# LAB MANUAL

## Course: CSC322-Operating Systems


Department of Computer Science

### Java Learning Procedure

1) Stage J (Journey inside-out the concept)
2) Stage $a_1$ (Apply the learned)
3) Stage V (Verify the accuracy)
4) Stage $a_2$ (Assess your work)

COMSATS University Islamabad (CUI)

# Table of Contents

# LAB # 01

## Statement Purpose:

This lab will introduce the Linux Operating System to you. You will learn the how to create VM using Virtual-Box, Installing Ubuntu on VM and the basic syntax of Linux Commands

## Activity Outcomes:

This lab teaches you the following topics:

- Introduction of Linux OS, Linux Distros and Virtual Machines
- Creating VM in Virtual-Box
- Installing Ubuntu on VM
- Writing basic commands in CLI

## Instructor Note:

As pre-lab activity, read Chapter 1 to 6 from the book “The Linux Command Line”, William E. Shotts, Jr.

# 1) Stage J (Journey)

# 1. Linux

**Linux** is a generic term referring to Unix-like computer operating systems based on the Linux kernel. Their development is one of the most prominent examples of free and open source software collaboration; typically all the underlying source code can be used, freely modified, and redistributed by anyone. Many quantitative studies of free / open source software focus on topics including market share and reliability, with numerous studies specifically examining Linux. The Linux market is growing rapidly.

## 1.1 History

All modern operating systems have their roots in 1969 when Dennis Ritchie and Ken Thompson developed the C language and the Unix operating system at AT&T Bell Labs. They shared their source code (yes, there was open source back in the Seventies) with the rest of the world, including the hippies in Berkeley California. By 1975, when AT&T started selling Unix commercially, about half of the source code was written by others. The hippies were not happy that a commercial company sold software that they had written; the resulting (legal) battle ended in there being two versions of Unix: the official AT&T Unix, and the free BSD Unix.

In the Eighties many companies started developing their own Unix: IBM created AIX, Sun SunOS (later Solaris), HP HP-UX and about a dozen other companies did the same. The result was a mess of Unix dialects and a dozen different ways to do the same thing. And here is the first real root of Linux, when Richard Stallman aimed to end this era of Unix separation and everybody re-inventing the wheel by starting the GNU project (GNU is Not Unix). His goal was to make an operating system that was freely available to everyone, and where everyone could work together (like in the Seventies). Many of the command line tools that you use today on Linux are GNU tools.

The Nineties started with Linus Torvalds, a Swedish speaking Finnish student, buying a 386 computer and writing a brand new POSIX compliant kernel. He put the source code online, thinking it would never support anything but 386 hardware. Many people embraced the combination of this kernel with the GNU tools, and the rest, as they say, is history. Linux kernel version 4.0 was released in April 2015. Its source code grew by several hundred thousand lines (compared to version 3.19 from February 2015) thanks to contributions of thousands of developers paid by hundreds of commercial companies including Red Hat, Intel, Samsung, Broadcom, Texas Instruments, IBM, Novell, Qualcomm, Nokia, Oracle, Google, AMD and even Microsoft (and many more).

## 1.2 Popularity

Today more than 97 percent of the world's supercomputers (including the complete top 10), more than 80 percent of all smartphones, many millions of desktop computers, around 70 percent of all web servers, a large chunk of tablet computers, and several appliances (DVD players, washing machines, DSL modems, routers, self-driving cars, space station laptops...) run Linux. Linux is by far the most commonly used operating system in the world.

## 1.3 Linux Distribution

**The Linux kernel:** This is a small, but essential part of an operating system. The kernel is responsible for interfacing with a device's hardware, providing services to the rest of the system, and performing tasks such as managing the device's CPU and memory. The Linux kernel, like any kernel, can only function as part of a wider operating system. It's impossible to have an operating system that consists solely of a Linux kernel. Since Android is a complete operating system, we can immediately rule out classifying Android as a Linux kernel.

**A Linux distribution or distro:** This is an operating system that contains the Linux kernel and additional software such as utilities, libraries and a GUI, plus pre-installed applications such as web browsers, text editors, and music players. Even if this additional software was designed specifically to run on the Linux kernel, it's not part of the Linux kernel. When discussing operating systems that use the Linux kernel, the terms 'distribution,' 'distro' and 'operating system' are interchangeable. Since anyone can take the Linux kernel, add their own software, and create a complete operating system, there are countless Linux distros currently available.

A Linux distribution -- often shortened to "Linux distro" -- is a version of the open source Linux operating system that is packaged with other components, such as an installation programs, management tools and additional software such as the KVM hypervisor. Linux distributions, which are based on the Linux kernel, are often easier for users to deploy than the traditional open source version of Linux. This is because most distributions eliminate the need for users to manually compile a complete Linux operating system from source code, and because they are often supported by a specific vendor.
Hundreds of Linux distributions are available today, and each targets specific users or systems such as desktops, servers, mobile devices or embedded devices. Most distributions come ready to use, while others are packaged as source code that a user must compile during installation. A list of most popular Linux distros is given below

1. Ubuntu
2. Slackware
3. SuSE
4. Debain
5. RedHat
6. Fedora
7. Turbo Linux

In this course, we will use the Ubuntu distro. Ubuntu is a popular and to use graphical Linux distro. It was developed and released by Canonical Ltd. in 2004. It is freely available and can be downloaded from http://www.ubuntu.com/download/desktop.

## 2. Installing Ubuntu

Before discussing the options available to install Ubuntu, we discuss the basic system requirement. It is recommended to Ubuntu should be installed on a system that has a 2 GHz dual core processor with 2GB RAM and 25GB of free hard disk space.

There are many ways to use Ubuntu. It can be installed on a system as a stand-alone OS. Similarly, it can be installed as multi-boot system where it is installed on a system that already has any other OS like windows. Further, it can also be used without installing from a bootable USB. However, in this course we will run the Ubuntu on virtual machine. To create virtual machine we will use Oracle VM Virtual-box. In the following, first we give an overview of Virtual-Box and then discuss the installation process of Ubuntu on VM.

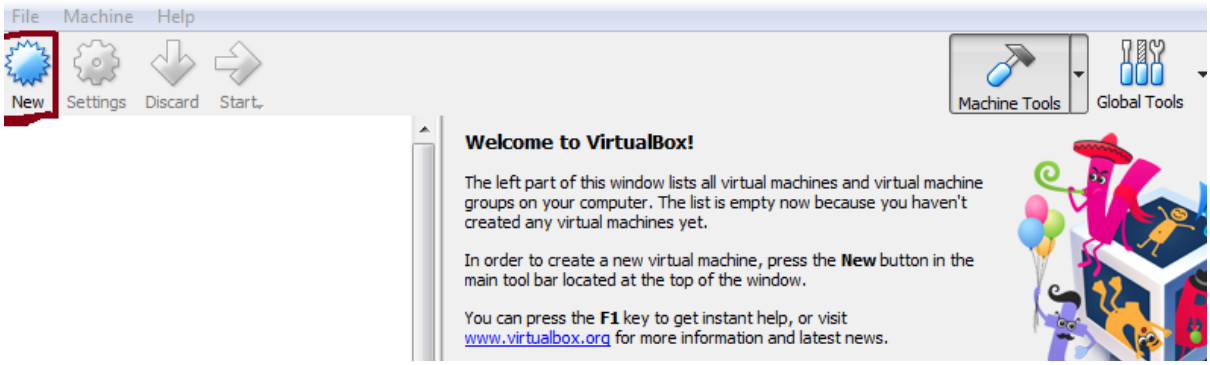
### 2.1 Oracle VM Virtual-Box

Virtual-Box is a cross-platform virtualization application. What does that mean? For one thing, it installs on your existing Intel or AMD-based computers, whether they are running Windows, Mac, Linux or Solaris operating systems. Secondly, it extends the capabilities of your existing computer so that it can run multiple operating systems (inside multiple virtual machines) at the same time. So, for example, you can run Windows and Linux on your Mac, run Windows Server 2008 on your Linux server, run Linux on your Windows PC, and so on, all alongside your existing applications. You can install and run as many virtual machines as you like -- the only practical limits are disk space and memory. Virtual-Box can be downloaded from https:// www.virtualbox.org/ wiki/ Downloads for free.

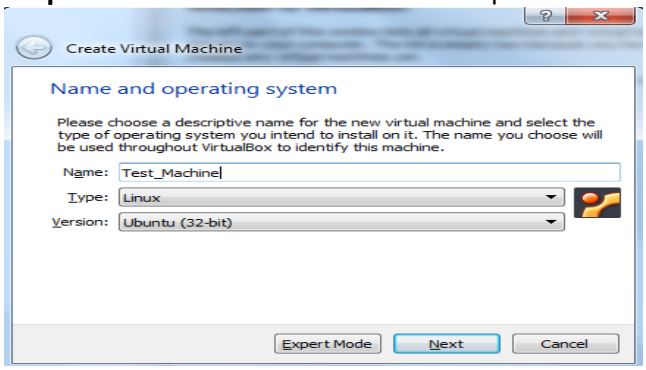
### 2.2 Installing Ubuntu on VM

Before installing the Ubuntu, we need to ensure that Virtual-Box has been installed on the machine and a compatible version of Ubuntu has been downloaded. To install Ubuntu on VM we need to create a virtual machine on Virtual-Box. The process to create a VM on Virtual-Box is given below.

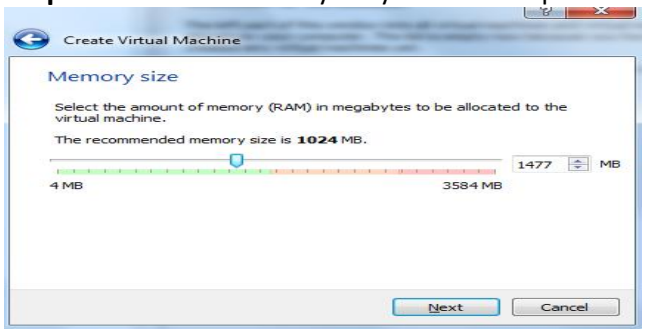
**Step 1:** Start the Virtual-Box and click the new button to start creating a new VM



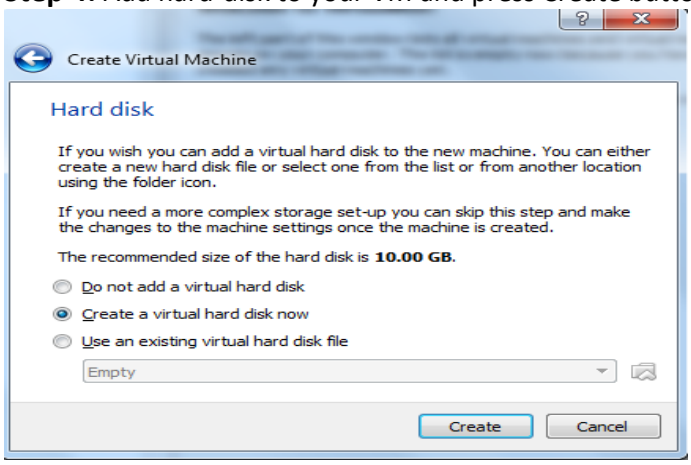

**Step 2:** Enter the basic information and press Next



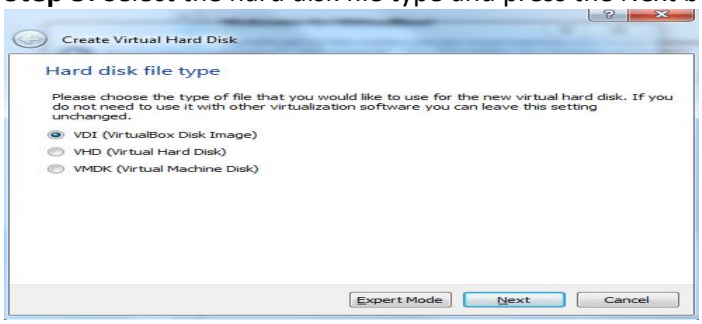

**Step 3:** Select the memory for your VM and press Next



**Step 4:** Add hard-disk to your VM and press Create button

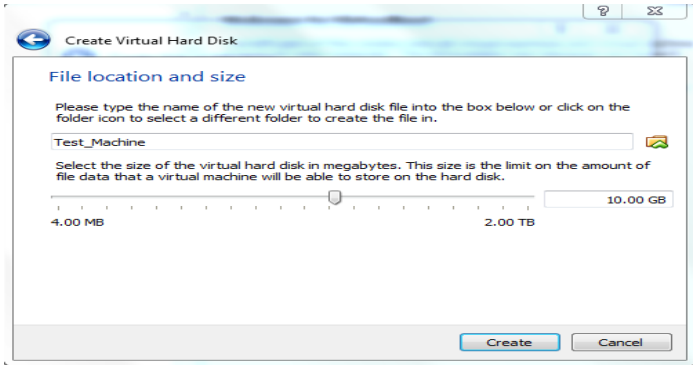



**Step 5:** Select the hard disk file type and press the Next button

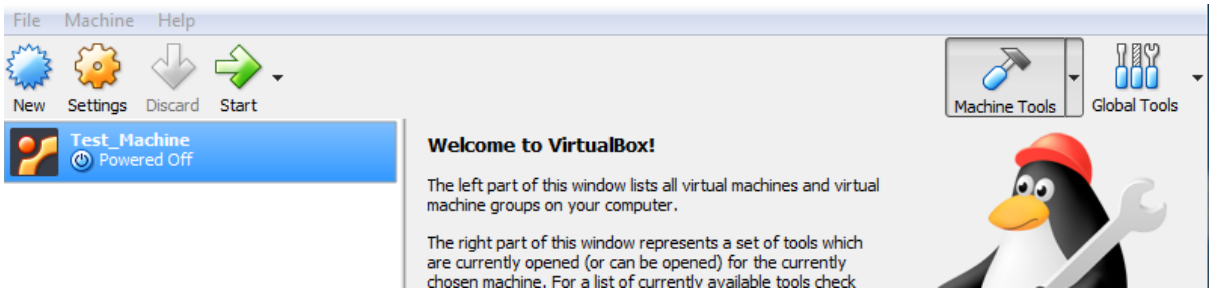



**Step 6:** Select the storage type on Physical hard disk and press Next


**Step 7:** Select the location and size on physical hard disk to store the VM and press the Create button

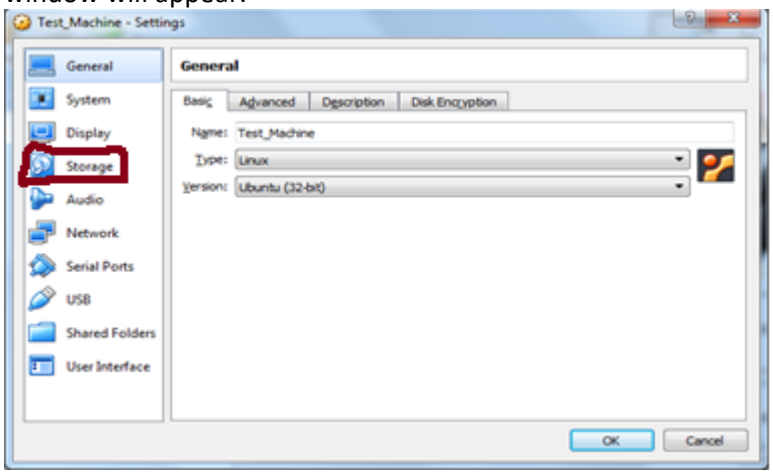


It completes the creation process of VM on Virtual-Box. The newly created VM can be seen on main screen

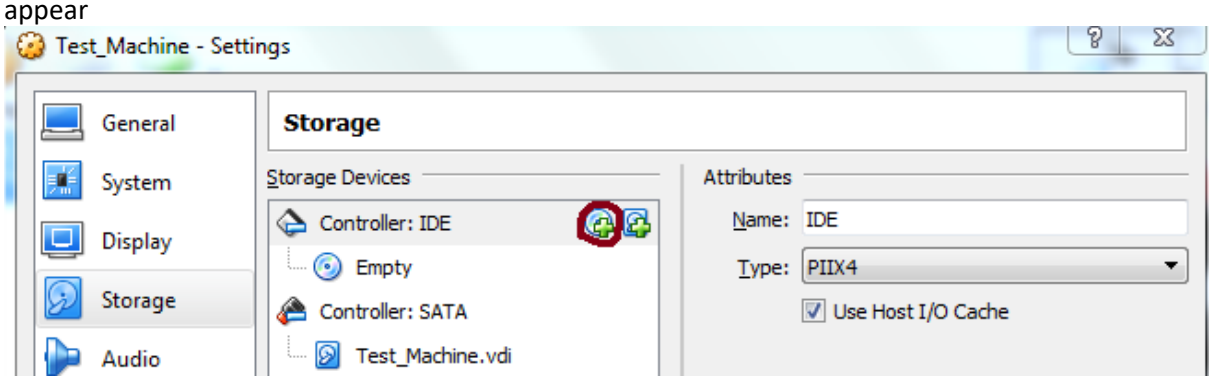



To install Ubuntu on the newly created VM, we need to attach the Ubuntu installation file with it. It can be done as given below.

**Step 1:** Select the VM on main screen of Virtual-Box and click the settings button. The following window will appear.

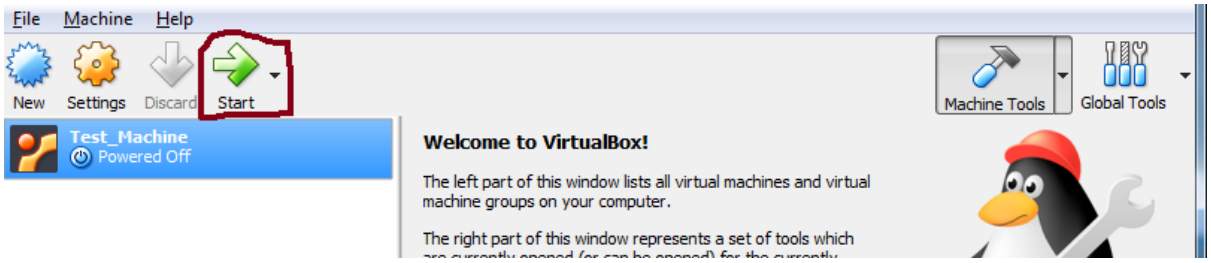



**Step 2:** Select the storage button as highlighted in the above figure. The following window will appear

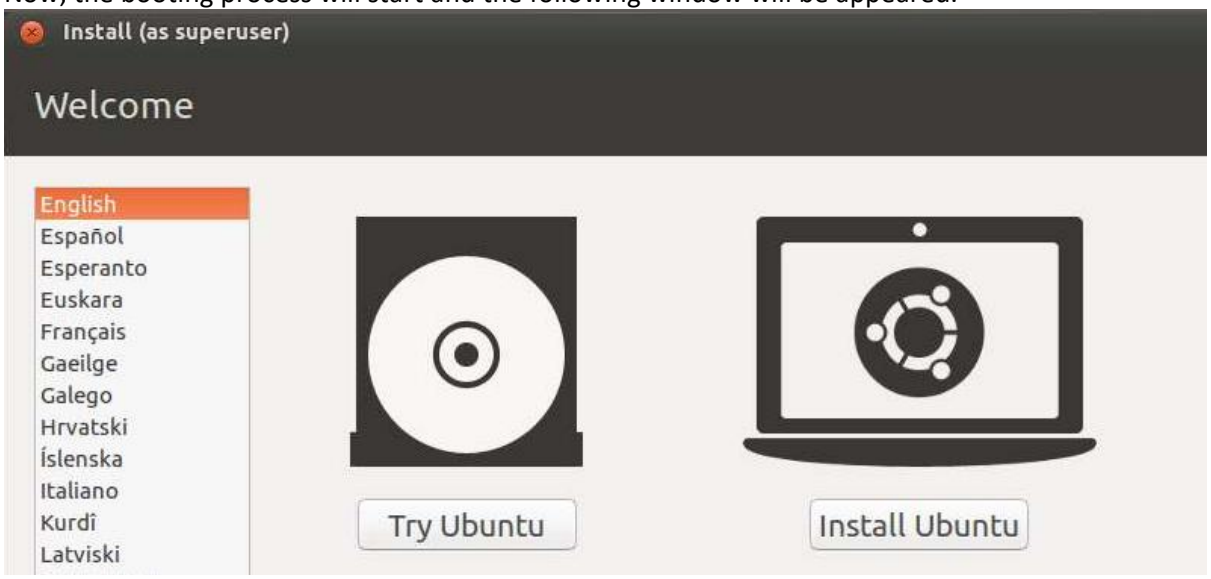



**Step 3:** Select + button (as highlighted in above figure) and browse and select the Ubuntu installation file downloaded earlier. Now, press the OK button to complete the process of attaching Ubuntu with VM.

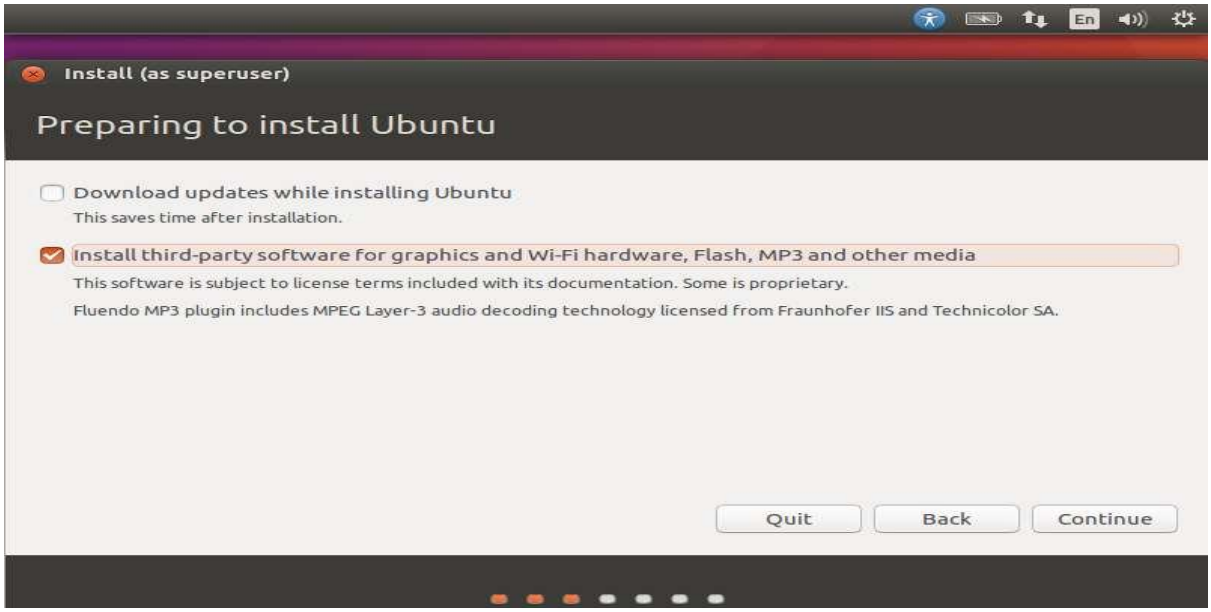
Once the VM has been created and Ubuntu Installation file is attached with it, we can start the installation process. To do this, select the VM on main window of Virtual-Box and click the start button as given below.




Now, the booting process will start and the following window will be appeared.




Here, we can try the Ubuntu without installing. To install Ubuntu, click Install Ubuntu button. The next screen gives you 2 options. One is to download updates in the background while installing and the other is to install 3rd party software. Check the option to install 3 party software. Then click the Continue button.

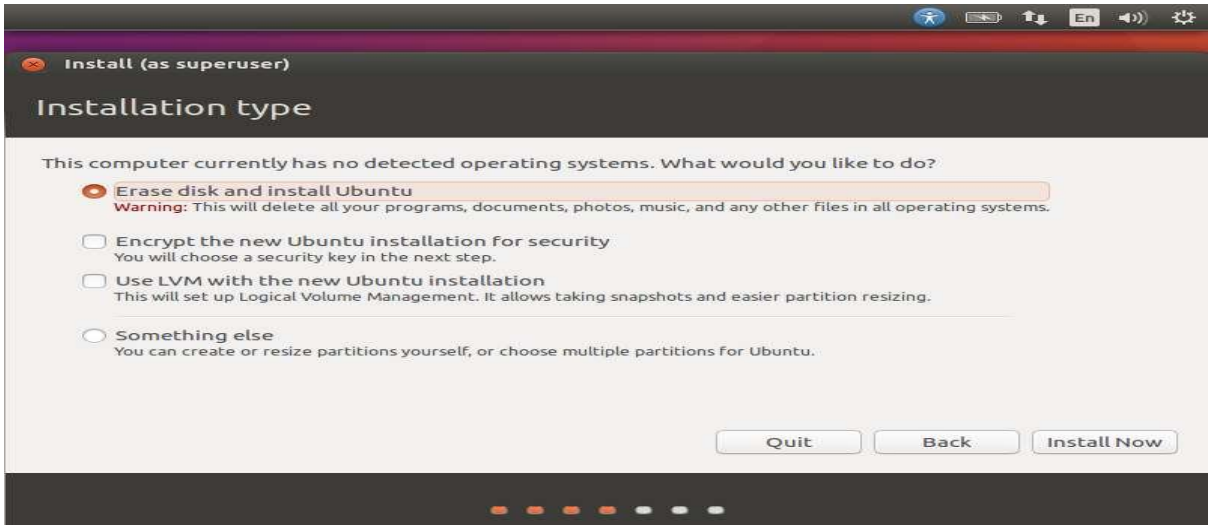



In the next screen, the following options are presented -

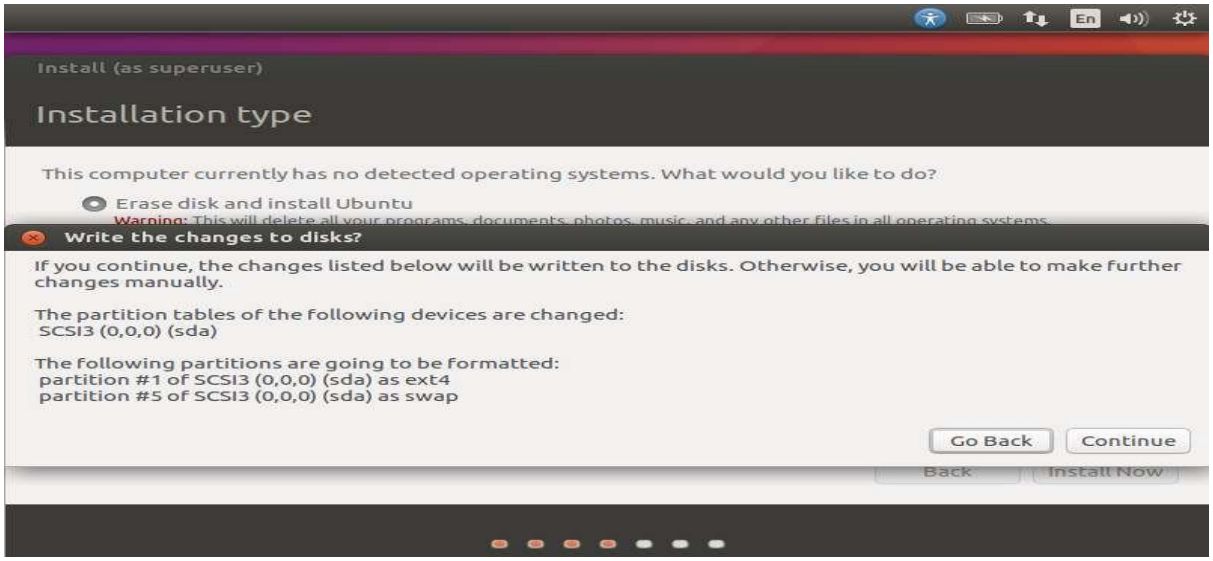
- The disk is erased and the installation is carried out. If there was another operating system already on the disk, then Ubuntu would detect it and give the user the option to install the operating system side by side.

- There is an option to encrypt the installation. This is so that if anybody else were to steal the data, they would not be able to decrypt the data.
- Finally, Linux offers a facility called LVM, which can be used for taking snapshots of the disk.

For the moment, to make the installation simple, let's keep the options unchecked and proceed with the installation by clicking the Install Now button.

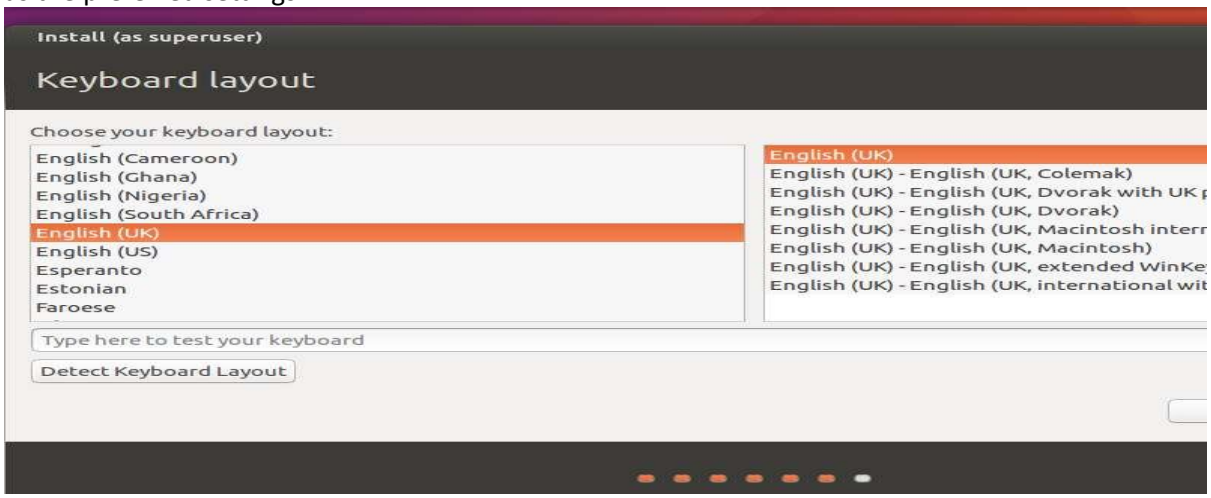



In the following screen, we will be prompted if we want to erase the disk. Click the Continue button to proceed.

