Lab Manual 01
Introduction and Shell Scripting

1 Operating Systems

1. Definition

   An Operating System, or OS, is low-level software that enables a user and higher-level application software to interact with a computer's hardware and the data and other programs stored on the computer.

2. Basic Functions

   (a) Program execution.
   (b) Interrupts.
   (c) Memory management.
   (d) Multitasking

3. Goals

   (a) Execute user programs and make solving user problems easier.
   (b) Make the computer system convenient to use.

2 Shell

A shell is simply a program which is used to start other programs. It takes the commands from the keyboard and gives them to the operating system to perform the particular task.
There are many different shells, but all derive several of their features from the Bourne shell, a standard shell developed at Bell Labs for early versions of Unix. Linux uses an enhanced version of the Bourne shell called bash or the "Bourne-again" shell. The bash shell is the default shell on most Linux distributions, and **/bin/sh** is normally a link to bash on a Linux system.

The Shell Window

After logging in, open a shell window (often referred to as a terminal). The easiest way to do so from a GUI like Ubuntu's Unity is to open a terminal application, which starts a shell inside a new window. The window display a prompt at the top that usually ends with a dollar sign $. If # is the last character it means you are running the command as root user.

3 Basic Commands

In this section we will have an insight of some basic commands. Different commands take multiple arguments and options (where option starts with a dash - sign).

1. **echo**
   The echo command prints its arguments to the standard output.

   ```
   $ echo Hello World
   ```

   Output Hello World on your terminal screen.

2. **ls**
   The ls command lists the contents of a directory. The default is the current directory. Use ls -l for a detailed (long) listing where -l is an option. Output includes the owner of the file (column 3), the group (column 4), the file size (column 5), and the modification date/time (between column 5 and the filename).

3. **cp**
   cp copies files. For example, to copy file1 to file2, enter this:

   ```
   $ cp file1 file2
   ```

   where file1 and file2 should be in current working directory.

   ```
   $cp hSourcei hdestinationi
   ```

   where source and destination are ful path from the root directory. To copy a num-ber of files to a directory (folder) named dir, try this instead:

   ```
   $ cp file1 ... fileN dir
   ```

4. **cat**
   It simply outputs the contents of one or more files. The general syntax of the cat command is as follows:

   ```
   $ cat file1 file2 ...
   ```

   where file1 and file2 should be in current working directory or otherwise write down full path starting from root. When you run this command, cat prints the contents of file1, file2, and any other files that you specify (denoted by ...), and then exits. The command is called cat because it performs concatenation when it prints the contents of more than one file.

5. **mv**
   It renames a file. For example, to rename file1 to file2, enter this:

```
$ mv file1 file2
```

You can also use mv to move a number of files to a different directory:

```
$ mv file1 ... fileN dir
```

6. **rm**
   To delete (remove) a file, use rm. After a file is removed, it’s gone from the system and generally cannot be undeleted.

   ```
   $ rm file
   ```

7. **touch**
   The touch command creates a file. If the file already exists, touch does not change it, but it does update the file’s modification time stamp.

   ```
   $ touch file
   ```

8. **pwd**
   The pwd (print working directory) program simply outputs the name of the current working directory.
9. **date**
   Displays current time and date
10. **clear**
    Clears the terminal screen.
11. **exit**
    Exit the Shell
12. **head and tail**
    To quickly view a portion of a file or stream of data, use the head and tail commands. For example:

    ```
    head /etc/passwd
    ```

    shows the first 10 lines of the password file.

    ```
    tail /etc/passwd
    ```

    shows the last 10 lines. To change the number of lines to display, use the -n option, where n is the number of lines you want to see.
13. **sort**
    The sort command quickly puts the lines of a text file in alphanumeric order. The -r option reverses the order of the sort.

14. **grep**
    The grep command prints the lines from a file or input stream that match an expression. For example, to print the lines in the /etc/passwd file that contain the text root, enter this:

    ```
    $ grep root /etc/passwd
    ```

    The grep command is extraordinarily handy when operating on multiple files at once because it prints the filename in addition to the matching line. For example, if you want to check every file in /etc that contains the word root, you could use this command:

    ```
    $ grep root /etc/*
    ```

    Two of the most important grep options are -i (for case-insensitive matches) and -v (which inverts the search, that is, prints all lines that don't match).

15. **mkdir**
    The mkdir command creates a new directory dir:

    ```
    $ mkdir dir
    ```

16. **rmdir**
    The rmdir command removes the directory dir:

    ```
    $ rmdir dir
    ```

    If dir isn't empty, this command fails.
    rm -rf dir is used to delete a directory and its contents where -r option specifies recursive delete to repeatedly delete everything inside dir, and -f forces the delete operation.

17. **cd**
    The current working directory is the directory that a process (such as shell) is cur-rently in. The cd command changes the shell's current working directory:

    ```
    $ cd dir
    ```

    If you omit dir, the shell returns to your home directory, the directory you started in when you first logged in.

18. **less**
    The less command comes in handy when a file is really big or when a command's output is long and scrolls off the top of the screen. To page through a big file like /usr/share/dict/words, use the command

```
$ less /usr/share/dict/words
```

When running less, you'll see the contents of the file one screenful at a time. Press the spacebar to go forward in the file and the b key to skip back one screenful. To quit, type q.

19. **man**
    Linux systems come with a wealth of documentation. For basic commands, the manual pages (or man pages) will tell you what you need to know. For example, to see the manual page for the ls command, run man as follows:

```
$ man ls
```

4 OUTPUT REDIRECTION

To send the output of command to a file instead of the terminal, use the > redirection character:

```
$ command > file
```

The shell creates file if it does not already exist. If file exists, the shell erases the original file first. You can append the output to the file instead of overwriting it with the » redirection syntax:

```
$ command » file
```

5 NAVIGATING DIRECTORIES

Unix has a directory hierarchy that starts at /, sometimes called the root directory. The directory separator is the slash (/).

PATHNAME

When you refer to a file or directory, you specify a path or pathname. There are two different ways of writing files path:

1. Absolute Path
   When a path starts with root, it is a full or absolute path.

```
/home/cslab/Desktop/Lab01
```

2. Relative Path
   A path starts at current directory is called a relative path.

```
./Lab01
```

DIRECTORY REFERENCES

1. Home Directory (~)
2. Root Directory(/)
3. Current Working Directory(.)
   One dot (.) refers to the current directory; for example, if you're in /usr/lib, the path dot(.) is still /usr/lib, and ./X11 is /usr/lib/X11.
4. Parent Directory(..)
   A path component identified by two dots (..) specifies the parent of a directory. For example, if you're working in /usr/lib, the path .. would refer to /usr. Similarly, ../bin would refer to /usr/bin.

6 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
#First shell script
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
```

7 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 .

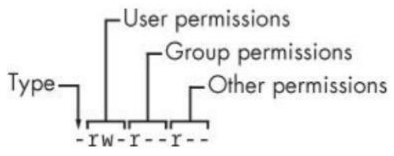

Figure 1: File Mode

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
```

8 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.

9 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=Hello
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 com-mands 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:

   ```
   # !/bin/bash
   echo First 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.

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.