# OPERATING SYSTEM LABORATORY MANUAL

## UNIVERSITY OF THE PUNJAB

**FACULTY OF COMPUTING & INFORMATION TECHNOLOGY, LAHORE**

**DEPARTMENT OF COMPUTER SCIENCE**

| | | |
|---|---|---|
| **Course:** | **Operating System Lab** | **Date:** |
| **Course Code:** | **CC-217-3L** | **Max Marks: 40** |
| **Faculty/Instructor's Name & Email:** | **Dr. Ahmad Hassan Butt (ahmad.hassan@pucit.edu.pk)** | |

**LAB MANUAL # 5**

**(SPRING 2023)**

Name:______________________________________ Enroll No: ____________________________

## Objective(s) :

To study and execute the Linux Shell Programming.

## Lab Tasks :

**Task 1 :** Write the output of the programs given.

**Task 2:** Write the output of the arithmetic programs provided in the lab.

**Task 3 :** Write the output of the following Control Structures.

**Task 4 :** Write a program to enter the numbers till the user wants and at the end it should display the maximum and minimum number entered.

## Lab Grading Sheet :

| **Task** | **Max Marks** | **Obtained Marks** | **Comments(*if any*)** |
|---|---|---|---|
| 1. | 10 | | |
| 2. | 10 | | |
| 3. | 10 | | |
| 4. | 10 | | |
| **Total** | **40** | | **Signature** |

**Note : Attempt all tasks and get them checked by your Instructor**

# Lab 05: LINUX Shell Programming - I

**Objective(s):**

To understand VIM Editor.

**Tool(s) used:**

Ubuntu, VIM Editor

**Introduction**

Shell programming is a group of commands grouped together under single filename. After logging onto the system a prompt for input appears which is generated by a Command String Interpreter program called the shell. The shell interprets the input, takes appropriate action, and finally prompts for more input. 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.

**Common Shells**

C-Shell - csh

The default on teaching systems Good for interactive systems Inferior programmable features.

Bourne Shell

bash or sh - also restricted shell – bsh (The Bourne Again Shell) It was written by Steve Bourne. Over the years the original Bourne Shell has been expanded, but it remains the basic shell provided in many commercial versions of Linux.

Korn Shell

It was written by David Korn This shell extended many features of Bourne Again Shell and added many new features.

Thomas C-Shell - tcsh

The TC Shell performs the same functions as Bourne Again Shell. It is an interactive command line interpreter as well as for high level programming languages.

Shell Keywords

echo, read, if fi, else, case, esac, for , while , do , done, until , set, unset, readonly, shift, export, break, continue, exit, return, trap , wait, eval ,exec, ulimit , umask.

**General Shell Terminologies**

The shebang line or hashbang #!

The "shbang" line is the very first line of the script and lets the kernel know what shall will be interpreting the lines in the script. The shbang line consists of #! Followed by the full pathname to the shell, and can be followed by options to control the behavior of shell.

Example

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

Comments

Comments are descriptive material preceded by a # sign. They are ineffect until the end of a line and can be started anywhere on the line.

Example

```
#This text is not interpreted by the shell.
```

## FIRST “HELLO WORLD” SHELL PROGRAM

**Step 1** Open VIM with the filename.sh extension.

**Step 2** Write the Hello World Program

```
#!bin/bash
#YOUR FIRST HELLO WORLD PROGRAM
   echo ‘Hello World!’
```

**Step 3** Execution of SHELL Script:

By using change mode

command

```
$ chmod u + x  hello.sh
$ ./hello.sh
```

<u>Shell Variables</u>

A variable is something to which we assign a value. The value assigned could be a number, text, or any data type.

Rules

- A variable name is any combination of alphabets, digits and an underscore.
- No commas or blanks are allowed within a variable name.
- The first character of a variable name must either be an alphabet or and underscore.
- Variables names should be of any reasonable length.

<u>Shell Variables</u>

1. **PATH** - Directory paths to search for commands.
2. **HOSTNAME** - The name of the computer.
3. **USER** - The user id of the user running this shell.
4. **SHELL** - The shell currently is used.
5. **TERM** - The type of terminal being used.
6. **PS1** - The prompt to print when then shell is ready for another command.

**Task 1** Write the output of the below programs.

**i)**
```
#!/bin/bash
#Variable Assignment & Accessing
 echo "Variable Name : "
 Name="Operating System"
 echo $Name
```

**OUTPUT**

ECHO Statement

Similar to the output statement. To print output to the screen, the echo command is used.

**Syntax:** Echo "String" (or) echo $ b (for variable).

**Example:** echo "What is your name?"

## READ Statement

To get the input from the user.

**Syntax:** read x y (no need of commas between variables)

**Reading user input:** The read command takes a line of input from the user and assigns it to a variable(s) on the right-hand side. The read command can accept multiple variable names. Each variable will be assigned a word.

**ii)**
```
#!/bin/bash
#Input from user
       echo "Enter your name"
       read NAME
       echo "Enter your age"
       read AGE
       echo "Enter your enrollment"
       read ENROLLMENT
       echo  "Hello $NAME, Your age is : $AGE  Your enrollment is :
       $ENROLLMENT"
```

## **OUTPUT**

**iii)**
```
#!/bin/bash
   #readonly Variables
         echo "Readonly Variables"
         Name=”David”
         readonly Name
         Name=’John’
```
**OUTPUT**

**iv)**
```
#!/bin/bash
         echo "Unset Variables : "
         Name=”John”
         unset Name
         echo $Name
```
**OUTPUT**

**Wildcards**

There are some characters that are evaluated by the shell in a special way. They are called shell meta characters or “Wildcards. These characters are neither number nor letters.