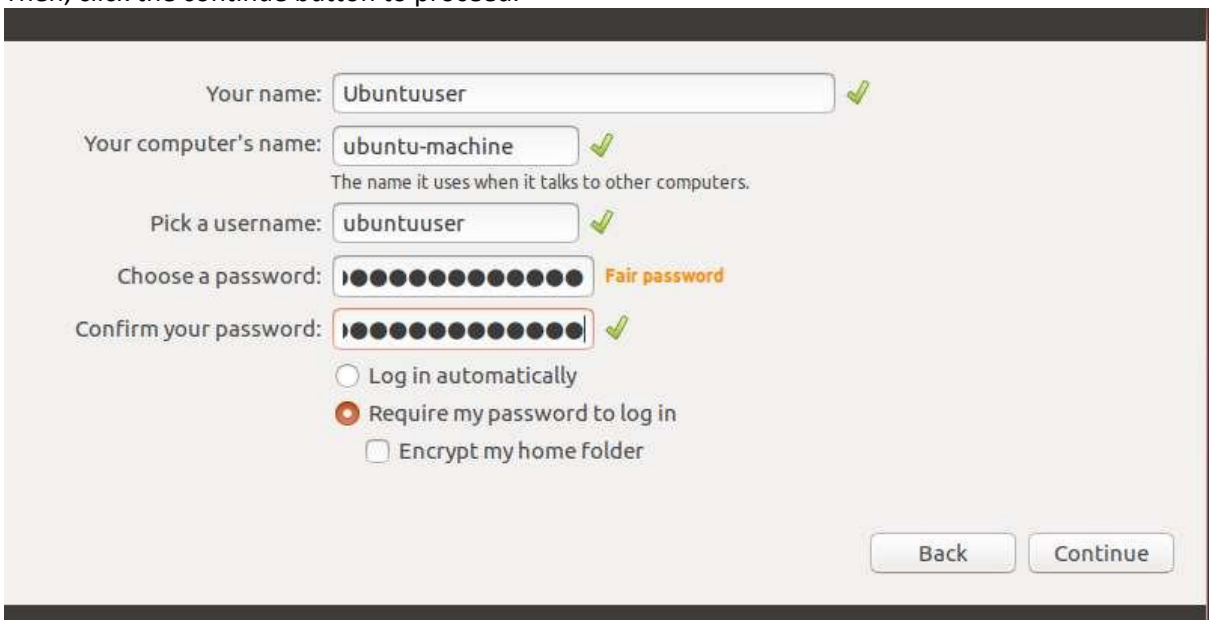



In next screen, we will be asked to confirm our location. Click the Continue button to proceed.

Now, we will be asked to confirm the language and the keyboard settings. Let us select English (UK) as the preferred settings.

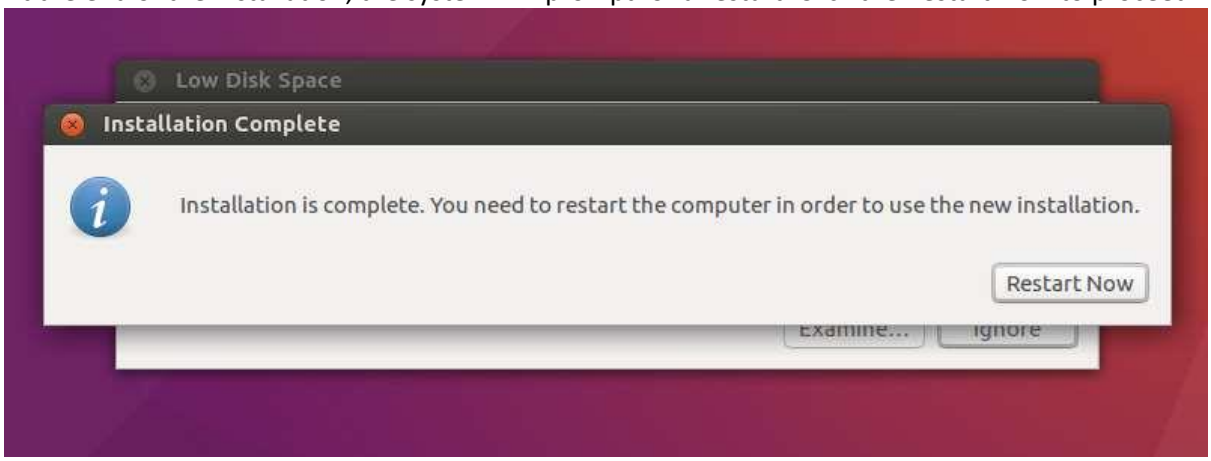



In the following screen, we will need to enter the user name, computer name and password which will be used to log into the system. Fill the necessary details as shown in the following screenshot. Then, click the continue button to proceed.

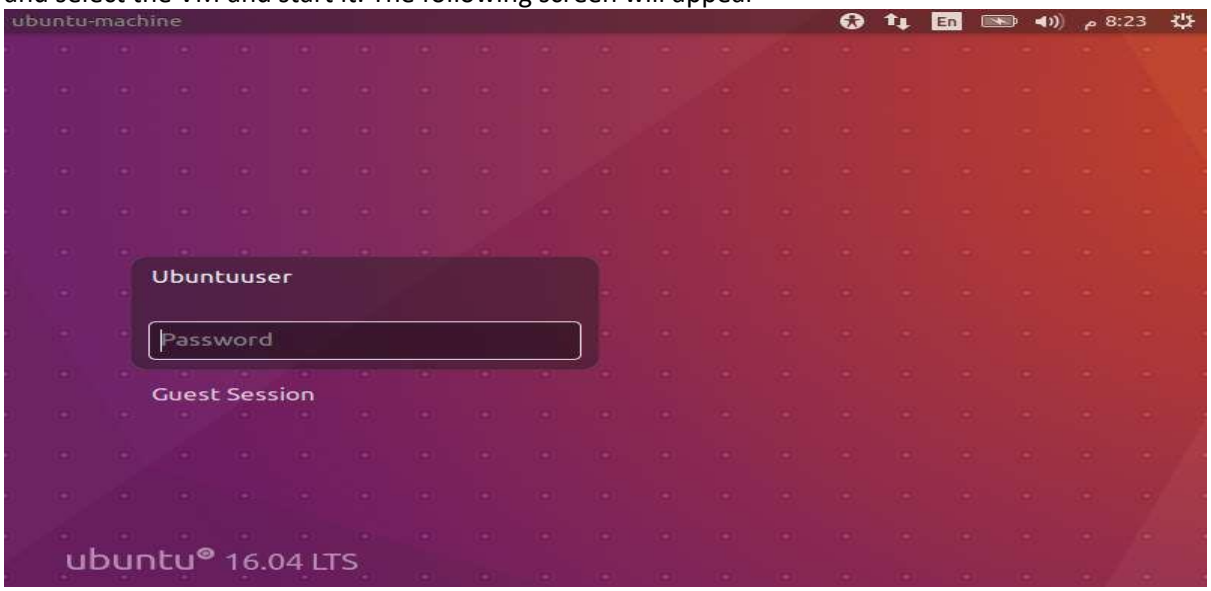



The system will now proceed with the installation and we will see the progress of the installation as shown in the following screenshot.

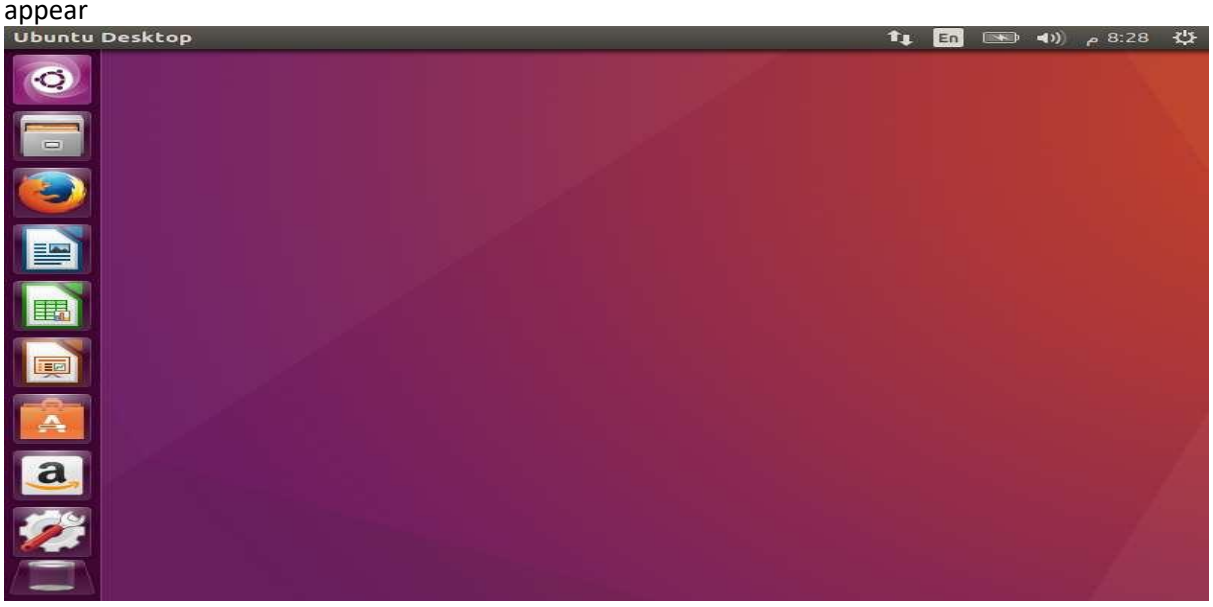
At the end of the installation, the system will prompt for a restart. Click the Restart Now to proceed.




When the installation is completed, we can login into the system. To do this, start the Virtual-Box and select the VM and start it. The following screen will appear




By entering username and password, we can login to the system and the following window will appear

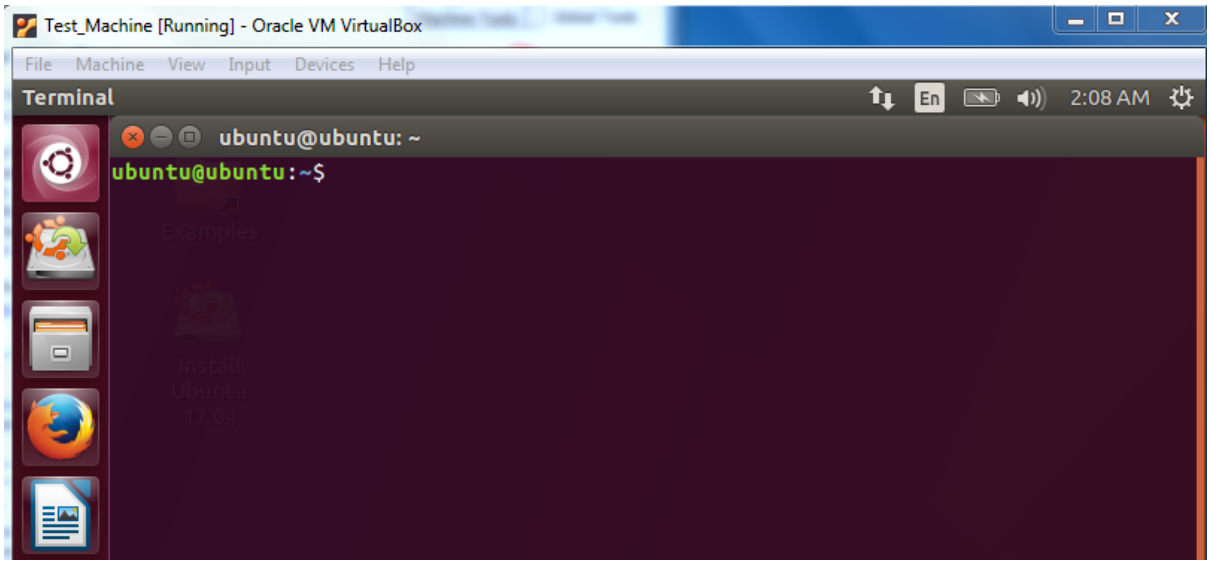


# 3. Writing Linux Commands

## 3.1 Command Line Interface

The Command Line Interface (CLI), is a non-graphical, text-based interface to the computer system, where the user types in a command and the computer then successfully executes it. The Terminal is the platform or the IDE that provides the command line interface (CLI) environment to the user. The CLI terminal accepts the commands that the user types and passes to a shell. The shell then receives and interprets what the user has typed into the instructions that can be executed by the OS (Operating System). If the output is produced by the specific command, then this text is displayed in the terminal. If any of the problems with the commands are found, then some error message is displayed.

We can open the terminal by typing Ctrl + Alt + T short-key or by right-clicking the mouse and selecting the Open New Terminal option. The terminal window looks like given below.




## 3.2 Basic syntax of Linux Commands

A command is an instruction given by a user telling a computer to do something, such a run a single program or a group of linked programs. Commands are generally issued by typing them in at the command line (i.e., the all-text display mode) and then pressing the ENTER key, which passes them to the shell.

A shell is a program that reads commands that are typed on a keyboard and then executes (i.e., runs) them. Shells are the most basic method for a user to interact with the system.

### Options and Arguments

This brings us to a very important point about how most commands work. Commands are often followed by one or more *options* that modify their behavior, and further, by one or more *arguments*, the items upon which the command acts. So most commands look kind of like this:

```
command -options arguments
```

Most commands use options consisting of a single character preceded by a dash, for example, "-l", but many commands, including those from the GNU Project, also support long options, consisting of a word preceded by two dashes. Also, many commands allow multiple short options to be strung together.

**Command History**

Most Linux distributions remember the last 500 commands by default. Press the down-arrow key and the previous command disappears.

## 3.3 Some Basic Linux Commands

1. **Date** Command: This command is used to display the current data and time.

   **Syntax:**

   $date

   $date +%ch

   **Options:**

   a = Abbreviated weekday.
   A = Full weekday.
   b = Abbreviated month.
   B = Full month.
   c = Current day and time.
   C = Display the century as a decimal number.
   d = Day of the month.
   D = Day in „mm/dd/yy" format
   h = Abbrevated month day.
   H = Display the hour.
   L = Day of the year.
   m = Month of the year.
   M = Minute.
   P = Display AM or PM
   S = Seconds
   T = HH:MM:SS format
   u = Week of the year.
   y = Display the year in 2 digit.
   Y = Display the full year.
   Z = Time zone

   To change the format:

   **Syntax:**

   $date „+%H-%M-%S"

2. Calendar Command: This command is used to display the calendar of the year or the particular month of calendar year.

   **Syntax :**

   a.$cal <year>
   b.$cal <month> <year>

   Here the first syntax gives the entire calendar for given year & the second Syntax gives the calendar of reserved month of that year.

3. To see the current amount of free space on your disk drives, enter **df**:

4. Likewise, to display the amount of free memory, enter the **free** command.

5. We can end a terminal session by either closing the terminal emulator window, or by entering the **exit** command at the shell prompt

6. **'who'** Command: It is used to display who are the users connected to our computer currently.
   **Syntax:**
   $who – option‟s
   **Options : -**
   H–Display the output with headers
   b–Display the last booting date or time or when the system was lastly rebooted

7. **'who am i'** Command: Display the details of the current working directory.
   **Syntax:**
   $who am i

8. CLEAR' Command**:** It is used to clear the screen.
   **Syntax:**
   $clear

9. MAN' Command: It help us to know about the particular command and its options & working. It is like„help‟ command in windows .
   **Syntax:**
   $man <command name>

## 2) Stage a1 (apply)

# Lab Activities:

## Activity 1:

In this activity, you are required to perform tasks given below:

1. Display the current date
2. Display the calendar for the current year
3. Display the calendar of 2012
4. Display the calendar of Feb 2015

## Solution:

1. date
2. cal 2019
3. cal 2012
4. cal 2 2012

### Activity 2:

Perform the following tasks using Linux CLI commands

1. Display the amount of free storage on your machine
2. Display the amount of free memory on your machine
3. Display the user name of the current user
4. Open the man of date free command

### Solution:

1. df
2. free
3. who
4. man date

## 3) Stage V (verify)

## Home Activities:

1. In GUI open the Libre Office writer tool create a document that contains information about your favorite place. Try the following short- keys while formatting the document

| Keyboard Shortcuts | Functions |
|---|---|
| Ctrl + C | Copy the Selected text or Object |
| Ctrl + X | Cut the selected text or object |
| Ctrl + V | Paste the Copied text or Object |
| Ctrl + A | Select all text or All files and folder in a Parent folder |
| Ctrl + B | Make the Selected text as **BOLD** |
| Ctrl + I | Mark the selected text as *italic* |
| Ctrl + U | Mark the Selected text Underline |
| Ctrl + N | Open a New document or Window |
| Ctrl + S | Save the Current Document |
| Ctrl + O | Open another Document |
| Ctrl + P | Print the Document (Print option) |
| Ctrl + Z | Undo the Last Change you made |
| Ctrl + Shift + Z | Redo a change that you just undeed |

## 4) Stage $a_2$ (assess)

### Lab Assignment and Viva voce

# LAB # 02

## Statement of Purpose:

This lab will introduce the Directory and File related commands to you. We will start with the some basic but important commands used to navigate through the Linux file system. Then we will discuss Directory and File related commands. Finally, we will introduce the I/O redirection in Linux

## Activity Outcomes:

This lab teaches you the following topics:

- Navigation through Linux file system using CLI
- Working with directories in Linux using CLI
- Handling Files in Linux using CLI
- Using I/O redirection in Linux.

## Instructor Note:

As pre-lab activity, read Chapter 1 to 6 from the book “The Linux Command Line”, William E. Shotts, Jr.

# 1) Stage J (Journey)

## Introduction

Linux organizes its files in a *hierarchical directory structure*. The first directory in the file system is called the *root directory*. The root directory contains files and subdirectories, which contain more files and subdirectories and so on and so on. If we map out the files and directories in Linux, it would look like an upside-down tree. At the top is the root directory, which is represented by a single slash (/). Below that is a set of common directories in the Linux system, such as bin, dev, home, lib , and tmp , to name a few. Each of those directories, as well as directories added to the root, can contain subdirectories.

### 1. Navigation

The first thing we need to learn is how to navigate the file system on our Linux system. In this section we will introduce the commands used for navigation in Linux system.

#### 1.1 Print Working Directory

The directory we are standing in is called the current working directory. To display the current working directory, we use the pwd (print working directory) command. When we first log in to our system our current working directory is set to our home directory. Suppose, a user is created with name ***me*** on machine Ubuntu; we display its current working directory as given below

```
[me@ubuntu  ~] $ pwd
/home/me
```

#### 1.2 Listing The Contents Of A Directory

To list the files and directories in the current working directory, we use the **ls** command. Suppose, a user ***me*** is in its home directory; to display the contents of current working directory can be displayed as follows:

```
[me@ubuntu  ~] $ ls
Desktop  Documents  Music  Pictures  Pulic  Templates  Videso
```

Besides the current working directory, we can specify the directory to list, like so:

```
[me@ubuntu  ~] $ ls  /usr
bin   games   kerberos  libexec   sbin    src
etc   include  lib          local      share  tmp
```

Or even specify multiple directories. In this example we will list both the user's home directory (symbolized by the "~" character) and the /usr directory:

```
[me@ubuntu ~] $ ls ~ /usr

/home/me:
Desktop Documents Music Pictures Pulic Templates Videso
/usr:
bin games kerberos libexec sbin src
etc include lib local share tmp
```

The following options can also be used with ls command

| Options | Long-options | Description |
|---|---|---|
| -a | - - all | List all files, even those with names |
| -d | - - directory | Ordinarily, if a directory is specified, ls will list the contents of the directory, not the directory itself. Use this option in conjunction with the -l option to see details about the directory rather than its contents. |
| -h | - - human-readable | In long format listings, display file sizes in human readable format rather than in bytes. |
| -r | - - reeverse | Display the results in reverse order. Normally,ls displays its results in ascending alphabetical order. |
| -S | - | Sort results by file size. |
| -t | | Sort by modification time |
| -l | | Display results in long format. |

### 1.3 Changing the Current Working Directory

To change your working directory, we use the **cd** command. To do this, type cd followed by the pathname of the desired working directory. A pathname is the route we take along the branches of the tree to get to the directory we want. Pathnames can be specified in one of two different ways; as **absolute pathnames** or as **relative pathnames**. An absolute pathname begins with the root directory and follows the tree branch by branch until the path to the desired directory or file is completed. On the other hand a relative pathname starts from the working directory.

Suppose, a user ***me*** is in its home directory and we want to go into the Desktop directory, then it can be done as follows:

```
[me@ubuntu ~] $ cd /home/me/Desktop (absolute path)
or
[me@ubuntu ~] $ cd Desktop (relative path)
```

The ".." operator is used to go to the parent directory of the current working directory. In continuation of the above example, suppose we are in the Desktop directory and we have to go to the Documents directory. To this task, first we will go the parent directory of Desktop (i.e. me, home directory of the user) that contains the Documents directory then we will go into the Documents directory as given below.

```
[me@ubuntu Desktop ] $ cd /home/me/Documents (absolute path)
or
[me@ubuntu Desktop ] $ cd ../Documents (relative path)
```

## 2. Working With Directories

In this Section, we introduce the most commonly used commands related to Directories.

### 2.1. Creating a Directory

In Linux, ***mkdir*** command is used to create a directory. We pass the directory name as the argument to the mkdir command. Suppose, the user me is in its home directory and we want to create a new directory named **mydir** in the Desktop directory. To do this, first we will change the current directory to Desktop and then we will create the new directory. It is shown below:

```
[me@ubuntu ~] $   cd  /home/me/Desktop
[me@ubuntu  Desktop ] $  mkdir  mydir
```

Multiple directories can also be created using single mkdir command as given below:

```
[me@ubuntu  Desktop ] $  mkdir  mydir  mydir1  mydir2
```

### 2.2. Copying Files and Directories

**cp** command is used to copy files and directories. The syntax to use cp command is given below:

```
cp  item1  item2
```

Here, item1 and item2 may be files or directories. Similarly, multiple files can also be copied using single cp command.

```
cp  item ....  directory
```

The common options that can be used with cp commands are:

| Option | Long Option | Explanation |
|---|---|---|
| -a | --archive | Copy the files and directories and all of their attributes |
| -i | --interactive | Before overwriting an existing file, prompt the user for confirmation |
| -r | --recursive | Recursively copy directories and their contents |
| -u | -update | When copying files from one directory to another, only copy files that either don't exist, or are newer |

### 2.3. Moving and Renaming Files and Directories

**mv** command is used to move files and directories. This command can also be used to rename files and folder. To rename files and directories, we just perform the move operation with old name and new name. As a result, the files or directory is created again with a new name. The syntax to use mv command is given below:

```
mv  item1  item2
```

Similarly, multiple files can be moved to a directory as given below

```
mv   items .....  directory
```

Common options, used with mv command are:

| Option | Long Option | Explanation |
|---|---|---|
| -i | --interactive | Before overwriting an existing file, prompt the user for confirmation |
| -u | -update | When moving files from one directory to another, only copy files that either don't exist, or are newer |

### 2.4. Removing and Files and Directories

To remove or delete a files and directories, **rm** command is used. Empty directories can also be deleted using rmdir command but rm can be used for both empty and non-empty directories as well as for files. The syntax is given below:

```
rm   item1 item2 .....
```

The common options, used with rm command are:

| Option | Long Option | Explanation |
|---|---|---|
| -i | --interactive | Before deleting an existing file, prompt the user for confirmation. |
| -r | --recursive | Recursively delete directories. |

## 3. Working with Files

In this section, we will introduce the file related commands.

### 3.1 Creating and Empty Text File

In Linux, there are several ways to create an empty text file. Most commonly the **touch** command is used to create a file. We can create a file with name myfile.txt using touch command as given below:

```
touch  myfile.txt
```

Another, way to create a file in Linux is the cat command.

```
cat  > myfile.txt
```

Similarly, a file can be created using some editors. For example, to create a file using gedit editor

```
gedit myfile.txt
```

### 3.2 Reading the File Contents

**cat** command can also be used to read the contents of a file.

```
cat  myfile.txt
```

Another option to view the contents of a text file is the use of less command.

```
less   myfile.txt
```

Similarly, an editor can also be used to view the contents of a file.

```
gedit   myfile.txt
```

### 3.3 Appending text files

**cat** command is also used to append a text file. Suppose we want to add some text at the end of **myfile.txt**

```
cat  >>  myfile.txt
```

Now, type the text and enter ctrl+d to copy the text to myfile.txt.

### 3.4 Combining multiple text files

Using cat command, we can view the contents of multiple files. Suppose, we want to view the contents of file1, file2 and file3, we can use the cat command as follows:

```
cat  file1  file2   file3
```

Similarly, we can redirect the output of multiple files to file instead of screen using cat command. Suppose, in the above example we want to write the contents of file1, file2 and file3 into another file file4 we can do this as shown below:

```
cat  file1    file2   file3 >  file4
```

### 3.5 Determining File Type

To determine the type of a file we can use the **file** command. The syntax is given below:

```
file  filename
```

## 4. Redirecting I/O

Many of the programs that we have used so far produce output of some kind. This output often consists of two types. First, we have the program's results; that is, the data the program is designed to produce, and second, we have status and error messages that tell us how the program is getting along. If we look at a command like ls, we can see that it displays its results and its error messages on the screen.

Keeping with the Unix theme of “everything is a file,” programs such as ls actually send their results to a special file called standard output (often expressed as stdout) and their status messages to another file called standard error (stderr). By default, both standard output and standard error are linked to the screen and not saved into a disk file. In addition, many programs take input from a facility called standard input (stdin) which is, by default, attached to the keyboard. I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.

### 4.1 Redirecting Standard Output

I/O redirection allows us to write the output on another file instead of standard output i.e. screen. To do this, we use the redirection operator i.e. <. For example, we want to write the output of ls command in a text file myfile.txt instead of screen. This can be done as given below:

```
ls -l > myfile.txt
```

If we write the output of some other program to myfile.txt using > operator, its previous contents will be overwritten. Now, if want to append the file instead of over-writing we can use the << operator.

### 4.2 Redirecting Standard input

Redirecting input enables us to take input from another file instead of standard input i.e. keyboard. We have already discussed this in previous section while discussing cat command where we used the text file as input instead of keyboard and wrote it to another file.

### 4.3 Pipelines

The ability of commands to read data from standard input and send to standard output is utilized by a shell feature called pipelines. Using the pipe operator "|" (vertical bar), the standard output of one command can be piped into the standard input of another:

```
[me@ubuntu ~] $ ls -l | less
```

## 2) Stage a1 (apply)

# Lab Activities:

## Activity 1:

In this activity, you are required to perform tasks given below:

1. Display your current directory.
2. Change to the /etc directory.
3. Go to the parent directory of the current directory.
4. Go to the root directory.
5. List the contents of the root directory.
6. List a long listing of the root directory.
7. Stay where you are, and list the contents of /etc.
8. Stay where you are, and list the contents of /bin and /sbin.
9. Stay where you are, and list the contents of ~.
10. List all the files (including hidden files) in your home directory.
11. List the files in /boot in a human readable format.

## Solution:

1. pwd
2. cd /etc
3. cd ..
4. cd /
5. ls
6. ls -l
7. ls /etc
8. ls /bin /sbin
9. ls ~
10. ls -al ~
11. ls -lh /boot

## Activity 2:

Perform the following tasks using Linux CLI

5. Create a directory “mydir1” in Desktop Directory. Inside mydir1 create another directory “mydir2”.
6. Change your current directory to “mydir2” using absolute path
7. Now, change you current directory to Documents using relative path
8. Create mydir3 directory in Documents directory and go into it
9. Now, change your current directory to mydir2 using relative path

## Solution:

1. **cd /home/Ubuntu/Desktop** (suppose the user name is ubuntu)
   **mkdir mydir1**
   **cd mydir1**
   **mkdir mydir2**

2. **cd /home/ubuntu/Desktop/mydir1/mydir2**

3. **cd ../../../Desktop**

4. **mkdir mydir3**
   **cd mydir3**

5. **cd ../../Desktop/mydir1/mydir2**

## Activity 3:

Considering the directories created in Activity 2, perform the following tasks

1. Go to mydir3 and create an empty file **myfile** using cat command
2. Add text the text “Hello World” to myfile
3. Append myfile with text “Hello World again”

4. View the contents of myfile

## Solution:

1. **cd /home/Documents/mydir3** (suppose the user name is ubuntu)
   **cat >myfile**

2. **cat > myfile**
   **type: Hello World**
   **type: ctrl + d**

3. **cat >> myfile**
   **type: Hello World again**
   **type: ctrl + d**

4. **cat myfile**

## Activity 4:

Considering the above activities, perform the following tasks

1. move myfile to mydir1
2. copy myfile to mydir2
3. copy mydir2 on Desktop
4. delete mydir1 (get confirmation before deleting)
5. Rename myfile to mynewfile

## Solution:

1. **mv /home/ubuntu/Document/mydir3/myfile /home/ubuntu/Desktop/mydir1**

2. **cp /home/ubuntu/Desktop/mydir1/mydir3/myfile**
   **/home/ubuntu/Desktop/mydir1/mydir2**

3. **cp -r /home/ubuntu/Desktop/mydir1/mydir2 /home/ubuntu/Desktop**

4. **rm -ri /home/ubuntu/Desktop/mydir1**

5. **mv /home/ubuntu/Desktop/mydir2/myfile**
   **/hoem/ubuntu/Desktop/mydir2/mynewfile**

## Activity 5:

This activity is related to I/O redirection

1. Go to Desktop directory

2. Write the long-listing of contents of Desktop on an empty file out-put-file
3. View contents of out-put-file

## Solution:

1. **cd /home/ubuntu/Desktop**

2. **ls -l > out-put-file**

3. **cat out-put-file**

## 3) Stage V (verify)

## Home Activities:

1. Considering the lab activities, perform the following tasks
   1. Go to Desktop directory
   2. write the contents of mynewfile to newfile
   3. view the output of both mynewfile and newfile on screen
   4. write the combined output of mynewfile and newfile to a third file out-put-file

2. Long list all files and directories in your system and write out-put on a text-file.

## 4) Stage $a_2$ (assess)

## Lab Assignment and Viva voce

# LAB # 03

## Statement of Purpose:

This lab will introduce the basic concept of file access permissions and package management in Linux to you.

## Activity Outcomes:

This lab teaches you the following topics:

1. Reading and setting file permissions.
2. Setting the default file permissions.
3. Performing package management tasks.

# 1) Stage J (Journey)

## 1. File Permissions

Linux is a multi-user system. It means that more than one person can be using the computer at the same time. While a typical computer will likely have only one keyboard and monitor, it can still be used by more than one user. For example, if a computer is attached to a network or the Internet, remote users can log in via ssh (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the graphical output appear on a remote display.

In a multi-user environment, to ensure the operational accuracy, it is required to protect the users from each other. After all, the actions of one user could not be allowed to crash the computer, nor could one user interfere with the files belonging to another user.

### 1.1 id command

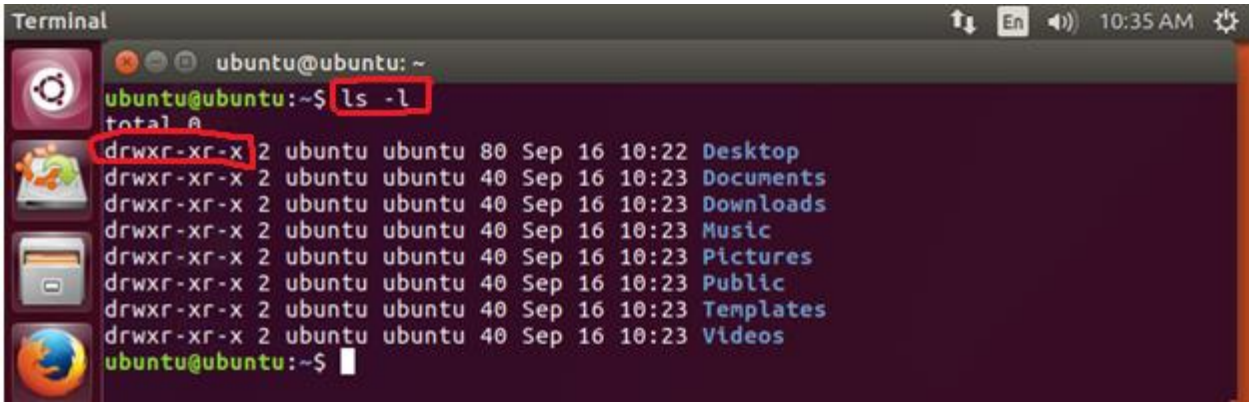
In the Linux security model, a user may own files and directories. When a user owns a file or directory, the user has control over its access. Users can, in turn, belong to a group consisting of one or more users who are given access to files and directories by their owners. In addition to granting access to a group, an owner may also grant some set of access rights to everybody, which in Linux terms is referred to as the world.

