true when string size is zero	[-z $a]

-n	Returns true when string size is non-zero	[-n $a]
str	Returns true when string is non-empty	[$a]

Conditional statements:

As all programming languages UNIX shell also supports conditional statements.

1. if … fi statement: this is simple if condition, which does not have else part.

```
if [ condition ]
then
Statements in true condition
fi
```

2. if … else … fi statement: this is simple if and else condition.

```
if [ condition ]
then
Statements in true condition
else
Statements in false condition
fi
```

3. if … elif … else … fi statement: this is else if condition where you can check more conditions as you need.

```
if [ condition_1 ]
then
Statements in true condition_1
elif [ condition_2 ]
then
Statements in true condition_2
else
Statements in false conditions
fi
```

4. case … esac: this is switch case condition where you give the case situation to select the next step.

```
case $option in
option_1)
Statements for option_1
;;
```

```
option_2)
Statements for option_2
;;
option_3)
Statements for option_3
;;
*)
Default statement (No options are matched)
;;
esac
```

Loops:

UNIX shell has four types of loops:

1. while loop: this is simple while loop; same as c programming language

```
while [ condition ]
do
Statements will execute until condition is true
done
```

2, for loop

```
for var in var1 var2 … varN
do
Statements will execute till N variable
done
```

3. until loop

This loop is exactly opposite of while loop. Which means, the loop will continue in false condition, however for until loop false condition is true condition.

```
until [ condition ]
do
Statements will execute until condition is true
done
```

4. select loop

This loop is unique, which never ends by itself, means this loop does not have conditions that can be false. To end the select loop user need to decide the end case for it. This loop was introduced in KornShell and then adapted by Bourne shell. The main idea behind it to give a visible option to end-user as a selection.

```
select option in option1 option2 … optionN
do
Statements will execute
done
```

Basic UNIX commands:

- date: This command output the current system date.

- cp: This is a copy command, which copies a file. Copy command need source and destination path to copy a file.
- mkdir: Make directory command creates directory at given location.
- mv: move or rename command to move or rename file or directory. This command needs source and destination path to execute.
- rm: remove command use to delete files from given location
- cat: this command displays the content of given inputs.

- head: this command displays the first certain number of lines from given input (you need to specify how many lines want to display; default is 10 lines)
- tail: this command displays the last certain number of lines from given input (you need to specify how many lines want to display; default is 10)
- find: this command is to find files with specific properties and patterns
- grep: this command is used to grep a specific pattern from select file/lines.

Use man command read more about all these commands.

External program execution:

As you know shell is user interface which allow you to communicate with the kernel and operating system. So, by using shell scripting you can run any external programs such as JAVA, C/C++, or any application which does not need any human interaction during their execution. This feature allows you to run your application as jobs that executes using fork and exec.

Now let's see simple scripts that calls simple c program.

hello.c

```
#include <stdio.h>

int main(int argc, char** argv){
        printf("Hello World!\n");
        return 0;
}
```

callexternal.sh

```
#!/bin/bash
gcc -o hello hello.c
./hello
echo $?
```

Now run the shell script.