Example

*,?,[ ],$

$$echo $$ -- It represents process ID Number, or PID of the current shell

The following table shows a number of special variables that can be used in shell scripts.

| Variable | Description |
|---|---|
| $0 | The filename of the current script |
| $n | These variables correspond to the arguments with which a script is invoked. |
| $# | The number of arguments supplied to a script. |
| $* | All arguments are double quoted. If a script receives two arguments $* is equivalent to $1,$2 |
| $@ | All arguments are individually double quoted, equivalent to $1,$2 |
| $$ | Shows the number of current shell. |
| $! | The process number of last background command. |

**v)**
```
 #!/bin/bash
#Unix Special Commands
      echo "File Name = $0"
      echo "First Parameter = $1"
      echo "Second Parameter = $2"
      echo "Quoted Values = $@"
      echo "Quoted Values = $*"
      echo "Total number of parameters = $#"
```

**OUTPUT**

**vi)**
```
   #!/bin/bash
#Special Commands
      echo "File Name = $0"
      echo "First Parameter = $1"
      echo "Second Parameter = $2"
      echo "Quoted Values = $@"
      echo "Quoted Values = $*"
      echo "Total number of parameters = $#"
      echo $?
```

**OUTPUT**

**EXPRESSION Command**

Arithmetic Operations

To perform all arithmetic operations. The Bourne shell does not support arithmetic. LINUX/Linux commands must be used to perform calculations.

**Task 2** Write the output of the following arithmetic programs

```
#!/bin/bash
var1=10
var2=20
add=$(expr $var1 + $var2)
sub=$(expr $var1 - $var2)
multi=$(expr $var1 \* $var2)
div=$(expr $var1 / $var2)
mod=$(expr $var1 % $var2)
echo "Addition : $add"
echo "Subtraction : $sub"
echo "Multiplication : $multi"
echo "Division : $div"
echo "Modulus : $mod"
```

**OUTPUT**

## Operators

The Bourne shell uses the built-in test command operators to test numbers and strings.

**Example**

**Equality:**

| | |
|---|---|
| = | *string* |
| != | *string* |
| -eq | *number* |
| -ne | *number* |

**Logical:**

| | |
|---|---|
| -a | *and* |
| -o | *or* |
| ! | *not* |

**Relational:**

| | |
|---|---|
| -gt | *greater than* |
| -ge | *greater than, equal to* |
| -lt | *less than* |
| -le | *less than, equal to* |

**Arithmetic:**

+, -, *, /, %

## Control Structures

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. Two decision making statements are mentioned below:

- if…else statements
- case…esac statements

### if..fi Statements

The if construct is followed by a command. If an expression is to be tested, it is enclosed in square brackets. The then keyword is placed after the closing parenthesis. An if must end with a fi.

**Syntax:**

```
if [ expression ]
```

```
then
     Statements to be executed if true
fi
```

**Task 3** Write the output of the following Control Structures.

**i)**

```
 #!/bin/bash
a=10
b=20
if [ $a == $b ]
then
       echo "a is equal to b"
fi
if [ $a != $b ]
then
       echo "a is not equal to b"
fi
```

**<u>OUTPUT</u>**

if..else..Statements

The if…else…fi statements is the next form of control statements that allow shell to execute statements in more controlled way and making decision in two ways.

**Syntax**

```
if [ expression ]
then
        Statement(s) to be executed if true
 else
        Statement(s) to be executed if true
```

**ii)**

```
#!/bin/bash
a=10
b=20
if [ $a == $b ]
then
       echo  "a is equal to b"
else
       echo  "a is not equal to b"
fi
```

**OUTPUT**

## if…else…elif…fi Statement

The if…elif…fi statement is to make correct decision out of several conditions.

**Syntax**

```
if [ expression 1 ]
then
        Statement(s) to be executed if expression 1 is true
elif [ expression 2]
        Statement(s) to be executed if expression 2 is true
elif [ expression 3]
        Statement(s) to be executed if expression 3 is true
else
        Statement(s) to be executed if no expression is true
fi
```

**iii)**

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b]
then
   echo "a is less than b"
else
   echo "None of the conditions met."
fi
```

**OUTPUT**

case…easc Statement

You can handle multiple if…elif statements to perform a multiway branch. However, this is not the best solution, especially when all the branches depend the value of single variable.

**Syntax**

```
case word in
        pattern1)
Statement(s) to be executed if pattern1 matches;;
        pattern2)
        Statement(s) to be executed if pattern1 matches;;
        pattern3)
        Statement(s) to be executed if pattern1 matches;;
       easc
```

**iv)**

```
#!/bin/bash
echo "Enter fruit name"
read Fruit
case "$Fruit" in
       "apple")  echo "Apple pie";;
        "banana") echo "I like banana";;
        "kiwi")  echo "New Zealand famous for kiwi";;
esac
```

**OUTPUT**

## LOOPS

There are three types of loops: while, until and for. The while loop is followed by a command or an expression enclosed in square brackets, a do keyword, a block of statements, and terminated with the done keyword. As long as the expression is true, the body of statements between do and done will be executed.

The until loop is just like the while loop, except the body of the loop will be executed as long as the expression is false.

The for loop used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends. The for loop is followed by a variable name, the in keyword, and a list of words then a block of statements, and terminates with the done keyword.

The loop control commands are break and continue.

Syntax

While Loop

```
while command
do
      block of statements
done
```

For Loop

```
for var in word1……word
do
      Statement(s) to be executed for every word
done
```

Until Loop

```
Until command
do
        Statement(s) to be executed until command is true
done
```

Example

```
for var in 0 1 2 3 4 5 6 7 8 9
do
       echo $var
done
```

**Task 4** Write a program to enter the numbers till the user wants and at the end it should display the maximum and minimum number entered.