User accounts are defined in the /etc/passwd file and groups are defined in the /etc/group file. When user accounts and groups are created, these files are modified along with /etc/shadow which holds information about the user's password.


| Option | Explanation |
|---|---|
| -g | Print only the effective group id |
| -G | Print all Group ID's |
| -n | Prints name instead of number. |
| -r | Prints real ID instead of numbers. |
| -u | Prints only the effective user ID. |

### 1.2 Reading, Writing, and Executing

Access rights to files and directories are defined in terms of read access, write access, and execution access. If we look at the output of the ls command, we can get some clue as to how this is implemented:




The first ten characters of the listing are the file attributes. The first of these characters is the file type. Here are the file types you are most likely to see:

| Attribute | File Type |
|---|---|
| - | A regular file. |
| d | A directory. |
| l | A symbolic link. |
| c | A character special file. This file type refers to a device that handles data as a stream of bytes, such as a terminal or modem. |
| b | A block special file. This file type refers to a device that handles data in blocks, such as a hard drive or CD-ROM drive. |

The remaining nine characters of the file attributes, called the file mode, represent the read, write, and execute permissions for the file's owner, the file's group owner, and everybody else.

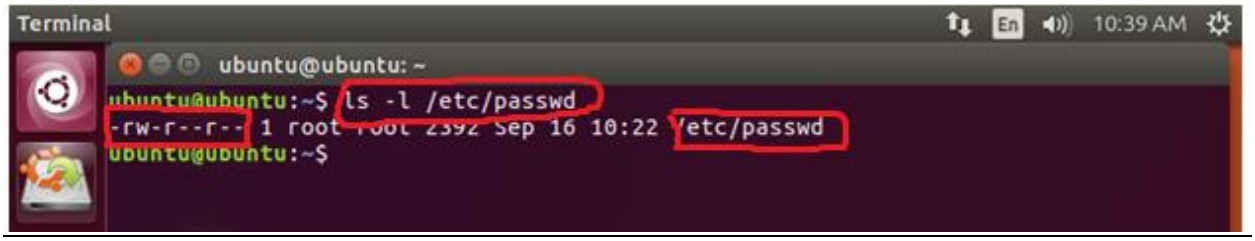
| User | Group | World |
|---|---|---|
| **rwx** | rwx | rwx |

| Attribute | Files | Directories |
|---|---|---|
| r | Allows a file to be opened and read. | Allows a directory's contents to be listed if the execute attribute is also set. |
| w | Allows a file to be written | Allows files within a directory to be created, deleted, and renamed if the execute attribute is also set. |
| x | Allows a file to be treated as a program and executed. | Allows a directory to be entered, e.g., cd directory. |

For example: **-rw-r--r-** A regular file that is readable and writable by the file's owner. Members of the file's owner group may read the file. The file is world-readable.

## 1.3 Reading File Permissions

The ls command is used to read the permission of a file. In the following example, we have used ls command with -l option to see the information about /etc/passwd file. Similarly, we can read the current permissions of any file.

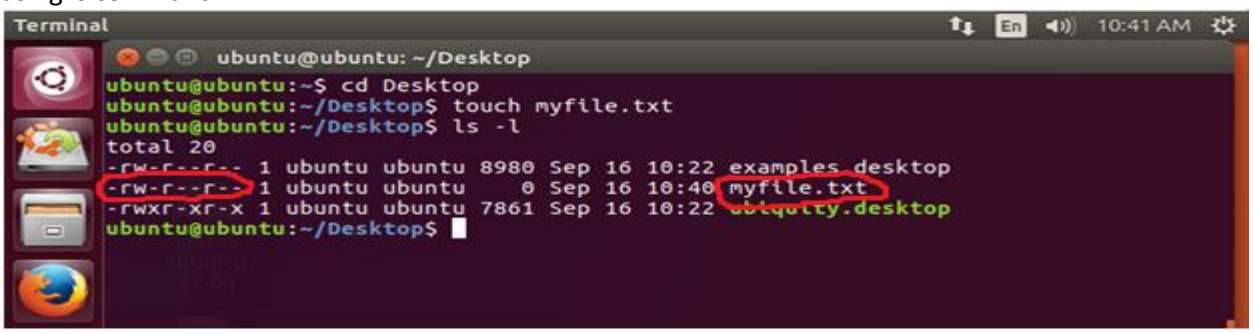



## 1.4 Change File Mode (Permissions)

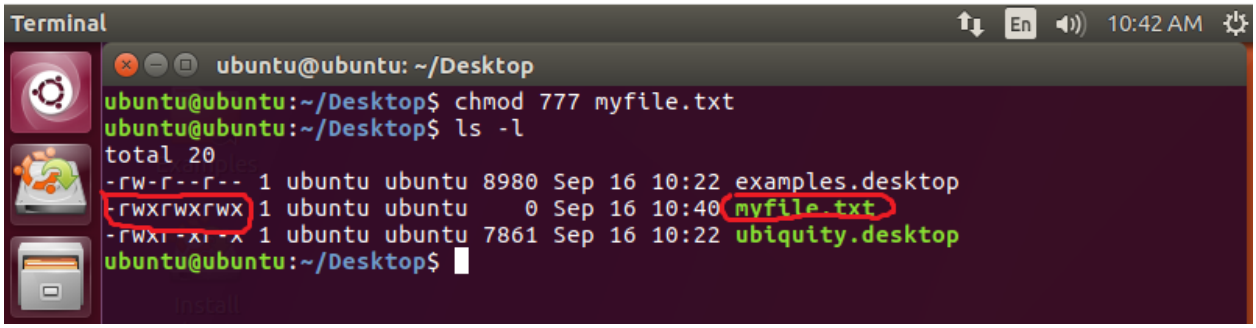
To change the mode (permissions) of a file or directory, the **chmod** command is used. Beware that only the file's owner or the super-user can change the mode of a file or directory. **chmod** supports two distinct ways of specifying mode changes: octal number representation, or symbolic representation. With octal notation we use octal numbers to set the pattern of desired permissions. Since each digit in an octal number represents three binary digits, these maps nicely to the scheme used to store the file mode.

| Octal | Binary | File Mode |
|---|---|---|
| 0 | 000 | --- |
| 1 | 001 | --x |
| 2 | 010 | -w- |
| 3 | 011 | -wx |
| 4 | 100 | r-- |
| 5 | 101 | r-x |
| 6 | 110 | rw- |
| 7 | 111 | rwx |

In the following example, we first go to the Desktop directory using cd command. In Desktop directory, we create a text file **"myfile.txt"** using touch command and read its current permissions using ls command.




Now, we change the permission of myfile.txt and set it to 777 that is everyone can read, write and execute the file.

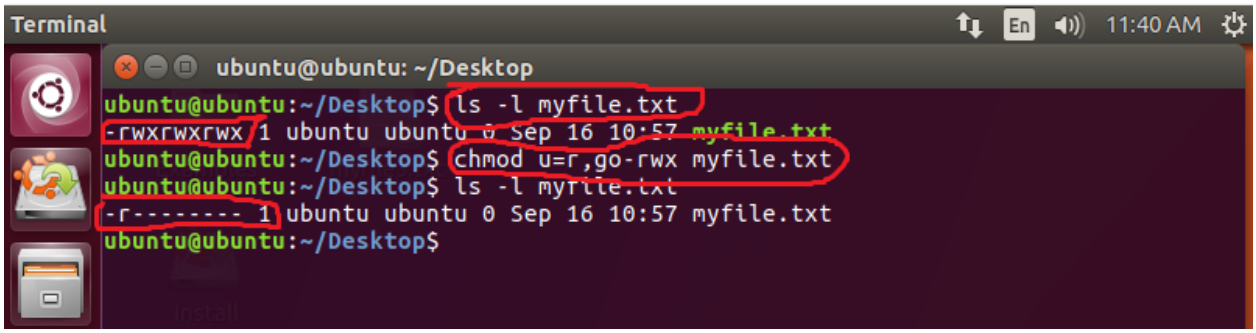



chmod also supports a symbolic notation for specifying file modes. Symbolic notation is divided into three parts: who the change will affect, which operation will be performed, and what permission will be set. To specify who is affected, a combination of the characters “u”, “g”, “o”, and “a” is used as follows:

| Character | Meaning |
|---|---|
| u | Owner |
| g | Group |
| o | Others |
| a | all |

If no character is specified, “all” will be assumed. The operation may be a “+” indicating that a permission is to be added, a “-” indicating that a permission is to be taken away, or a “=” indicating that only the specified permissions are to be applied and that all others are to be removed. For example: **u+x,go=rx** Add execute permission for the owner and set the permissions for the group and others to read and execute. Multiple specifications may be separated by commas.

In the following example, we change the permissions of myfile.txt using symbolic codes. As all of the permissions of myfile.txt were set previously, now we make it readable only to the user while the rest cannot read, write or execute the file.



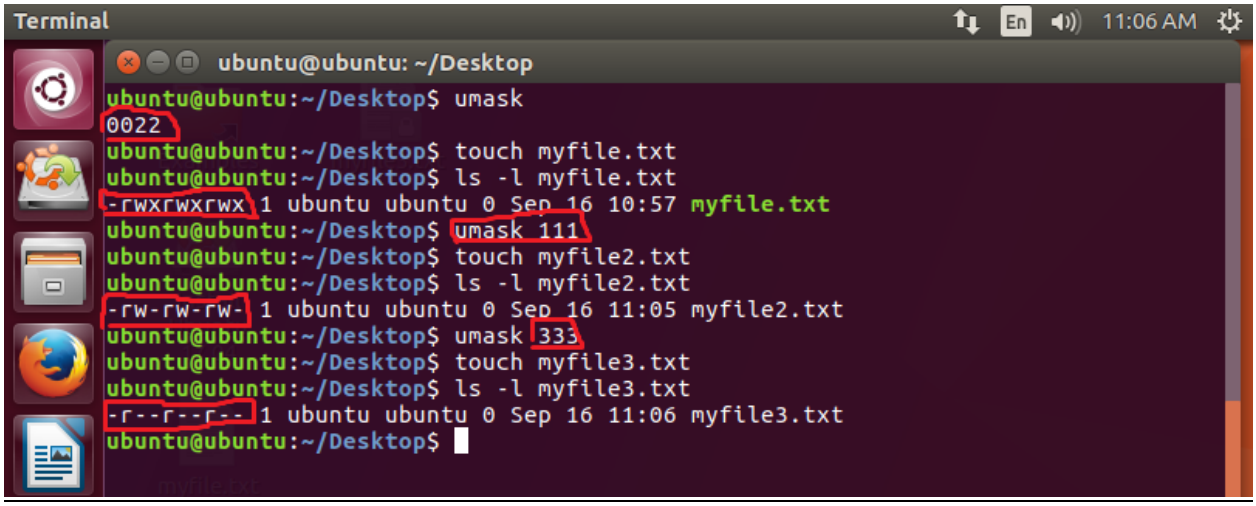

## 1.5 Controlling the Default Permissions

On Unix-like operating systems, the **umask** command returns, or sets, the value of the system's file mode creation mask. When user create a file or directory under Linux or UNIX, he/she creates it with a default set of permissions. In most case the system defaults may be open or relaxed for file sharing purpose.
**umask** command with no arguments can be used to return the current mask value. Similarly, If the **umask** command is invoked with **an octal argument**, it will directly set the bits of the mask to that argument.

The three rightmost octal digits address the "owner", "group" and "other" user classes respectively. If fewer than 4 digits are entered, leading zeros are assumed. An error will result if the argument is not a valid octal number or if it has more than 4 digits. If a fourth digit is present, the leftmost (high-order) digit addresses three additional attributes, the setuid bit, the setgid bit and the sticky bit.

| Octal Value | Permissions |
|---|---|
| 0 | read, write, execute |
| 1 | read and write |
| 2 | read and execute |
| 3 | read only |
| 4 | write and execute |
| 5 | write only |
| 6 | execute only |
| 7 | no permissions |

In the following example, we first read the current mask that is 0022 and then we created a file myfile.txt using this mask and display its permission. Then we reset the mask with 111 and 333 octal values and create new files. It can be seen clearly that new files are created with different default permissions.




## 1.6 Changing User Identity

At various times, we may find it necessary to take on the identity of another user. Often, we want to gain superuser privileges to carry out some administrative task, but it is also possible to “become” another regular user for such things as testing an account.

### Run A Shell with Substitute User and Group IDs

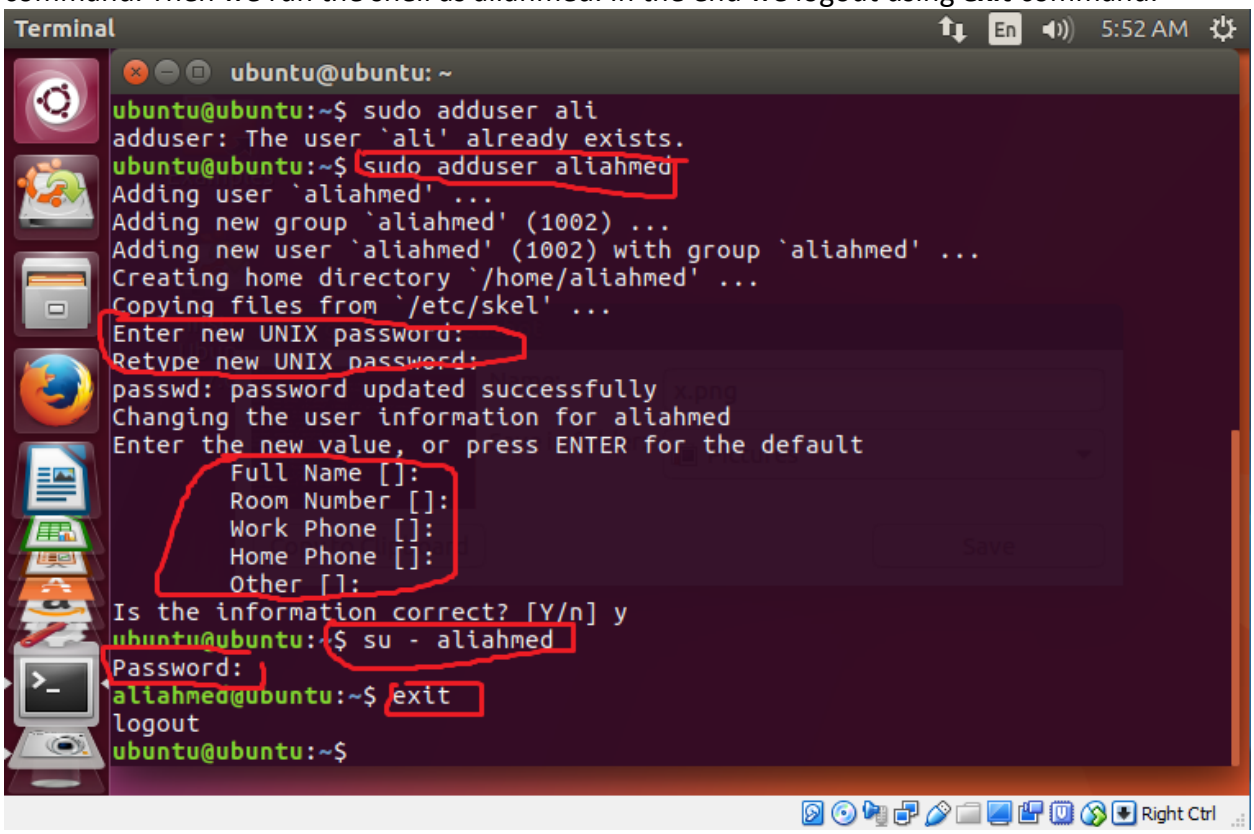
The su command is used to start a shell as another user. The command syntax looks like this:

**su -l *username***

### Execute A Command as Another User

On Unix-like operating systems, the **sudo** command ("superuser do", or "switch user, do") allows a user with proper permissions to execute a command as another user, such as the superuser.

**Example**: In the following example first, we created a new user “aliahmed” using **adduser** command. Then we run the shell as aliahmed. In the end we logout using **exit** command.

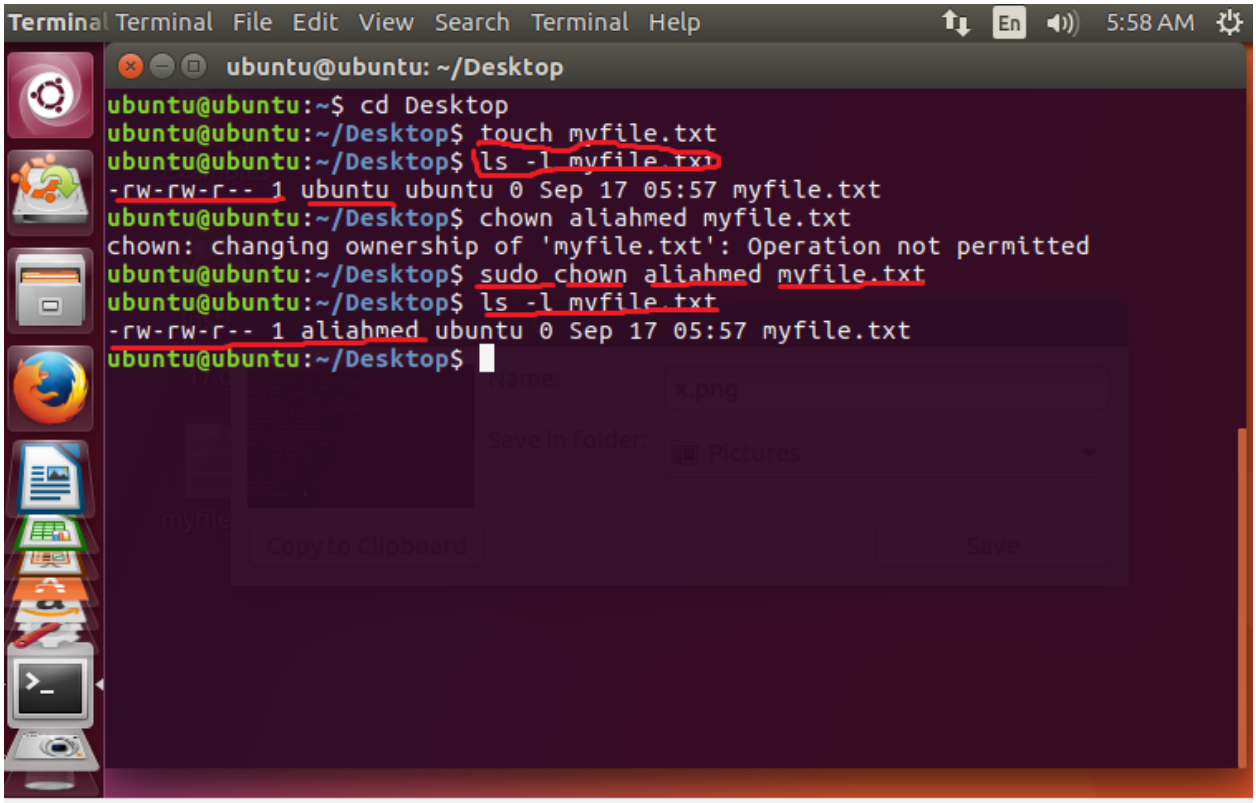



## 1.7 Change File Owner and Group

The chown command is used to change the owner and group owner of a file or directory. Superuser privileges are required to use this command. The syntax of chown looks like this:

### chown owner:group file/files

**Example:** In the following example first, we created a file named myfile.txt as user ubuntu. Then we changed the ownership of myfile.txt from ubuntu to aliahmed.




## 2. Package Management

Package management is a method of installing and maintaining software on the system. Linux doesn't work that way. Virtually all software for a Linux system will be found on the Internet. Most of it will be provided by the distribution vendor in the form of *package files* and the rest will be available in source code form that can be installed manually.
Different distributions use different packaging systems and as a general rule, a package intended for one distribution is not compatible with another distribution. Most distributions fall into one of two camps of packaging technologies: the Debian ".deb" camp and the Red Hat ".rpm" camp. There are some important exceptions such as Gentoo, Slackware, and Foresight, but most others use one of these two basic systems.

**Package Files**

The basic unit of software in a packaging system is the package file. A package file is a compressed collection of files that comprise the software package. A package may consist of numerous programs and data files that support the programs. In addition to the files to be installed, the package file also includes metadata about the package, such as a text description of the package and its contents. Additionally, many packages contain pre- and post-installation scripts that perform configuration tasks before and after the package installation.

**Repositories**
While some software projects choose to perform their own packaging and distribution, most packages today are created by the distribution vendors and interested third parties.

Packages are made available to the users of a distribution in central repositories that may contain many thousands of packages, each specially built and maintained for the distribution.

#### Dependencies

Programs seldom "standalone"; rather they rely on the presence of other software components to get their work done. Common activities, such as input/output for example, are handled by routines shared by many programs. These routines are stored in what are called *shared libraries*, which provide essential services to more than one program. If a package requires a shared resource such as a shared library, it is said to have a *dependency*. Modern package management systems all provide some method of *dependency resolution* to ensure that when a package is installed, all of its dependencies are installed too

#### High and Low-level Package Tools

Package management systems usually consist of two types of tools: low-level tools which handle tasks such as installing and removing package files, and high-level tools that perform metadata searching and dependency resolution. For Debian based systems low-level tools are defined in dpkg while high-level tools are defined in apt-get, aptitude.

## Common Package Management Tasks

### 2.1 Finding a Package in a Repository

Using the high-level tools to search repository metadata, a package can be located based on its name or description. In Debian based systems it can be done as given below:

**apt-get update**
**apt-cache search *search_string***

Example: To search apt repository for the emacs text editor, this command could be used:

**apt -get update**
**apt-get search emacs**

### 2.2 Installing a Package from a Repository

High-level tools permit a package to be downloaded from a repository and installed with full dependency resolution.

Example: To install the emacs text editor from an apt repository:

**apt-get update; apt-get install emacs**

### 2.3 Installing a Package from a Package File

If a package file has been downloaded from a source other than a repository, it can be installed directly (though without dependency resolution) using a low-level tool.

**dpkg-install *package_file***

Example: If the emacs-22.1-7.fc7-i386.deb package file had been downloaded from a non-repository site, it would be installed this way:

**dpkg –install emacs-22.1-7.fc7-i386.deb**

## 2.4 Removing A Package

Packages can be uninstalled using either the high-level or low-level tools. The high-level tools are shown below.

**apt -get remove *package_name***

Example: To uninstall the emacs package from a Debian-style system:

**apt -get remove emacs**

## 2.5 Updating Packages from a Repository

The most common package management task is keeping the system up-to-date with the latest packages. The high-level tools can perform this vital task in one single step.

Example: To apply any available updates to the installed packages on a Debian-style system:

**apt -get update; apt-get upgrade**

## 2.6 Upgrading a Package from a Package File

If an updated version of a package has been downloaded from a non-repository source, it can be installed, replacing the previous version:

**dpkg --install *package_file***

## 2.7 Listing Installed Packages

These commands can be used to display a list of all the packages installed on the system:

**dpkg --list**

## 2.8 Determining If A Package Is Installed

These low-level tools can be used to display whether a specified package is installed:

**dpkg --status *package_name***

*Example:*
**dpkg --status emacs**

## 2.9 Displaying Information About an Installed Package

If the name of an installed package is known, the following commands can be used to display a description of the package:

**apt -cache show *package_name***

*Example:*
**apt -cache show emacs**

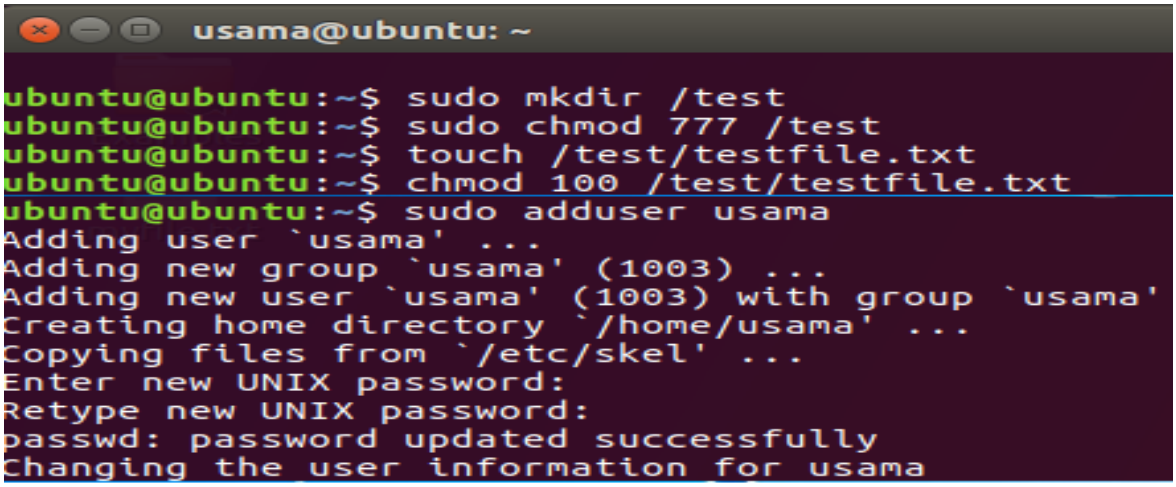
# 2) Stage a1 (apply)

## Lab Activities

**Activity 1:**
This activity is related to file permission. Perform the following tasks

1. Create a new directory named test in root directory as superuser
2. Make this directory public for all
3. Create a file “testfile.txt” in /test directory
4. Change its permissions that no boy can write the file, but the owner can read it.
5. Create another user “Usama”
6. Run the shell with user Usama
7. Try to read the “testfile.txt”
8. Logout as Usama
9. Change the permission of testfile.txt so that everyone can read, write and execute it
10. Run shell as Usama again
11. Now, read the file

**Solution:**



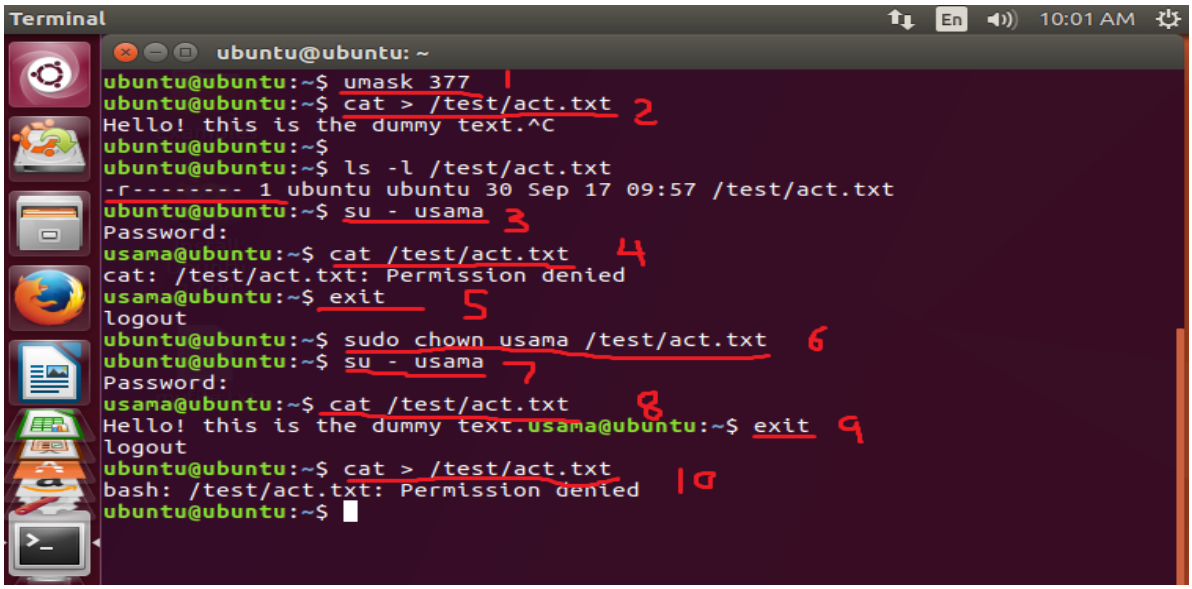
```
Enter the new value, or press ENTER for the defaul
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] y
ubuntu@ubuntu:~$ su - usama
Password:
usama@ubuntu:~$ less /test/testfile.txt
/test/testfile.txt: Permission denied
usama@ubuntu:~$ exit
logout
ubuntu@ubuntu:~$ chmod 777 /test/testfile.txt
ubuntu@ubuntu:~$ su - usama
Password:
usama@ubuntu:~$ less /test/testfile.txt
usama@ubuntu:~$
```

## Activity 2:

Perform the following tasks

1. Set the permission such that a newly created file is readable only to the owner
2. Create a text file “act.txt” in /test directory crated in previous activity
3. Run the shell as user “usama” (created previously)
4. Access the file “act.txt”
5. Logout as usama
6. Now change the ownership of the file from ubuntu to usama
7. Run the shell again as usama
8. Read the file “act.txt” using cat command
9. Logout as usama
10. Now access the act.txt with user ubuntu

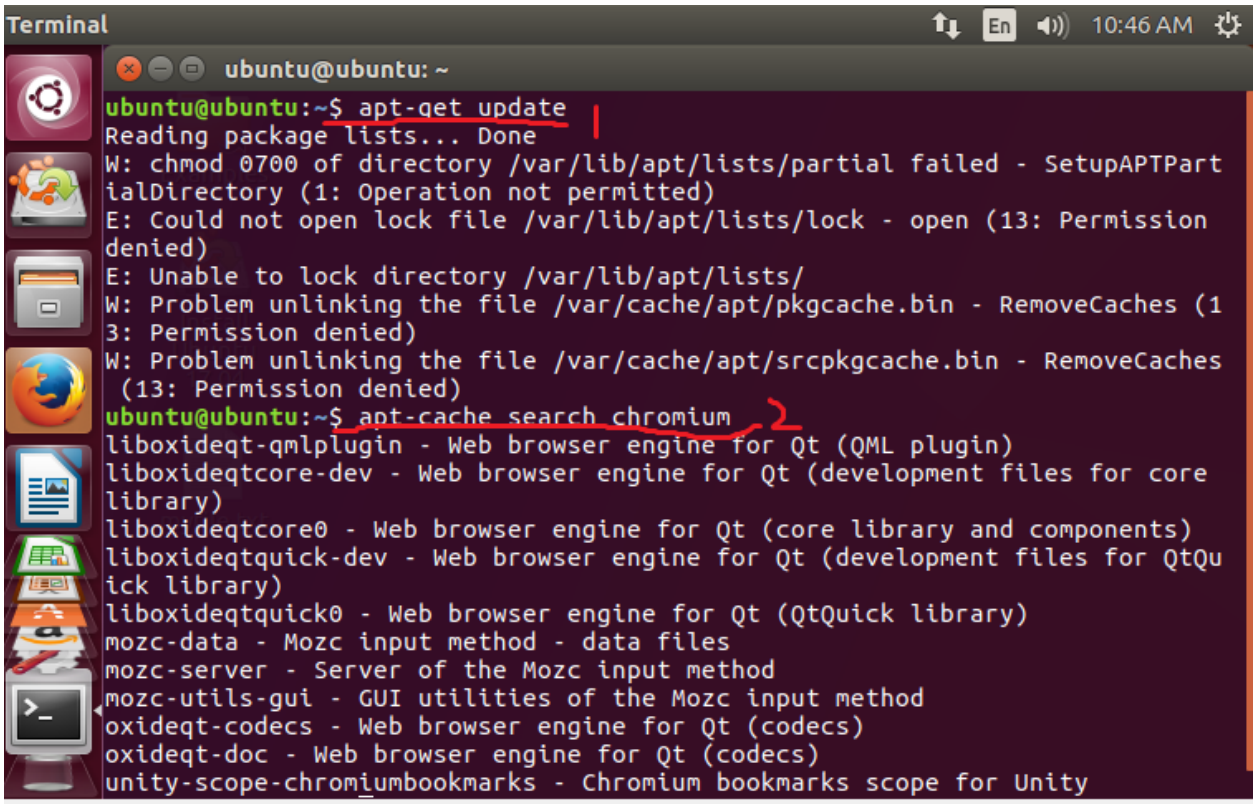
## Solution:

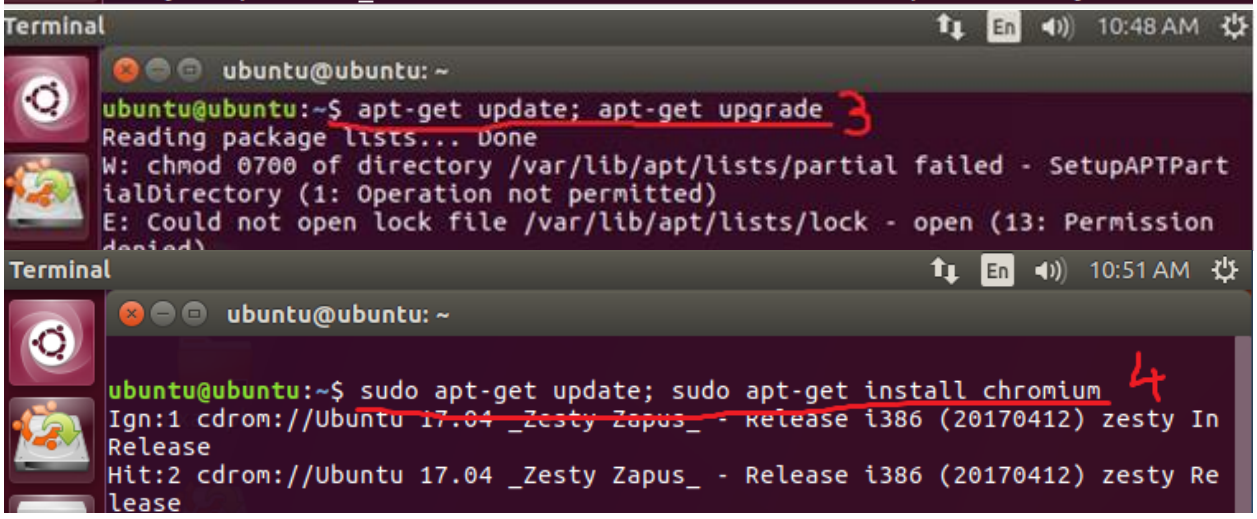


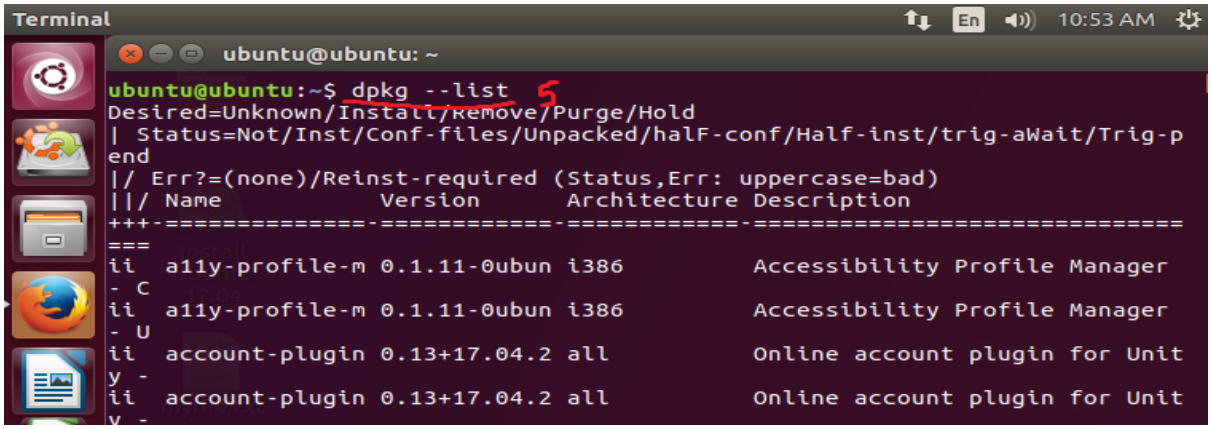
## Activity 3:

Perform the following tasks

1. search apt repository for the chromium browser
2. install the chromium browser using command line
3. write the command to update and upgrade the repository
4. list the software installed on your machine and write output on a file list.txt
5. read the list.txt file using cat command






## 3) Stage v (verify)

## Home Activities:

familiar with adduser command using: man adduser/useradd, man groupadd useradd - create a new user or update default new user information. Create 3 user accounts (user1, user2, user3) and add 2 groups (gr1, gr2). add user1 to gr1 and add user2, user2 to gr2. Check user ID (UID) and group ID (GID) by listing file /etc/passwd. Find lines with added user. What is the UID and GID for these accounts? Write command which show UID and GID for your user name:

2. create 3 files with touch command: files1, files2, files3.
3. Write the command line by using letters with chmod to set the following permissions:

- rwxrwxr-x for file1
- r-x—x—x for file2
- ——xrwx for file3

4. Write the command line by using numbers with chmod to set the following permissions:

- rwxrwxrwx for file4 (you have to prepare this file)
- -w------- for file5 (you have to prepare this file)
- rwx--x—x for folder1 (you have to prepare this folder)

5. Create two user accounts: tst1 and tst2 Logging in id: tst1, group users, with bash shell, home directory /home/tst1 Logging in id: tst2, group public, with bash shell, home directory home/tst2 For the two accounts set a password.

Logging in as tst1 and copy /bin/ls into tst1 home directory as myls file. Change the owner of myls to tst1 and the permissions to 0710. What does this permission value mean?

Logging in as tst2 and try to use /home/tst1/myls to list your current directory. Does it work ?

Create a new group labo with tst1 and tst2. Change the owner group of myls file to labo. Try again from tst2 account to execute /home/tst1/myls to list your current directory. Does it work?

## 4) Stage a2 (assess)

## Lab Assignment and Viva voce

# LAB # 04

## Statement of Purpose:

This lab will introduce the basic concept of **Text Processing Tools and Basic System Configuration Tools** in Linux to you.

## Activity Outcomes:

This lab teaches you the following topics:

An introduction to some of the most useful text-processing utilities

Using Linux's graphical and text-based configuration tools to manage networking, printing and date/time settings

---

# 1) Stage J (Journey)

**Viewing File Contents With less**

The less command is a program to view text files. Throughout our Linux system, there are many files that contain human-readable text. The less program provides a convenient way to examine them.

Why would we want to examine text files?

Because many of the files that contain system settings (called configuration files) are stored in this format and being able to read them gives us insight about how the system works. In addition, many of the actual programs that the system uses (called scripts) are stored in this format.

The less command is used like this:

```
$less <filename>
```

Once started, the less program allows you to scroll forward and backward through a text file. For example, to examine the file that defines all the system's user accounts, enter the following command:

```
$less /etc/passwd
```

Once the less program starts, we can view the contents of the file. If the file is longer than one page, we can scroll up and down. To exit less, press the “q” key.

The table below lists the most common keyboard commands used by less.

| Command | Action |
|---|---|
| Page Up or `b` | Scroll back one page |
| Page Down or space | Scroll forward one page |
| Up Arrow | Scroll up one line |
| Down Arrow | Scroll down one line |
| `G` | Move to the end of the text file |
| `1G` or `g` | Move to the beginning of the text file |
| */characters* | Search forward to the next occurrence of *characters* |
| `n` | Search for the next occurrence of the previous search |
| `h` | Display help screen |
| `q` | Quit `less` |

**wc – Print Line, Word, and Byte Counts**

The wc (word count) command is used to display the number of lines, words, and bytes contained in files. For example:

```
$wc Myfile.txt
```

In this case it prints out three numbers: lines, words, and bytes contained in Myfile.txt.

**Pipelines**

The ability of commands to read data from standard input and send to standard output is utilized by a shell feature called pipelines. Using the pipe operator "|" (vertical bar), the standard output of one command can be piped into the standard input of another: For example, we can use less to display, page-by-page, the output of any command that sends its results to standard output:

```
$ls -l /usr/bin | less
```

**Filters**

Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as filters. Filters take input, change it somehow and then output it. The first one we will try is **sort**.

**Sort**

Imagine we wanted to make a combined list of all of the executable programs in /bin and /usr/bin, put them in sorted order and view it:

```
$ls /bin /usr/bin | sort | less
```

Since we specified two directories (/bin and /usr/bin), the output of ls would have consisted of two sorted lists, one for each directory. By including sort in our pipeline, we changed the data to produce a single, sorted list.

**grep – Print Lines Matching A Pattern**

grep is a powerful program used to find text patterns within files. It's used like this:

```
grep pattern [file...]
```

When grep encounters a “pattern” in the file, it prints out the lines containing it. The patterns that grep can match can be very complex, but for now we will concentrate on simple text matches.

For example, find all the files in our list of programs that had the word “zip” embedded in the name:

```
$ ls /bin /usr/bin | sort | grep zip
```

There are a couple of handy options for grep: “-i” which causes grep to ignore case when performing the search (normally searches are case sensitive) and “-v” which tells grep to only print lines that do not match the pattern.

head / tail – Print First / Last Part of Files

Sometimes you don't want all the output from a command. You may only want the first few lines or the last few lines. The head command prints the first ten lines of a file and the tail command prints the last ten lines. By default, both commands print ten lines of text, but this can be adjusted with the “-n” option:

```
$head -n 5 Myfile.txt
$tail -n 5 Myfile.txt
```

These can be used in pipelines as well:

```
$ ls /usr/bin | tail -n 5
```

**Networking**

There are number of commands that can be used to configure and control networking including commands used to monitor networks and those used to transfer files. In this lab, we will also see the above-mentioned commands. In addition, we are going to explore the **ssh** program that is used to perform remote logins.

**Examining and Monitoring A Network**

The ping command sends a special network packet called an ICMP ECHO_REQUEST to a specified host. Most network devices receiving this packet will reply to it, allowing the network connection to be verified.

For example,

| **$ping localhost** |
|---|

After it is interrupted by pressing Ctrl-c, ping prints performance statistics. A properly performing network will exhibit zero percent packet loss. A successful “ping” will indicate that the elements of the network (its interface cards, cabling, routing, and gateways) are in generally good working order.

## Set Date and Time

### Display Current Date and Time

Just type the date command:

$ date
Sample outputs:
Mon Jan 21 01:31:40 IST 2019

### Display the Hardware Clock (RTC)

Type the following hwclock command to read the Hardware Clock and display the time on screen:

# hwclock -r

OR

# hwclock --show

$ sudo hwclock --show --verbose

OR show it in Coordinated Universal time (UTC):

# hwclock --show --utc

Sample outputs:

2019-01-21 01:30:50.608410+05:30

### Set Date Command Example

Use the following syntax to set new data and time:

date –-set "STRING"

For example, set new data to 2 Oct 2006 18:00:00, type the following command as root user:

# date -s "2 OCT 2006 18:00:00"

OR

# date –-set "2 OCT 2006 18:00:00"

You can also simplify format using following syntax:

# date +%Y%m%d -s "20081128"

**Set Time Examples**

To set time use the following syntax:

# date +%T -s "10:13:13"

Where,

1: Hour (hh)

2: Minute (mm)

3: Second (ss)

Use %p locale's equivalent of either AM or PM, enter:

# date +%T%p -s "6:10:30AM"

# date +%T%p -s "12:10:30PM"

**How do I set the Hardware Clock to the current System Time?**

Use the following syntax:
# hwclock --systohc
OR
# hwclock -w
timedatectl: Display the current date and time
Type the following command:
$ timedatectl

How do I change the current date using the timedatectl command?
To change the current date, type the following command as root user:
# timedatectl set-time YYYY-MM-DD
OR
$ sudo timedatectl set-time YYYY-MM-DD
For example set the current date to 2015-12-01 (1st, Dec, 2015):
# timedatectl set-time '2015-12-01'
# timedatectl
Sample outputs:

```
      Local time: Tue 2015-12-01 00:00:03 EST
  Universal time: Tue 2015-12-01 05:00:03 UTC
        RTC time: Tue 2015-12-01 05:00:03
       Time zone: America/New_York (EST, -0500)
     NTP enabled: no
NTP synchronized: no
 RTC in local TZ: no
      DST active: no
 Last DST change: DST ended at
                  Sun 2015-11-01 01:59:59 EDT
```

Sun 2015-11-01 01:00:00 EST
Next DST change: DST begins (the clock jumps one hour forward) at
Sun 2016-03-13 01:59:59 EST
Sun 2016-03-13 03:00:00 EDT

**To change both the date and time, use the following syntax:**

# timedatectl set-time YYYY-MM-DD HH:MM:SS

Where,

1. HH : An hour.
2. MM : A minute.
3. SS : A second, all typed in two-digit form.
4. YYYY: A four-digit year.
5. MM : A two-digit month.
6. DD: A two-digit day of the month.

For example, set the date ’23rd Nov 2015’ and time to ‘8:10:40 am’, enter:

# timedatectl set-time '2015-11-23 08:10:40'

# date

How do I set the current time only?

The syntax is:

# timedatectl set-time HH:MM:SS

# timedatectl set-time '10:42:43'

# date

Sample outputs:

Mon Nov 23 08:10:41 EST 2015

How do I set the time zone using timedatectl command?

To see list all available time zones, enter:

$ timedatectl list-timezones

$ timedatectl list-timezones | more

$ timedatectl list-timezones | grep -i asia

$ timedatectl list-timezones | grep America/New

To set the time zone to ‘Asia/Kolkata’, enter:

# timedatectl set-timezone 'Asia/Kolkata'

Verify it:

# timedatectl

Local time: Mon 2015-11-23 08:17:04 IST

Universal time: Mon 2015-11-23 02:47:04 UTC

RTC time: Mon 2015-11-23 13:16:09

Time zone: Asia/Kolkata (IST, +0530)

NTP enabled: no

NTP synchronized: no

RTC in local TZ: no

DST active: n/a

Lab Activities

## 2) Stage a1 (apply)

**Activity 1:**

This activity is related to file permission. Perform the following tasks

0. Create a file “testfile.txt” in /test directory
1. View and read the file using Less command
2. Use wc command to print lines, words and bytes
3. Find any text pattern in the file using grep
4. Print the first 3 lines of the file using head
5. Print the last 3 lines of the file using tail

**Solution:**

0. Touch /test/testfile.txt
1. Less testfile.txt
2. wc testfile.txt
3. grep testfile.txt
4. head -n 3 testfile.txt
5. tail -n 3 testfile.txt

**Activity 2:**

Perform the following tasks

1. Find all the files with extension txt in the /test directory
2. Find the first line of the list of files in the /test directory
3. Find the last line of the list of files in the /test directory
4. Produce and view a single sorted list of files by combining two directories: /Desktop and /bin

**Solution**

1. ls /test | grep zip
2. ls /test | head -n 1
3. ls /test | tail -n 1
4. ls /test /Desktop | sort | less

**Activity 3:**

Perform the following tasks

1. Examine the network using Ping command and view the performance statistics

**Solution**

1. ping localhost

## 3) Stage v (verify)

Write Linux commands to

1. Display all printer information
2. Display only those printers that are currently accepting print requests
3. Displays which printer is currently the default.
4. Print the double-sided legal document file name to printer myprinter.
5. Print a text file with 12 characters per inch, 8 lines per inch, and a 1 inch left margin.
6. Cancels all pending print jobs.

## 4) Stage $a_2$ (assess)

# Lab Assignment and Viva voce

# LAB # 05

## Statement of Purpose:

This lab will introduce the basic concepts related to process management through and Writing C++ programs in Linux.

## Activity Outcomes:

This lab teaches you the following topics:

1. How to manage processes in Linux.
2. Compiling and Executing C++ programs in Linux.

# 1) Stage J (Journey)

## 1. Process management in Linux

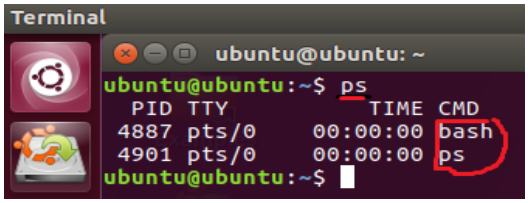
A process is the instance of a computer program that is being executed. While a computer program is a passive collection of instructions, a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often results in more than one process being executed. When a system starts up, the kernel initiates a few of its own activities as processes and launches a program called init. init, in turn, runs a series of shell scripts (located in /etc) called init scripts, which start all the system services. Many of these services are implemented as daemon programs, programs that just sit in the background and do their thing without having any user interface. So even if we are not logged in, the system is at least a little busy performing routine stuff.

The kernel maintains information about each process to help keep things organized. For example, each process is assigned a number called a process ID or PID. PIDs are assigned in ascending order, with init always getting PID 1. The kernel also keeps track of the memory assigned to each process, as well as the processes' readiness to resume execution. Like files, processes also have owners and user IDs, effective user IDs, etc. In the following, we will discuss the most common commands; available in Linux to mange processes.

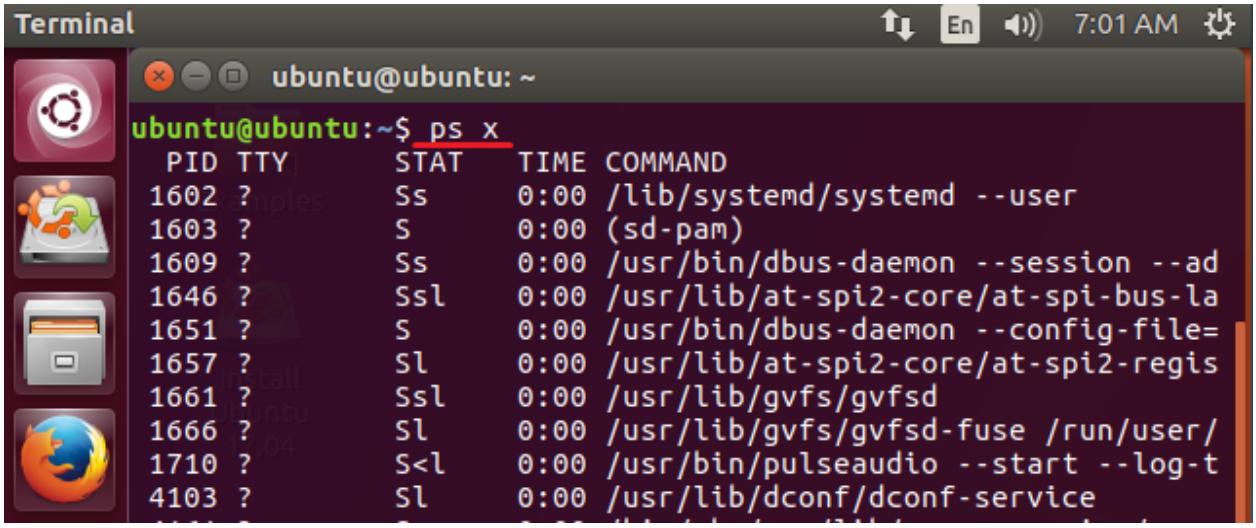
## b. Displaying Processes in the System

### 1.1.1 Static view

The most commonly used command to view processes is ps. This command displays the processes for the current shell. We can use the ps command as given below




We can display all of the processes owned by the current user by using the x option.



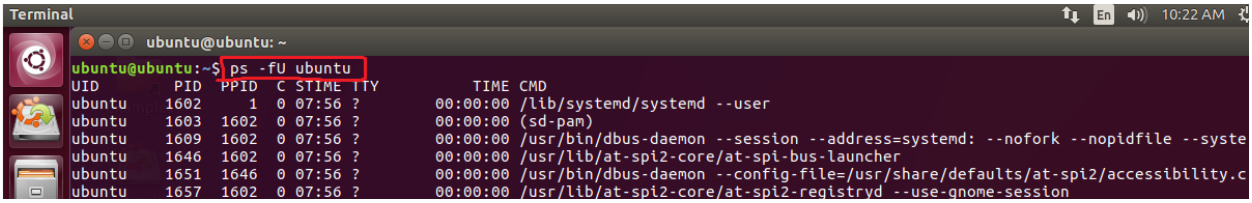

Here, a new column is also added in the output that is STAT. This column shows the current state of the process. The most common states are given below.

| State | Meaning |
|---|---|
| R | Running |
| S | Sleeping or waiting for an event such as keystroke |
| D | Uninterruptible Sleep. Process is waiting for I/O such as a disk drive |
| T | Stopped |
| Z | A difunctional or zombie process |
| l | multithreaded |
| s | Session leader |
| + | Foreground process |
| < | A high priority process |
| N | A low priority process |

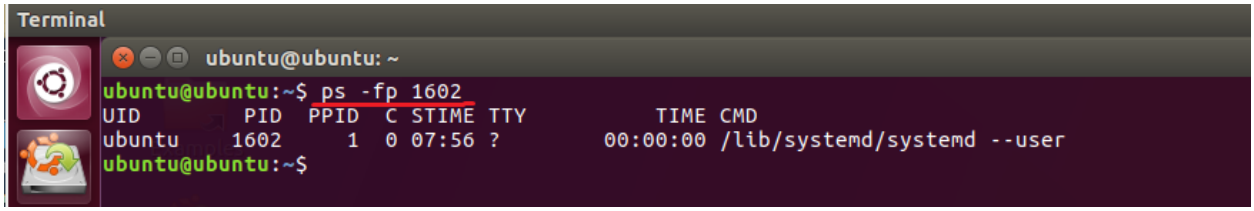
Following are the most common option that can be used with ps command.

| Option | Description |
|---|---|
| -A or -e | Display every active process on a Linux system |
| -x | Display all of the processes owned by the user |
| -F | Perform a full-format listing |
| -U | Select by real user ID or name |
| -u | Select by effective user ID or name |
| -p | Select process by pid |
| -r | Display only running processes |
| -L | Show number of threads in a process |
| -G | Show processes by Group |

In the following example, we display processes that are related to the user ubuntu.



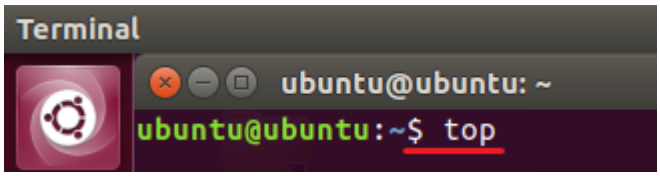

Now, we select a process with id 1602



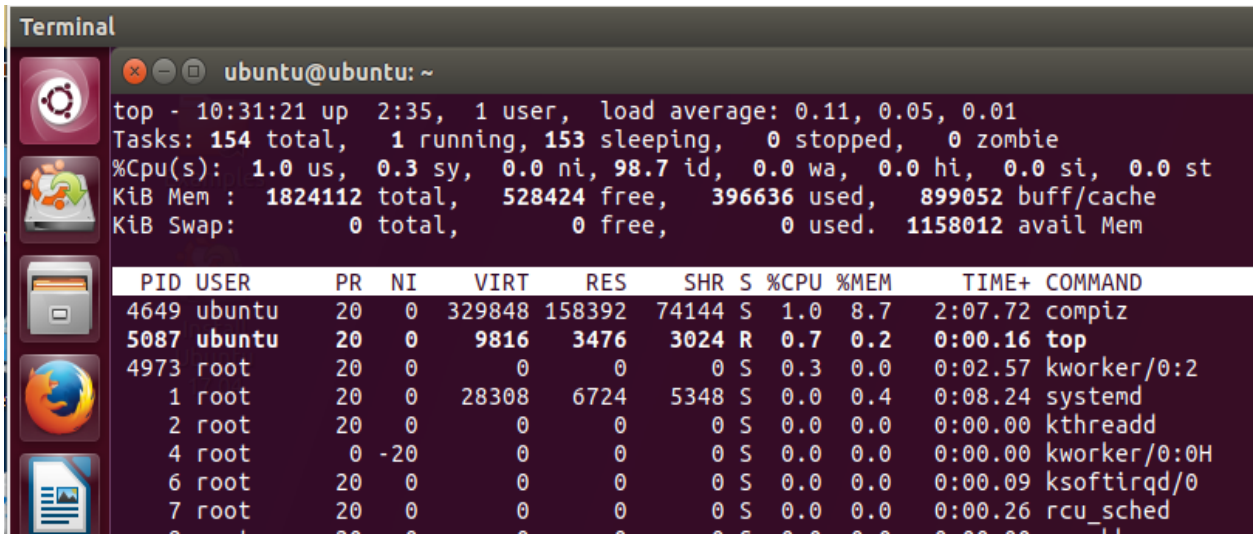

### 1.1.2 Dynamic view

The top command is the traditional way to view your system's resource usage and see the processes that are taking up the most system resources. Top displays a list of processes, with the ones using the most CPU at the top. An improved version of top command is htop but it is usually not pre-installed in most distributions. When a top program is running, we can highlight the running programs by pressing **z**. we can quit the top program by press **q** or **Ctrl + c**.

In the following example we display the dynamic view of the system process and resource usage.




The output of the above command is given below

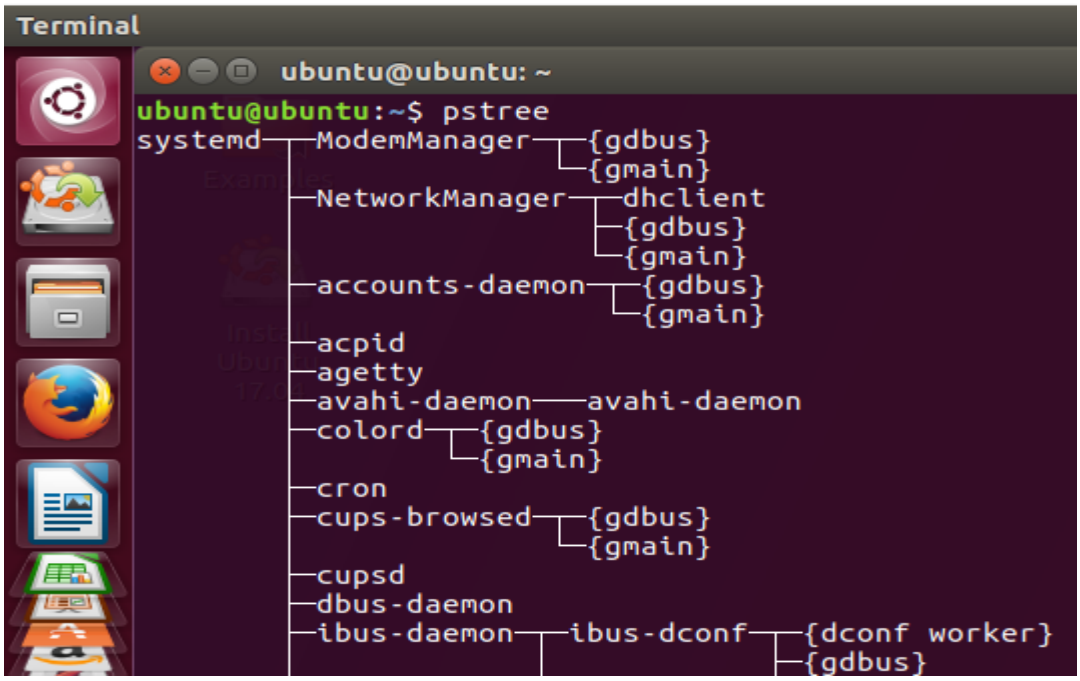



Some of the basic options available with top commands are given below

| Options | Description |
|---|---|
| -u | display specific User process details |
| -d | To set the screen refresh frequency |

### 1.1.3 Displaying processes in Treelike structure

The pstree command is used to display processes in tree-like structure showing the parent/child relationships between processes.

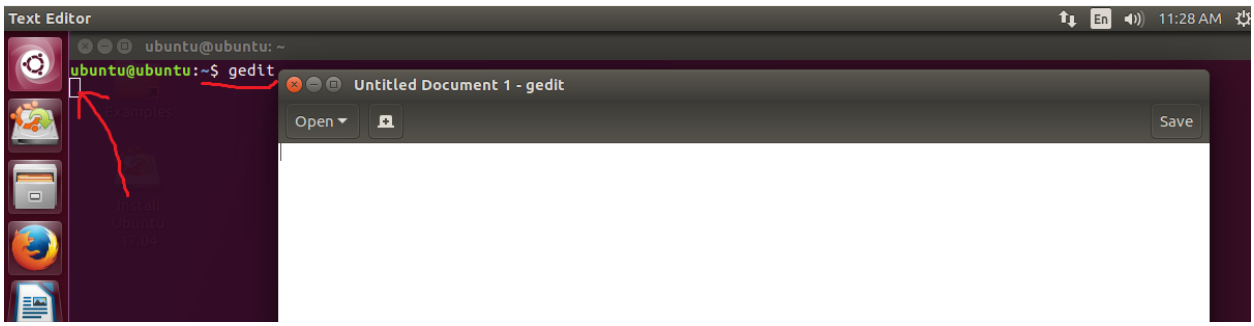



## 1.2 Interrupting A Process

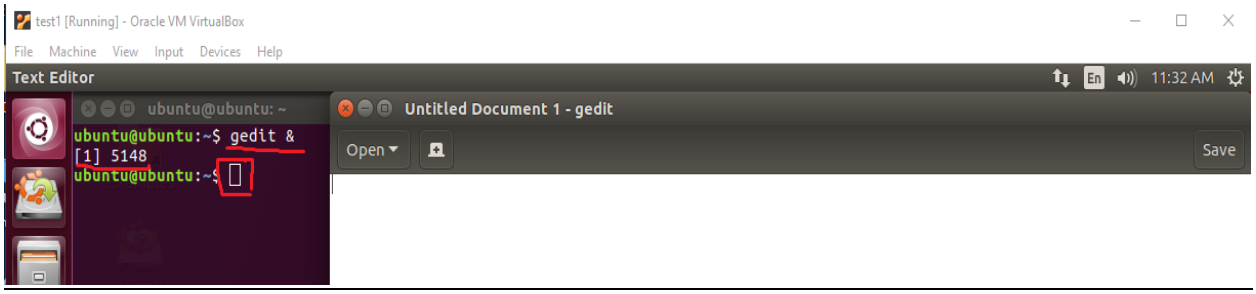
A program can be interrupted by pressing Ctrl + C. This will interrupt the given processes and stop the process. Ctrl+C essentially sends a SIGINT signal from the controlling terminal to the process, causing it to be killed.

## 1.3 Putting a Process in the Background

A foreground process is any command or task you run directly and wait for it to complete. Unlike with a foreground process, the shell does not have to wait for a background process to end before it can run more processes. Normally, we start a program by entering its name in the CLI. However, if we want to start a program in background, we will put an & after its name.
In the following example, first we open the gedit program normally as given below.



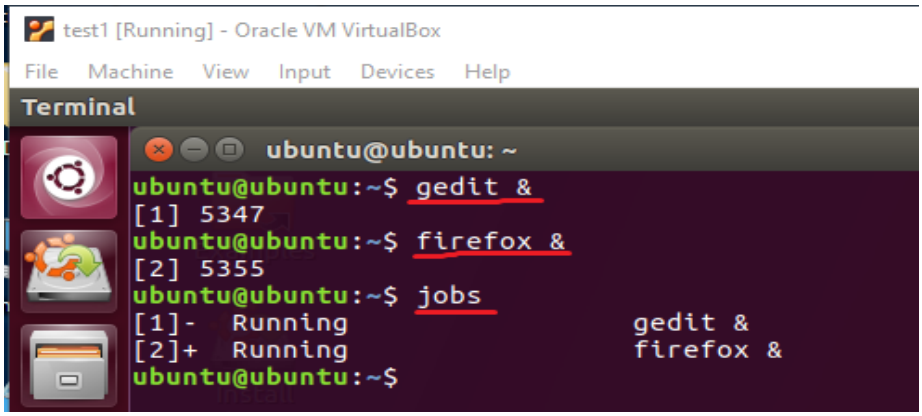
It can be noted that gedit is opened as foreground process and control does not returns to terminal unless it is closed. Now, we start the gedit again as a background process.




It can be seen that after starting the gedit program control returns to the terminal and user can interact with both terminal and gedit.

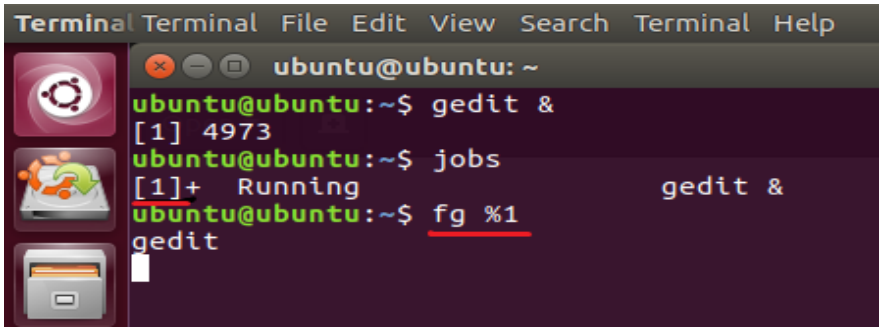
## 1.4 jobs command

The shell's job control facility also gives us a way to list the jobs that have been launched from our terminal. Using the **jobs** command, we can see this list.
In the following example, we first launch two jobs in background and then use the jobs command to view the running jobs.



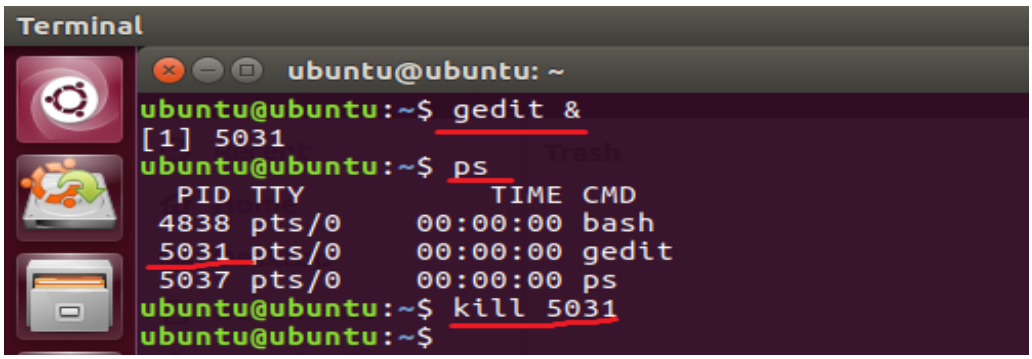

## 1.5 Bringing a process to the foreground

A process in the background is immune from keyboard input, including any attempt to interrupt it with a Ctrl-c. fg command is used to bring a process to the foreground. In the following example, we start the gedit editor in background. Then use the jobs command to see the list of jobs launched and then, bring this process to the foreground.




## 1.6 Killing a process

We can kill a process using the kill command. To kill a process, we provide the process id as an argument (We could have also specified the process using a jobspec). In the following example, we start the gedit program and then kill it using kill command.



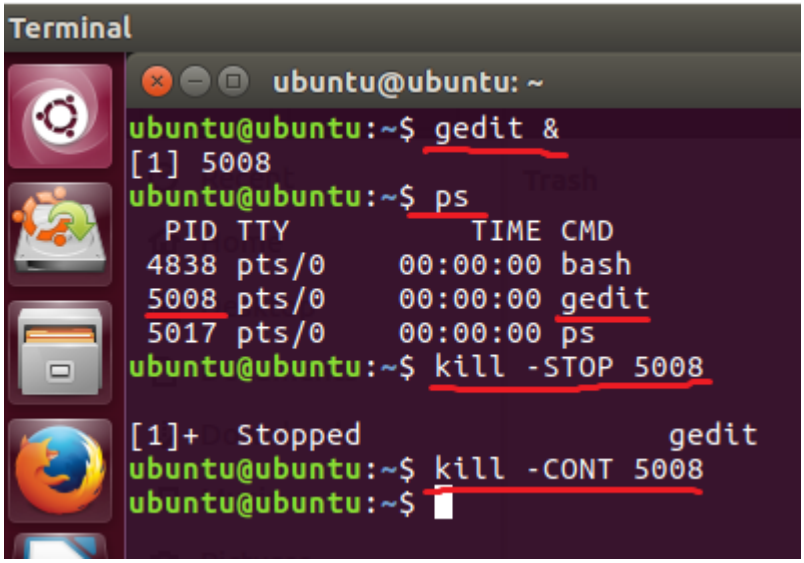

The kill command doesn't exactly "kill" processes, rather it sends them *signals*. Signals are one of several ways that the operating system communicates with programs. Programs, in turn, "listen" for signals and may act upon them as they are received. Following are most common signals that can be send with kill command.

| Signal | Meaning |
|---|---|
| INT | INT Interrupt. Performs the same function as the Ctrl-c key sent from the terminal. It will usually terminate a program. |
| TERM | Terminate. This is the default signal sent by the kill command. If a program is still "alive" enough to receive signals, it will terminate. |
| STOP | Stop. This signal causes a process to pause without terminating. |
| CONT | Continue. This will restore a process after a STOP signal. |

## 1.7 Pausing a process

Linux allows you to pause a running process rather than quitting or killing it. Pausing a process just suspends all of its operation so it stops using any of your processor power even while it still resides in memory. This may be useful when you want to run some sort of a processor intensive task, but don't wish to completely terminate another process you may have running. Pausing it would free up valuable processing time while you need it, and then continue it afterwards.

We can paus a process by using kill command with STOP option. The process id is required as an argument in kill command. In the following example, we start the gedit program in the background. Then we find the pid of gedit using ps command and then; we pause the process using kill command. Later, we can resume the process by using the kill command with CONT option.

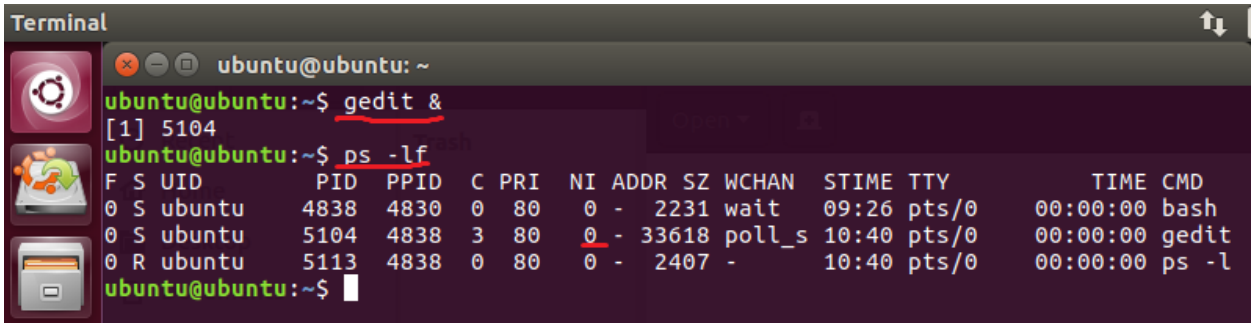



## 1.8 Changing process priority

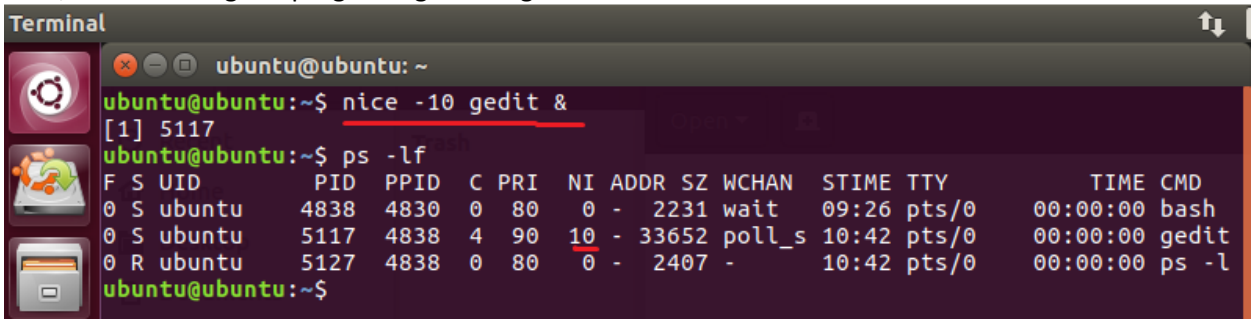
Every running process in Linux has a priority assigned to it. We can change the process priority using nice and renice utility. Nice command will launch a process with a user defined scheduling priority. Renice command will modify the scheduling priority of a running process. In Linux system priorities are 0 to 139 in which 0 to 99 for real time and 100 to 139 for users. nice value range is -20 to +19 where -20 is highest, 0 default and +19 is lowest. Relation between nice value and priority is:

PR = 20 + NI
So, the value of PR = 20 + (-20 to +19) is 0 to 39 that maps to 100-139.

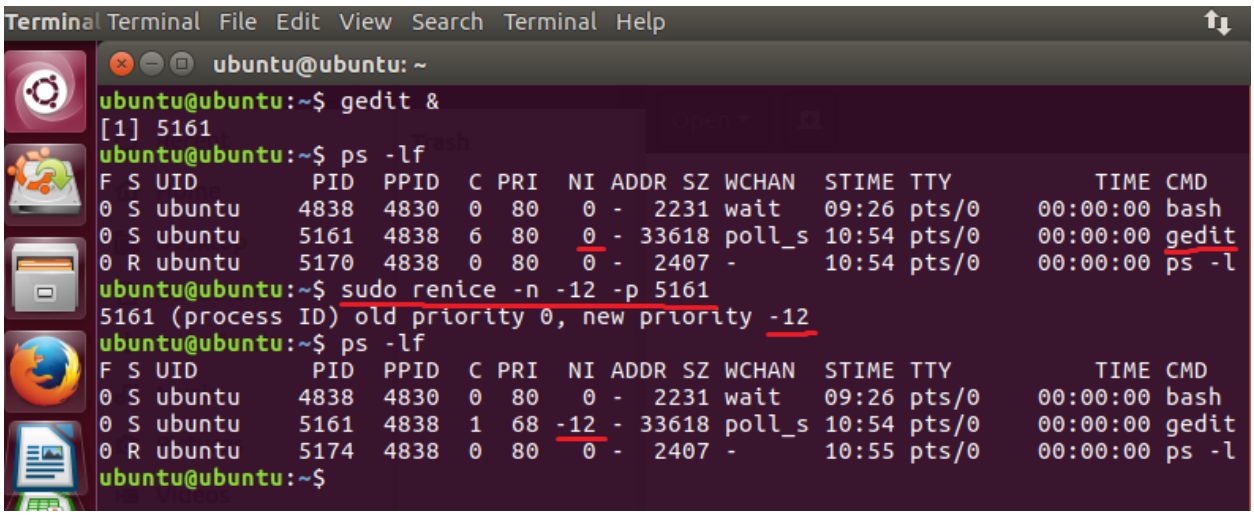
In the following example, we start the gedit program in background and see its nice value using ps command.



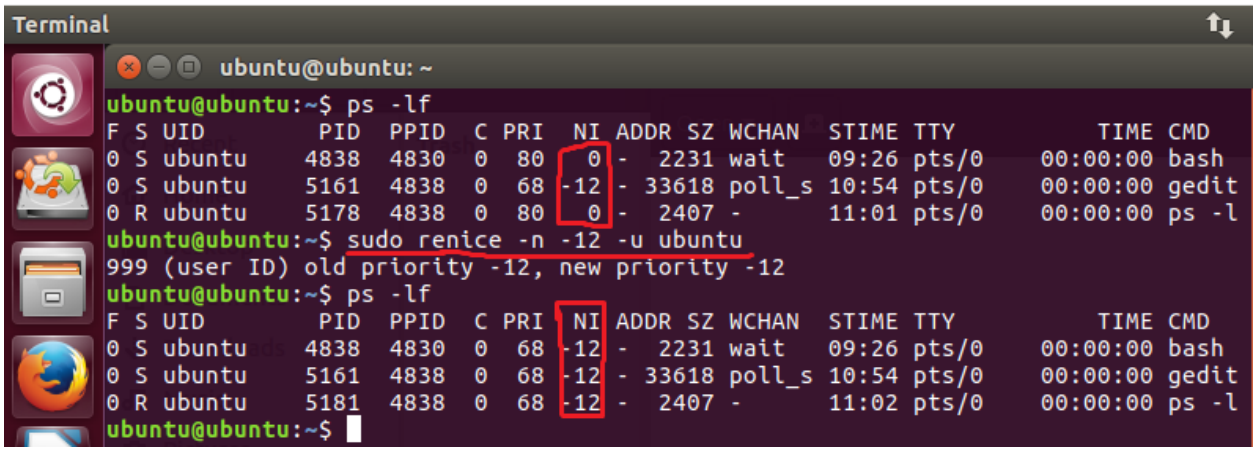

Now, we start the gedit program again using nice command.



We can change the priority of a running process using renice command. In the following example, we first start the gedit program in background. Then we change its priority using renice command.



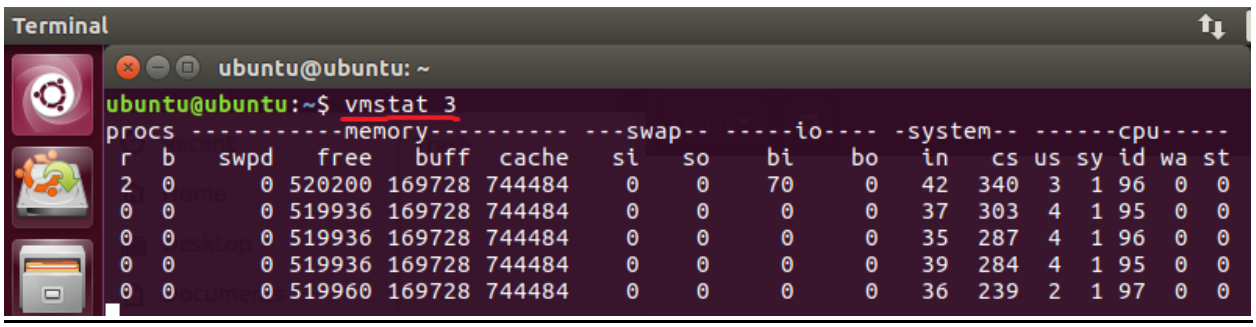

We can also use the renice command to change the priority of all of the processes belonging to a user or a group. In the following example, we change the nice value of all of the process belonging to user ubuntu.




## 1.9 Few more useful commands related to processes

### 1.9.1 vmstat command

Outputs a snapshot of system resource usage including, memory, swap and disk I/O. To see a continuous display, follow the command with a time delay (in seconds) for updates. For example: vmstat 5.



### 1.9.2 xload command

Displays the system load over time.

### 1.9.3 tload command

This command is similar to xload but displays output in the terminal.

## 2. Compiling and Executing C++ Programs

Compiling is the way toward making an interpretation of source code into the local language of the PC's processor. The PC's processor works at an extremely basic level, executing programs in what is called machine language. This is a numeric code that portrays extremely little activities, for example, "include this byte," "point to this area in memory," or "duplicate this byte". Each of these instructions is expressed in binary which were hard to write. This issue was overwhelmed by the appearance of assembly language, which supplanted the numeric codes with (marginally) simpler to utilize character mnemonics, for example, CPY (for duplicate) and MOV (for move). Programs written in assembly language are processed into machine language by a program called an assembler.

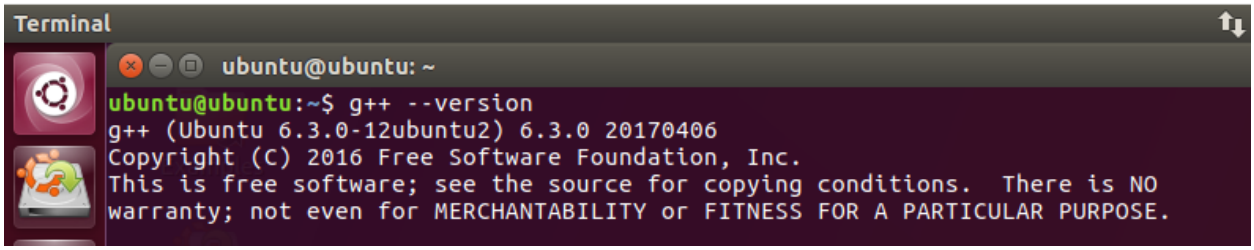
We next come to what are called high-level languages. They are called this since they enable the developer to be less worried about the details of what the processor is doing and more with taking care of the current issue. Programs written in high-level programming languages are converted into machine language by processing them with another program, called a compiler.

#### a. C++ compiler for Linux

C++ is a general-purpose programming language and widely used nowadays for competitive programming. It has imperative, object-oriented and generic programming features. C++ runs on lots of platform like Windows, Linux, Unix, Mac etc. Before we start programming with C++. We will need an environment to be set-up on our local computer to compile and run our C++ programs successfully. GNU C++ Compiler (g++) is a compiler in Linux which is used to compile C++ programs. It compiles both files with extension .c and .cpp as C++ files.

##### i. Installing g++ compiler

By default, g++ is provided with most of the Linux distributions. We can find the details of installed g++ compiler by writing the following command:




If g++ is not installed on your system; then it can be installed by writing the following commands

```
$ sudo apt-get update
$ sudo apt install g++
```

## 2.2 Compiling C++ program

We can compile a C++ program using the following command

```
$ sudo g++ source-file.cpp
```

The above command will generate an executable file a.out. We can use -o option to specify the output file name for the executable.

```
$ sudo g++ source-file.cpp -o executable-file
```

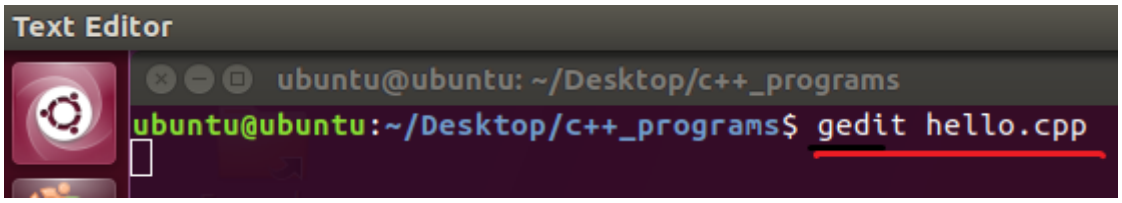
## 2.3 Running a C++ program

Onc a C++ program is successfully compiled, we can execute the created executable file by using the following command:

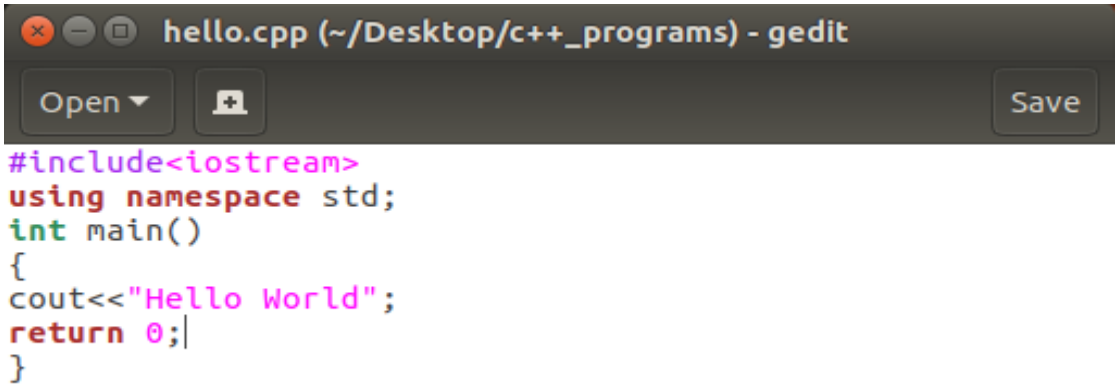
```
$ ./executable-file
```

In the following example, we write a simple C++ program that displays a Hello World message and the then compile and execute this program using the above commands. To write the program, we use the gedit text editor. The source code file is saved with .cpp extension.

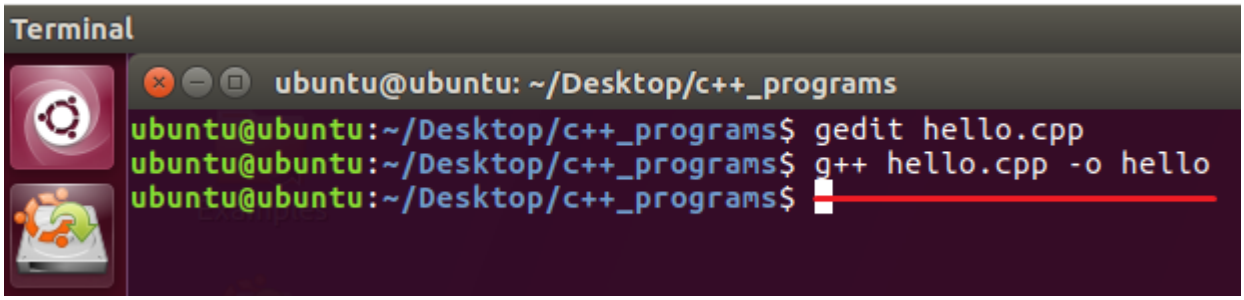
Open the gedit editor and pass the name of the file (hello.cpp) to be created



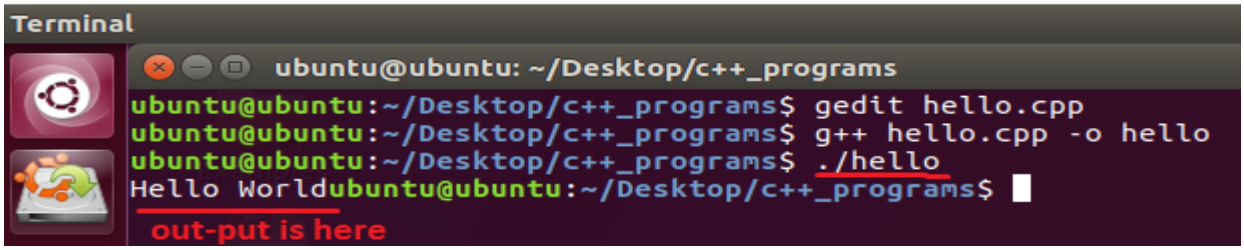

Write the following code




Now, close the source file and write the following command to compile hello.cpp




To execute the hello executable file, write the following command.




## 2.4 Passing command-line arguments to a C++ program

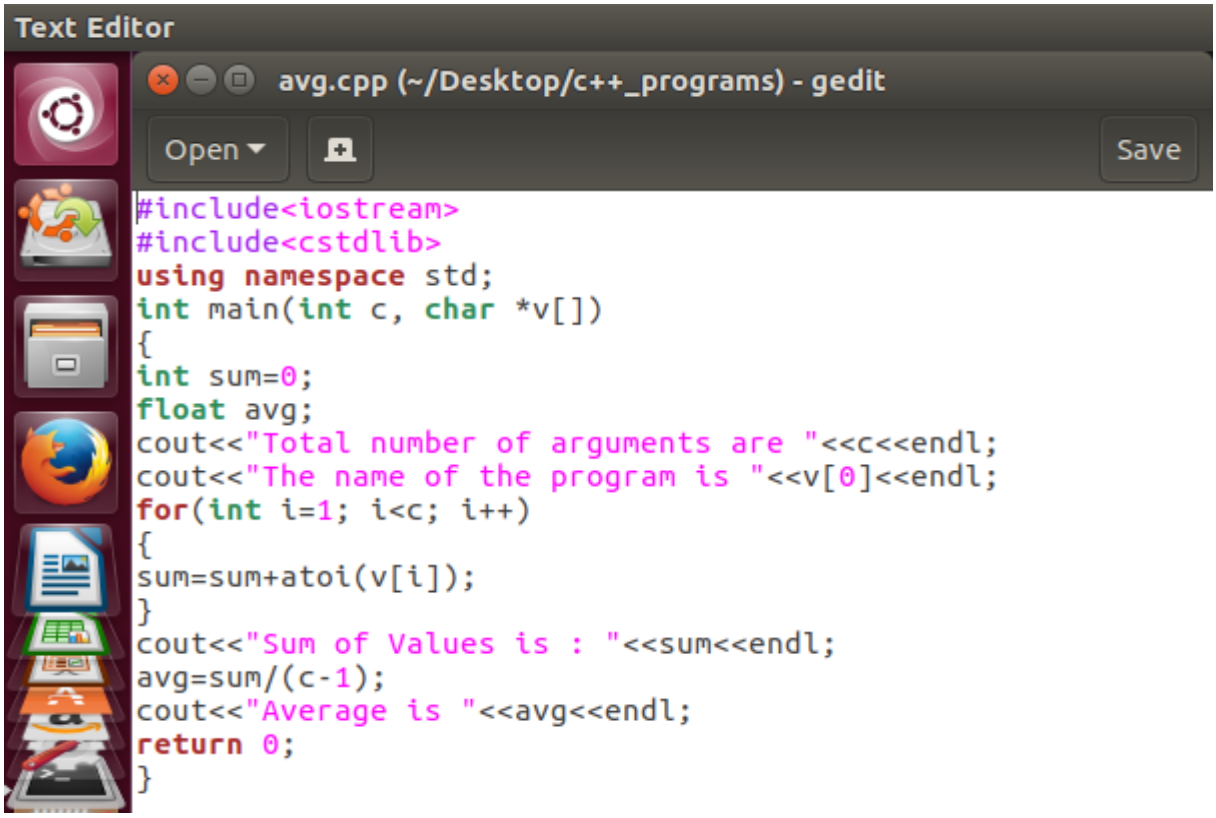
Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C++ programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main () method.

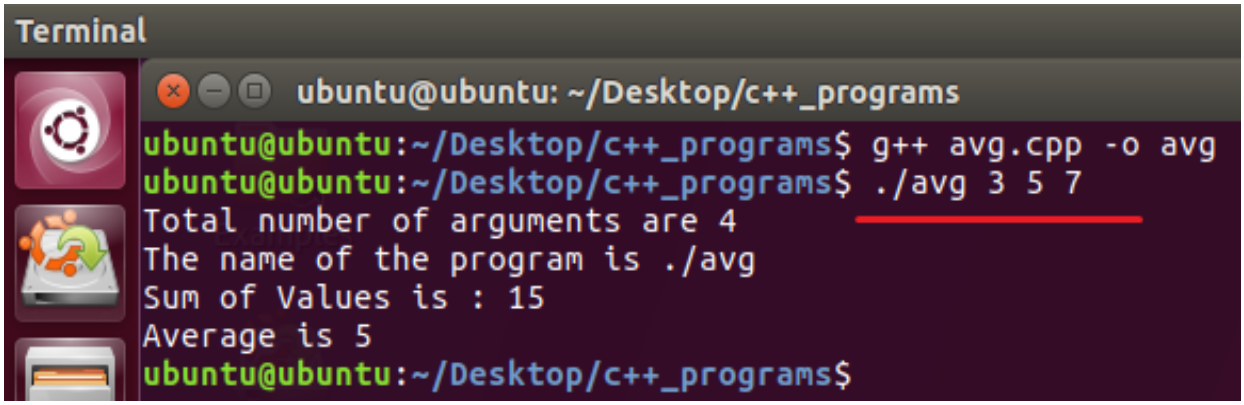
To pass command line arguments, we typically define main () with two arguments: first argument counts the number of arguments on the command line and the second is a pointer array which holds pointers of type char which points to the arguments passed to the program. The syntax to define the main method is

int main (int  argc, char *argv[])

Here, argc variable will hold the number of arguments pass to the program while the argv will contain pointers to those variables. argv[0] holds the name of the program while argv[1] to argv[argc] hold the arguments. Command-line arguments are given after the name of the program in command-line shell of Operating Systems. Each argument separated by a space. If a space is included in the argument, then it is written in “”.

In the following example, we calculate the average of numbers; passed in the command-line. Write the following code and save the file named as avg.cpp. Compile the avg.cpp and create an executable file named avg and execute it. While writing the command to execute avg pass the numbers whose average is required. It is shown below.





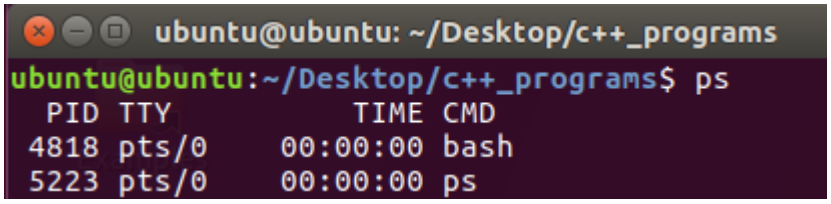

## 2) Stage a1 (apply)

## Lab Activities

**Activity 1:**
This activity is related to process monitoring in Linux

1. Display all of the process in current shell
2. Display every active process in the system
3. Provide a full-format listing of process owned by you
4. Display a full-format listing of processes owned by user ubuntu
5. Display a process with id 1
6. Display the dynamic view of the current processes in the system and set the refresh interval 0.5 second
7. Display the dynamic view of the processes owned by user with id 999 (or name ubuntu)
8. Start the gedit program in background and then bring it foreground
9. Suspend the gedit program
10. Resume the gedit program

**Solution:**

1. 

2.

```
ubuntu@ubuntu:~/Desktop/c++_programs$ ps -A
  PID TTY          TIME CMD
    1 ?        00:00:06 systemd
    2 ?        00:00:00 kthreadd
    4 ?        00:00:00 kworker/0:0H
    6 ?        00:00:00 ksoftirqd/0
```

3.

```
ubuntu@ubuntu:~/Desktop/c++_programs$ ps -lx
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4   999  1593     1  20   0   9632  6344 ep_pol Ss   ?          0:00 /lib/systemd/systemd --user
5   999  1594  1593  20   0  12972  1472 -      S    ?          0:00 (sd-pam)
0   999  1600  1593  20   0   7088  4592 ep_pol Ss   ?          0:01 /usr/bin/dbus-daemon --session --address=systemd: --no
for
```

4.

```
ubuntu@ubuntu:~/Desktop/c++_programs$ ps -lu ubuntu
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S   999  1593     1  0  80   0 -  2408 ep_pol ?        00:00:00 systemd
5 S   999  1594  1593  0  80   0 -  3243 -      ?        00:00:00 (sd-pam)
0 S   999  1600  1593  0  80   0 -  1772 ep_pol ?        00:00:01 dbus-daemon
```

5.

```
ubuntu@ubuntu:~/Desktop/c++_programs$ ps -lp 1
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 -  6818 -      ?        00:00:06 systemd
```

6.

```
ubuntu@ubuntu:~/Desktop/c++_programs$ top -d 0.5

top - 08:54:14 up  3:34,  1 user,  load average: 0.13, 0.04, 0.
Tasks: 153 total,   1 running, 152 sleeping,   0 stopped,   0 z
%Cpu(s):  5.9 us,  2.0 sy,  0.0 ni, 92.2 id,  0.0 wa,  0.0 hi,
KiB Mem :  1823696 total,   444780 free,   390968 used,   98794
KiB Swap:        0 total,        0 free,        0 used.  115920

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM
 4740 ubuntu    20   0  308300 137772  74772 S  4.0  7.6
```

7.

```
ubuntu@ubuntu:~/Desktop/c++_programs$ top -u 999

top - 08:54:36 up  3:34,  1 user,  load average: 0.09, 0.04, 0.
Tasks: 153 total,   1 running, 152 sleeping,   0 stopped,   0 z
%Cpu(s):  2.0 us,  0.3 sy,  0.0 ni, 97.7 id,  0.0 wa,  0.0 hi,
KiB Mem :  1823696 total,   444808 free,   390940 used,   98794
KiB Swap:        0 total,        0 free,        0 used.  115923
```

8.

```
ubuntu@ubuntu:~/Desktop/c++_programs$ gedit &
[1] 5230
ubuntu@ubuntu:~/Desktop/c++_programs$ fg %1
gedit
```

9.

```
ubuntu@ubuntu:~/Desktop/c++_programs$ kill -STOP 5249
```

10.

```
ubuntu@ubuntu:~/Desktop/c++_programs$ kill -CONT 5249
```

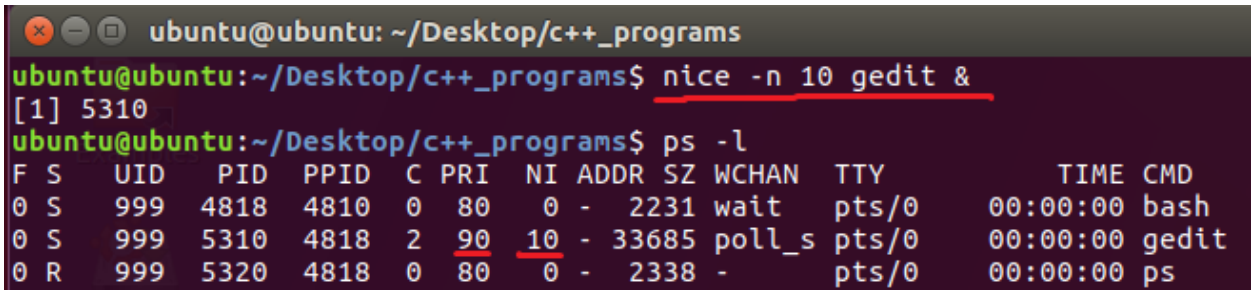
## Activity 2:

Perform the following tasks

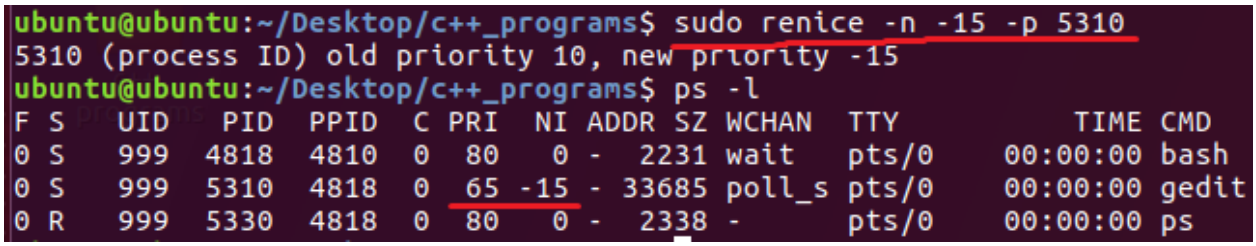
11. Start the gedit program with priority 90
12. Reset the priority of gedit to 65

## Solution:

1.




2.




## Activity 3:

Write a program in C++ that find the maximum and minimum number from an array.

## Solution:

```
#include<iostream>
using namespace std;
int main ()
{
  int arr[10], n, i, max, min;
  cout << "Enter the size of the array : ";
  cin >> n;
  cout << "Enter the elements of the array : ";
     for (i = 0; i < n; i++)
           cin >> arr[i];
       max = arr[0];
   for (i = 0; i < n; i++)
   {
      if (max < arr[i])
         max = arr[i];
   }
   min = arr[0];
   for (i = 0; i < n; i++)
   {
      if (min > arr[i])
         min = arr[i];
```

```
   }
   cout << "Largest element : " << max;
   cout << "Smallest element : " << min;
   return 0;
}
```

## Activity 4:

Change the above program such that it accepts the input at command-line.

## Solution:

```
#include<iostream>
#include<cstdlib>
using namespace std;
int main (int a_c, char *arr[ ])
{
  int arr[10], n, i, max, min,cout,num;
        count=a_c;
        max = atoi(arr[1]);
   for (i = 0; i < n; i++)
   {
      num=atoi(arr[i]);
      if (max <num)
         max = num;
   }
   min = atoi(arr[1]);
   for (i = 0; i < n; i++)
   {
     num=atoi(arr[i]);
      if (min > num)
         min = num;
   }
   cout << "Largest element : " << max;
   cout << "Smallest element : " << min;
   return 0;
}
```

# LAB # 06

## Statement of Purpose:

This lab describes how a program can create, terminate, and control processes using system calls. We will start with the some basic process creation commands and later will execute complex process termination, and wait system calls. Moreover, you will learn how to create orphans and zombie processes and how to replace one process with another during its execution using exec( ).

## Activity Outcomes:

This lab teaches you the following topics:

- Process creation system call fork ( )
- Exec system call
- Wait system call
- Sleep system call

## Instructor Note:

As pre-lab activity, introduce students to basic process creation using fork and exec.

## 1) Stage J (Journey)

# 1. Process Creation Concepts

Processes are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.

Processes are organized hierarchically. Each process has a **parent process**, which explicitly arranged to create it. The processes created by a given parent are called its **child processes**. A child inherits many of its attributes from the parent process.

A **process ID** number names each process. A unique process ID is allocated to each process when it is created. The lifetime of a process ends when its termination is reported to its parent process; at that time, all of the process resources, including its process ID, are freed.

To monitor the state of your processes under Unix use the **ps** command:

**ps [-option]**

Used without options this produces a list of all the processes owned by you and associated with your terminal.

These are some of the column headings displayed by the different versions of this command.

| | | | | |
|---|---|---|---|---|
| **PID** | SZ(size in Kb) | TTY(controlling terminal) | TIME(used by CPU) | COMMAND |

This PID is the process ID which you will b using to identify parent and child processes.

## Process Identification:

The **pid_t** data type represents process IDs which is basically a signed integer type (int). You can get the process ID of a process by calling:

**getpid()** - returns the process ID of the parent of the current process (the parent process ID).

**getppid()** - returns the process ID of the parent of the current process (the parent process ID).

Your program should include the header files ‘unistd.h’ and ‘sys/types.h’ to use these functions.

Function: pid_t getpid (void)

The `getpid()` function returns the process ID of the current process.

Function: pid_t getppid (void)

The `getppid()` function returns the process ID of the parent of the current process.

## 1.1 fork () | Process Creation

Processes are created with the **fork ()** system call (so the operation of creating a new process is sometimes called forking a process). The child process created by fork is a copy of the original parent process, except that it has its **own process ID**.

After forking a child process, both the parent and child processes continue to execute normally. If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling **wait () or waitpid ().** These functions give you limited information about why the child terminated--for example, its exit status code.

A newly forked child process continues to execute the same program as its parent process, at the point where the fork call returns. You can use the **return value** from fork to tell whether the program is running in the parent process or the child process.

When a **child process terminates**, its death is communicated to its parent so that the parent may take some appropriate action. If the fork () operation is **successful**, there are then both parent and child processes and both see **fork return**, but with different values: it returns a value of 0 in the child process and returns the child's process ID in the parent process. If process creation failed, fork returns a value of -1 in the parent process and no child is created.

**Example 1 – Single fork() :**

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
printf("Hello World!\n");
fork( );
printf("I am after forking\n");
printf("\tI am process %d.\n", getpid( ));
}
```

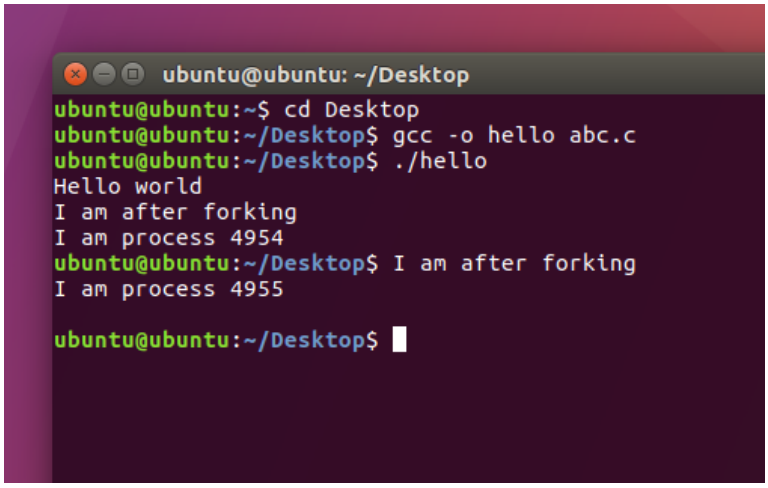
Write this program and run using the following commands:

**gedit hello.c //write the above C code and close the editor**

**gcc -o hello hello.c //compile using built-in GNU C compiler**

**./hello //run the code**

**Output:**




When this program is executed, it first prints Hello World! . When the fork is executed, an identical process called the child is created. Then both the parent and the child process begin execution at the next statement.

**Note that:**

1. There is no guarantee which process will print I am a process first.

2. The child process begins execution at the statement immediately after the fork, not at the beginning of the program.

3. A parent process can be distinguished from the child process by examining the return value of the fork call. Fork returns a zero to the child process and the process id of the child process to the parent.

**Example 2:**

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
int pid;
printf("Hello World!\n");
printf("I am the parent process and pid is : %d .\n",getpid());
printf("Here i am before use of forking\n");
pid = fork();
printf("Here I am just after forking\n");
if (pid == 0)
printf("I am the child process and pid is :%d.\n",getpid());
else
printf("I am the parent process and pid is: %d .\n",getpid());
}
```

Executing the above code will give the following output:

```
ubuntu@ubuntu:~/Desktop$ gedit fork1.c
ubuntu@ubuntu:~/Desktop$ gcc -o fork1 fork1.c
ubuntu@ubuntu:~/Desktop$ ./fork1
Hello World!
I am the parent process and pid is : 5292 .
Here i am before use of forking
Here I am just after forking
I am the parent process and pid is: 5292 .
ubuntu@ubuntu:~/Desktop$ Here I am just after forking
I am the child process and pid is :5293.
```

This programs give Ids of both parent and child process.

The above output shows that parent process is executed first and then child process is executed.

**Example 3 – Multiple Fork() :**

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
printf("Here I am just before first forking statement\n");
fork();
printf("Here I am just after first forking statement\n");
fork();
printf("Here I am just after second forking statement\n");
printf("\t\tHello World from process %d!\n", getpid());
}
```

**Output:**

```
ubuntu@ubuntu:~/Desktop$ gedit fork2.c
ubuntu@ubuntu:~/Desktop$ gcc -o fork2 fork2.c
ubuntu@ubuntu:~/Desktop$ ./fork2
Here I am just before first forking statement
Here I am just after first forking statement
Here I am just after second forking statement
                Hello World from process 5217!
ubuntu@ubuntu:~/Desktop$ Here I am just after second forking statement
                Hello World from process 5219!
Here I am just after first forking statement
Here I am just after second forking statement
                Hello World from process 5218!
Here I am just after second forking statement
                Hello World from process 5220!
```

**Example 4:**

**Using fork ( ) inside a function**

**Predict the output and verify after executing the following code**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
   // child process because return value zero
   if (fork() == 0)
      printf("Hello from Child!\n");

   // parent process because return value non-zero.
   else
      printf("Hello from Parent!\n");
}
int main()
{
   forkexample();
   return 0;
}
```

**Output:**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    // child process because return value zero
    if (fork() == 0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

```
Hello from Parent!
Hello from Child!
```

### 1.2 Wait () | Process Completion

A process **wait ()** for a child process to terminate or stop, and determine its status.

These functions are declared in the header file "**sys/wait.h**"

**1. wait ():**

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction. Child process may terminate due to any of these:

- It calls exit();
- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate.

Wait () will force a parent process to wait for a child process to stop or terminate. Wait () return the pid of the child or -1 for an error.

## 2. Exit ():

Exit () terminates the process which calls this function and returns the exit status value. Both UNIX and C (forked) programs can read the exit status value.

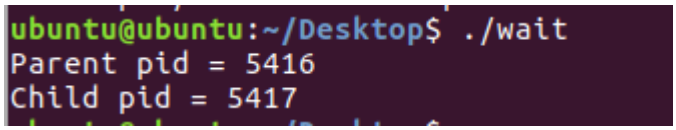
By convention, **a status of 0** means normal termination. Any other value indicates an error or unusual occurrence.

**Example 1:**

```
// C program to demonstrate working of wait() and exit()
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
int main() {
   pid_t cpid;
   if (fork()== 0)
      exit(0);          /* terminate child – exit (0) means normal termination */
   else
      cpid = wait(NULL); /* parent will wait until child terminates */
   printf("Parent pid = %d\n", getpid());
```

```
    printf("Child pid = %d\n", cpid);
    return 0;
}
```

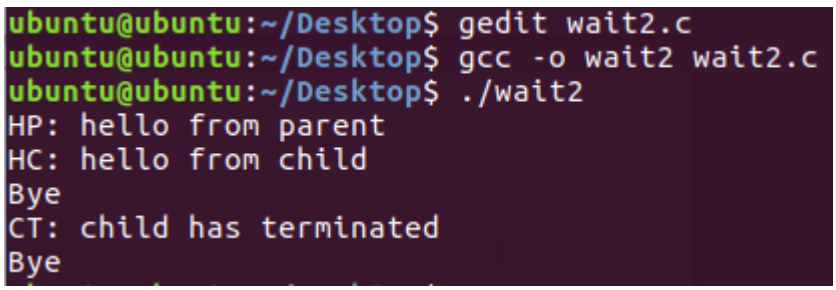
**Output:**




**Example 2:**

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
    if (fork()== 0)
        printf("HC: hello from child\n");
    else {
        printf("HP: hello from parent\n");
        wait(NULL);  /*waiting till child terminates
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    return 0;
}
```

**Output:**



## 3. Sleep ( ):

A process may suspend for a period of time using the sleep () command:

**Orphan Process:**

When a parent dies before its child, the child is automatically adopted by the original "init" process whose PID is 1.

**Zombie Process:**

A process that terminates cannot leave the system until its parent accepts its return code. If its parent process is already dead, it'll already have been adopted by the "init" process, which always accepts its children's return codes (orphan). However, if a process's parent is alive but never executes a wait ( ), the process's return code will never be accepted and the process will remain a *zombie*.

**Creation of orphan and zombie is left as an exercise for students in activities at the end of lab.**

## 4. Ecec ( ):

The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program. It comes under the header file **unistd.h.** The program that the process is executing is called its process image. Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image.

There are a lot of exec functions included in exec family of functions, for executing a file as a process image e.g. execv(), execvp() etc. You can use these functions to make a child process execute a new program after it has been forked. The functions in this family differ in how you specify the arguments, but otherwise they all do the same thing.

**Execv () :**
Using this command, the created child process does not have to run the same program as the parent process does. The exec type system calls allow a process to run any program files, which include a binary executable or a shell script .
Syntax:

```
int execv (const char *file, char *const argv[]);
```

**file:** points to the file name associated with the file being executed.
**argv:** is a null terminated array of character pointers.

**Example 1:**
Let us see a small example to show how to use execv () function in C. We will have two .C files: example.c and hello.c and we will replace the example.c with hello.c by calling execv() function in example.c

example.c

**CODE:**

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
    printf("Back to example.c");
    return 0;
}
```

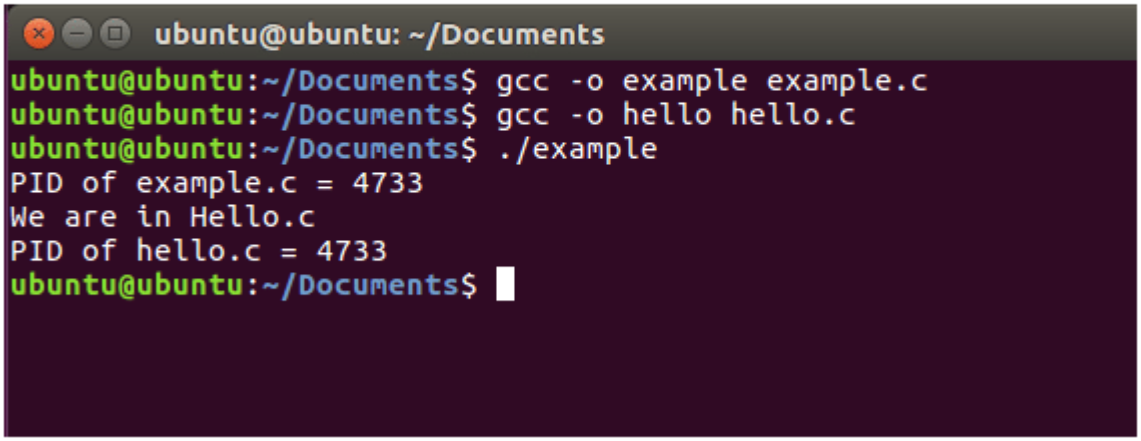
hello.c

**CODE:**

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("We are in Hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}
```

**Output:**

```
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
```



## 2) Stage a1 (apply)

# Lab Activities:

## Activity 1:

In this activity, you are required to perform tasks given below:

1. Print something and Check id of the parent process
2. Create a child process and print child process id in parent process
3. Create a child process and print child process id in child process

## Solution:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main()
{
int forkresult;
printf("%d: I am the parent. Remember my number!\n", getpid());
printf("%d: I am now going to fork ... \n", getpid());
forkresult = fork();
if (forkresult != 0)
{ /* the parent will execute this code */
printf("%d: My child's pid is %d\n", getpid(), forkresult);
}
else /* forkresult == 0 */
{ /* the child will execute this code */
printf("%d: Hi! I am the child.\n", getpid());
}
printf("%d: like father like son. \n", getpid());
return 0;
}
```

```
ubuntu@ubuntu:~/Desktop$ gcc -o activity1 activity1.c
ubuntu@ubuntu:~/Desktop$ ./activity1
5618: I am the parent. Remember my number!
5618: I am now going to fork ...
5618: My child's pid is 5619
5618: like father like son.
ubuntu@ubuntu:~/Desktop$ 5619: Hi! I am the child.
5619: like father like son.
```

## Activity 2:

**1. Create a process and make it an orphan.**

Hint: To, illustrate this insert a sleep statement into the child's code. This ensured that the parent process terminated before its child.

**Steps to create an orphan process:**

1. print something and get its pid and ppid

2. create a child process

3. Now as a parent process print parent id and id of child process

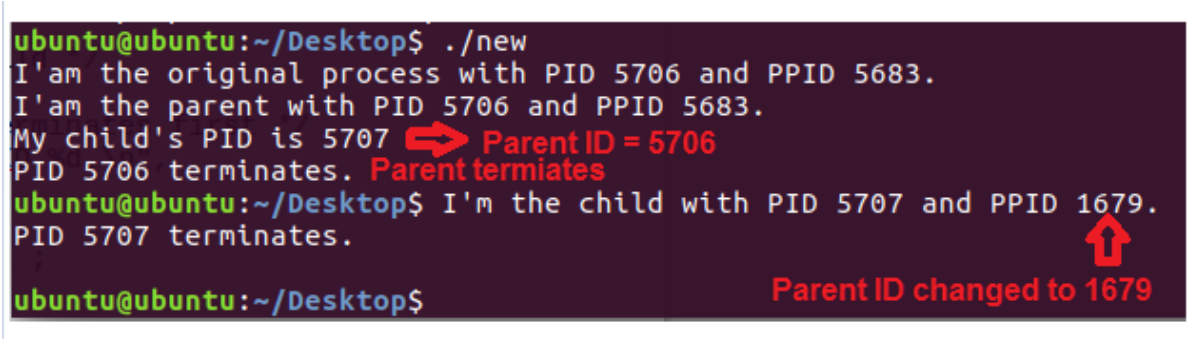
4. Make child sleep for 5 seconds

5. Now while child is sleeping parent will terminate. Print parent id of child to make sure it is orphaned (PPID has been changed)

## Solution:

```
#include <stdio.h>
main()
{
int pid ;
printf("I'am the original process with PID %d and PPID %d.\n",
getpid(), getppid()) ;
pid = fork ( ) ; /* Duplicate. Child and parent continue from here */
if ( pid != 0 ) /* pid is non-zero,so I must be the parent*/
{
printf("I'am the parent with PID %d and PPID %d.\n",
getpid(), getppid()) ;
printf("My child's PID is %d\n", pid ) ;
}
else /* pid is zero, so I must be the child */
{
sleep(4); /* make sure that the parent terminates first */
printf("I'm the child with PID %d and PPID %d.\n",
getpid(), getppid()) ;
}
printf ("PID %d terminates.\n", getpid()) ;
}
```

**Output:**



## Activity 3:

**Create a process and make it a Zombie.**

1. Execute fork to create a child
2. In parent process (using if statement) Create an infinite loop so that it never terminates and never executes wait ()
3. Make parent sleep for 100 sec
4. Terminate child process exit using exit.

Now this child is a zombie because no parent is waiting for him

To view status of processes, use the following commands on command line:

1. Execute the c code in background using ./abc &
2. View process status using 'ps -lf'
3. You will see zombie process with state 'Z' in STAT column
4. Get parent id and kill parent process using "kill (parent id)" command
5. again execute 'ps -lf'
6. Note that Zombie is gone now

## Solution:

```
#include <stdio.h>
main ( )
{
int pid ;
pid = fork(); /* Duplicate. Child and parent continue from here */
if ( pid != 0 ) /* pid is non-zero, so I must be the parent */
{
while (1) /* Never terminate and never execute a wait ( ) */
sleep (100) ; /* stop executing for 100 seconds */
}
else /* pid is zero, so I must be the child */
{
exit (42) ; /* exit with any number */
}
}
```

```
ubuntu@ubuntu:~/Desktop$ ./zombie &
[1] 5769
ubuntu@ubuntu:~/Desktop$ ps -lf
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S ubuntu      5759    5750  0  80   0 -  2103 wait   16:40 pts/0    00:00:00 bash
0 S ubuntu      5769    5759  0  80   0 -   522 hrtime 16:40 pts/0    00:00:00 ./zom
1 Z ubuntu      5770    5769  0  80   0 -     0 -      16:40 pts/0    00:00:00 [zomb
0 R ubuntu      5771    5759  0  80   0 -  2407 -      16:40 pts/0    00:00:00 ps -l
ubuntu@ubuntu:~/Desktop$ kill 5769
[1]+  Terminated              ./zombie
ubuntu@ubuntu:~/Desktop$ ps -lf
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S ubuntu      5759    5750  0  80   0 -  2231 wait   16:40 pts/0    00:00:00 bash
0 R ubuntu      5772    5759  0  80   0 -  2407 -      16:42 pts/0    00:00:00 ps -l
ubuntu@ubuntu:~/Desktop$
```

## Activity 4:

Write a C/C++ program in which a parent process creates a child process using a fork() system call. The child process takes your age as input and parent process prints the age.

## Solution:

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main()
{
int i;
pid_t p=fork();
if (p==0)
        {
        printf("%d: I am the child\n", getpid());
        printf("enter your age" );
        scanf("%d",&i);
        exit(i);
        }
else
{
wait(&i);
printf("%d: Hello I am parent \n", getpid());
printf("%d is your age", i/256);
}
return 0;
}
```

## Output:

```
ubuntu@ubuntu:~/Desktop$ ./act
5941: I am the child
enter your age4
5940: Hello I am parent
4 is your ageubuntu@ubuntu:~/Desktop$
```

## Activity 5:

5. Write a C/C++ program that asks user to enter his name and his university name. Within the same program, execute another program that asks the user to enter his degree name and department name.
6. Hint: write 2 separate programs and execute using  execv()

## Solution:

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
char uni[100];
char name[100];

        printf("Enter your name\n");
        scanf("%s",name);
        printf("enter your University name \n" );
        scanf("%s",uni);
        char *args[] = {"dept", "C", "PROGRAMMING" , NULL};
        execv("./dept",args);
return 0;
}
```

## Output:

```
ubuntu@ubuntu:~/Desktop$ gedit uni.c
ubuntu@ubuntu:~/Desktop$ gcc -o uni uni.c
ubuntu@ubuntu:~/Desktop$ ./uni
Enter your name
qurat
enter your University name
comsats
Enter your degree name
MS
enter your department name
Computer science
ubuntu@ubuntu:~/Desktop$
```

# 3) Stage V (verify)

## Home Activities:

Practice fork, exec, sleep and wait at home.

## 4) Stage $a_2$ (assess)

**Activity assessment and Viva voce at the end of lab.**

# LAB # 07

## Statement Purpose:

This lab will explain to you the inter process communication through shared memory and message passing in Linux.

## Activity Outcomes:

This lab describes

- How two processes communicate via shared memory
- How two processes communicate via message passing

## Instructor Note:

As pre-lab activity, introduce students to the concept of inter-process communication

## 1) Stage J (Journey)

A process can be of two type:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

1. Shared Memory
2. Message passing

The Figure below shows a basic structure of communication between processes via shared memory method and via message passing.

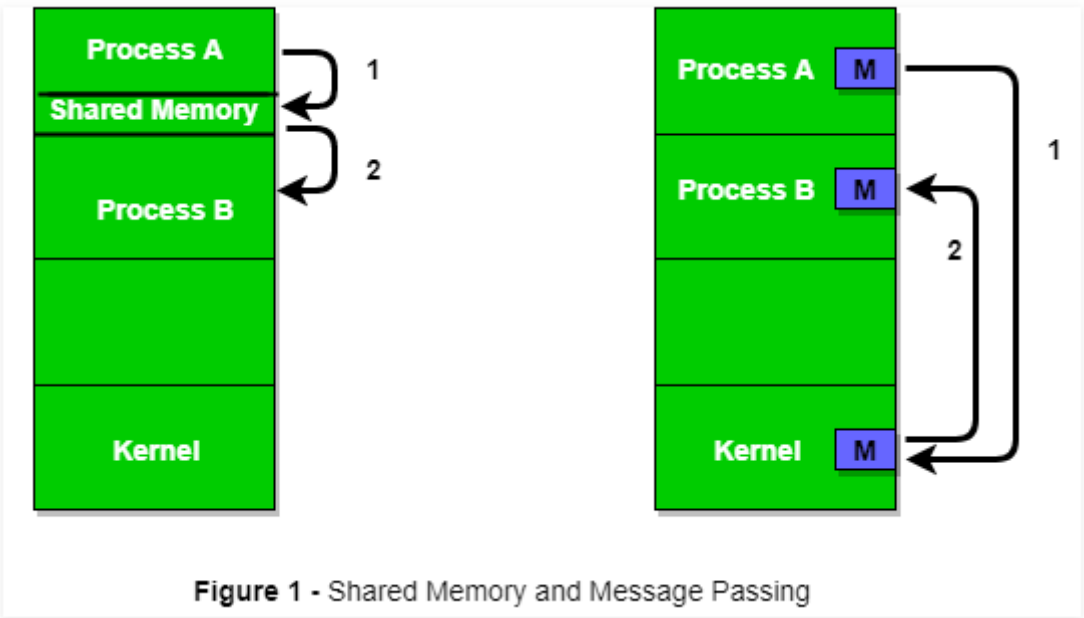



**Figure 1 -** Shared Memory and Message Passing

An operating system can implement both methods of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process 2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

Let's discuss an example of communication between processes using shared memory method.

**i. Shared Memory Method**

**Ex: Producer-Consumer problem**

There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Similarly, the consumer first check for the availability of the item and if no item is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it. The pseudo code are given below:

**Shared Data between the two Processes**

```
#define buff_max 25
#define mod %

    struct item{

        // diffrent member of the produced data
        // or consumed data
        ---------
    }

    // An array is needed for holding the items.
    // This is the shared place which will be
    // access by both process
    // item shared_buff [ buff_max ];

    // Two variables which will keep track of
    // the indexes of the items produced by producer
    // and consumer The free index points to
    // the next free index. The full index points to
    // the first full index.
    int free_index = 0;
    int full_index = 0;
```

**Producer Process Code**

```
item nextProduced;

    while(1){

        // check if there is no space
        // for production.
        // if so keep waiting.
        while((free_index+1) mod buff_max == full_index);

        shared_buff[free_index] = nextProduced;
        free_index = (free_index + 1) mod buff_max;
    }
```

**Consumer Process Code**

```
item nextConsumed;

    while(1){

        // check if there is an available
        // item  for consumption.
        // if not keep on waiting for
        // get them produced.
        while((free_index == full_index);

        nextConsumed = shared_buff[full_index];
        full_index = (full_index + 1) mod buff_max;
    }
```

In the above code, The producer will start producing again when the (free_index+1) mod buff max will be free because if it it not free, this implies that there are still items that can be consumed by the Consumer so there is no need to produce more. Similarly, if free index and full index points to the same index, this implies that there are no item to consume.

### ii) Messaging Passing Method

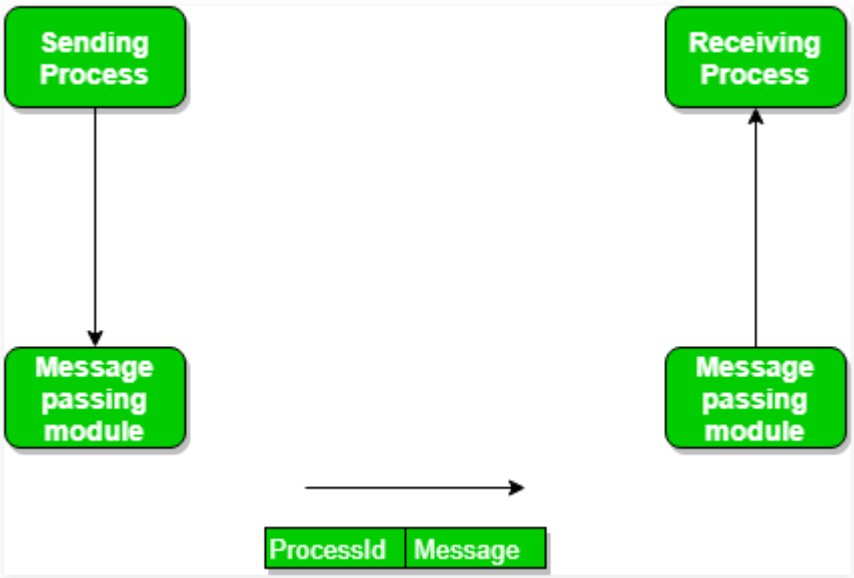
Now,We will start our discussion for the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.

  We need at least two primitives:

  – **send**(message, destinaion) or **send**(message)

  – **receive**(message, host) or **receive**(message)




The message size can be of fixed size or of variable size. if it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**

The **header part** is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

### A. IPC through shared memory

Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.
The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.

- The client reads the data from the IPC channel again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

SYSTEM CALLS USED ARE:

***ftok()****: is use to generate a unique key.*

***shmget()****: int shmget(key_t,size_tsize,intshmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.*

***shmat()****: Before you can use a shared memory segment, you have to attach yourself to it using shmat(). void *shmat(int shmid ,void *shmaddr ,int shmflg);*
*shmid is shared memory id. shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.*

***shmdt()****: When you're done with the shared memory segment, your program should detach itself from it using shmdt(). int shmdt(void *shmaddr);*

***shmctl()****: when you detach from shared memory,it is not destroyed. So, to destroy shmctl() is used. shmctl(int shmid,IPC_RMID,NULL);*

**SHARED MEMORY FOR WRITER PROCESS**

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    cout<<"Write Data : ";
    gets(str);

    printf("Data written in memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    return 0;
}
```

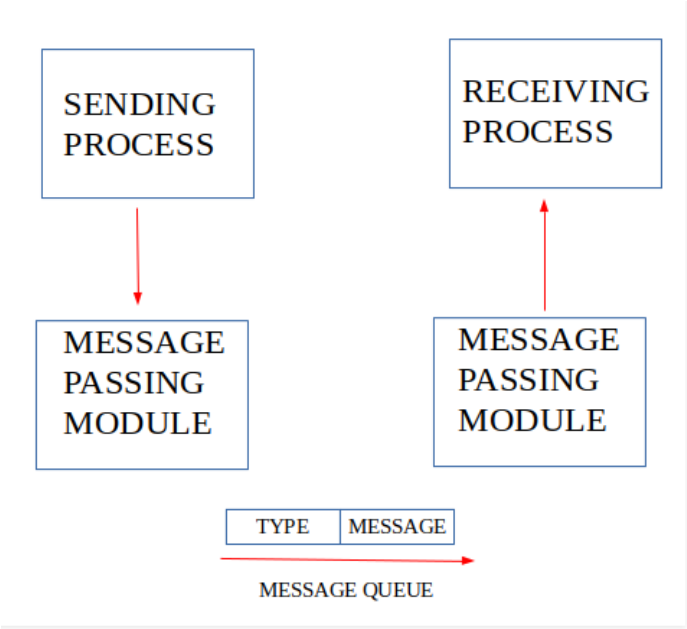
**SHARED MEMORY FOR READER PROCESS**

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Data read from memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    // destroy the shared memory
    shmctl(shmid,IPC_RMID,NULL);

    return 0;
}
```

### B. IPC using Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**. New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.




System calls used for message queues:

**ftok()**: is use to generate a unique key.

**msgget()**: either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.

**msgsnd()**: Data is placed on to a message queue by calling msgsnd().

**msgrcv()**: messages are retrieved from a queue.

**msgctl()**: It performs various operations on a queue. Generally it is use to destroy message queue.

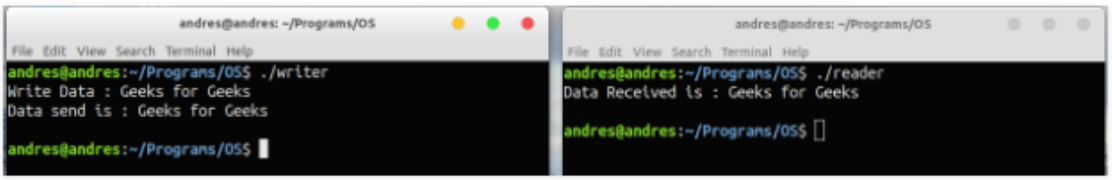
**MESSAGE QUEUE FOR WRITER PROCESS**

```c
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;

    printf("Write Data : ");
    gets(message.mesg_text);

    // msgsnd to send message
    msgsnd(msgid, &message, sizeof(message), 0);

    // display the message
    printf("Data send is : %s \n", message.mesg_text);

    return 0;
}
```

**MESSAGE QUEUE FOR READER PROCESS**

```c
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n",
                    message.mesg_text);

     // to destroy the message queue
     msgctl(msgid, IPC_RMID, NULL);

     return 0;
 }
```

Output:



## 2) Stage a1 (apply)

### Lab Activities:

1. Implement Merge sort in shared memory

**Solution:**

Given a number 'n' and a n numbers, sort the numbers using **Concurrent** Merge Sort. (Hint: Try to use shmget, shmat system calls).

**Part1: The algorithm (HOW?)**

Recursively make two child processes, one for the left half, one of the right half. If the number of elements in the array for a process is less than 5, perform a Insertion Sort. The parent of the two children then merges the result and returns back to the parent and so on. But how do you make it concurrent?

**Part2: The logical (WHY?)**

The important part of the solution to this problem is not algorithmic, but to explain concepts of Operating System and kernel.

To achieve concurrent sorting, we need a way to make two processes to work on the same array at the same time. To make things easier Linux provides a lot of system calls via simple API endpoints. Two of them are, **shmget()** (for shared memory allocation) and **shmat()** (for shared memory operations). We create a shared memory space between the child process that we fork. Each segment is split into left and right child which is sorted, the interesting part being they are working concurrently! The shmget() requests the kernel to allocate a shared page for both the processes.

```c
// C program to implement concurrent merge sort
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void insertionSort(int arr[], int n);
void merge(int a[], int l1, int h1, int h2);

void mergeSort(int a[], int l, int h)
{
    int i, len=(h-l+1);

    // Using insertion sort for small sized array
    if (len<=5)
    {
        insertionSort(a+l, len);
        return;
    }

    pid_t lpid,rpid;
    lpid = fork();
    if (lpid<0)
    {
        // Lchild proc not created
        perror("Left Child Proc. not created\n");
        _exit(-1);
    }
  else if (lpid==0)
  {
      mergeSort(a,l,l+len/2-1);
      _exit(0);
  }
  else
  {
      rpid = fork();
      if (rpid<0)
      {
          // Rchild proc not created
          perror("Right Child Proc. not created\n");
          _exit(-1);
      }
      else if(rpid==0)
      {
          mergeSort(a,l+len/2,h);
          _exit(0);
      }
  }

  int status;

  // Wait for child processes to finish
  waitpid(lpid, &status, 0);
  waitpid(rpid, &status, 0);
```

```
    // Merge the sorted subarrays
    merge(a, l, l+len/2-1, h);
}

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
       key = arr[i];
       j = i-1;

       /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
       while (j >= 0 && arr[j] > key)
       {
           arr[j+1] = arr[j];
           j = j-1;
       }
       arr[j+1] = key;
   }
}

// Method to merge sorted subarrays
void merge(int a[], int l1, int h1, int h2)
{
    // We can directly copy  the sorted elements
    // in the final array, no need for a temporary
    // sorted array.
    int count=h2-l1+1;
    int sorted[count];
    int i=l1, k=h1+1, m=0;
    while (i<=h1 && k<=h2)
    {
        if (a[i]<a[k])
            sorted[m++]=a[i++];
        else if (a[k]<a[i])
            sorted[m++]=a[k++];
        else if (a[i]==a[k])
        {
            sorted[m++]=a[i++];
            sorted[m++]=a[k++];
        }
    }

    while (i<=h1)
        sorted[m++]=a[i++];

    while (k<=h2)
        sorted[m++]=a[k++];
```

\

```
    int arr_count = l1;
    for (i=0; i<count; i++,l1++)
        a[l1] = sorted[i];
}

// To check if array is actually sorted or not
void isSorted(int arr[], int len)
{
    if (len==1)
    {
        printf("Sorting Done Successfully\n");
        return;
    }

    int i;
    for (i=1; i<len; i++)
    {
        if (arr[i]<arr[i-1])
        {
            printf("Sorting Not Done\n");
            return;
        }
    }
    printf("Sorting Done Successfully\n");
    return;
}

// To fill randome values in array for testing
// purpise
void fillData(int a[], int len)
{
    // Create random arrays
    int i;
    for (i=0; i<len; i++)
        a[i] = rand();
    return;
}

// Driver code
int main()
{
    int shmid;
    key_t key = IPC_PRIVATE;
    int *shm_array;


    // Using fixed size array.  We can uncomment
    // below lines to take size from user
    int length = 128;

    /* printf("Enter No of elements of Array:");
    scanf("%d",&length); */

    // Calculate segment length
    size_t SHM_SIZE = sizeof(int)*length;
```

```
    // Create the segment.
    if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget");
        _exit(1);
    }

    // Now we attach the segment to our data space.
    if ((shm_array = shmat(shmid, NULL, 0)) == (int *) -1)
    {
        perror("shmat");
        _exit(1);
    }

    // Create a random array of given length
    srand(time(NULL));
    fillData(shm_array, length);

    // Sort the created array
    mergeSort(shm_array, 0, length-1);

    // Check if array is sorted or not
    isSorted(shm_array, length);

      // Check if array is sorted or not
      isSorted(shm_array, length);

      /* Detach from the shared memory now that we are
         done using it. */
      if (shmdt(shm_array) == -1)
      {
          perror("shmdt");
          _exit(1);
      }

      /* Delete the shared memory segment. */
      if (shmctl(shmid, IPC_RMID, NULL) == -1)
      {
          perror("shmctl");
          _exit(1);
      }

      return 0;
}
```

Output:

```
Sorting Done Successfully
```

2. Implement chat application using message queue

## 3) Stage V (verify)

### Home Activities:

Practice the Linux commands discussed in the lab.

## 4) Stage $a_2$ (assess)

**Activity assessment and Viva voce at the end of lab.**

# LAB # 08

## Statement Purpose:

This lab will give you'll implement multithreading in C++ language using POSIX library, and implement synchronization Peterson's solution using threads.

## Activity Outcomes:

The primary objective of this lab is to implement the Thread Management Functions:

- Creating Threads
- Terminating Thread Execution
- Passing Arguments to Threads
- Thread Identifiers
- Joining Threads
- Detaching / Undetaching Threads
- Peterson's solution using threads

## Instructor Note:

https://www.geeksforgeeks.org/multithreading-in-cpp/
https://www.geeksforgeeks.org/petersons-algorithm-for-mutual-exclusion-set-2-cpu-cycles-and-memory-fence/

# 1) Stage J (Journey)

## Introduction

Multitasking is the feature that allows your computer to run two or more programs concurrently and Multithreading is a specialized form of multitasking. In general, there are two types of multitasking: process-based and thread-based.

Process-based multitasking handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A thread is a semi-process, that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

C++ does not contain any built-in support for multithreaded applications. Instead, it relies entirely upon the operating system to provide this feature.

**What are pthreads?** Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's.

Pthreads are defined as a set of C language programming types and procedure calls. Vendors usually provide a Pthreads implementation in the form of a header/include file and a library, which you link with your program.

In this lab we are going to write multi-threaded C++ program using POSIX. POSIX Threads, or Pthreads provides API which are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris.

# 2) Stage a1 (apply)

## Lab Activities:

**Creating Threads**

The following routine is used to create a POSIX thread

```
#include <pthread.h>
pthread_create (thread, attr, start_routine, arg)
```

Here, **pthread_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code. Here is the description of the parameters

| Sr.No | Parameter & Description |
|---|---|
| 1 | **Thread**<br>An unique identifier for the new thread returned by the subroutine. |
| 2 | **Attr**<br>An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values. |
| 3 | **start_routine**<br>The C++ routine that the thread will execute once it is created. |
| 4 | **Arg**<br>A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed. |

The maximum number of threads that may be created by a process is implementation dependent. Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

**Terminating Threads**

Following routine terminates a POSIX thread

```
#include <pthread.h>
pthread_exit (status)
```

Here **pthread_exit** is used to explicitly exit a thread. Typically, the pthread_exit routine is called after a thread has completed its work and is no longer required to exist.

If main finishes before the threads it has created, and exits with pthread_exit, the other threads will continue to execute. Otherwise, they will be automatically terminated when main finishes.

## Activity 1:

**Write a program in C++ to create and terminate threads.**

This simple example code creates 5 threads with the pthread_create routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit.

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

void *PrintHello(void *threadid) {
  long tid;
  tid = (long)threadid;
  cout << "Hello World! Thread ID, " << tid << endl;
  pthread_exit(NULL);
}

int main () {
  pthread_t threads[NUM_THREADS];
  int rc;
  int i;

  for( i = 0; i < NUM_THREADS; i++ ) {
    cout << "main() : creating thread, " << i << endl;
    rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);

    if (rc) {
      cout << "Error:unable to create thread," << rc << endl;
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Compile the following program using -lpthread library as follows −

```
$gcc test.cpp -lpthread
```

Now, execute your program which gives the following output −

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2
Hello World! Thread ID, 3
Hello World! Thread ID, 4
```

## Activity 2:

**Passing Arguments to Threads**

This example shows how to pass multiple arguments via a structure. You can pass any data type in a thread callback because it points to void as explained in the following example

```
 #include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

struct thread_data {
  int  thread_id;
  char *message;
};

void *PrintHello(void *threadarg) {
  struct thread_data *my_data;
  my_data = (struct thread_data *) threadarg;

  cout << "Thread ID : " << my_data->thread_id ;
  cout << " Message : " << my_data->message << endl;

  pthread_exit(NULL);
}

int main () {
  pthread_t threads[NUM_THREADS];
  struct thread_data td[NUM_THREADS];
  int rc;
  int i;

  for( i = 0; i < NUM_THREADS; i++ ) {
    cout <<"main() : creating thread, " << i << endl;
    td[i].thread_id = i;
    td[i].message = "This is message";
    rc = pthread_create(&threads[i], NULL, PrintHello, (void *)&td[i]);

    if (rc) {
      cout << "Error:unable to create thread," << rc << endl;
      exit(-1);
    }
  }
  pthread_exit(NULL);
```

```
}
```

When the above code is compiled and executed, it produces the following result −

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Thread ID : 3 Message : This is message
Thread ID : 2 Message : This is message
Thread ID : 0 Message : This is message
Thread ID : 1 Message : This is message
Thread ID : 4 Message : This is message
```

## Activity 3:

### Joining and Detaching Threads

There are following two routines which we can use to join or detach threads −

```
pthread_join (threadid, status)
pthread_detach (threadid)
```

The pthread_join subroutine blocks the calling thread until the specified threadid thread terminates. When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

This example demonstrates how to wait for thread completions by using the Pthread join routine.

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <unistd.h>

using namespace std;

#define NUM_THREADS 5

void *wait(void *t) {
  int i;
  long tid;

  tid = (long)t;
```

```
   sleep(1);
   cout << "Sleeping in thread " << endl;
   cout << "Thread with id : " << tid << "  ...exiting " << endl;
   pthread_exit(NULL);
}

int main () {
   int rc;
   int i;
   pthread_t threads[NUM_THREADS];
   pthread_attr_t attr;
   void *status;

   // Initialize and set thread joinable
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for( i = 0; i < NUM_THREADS; i++ ) {
      cout << "main() : creating thread, " << i << endl;
      rc = pthread_create(&threads[i], &attr, wait, (void *)i );

      if (rc) {
         cout << "Error:unable to create thread," << rc << endl;
         exit(-1);
      }
   }

   // free attribute and wait for the other threads
   pthread_attr_destroy(&attr);
   for( i = 0; i < NUM_THREADS; i++ ) {
      rc = pthread_join(threads[i], &status);
      if (rc) {
         cout << "Error:unable to join," << rc << endl;
```

```
      exit(-1);
    }

    cout << "Main: completed thread id :" << i ;
    cout << "  exiting with status :" << status << endl;
  }

  cout << "Main: program exiting." << endl;
  pthread_exit(NULL);
}
```

When the above code is compiled and executed, it produces the following result –

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Sleeping in thread
Thread with id : 0 .... exiting
Sleeping in thread
Thread with id : 1 .... exiting
Sleeping in thread
Thread with id : 2 .... exiting
Sleeping in thread
Thread with id : 3 .... exiting
Sleeping in thread
Thread with id : 4 .... exiting
Main: completed thread id :0  exiting with status :0
Main: completed thread id :1  exiting with status :0
Main: completed thread id :2  exiting with status :0
Main: completed thread id :3  exiting with status :0
Main: completed thread id :4  exiting with status :0
Main: program exiting.
```

## Activity 4

**Peterson solution for 2 processes critical section problem**

```
#include<pthread.h>
#include<stdio.h>
void *func1(void *);
void *func2(void *);
```

```
int flag[2];
int turn=0;
int global=100;
int main()
{
   pthread_t tid1,tid2;
   pthread_create(&tid1,NULL,func1,NULL);
   pthread_create(&tid2,NULL,func2,NULL);
   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
}

void *func1(void *param)
{
   int i=0;
   while(i<2)
   {
      flag[0]=1;
      turn=1;
      while(flag[1]==1 && turn==1);
      global+=100;
     // printf("FT: g: %d",global);
      // printf("value of i in func1 is %d \n",i);
      printf("value of global in func1 is %d \n",global);
      flag[0]=0;
      i++;
   }
}
void *func2(void *param)
{
   int i=0;
   while(i<2)
   {
      flag[1]=1;
      turn=0;
      while(flag[0]==1 && turn==0);
      global-=75;
     // printf("SP: g: %d",global);
      printf("value of i in func2 is %d \n",i);
        printf("value of global in func2 is %d \n",global);
      flag[1]=0;
      i++;
   }
}
```

Output

```
value of i in func2 is 0
value of global in func2 is 25
value of i in func2 is 1
```

```
value of global in func2 is -50
value of global in func1 is 50
value of global in func1 is 150
```

This **solution do not guarantee** mutual exclusion between the two process. The code in earlier Activity 4 might have worked on most systems, but is was not 100% correct. The logic was perfect, but most modern CPUs employ performance optimizations that can result in out-of-order execution. This reordering of memory operations (loads and stores) normally goes unnoticed within a single thread of execution, but can cause unpredictable behaviour in concurrent programs. When a thread was waiting for its turn, it ended in a long while loop which tested the condition millions of times per second thus doing unnecessary computation. There is a better way to wait, and it is known as ***"yield".***

In above activity the compiler considers the following 2 statements as independent of each other and thus tries to increase the code efficiency by re-ordering them, which can lead to problems for concurrent programs.

```
while (f = = 0);

// Memory fence required here

print x;
```

To avoid this we place a memory fence to give hint to the compiler about the possible relationship between the statements across the barrier.

So the modified code becomes,

```
#include<pthread.h>
#include<stdio.h>
void *func1(void *);
void *func2(void *);
int flag[2];
int turn=0;
int global=100;
int main()
{
   pthread_t tid1,tid2;
   pthread_create(&tid1,NULL,func1,NULL);
   pthread_create(&tid2,NULL,func2,NULL);
   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
}

void *func1(void *param)
{
```

```
   int i=0;
   while(i<2)
   {
      flag[0]=1;
      turn=1;
      __sync_synchronize();
      while(flag[1]==1 && turn==1) sched_yield();
      global+=100;
     // printf("FT: g: %d",global);
      // printf("value of i in func1 is %d \n",i);
      printf("value of global in func1 is %d \n",global);
      flag[0]=0;
      i++;
   }
}
void *func2(void *param)
{
   int i=0;
   while(i<2)
   {
      flag[1]=1;
      turn=0;
      __sync_synchronize();
      while(flag[0]==1 && turn==0)sched_yield();
      global-=75;
     // printf("SP: g: %d",global);
      printf("value of i in func2 is %d \n",i);
        printf("value of global in func2 is %d \n",global);
      flag[1]=0;
      i++;
   }
}
```

## 3) Stage V (verify)

### Home Activities:

1. Write a program that uses multiple threads to find which integer between 1 and 100,000 has the largest number of divisors, and how many divisors does it have?. By using threads, your program will take less time to do the computation when it is run on a multiprocessor computer. At the end of the program, output the elapsed time, the integer that has the largest number of divisors, and the number of divisors that it has.

### 3) Stage V (verify)

## Home Activities:

1) Practice the shell functions discussed in the lab

## 4) Stage $a_2$ (assess)

**Activity assessment and Viva voce at the end of lab.**

# LAB # 09

## Statement Purpose:

This lab will give you'll implement Process synchronization (i.e. Mutex and Semaphores) in C++ language using threads.

## Activity Outcomes:

The primary objective of this lab is to implement the process synchronization by:

- Creating/Destroying Mutexes
- Locking/Unlocking Mutexes
- Using Semaphores

# 1) Stage J (Journey)

## Introduction

When multiple threads are running they will invariably need to communicate with each other in order synchronize their execution. One main benefit of using threads is the ease of using synchronization facilities.
Threads need to synchronize their activities to effectively interact. This includes:

- Implicit communication through the modification of shared data
- Explicit communication by informing each other of events that have occurred.

This lab describes the synchronization types available with threads and discusses when and how to use synchronization. There are a few possible methods of synchronizing threads and here we will discuss:

1. Mutual Exclusion (Mutex) Locks
2. Semaphores

**Why we need Synchronization and how to achieve it?**
Suppose the multiple threads share the common address space (thru a common variable), and then there is a problem.

```
THREAD A                          THREAD B
x = common_variable ;             y = common_variable ;
x++ ;                             y-- ;
common_variable = x ;             common_variable = y ;
```

If threads execute this code independently it will lead to garbage. The access to the common_variable by both of them simultaneously is prevented by having a lock, performing the thing and then releasing the lock.

### 1. Mutexes

Mutual exclusion locks (mutexes) can prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.

**Mutex Variables:**
Mutex is a shortened form of the words "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization. A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.
Very often the action performed by a thread owning a mutex is the updating of global variables. This is a safe way to ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the update. The variables being updated belong to a "critical section".
A typical sequence in the use of a mutex is as follows:

a. Create and initialize a mutex variable
b. Several threads attempt to lock the mutex
c. Only one succeeds and that thread owns the mutex

d. The owner thread performs some set of actions
e. The owner unlocks the mutex
f. Another thread acquires the mutex and repeats the process
g. Finally the mutex is destroyed.

When several threads compete for a mutex, the losers block at that call an unblocking call is available with "trylock" instead of the "lock" call.

**Creating / Destroying Mutexes :**

```
pthread_mutex_init ( pthread_mutex_t mutex,  pthread_mutexattr_t   attr)

pthread_mutex_destroy ( pthread_mutex_t   mutex )

pthread_mutexattr_init ( pthread_mutexattr_t   attr )

pthread_mutexattr_destroy ( pthread_mutexattr_t   attr )
```

pthread_mutex_init( ) creates and initializes a new mutex mutex object, and sets its attributes according to the mutex attributes object, attr. The mutex is initially unlocked.

Mutex variables must be of type pthread_mutex_t. The attr object is used to establish properties for the mutex object, and must be of type pthread_mutexattr_t if used (may be specified as NULL to accept defaults). If implemented, the pthread_mutexattr_init( ) and pthread_mutexattr_destroy( ) routines are used to create and destroy mutex attribute objects respectively.
pthread_mutex_destroy( ) should be used to free a mutex object which is no longer needed.

**Locking / Unlocking Mutexes :**

```
pthread_mutex_lock ( pthread_mutex_t mutex )

pthread_mutex_trylock ( pthread_mutex_t mutex )

pthread_mutex_unlock ( pthread_mutex_t mutex )
```

pthread_mutex_lock( ) routine is used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, the call will block the calling thread until the mutex is unlocked.

pthread_mutex_trylock( ) will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.
Mutex contention: when more than one thread is waiting for a locked mutex, which thread will be granted the lock first after it is released? Unless thread priority scheduling (not covered) is used, the assignment will be left to the native system scheduler and may appear to be more or less random. pthread_mutex_unlock( ) will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data.
**An error will be returned:**

If the mutex was already unlocked

If the mutex is owned by another thread.

## 2. Semaphores

Semaphore is another tool to synchronize activities in a computer. The concept of semaphores as used in computer synchronization is due to the Dutch computer scientist *Edsgar Dijkstra*. They have the advantages of being very simple, but sufficient to construct just about any other synchronization function you would care to have.
There are several versions to implement semaphores. The most common versions are:

1) POSIX semaphores
2) System V IPC semaphores

This lab will consider only POSIX semaphore, since POSIX semaphores has very clear API functions to perform semaphore operations. However, it is more efficient to use System V semaphore than POSIX semaphore when semaphores are shared between processes.

### What is a Semaphore?

A semaphore is an integer variable with two atomic operations:

1) **wait**. Other names for **wait** are **P**, **down** and **lock.**
2) **signal**: Other names for **signal** are **V**, **up**, **unlock** and **post**.

The simplest way to understand semaphore is, of course, with code. Here is a little pseudo-code that may help.

```
typedef struct SEMAPHORE {
int value; /* the integer value for semaphore */
other stuff;
} sem_t;
init(sem_t *S, int i)
{
S->value = i;
}
void wait(sem_t *S)
{
S->value--;
if (S->value < 0)
block on semaphore
}
void signal(sem_t *S)
{
S->value++;
if (S->value <= 0)
unblock one process or thread that is blocked on semaphore
    }
```

### Basic Synchronization Patterns

This section presents a series of examples of basic synchronization problems. It shows ways of using semaphore to solve them.

### Example 1 (Signaling)

This example represents the simplest use for a semaphore (which is **signaling**), in which one process/thread sends a signal to another process/thread to indicate that something has happened.

Suppose process one must execute statement **a** before process two executes statement **b**. To solve this synchronization problem, we need a semaphore (say **sync**) with initial value **0**, and that both processes have shared access to it. The following pseudo-code describes the solutions:

```
init(&sync, 0);
```

Process One

```
a;
signal(&sync);
```

Process Two

```
wait(&sync);
b;
```

**Example 2 (Rendezvous)**

Puzzle: Generalize the signal pattern so that it works both ways. Process One has to wait for Process Two and vice versa. In other words, given this code.

Process One

```
a1;
a2;
```

Process Two

```
b1;
b2;
```

we want to guarantee that **a1** happens before **b2** and **b1** happens before **a2**. Please try to think how to solve this puzzle before proceeding to next paragraph.

**Hint:** You need to use two semaphores.

```
init(&sem1, 0);

init(&sem2, 0);
```

While working on the previous problem, you might have tried something like this :

Process One

```
a1;
wait(&sem2);
signal(&sem1);
a2;
```

Process Two

```
b1;
wait(&sem1);
signal(&sem2);
b2;
```

if so, I hope you will reject it quickly because it has serious problem of Deadlock! Solutions for this problem are:

**Solution 1:**

Process One

```
a1;
signal(&sem1);
wait(&sem2);
a2;
```

Process Two

```
b1;
signal(&sem2);
wait(&sem1);
b2;
```

**Solution 2:**

Process One

```
a1;
wait(&sem2);
signal(&sem1);
a2;
```

Process Two

```
b1;
signal(&sem2);
wait(&sem1);
b2;
```

This solution is probably less efficient, since it might have to switch between process One and Two more than necessary.

**Example 3 (Mutual exclusion)**

This example illustrates how we can use a semaphore to guarantee that only one process/thread accesses a shared memory (or global variables).
Suppose two threads try to access a global variable count. To enforce mutual exclusion, we need a semaphore (say **mutex**) with initial value **1**. The following pseudo-code describes the solution:

```
init(&mutex, 1);
```

Thread A

```
wait(&mutex);
/* CS */
count = count + 1;
signal(&mutex);
```

Thread B

```
wait(&mutex);
/* CS */
count = count + 1;
signal(&mutex);
```

**POSIX Semaphore**

Semaphores are part of the POSIX.1b standard adopted in 1993. The POSIX.1b standard defines two types of semaphores: *named* and *unnamed*. A POSIX.1b *unnamed semaphore* can be used by a single process or by children of the process that created them. A POSIX.1b *named semaphore* can be used by any processes. In this section, we will consider only how to initialize unnamed semaphore.
The following header summarizes how we can use POSIX.1b unnamed semaphore:

| ***Header file name*** | **#include <semaphore.h>** |
|---|---|
| ***Semaphore data type*** | **sem_t** |
| ***Initialization*** | **int sem_init(sem_t *sem, int pshared, unsigned value);** |
| ***Semaphore Operations*** | **int sem_destroy(sem_t *sem);**<br>**int sem_wait(sem_t *sem);** |

| | |
|---|---|
| | **int sem_post(sem_t *sem);**<br>**int sem_trywait(sem_t *sem);** |
| ***Compilation*** | **cc filename.c –o filename –lrt** |

All of the POSIX.1b semaphore functions return **-1** to indicate an error.
**sem_init** function initializes the semaphore to have the value value. The value parameter cannot be negative. If the value of pshared is not 0, the semaphore can be used between processes (i.e. the process that initializes it and by children of that process). Otherwise it can be used only by threads within the process that initializes it.
**sem_wait** is a standard semaphore wait operation. If the semaphore value is 0, the sem_wait blocks unit it can successfully decrement the semaphore value.
**sem_trywait** is similar to sem_wait except that instead of blocking when attempting to decrement a zero-valued semaphore, it returns -1.
**sem_post** is a standard semaphore signal operation. The POSIX.1b standard requires that sem_post be reentrant with respect to signals, that is, it is asynchronous-signal safe and may be invoked from a signal-handler.

# Lab Activities

**Activity 1.a:**

**Write a program in C++ to use Mutexes.**

This simple example code demonstrates the use of several Pthread mutex routines. The serial version may be reviewed first to compare how the threaded version performs the same task.

```
#include  <stdio.h>
#include <pthread.h>
/* The function run when the thread is created */
void* compute_thread (void*);
main( )
{
/* This is data describing the thread created */
   pthread_t tid; /* thread ID structure */
   pthread_attr_t attr; /* thread attributes */
   char hello[ ] = {"Hello, "}; /* some typical data */
   char thread[ ] = {"thread"};
/* Initialize the thread attribute structure */
   pthread_attr_init(&attr);
/* Create another thread. ID is returned in &tid */
/* The last parameter passed to the thread function */
   pthread_create(&tid, &attr, compute_thread, thread);
/* Continue on with the base thread */
  printf(hello);
  sleep(1);
   printf("\n");
   exit(0);
}
/* The thread function to be run */
```

```
void* compute_thread(void* dummy) {
   printf(dummy); return; }
```

Compile the following program using -lpthread library as follows −

```
$gcc test.cpp -lpthread
```

Now, execute your program which gives the following output −

```
Hello, thread
```

After creating a new thread, main process prints “Hello, ” and then sleeps. The new thread then prints “thread”.

**Activity 1.b:**

This example shows how to pass multiple arguments via a structure. You can pass any data type in a thread callback because it points to void as explained in the following example

```
#include <pthread.h>
#include <stdio.h>
/* Function run when the thread is created */
void* compute_thread (void*);
/* This is the lock for thread synchronization */
pthread_mutex_t my_sync;
main( )
{
/* This is data describing the thread created */
   pthread_t tid;
   pthread_attr_t attr;
   char hello[ ] = {"Hello, "};
   char thread[ ] = {"thread"};
/* Initialize the thread attributes */
   pthread_attr_init (&attr);
/* Initialize the mutex (default attributes) */
   pthread_mutex_init (&my_sync,NULL);
/* Create another thread. ID is returned in &tid */
/* The last parameter is passed to the thread function */
/* Note reversed the order of "hello" and "thread" */
   pthread_create(&tid, &attr, compute_thread, hello);
   sleep(1); /* Let the thread get started */
/* Lock the mutex when it's our turn to do work */
   pthread_mutex_lock(&my_sync);
   printf(thread);
   printf("\n");
   pthread_mutex_unlock(&my_sync);
   exit(0);
}
/* The thread function to be run */
```

```
void* compute_thread(void* dummy)
{
/* Lock the mutex when its our turn */
   pthread_mutex_lock(&my_sync);
   printf(dummy);
   pthread_mutex_unlock(&my_sync);
   sleep(1); return;
}
```

Now, execute your program which gives the following output −

```
Hello, thread
```

The main process sleeps after creating new thread which prints “Hello, ”. After waking up, the main process prints “thread”. Printf is treated as Critical Section and hence guarded with Mutex Lock so that no two process can print at the same time.

## Activity 2:

### Semaphores (Signaling):

**Lab1.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
/* Global variables */
int x = 0;
sem_t sync;
/* Thread function */
void *my_func(void *arg)
{
/* wait for signal from main thread */
sem_wait(&sync);
printf("X = %d\n", x);
}
void main ()
{
pthread_t thread;
/* semaphore sync should be initialized by 0 */
if (sem_init(&sync, 0, 0) == -1) {
perror("Could not initialize mylock semaphore");
exit(2);
}
if (pthread_create(&thread, NULL, my_func, NULL) < 0) {
perror("Error: thread cannot be created");
exit(1);
}
/* perform some operation(s) */
```

```
x = 55;
/* send signal to the created thread */
sem_post(&sync);
/* wait for created thread to terminate */
pthread_join(thread, NULL);
/* destroy semaphore sync */
sem_destroy(&sync);
exit(0);
}
```

The output of this code would be:

```
Final value of x is 10
```

**Example2 (Critical Section-Binary Semaphore):**

```
// in this code, two threads are displaying output by calling sem_wait() and sem_post() to ensure M.E //

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
//using namespace std;
sem_t lock; //lock is defined as semaphore
//routine for thread 1
void *thread1(void *varg)
{
   sem_wait(&lock);
          printf("this from Thread 1\n"); // CS
   sem_post(&lock);
          return NULL;
}
//routine for thread 2
void *thread2(void *varg)
{
   sem_wait(&lock);
          printf("this from Thread 2\n");  //CS
   sem_post(&lock);
   return NULL;
}
int main()
{
   sem_init(&lock,0,1); // lock is initialized as 1 with last argument. middle argument is for threads(i.e. 1 for
processes and 0 for threads)
          pthread_t t1, t2;
          printf("Before Thread\n");
          pthread_create(&t1, NULL, thread1,NULL );
   pthread_create(&t2, NULL, thread2,NULL );
          pthread_join(t1, NULL);
   pthread_join(t2, NULL);
```

```
        printf("After Thread\n");
return 0;
}
```

The output of the above code should look like:

```
Before Thread
This from Thread 1
This form Thread 2
After Thread
```

### Example3: Resource Manager (Counting Semaphore)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
//using namespace std;
sem_t lock; //lock is defined as semaphore
//routine for thread 1
void *thread1(void *varg)
{
   sem_wait(&lock);
        printf("this from Thread 1\n"); // CS
   sem_post(&lock);
        return NULL;
}
//routine for thread 2
void *thread2(void *varg)
{
   sem_wait(&lock);
        printf("this from Thread 2\n");  //CS
   sem_post(&lock);
   return NULL;
}
int main()
{
   sem_init(&lock,0,2); // lock is initialized as 2 for counting semaphore. middle argument is for threads(i.e.
1 for processes and 0 for threads)
        pthread_t t1, t2;
        printf("Before Thread\n");
        pthread_create(&t1, NULL, thread1,NULL );
   pthread_create(&t2, NULL, thread2,NULL );
   pthread_create(&t1, NULL, thread1,NULL );
   pthread_create(&t2, NULL, thread2,NULL );
        pthread_join(t1, NULL);
   pthread_join(t2, NULL);
        printf("After Thread\n");
return 0;
}
```

The output of the code would be:

```
Before Thread
this from Thread 2
this from Thread 1
this from Thread 1
this from Thread 2
After Thread
```

## 2) Stage V (verify)

### Home Activities:

Practice the shell functions discussed in the lab

## 4) Stage $a_2$ (assess)

**Activity assessment and Viva voce at the end of lab.**

# LAB # 10

## Statement Purpose:

This lab will give you'll implement Filing in C++ language.

## Activity Outcomes:

The primary objective of this lab is to implement the Filing in C++ language.

## 1) Stage J (Journey)

## Introduction

A process is a currently executing instance of a program. All programs by default execute in the user mode. A C program can invoke UNIX system calls directly. A system call can be defined as a request to the operating system to do something on behalf of the program. During the execution of a system call, the mode is change from user mode to kernel mode (or system mode) to allow the execution of the system call. The kernel, the core of the operating system program in fact has control over everything. All OS software is trusted and executed without any further verification. All other software needs to request kernel mode using specific system calls to create new processes and manage I/O. A high level programmer does not have to worry about the mode change from user-mode to kernel-mode as it is handled by a predefined library of system calls. Unlike processes in user mode, which can be replaced by another process at any time, a process in kernel mode cannot be arbitrarily replaced by another process. A process in kernel mode can only be suspended by an interrupt or exception. A C system call software instruction generates an OS interrupt commonly called the operating system trap. The system call interface handles these interruptions in a special way. The C library function passes a unique number corresponding to the system call to the kernel, so kernel can determine the specific system call user is invoking. After executing the kernel command the operating system trap is released and the system returns to user mode. Unix system calls are primarily used to manage the file system or control processes or to provide communication between multiple processes. A subset of the system calls include

creat( ), open( ), close( ) -- managing I/O channels

read( ), write( ) – handling input and output operations

lseek( ) – for random access of files

link( ), unlink( ) – aliasing and removing files

stat( ) – getting file status

access( ), chmod( ), chown( ) – for access control

exec( ), fork( ), wait( ), exit( ) --- for process control

getuid( ) – for process ownership

getpid( ) -- for process ID

signal( ) , kill( ), alarm( ) – for process control

chdir( ) – for changing working directory

mmap(), shmget(), mprotect(), mlock() – manipulate low level memory attributes

time(), gettimer(), settimer(),settimeofday(), alarm() – time management functions

pipe( ) – for creating inter-process communication

System calls interface often change and therefore it is advisable to place system calls in subroutines so subroutines can be adjusted in the event of a system call interface change. When a system call causes an error, it returns -1 and store the error number in a variable called “errno” given in a header file called /usr/include/errno.h. When a system call returns an error, the function perror can be used to print a diagnostic message. If we call perror( ), then it displays the argument string, a colon, and then the error message, as directed by "errno", followed by a newline.

if (unlink("text.txt")==-1){ perror(""); } If the file text.txt does not exists, unlink will return -1 and that in return will cause the program to print the message “File does not exists”

**Managing the File System with Sys calls**

File structure related system calls to manage the file system are quite common. Using file structure related system calls, we can create, open, close, read, write, remove and alias, set permission, get file information among other things. The arguments to these functions include either the relative or absolute path of the file or a descriptor that defines the IO channel for the file. A channel provides access to the file as an unformatted stream of bytes. We will now look at some of the file related functions provided as system calls.

**UNIX System I/O Calls**

The high level library functions given by <stdio.h> provide most common input and output operations. These high level functions are built on the top of low-level structures and calls provided by the operating system. In this section, we will look at some low level I/O facilities that will provide insight into how low level I/O facilities are handled and therefore may provide ways to use I/O in ways that are not provided by the stdio.h. In UNIX, I/O hardware devices are represented as special files. Input/Output to files or special files (such as terminal or printers) are handled the same way. UNIX also supports “pipes” a mechanism for input/output between processes. Although pipes and files are different I/O objects, both are supported by low level I/O mechanisms.

**Create System Call**

A file can be created using the creat function as given by the following prototype.

#include <sys/file.h> // can be replaced by <fcntl.h>

int creat(char* filename, mode_t mode)

The mode is specified as an octal number. For example, 0444 indicates that r access for USER, GROUP and ALL for the file. If the file exists, the creat is ignored and prior content and rights are maintained. The library

**/usr/include/sys/stat.h**

provides following constants that can be used to set permissions.

S_IREAD --- read permission for the owner

S_IWRITE --- write permission for the owner

S_IEXEC --- execute/search permission for the ownerS_

IRWXU --- read, write, execute permission for the userS_

IRGRP – read for group

S_IWGRP – write for group

S_IXGRP – execute for group

S_IRWXG – read, write, execute for the groupS_

IROTH --- read for others

S_IWOTH – write for othersS_

IXOTH -- execute for othersS_

IRWXO – read , write , execute for others

For example, to create a file with read and write access only to user, we can do the following.
**creat("myfile", S_IREAD | S_IWRITE);**

**Open System Call**

A file can be open using the open system call as follows.

**#include <sys/file.h> // can be replaced by <fcntl.h>**

**int open(char* filename, int flags, int mode);**

The above code opens the filename for reading or writing as specified by the access and returns an integer descriptor for that file. Descriptor is an integer (usually less than 16) that indicates a table entry for the file reference for the current process. The stdin(0), stdout(1) and stderr(3) are given. Other files will be provided with the next available descriptor when opening. File name can be given

as full path name, relative path name, or simple file name. If the file does not exist, then open creates the file with the given name. Let us take a detail look at the arguments to open.

filename : A string that represents the absolute, relative or filename of the file

flags : An integer code describing the access (see below for details)

mode : The file protection mode usually given by 9 bits indicating rwx permission

The flag codes are given by

O_RDONLY -- opens file for read only

O_WRONLY – opens file for write only

O_RDWR – opens file for reading and writing

O_NDELAY – prevents possible blocking

O_APPEND --- opens the file for appending

O_CREAT -- creates the file if it does not exists

O_TRUNC -- truncates the size to zero

O_EXCL – produces an error if the

O_CREAT bit is on and file exists If the open call fails, a -1 is returned; otherwise a descriptor is returned. For example, to open a file for read and truncates the size to zero we could use,

**open("filename", O_RDONLY | O_TRUNC, 0);**

We assume that the file exists and note that zero can be used for protection mode. For opening files, the third argument can always be left at 0.

**Reading and Writing to Files**

Reading and writing a file is normally sequential. For each open file, a current position points to the next byte to be read or written. The current position can be movable for an actual file, but not for stdin when connected to a keyboard.

**Read System Call**

**#include <sys/types.h> // or #include <unistd.h>**

**size_t read(int fd, char *buffer, size_t bytes);**

fd is the file descriptor, buffer is address of a memory area into which the data is read and bytes is the maximum amount of data to read from the stream. The return value is the actual amount of data read from the file. The pointer is incremented by the amount of data read. Bytes should not exceed the size of the buffer. Write System Call The write system call is used to write data to a file or other object identified by a file descriptor. The prototype is #include <sys/types.h> size_t write(int fd, char *buffer, size_t bytes); fd is the file descriptor, buffer is the address of the area of

memory that data is to be written out, bytes is the amount of data to copy. The return value is the actual amount of data written, if this differs from bytes then something may be wrong.

Example: Consider the C high level function readline(char [], int) that reads a line from stdin and store the line in a character array. This function can now be rewritten using low level read as follows

```
int readline(char s[], int size){

char* tmp = s;

while (--size>0 && read(0,tmp,1)!=0 && *tmp++ != '\n');

*tmp = '\0';

return (tmp-s);

}
```

**Close System Call**

The close system call is used to close files. The prototype is

**include <unistd.h>**

**int close(int fd);**

When a process terminates, all the files associated with the process are closed. But it is always a good idea to close a file as they do consume resources and systems impose limits on the number of files a process can keep open.

**lseek System Call**

Whenever a read() or write() operation is performed on a file, the position in the file at which reading or writing starts is determined by the current value of the read/write pointer. The value of the read/write pointer is often called the offset. It is always measured in bytes from the start of the file. The lseek() system call allows programs to manipulate this directly by providing the facility for direct access to any part of the file. In other words, the lseek allows random access to any byte of the file. It has three parameters and the prototype is

**#include <sys/types.h>**

**#include <unistd.h> long**

**lseek(int fd,int offset,int origin)**

| **origin** | **position** |
|---|---|
| 0 | beginning of the file |
| 1 | Current position |
| 2 | End of the file |

**Call** **Meaning**

lseek(fd,0,0) places the current position at the first byte

lseek(fd,0,2) places the current position at EOF

lseek(fd,-10,1) Backs up the current position by 10 bytes

**Other System Level Operations**

Other system level operations include creating a file using

**creat(filename, mode);**

The mode can be selected from the following octal bit patterns

| Octal Bit Pattern | Meaning |
|---|---|
| 0400 | Read by owner |
| 0200 | Write by Owner |
| 0100 | Execute or search by owner |
| 0070 | Read, Write, Execute(search) by group |
| 0007 | Read, Write, Execute(search) by others |

Other system calls include link a way to give alternative names to a file (aliases), and unlink a way to remove an alias to a file. The prototypes for the link and unlink are

**int link (char* file1, char* file2);**

**int unlink(char* name);**

The link( ) function creates an alias name file2 for file1, that exists in the current directory. Use of unlink with the original file name will remove the file.

**Creating and removing Directories**

Directories can be created and removed using mkdir and rmdir function calls. The function prototypes are

**int mkdir(char* name, int mode);**

**int rmdir(char* name); returns 0 or 1 for success or failure.**

For example, creating a new directory called “newfiles” with only the READ access for the user we can do the following.

**mkdir(“newfiles”, 0400);**

Later you can remove this directory by calling

**rmdir("newfiles");**

Caution: Be very careful about removing directories. These are system calls that are executed w/o further confirmation. You can also reset the file/dir permission with chmod function. The prototype is given by

**#include <sys/types.h>**

**#include <sys/stat.h>**

**int chmod(const char *path, mode_t mode);**

For example, the directory that was created above called "newfiles" has only read access on the owner. But if you decided to change the read access from owner to all groups, then you can call:

**chmod("newfiles", 0444);**

**Accessing Directories**

A UNIX directory contains a set of files that can be accessed using the sys/dir.h library. We include the library with

**#include <sys/dir.h>**

And the function

**DIR *opendir(char* dir_name)**

Opens a directory given by dir_name and provides a pointer to access files within the directory. The open DIR stream can be used to access a struct that contains the file information. The function

**struct dirent *readdir(DIR* dp)**

returns a pointer to the next entry in the directory. A NULL pointer is returned when the end of the directory is reached. The struct direct has the following format.

**struct dirent {**

**u-long d_ino; /* i-node number for the dir entry */**

**u_short d_reclen; /* length of this record */**

**u_short d_namelen ; /* length of the string in d_name */ char d_name[MAXNAMLEN+1] ; /* directory name */ };**

Using system libraries, we can write our own "find" function for example. The following function, search (char* file, char* dir) returns 0 if the file is found in the directory and returns 1 otherwise.

**#include <string.h>**

**#include <sys/types.h>**

**#include <sys/dir.h>**

```
int search (char* file, char* dir){

DIR  *dirptr=opendir(dir);

struct dirent *entry = readdir(dirptr);

while (entry != NULL) {

if ( strlen(entry->d_name) == strlen(file) && (strcmp(entry->d_name, file) == 0)

            return 0; /* return success */

entry = readdir(dirptr);     }

return 1;  /* return failure */ }
```

**Accessing File Status**

Status of a file such as file type, protection mode, time when the file was last modified can be accessed using stat and fstat functions. The prototypes of stat and fstat functions are

```
stat(char* file, struct stat *buf);

fstat(int fd, struct stat *buf);
```

stat and fstat functions are equivalent except that former takes the name of a file, while the latter takes a file descriptor. For example, we can get the status of a file as follows.

```
struct stat buf;  // defines a struct stat to hold file information

stat("filename", &buf) ;  // now the file information is placed in the buf
```

The status of the file that is retrieved and placed in the buf has many information about the file. For example, the stat structure contains useful information such as

**st_atime  --- Last access time**

**st_mtime --- last modify time**

**st_ctime  --- Last status change time**

**st_size --- total size of file**

**st_uid – user ID of owner**

**st_mode – file status (directory or not)**

**Example**

```
#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h>
```

```
struct stat statbuf;

char dirpath[256];

getcwd(dirpath,256);

DIR *dir = opendir(dirpath);

struct dirent *dp;

for (dp=readdir(dir);

dp != NULL ;

dp=readdir(dir)){

stat(dp->d_name, &statbuf);

printf("the file name is %s \n", dp->d_name);

printf("dir = %d\n", S_ISDIR(statbuf.st_mode));

printf("file size is %ld in bytes \n", statbuf.st_size);

printf("last modified time is %ld in seconds \n", statbuf.st_mtime);

printf("last access time is %ld in seconds \n", statbuf.st_atime);

printf("The device containing the file is %d\n", statbuf.st_dev);

printf("File serial number is %d\n\n", statbuf.st_ino);  }
```

## 3)Stage v (verify)

### Home Activities:

Practice the shell functions discussed in the lab

## 4) Stage $a_2$ (assess)

**Activity assessment and Viva voce at the end of lab.**

# LAB # 11

## Statement Purpose:

This lab will explain to you the basic concept of shell and shell scripting in Linux.

## Activity Outcomes:

This lab describes

- How two view, create, open shells in linux
- Writing shell programs
- Performing basic operations using shells

## Instructor Note:

As pre-lab activity, introduce students to the concept of shell

# 1) Stage J (Journey)

## 1. 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.

- Shell is an Interface between the user and the operating system.
- Bourne, Korn, C Shell, tcsh, bash, zsh, etc. are different UNIX Shells
- Recommended — **Korn** (ksh) Shell
    - **ksh** is a variant of the Bourne Shell.
    - Provides all features of the Bourne Shell.
    - Allows access to prior (history) commands with the **j, k** keys (after <escape>).
- Changing Shells: **chsh** command allows users to switch default login shells.
- History is a mechanism for saving recently executed commands (events).
    - **export HISTSIZE = 25** : command to save the last 25 commands.
    - **history** : command to display the history list.
    - **r 3** : executes command number 3 in the history list.
    - **r :** re-executes the last executed command.
    - **r prog**: re-executes the most recent command beginning with the string "prog".

## 2. 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.

- A shell script (or shell program) is a series of Unix commands placed in an ASCII text file.
- Each shell (ksh/csh/sh..) provides mechanisms for control (e.g. statements like if, while, for, etc).
- Unix commands + shell variables + control mechanisms = powerful programming language.

## 2.1 Executing shell scripts

- Explicitly choosing shell to use:

```
$ sh MyScript
$ csh MyOtherScript
$ ksh MyThirdScript
```

- Implicit selection of shell for script:
    - Suppose you have a script named "MyScript".
    - Make the first line of the script start with:

```
#!/usr/local/bin/ksh
```

(You could specify any command after "#!" to act as the "interpreter" for the commands in the script, and you can also give arguments.)

- Use chmod to make it executable (recall general format "chmod [ugoa][+-][rwx] filename"):

```
$ chmod a+x MyScript
```

- Then you can execute MyScript like a regular command:

```
$ MyScript
```

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).

## 2.2 Example

```
# !/bin/bash
#First shell script
clear
echo "Hello World"
```

```
# !/bin/bash
#Script to print user information 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
```

## 2.3 Example

Creating a simple shell script (you could do this with emacs):

$ cat > gohome

#!/usr/local/bin/ksh

cd ~

ls -F

^D

Change file permissions: $ chmod a+x gohome

Run it: $ gohome

## 2.4 Shell Variables

- Environment variables are variables that the shell, a shell script or any other program can access to get information unique to specific user. These variables are also called as shell variables.
- Shell variables can be: User Defined, System defined, or parameter variables.

- **System defined** variables are already defined and included in the environment when we login. Their names are in upper case letters. Some of them are as follows:

HISTFILE – filename of the history file. Default is $HOME/.sh_history.

HISTSIZE – Maximum number of commands retained in the history file

HOME – The pathname of your home directory

IFS – Inter Filed Separator. Zero or more characters treated by the shell as delimiters in parsing a command line into words. Default – Blank, Tab and Newline in succession

PATH – List of directories separated by colon that are searched for executables

PS1 – It is the primary prompt string. Korn Shell performs a full substitution of the value of PS1 before displaying it at the beginning of each command input line.

PS2 – Secondary prompt string. It is displayed when command is continued on the next line

PS3 – The value of this string is printed at the selection prompt by the select command

PS4 – The value of this string is printed in front of each line by the trace or the -x option

PWD – The present working directory

RANDOM – This is a random number from 0 – 32767. Its value will be different every time you examine it.

SHELL – The pathname of the shell

SECONDS – The number of seconds since the ksh was invoked

TERM – This is an alphanumeric string that identifies the type of your terminal. Its proper setting is imperative to the proper functioning of your terminal.

TMOUT–An integer specifying the number of seconds after which no terminal activity should cause the Korn Shell to log you out

USER – Your login name

$0 – The name of the shell script

$# – The number of parameters passed

$$ – The PID of the parent process i.e. shell

$? – exit status of last command run

- **Parameter variables** contain the values of the parameters passed to a shell script.

$1, $2,… Store the parameters given to the script

$* A list of all parameters, separated by the character defined in IFS

$@ Same as $* except the parameters are separated by space character

- **User defined** variables are subject to the following rules:
- **Rules:**

–Variable names can begin with a letter or an underscore and can contain an arbitrary number of letters, digits and underscores that is it can have any name of the form [A-Za-z][A-Za-z0-9_]*.

–No arbitrary upper limit to the variables you can define or use

–A variable retains its value from the time it is set – whether explicitly by you or implicitly by the shell – until its value is changed or the shell exits

–The variable is not passed to commands / shell scripts that you invoke unless it is exported

Note: You can assign values to variables and export them to pass variables downward to subshells of your current shells, but you cannot pass values upward to the higher-level shells or shell scripts

- For ksh:
    - variable=value
        - Notice that there are no spaces on either side of the equals sign.
    - export variable=value
        - Using export makes the value visible to child processes.

## 2.5 Using Variables

- To retrieve the value, precede the variable name with a dollar sign ($):

$ person=Terry

$ echo person

person

$ echo $person

Terry

- Unsetting (erasing) variables: $ unset person

Note: This is not the same as "person=''", which leaves the variable defined, but with the null value.

- For output, use "print" or "echo":

  $ print "Hello World"

  Hello World

- At the end of your script, use "return *N*" to give a specific return value, as in this script:

  #!/usr/local/bin/ksh

  print "Hello World"

  return 0

- You can use the built-in variable $? to access the exit status of the previously run command. If the above script were HelloScript:

$ HelloScript;print $?

0

## 2.6 Shell Environment Customization

- Customizing the Korn shell begins with your login profile script which is called **.profile** and is located in the home directory. All shell programming techniques valid in any shell script are valid in .profile.
- Some of the things that can be done here are

Set Control Keys with the stty command

Set Environment variables

- Terminal Type
- PATH – maybe to include a local bin directory
- Set local variables for shell control
- Local variables are not exported

Define aliases you like to use

alias ls='ls –F'

unalias ls – will remove the ls alias

- Define functions that you normally use
- Set your favorite shell options

set –o vi - command history and editing

set –o monitor- full job control support

- Execute commands that you want to run each time you login

***Example : Korn (ksh) Shell***

- Sample (*simple*) profile:

```
CDPATH=$HOME
PATH=.:./bin:$HOME/bin:/bin:/sbin:/usr/bin:/usr/sbin:/usr/ucb:\
/usr/local/bin:/usr/bin/X11
EDITOR=/usr/ucb/vi
ENV=$HOME/.kshrc
EXINIT='set ai nonumber showmatch wrapmargin=72 nowrapscan'
```

```
FCEDIT=/usr/ucb/vi
HISTSIZE=32
MAIL=/usr/spool/mail/$USER
PS1="! ${HOST%%.*}> "
SHELL=/usr/bin/ksh
export CDPATH EDITOR ENV EXINIT FCEDIT HISTSIZE HOST MAIL PATH PS1
export SHELL
biff y
tset -I
stty  erase \^? kill \^U intr \^C
umask 0022
alias mail=/usr/ucb/mailx
OSTYPE=`uname`
```

**How it works?**

**CDPATH** Directories searched by **cd** command.

**EDITOR** Sets default editor.

**ENV** Name of script that is executed at startup.

**EXINIT** Sets options for **ex** and **vi** editors.

**FCEDIT** Editor used by the **fc** command.

**HISTSIZE** Number of history commands available (must be set before **ksh** is started).

**PATH** Directories searched for any file (command) name. Most important shell variable.

**PS1** Primary prompt string; default is **$.**

**SHELL** Name of shell environment.

**export** Pass the value of shell variables, giving global meaning to them.

**tset** Initializes terminals, setting the TERM variable.

**biff y** System notifies user of new mail.

**stty erase** Sets delete character for command line edits.

**umask** Sets (eliminates) file default access permissions.

**alias** Allows commands (and options) to be abbreviated.

**Note:** .**kshrc** file: Contains shell commands executed whenever a new Korn shell is instigated e.g. a sample .**kshrc** file contains: **set -o trackall** **i.e.** Locate commands as they are defined.

## 2.7 Shell Variables and Quotes

- Quotes are used to specify values which contain spaces or special characters.
- Double quotes (") prevent shell interpretation of special characters **except** $ and `.
- Single quotes (') prevent shell interpretation of all special characters.
- The backslash (\) prevents the shell from interpreting the next character specially.
- Examples:

```
% people="jack and jill *"
% print "$people"
jack and jill *
% print $people
jack and jill file1 file2 file3
% print '$people'
$people
% print \$people
$people
```

## Numeric Variables

- ksh variables are **strings** or **integers**, depending on how they are defined. var=100 makes var the string '100'. integer var=100 makes var the integer 100.
- String variables that are all digits can be treated as numbers.
- To manipulate numeric variables using C-style expressions, use either $(( expression )) to return the value of expression, or (( expression )) to return only an exit status (true or false).

```
% integer x=1
% (( y=x*10 ))
% (( x+=1 ))
% print $x $y
2 10
```

- Within $(( expression )) and (( expression )) you can use parentheses for grouping, the arithmetic operators +, -, *, /, %, <<, >>, &, |, ~, ^, and the relational operators <, >, <=, >=, ==, !=, &&, and ||.
- Within the ((...)) syntax, variables need not be preceded by a dollar sign, and special characters need not be quoted or escaped.

## Array Variables

- An array variable provides a way to index a list of values and can be defined by the set command: Syntax: set -A arrayname [list]
- Items in an array can be accessed by position (first item is at index 0). Syntax: ${variable[index]}

Note: $variable refers to ${variable[0]}.

- **Example:**

```
set  -A array
array [0]=”one”
array[1]=”two”
array[2]=”three”
three=3
array[three]=”four”
for i  in 0 1 2 3 ; do
print ${array[i]}
            done
echo $ {array[three]}
```

- Alternatively we can declare the array as: set -A array one two three four

**Note:**

1. arrays are supported by ksh nor by bash
2. The number of defined elements in an array variable is given by ${#variable[*]}.

e.g. print ${#array[*]} for above array will display 4

3. set -A others ${array[*]} will create a new array others with the size and values of array where as: set -A others ${array[*]} “five” six seven will define the new array

**others** of 7 elements and initialize the last three elements with the three new values. Also: others[7]=eight ; others[8]=nine

$ print ${#others[*]} will display 9

$ print ${others[$(( ${#others[*]} - 1 ))]}

## 2) Stage a1 (apply)

**Lab Activities:**

1. Implement shell scripting using different fuctions

## 3) Stage V (verify)

## Home Activities:

Practice the shell scripting functions discussed in the lab.

## 4) Stage $a_2$ (assess)

**Activity assessment and Viva voce at the end of lab.**

# LAB # 012

## Statement Purpose:

This lab will give you insight of shell programs using C++ programming structs such as conditional statement and loops

.

## Activity Outcomes:

The primary objective of this lab is to implement:

- Conditional statements using If statement
- Coase statements
- Loops

# 1) Stage J (Journey)

## Introduction

## If Statements

- The basic form for if statements in korn shell ksh is:

```
if condition
then
   statements
[elif condition
then
   statements]
[else
   statements]
fi
```

- The then/else/elif/fi keywords take the place of braces ({})---braces mean something special to the shell!
- Examples of *condition* are (assuming integer a b):

```
(( a == 10 ))
(( b < 20 ))
[[ $person = steve ]]
[[ $person != todd ]]
(( (a < 10) || (a > 100) ))
[[ ($person != steve ) && ($person != todd ) ]]
```

- The elif or else parts can be omitted.

## Example If Statement

```
if [[ $person = steve ]]
then
   print $person is on the sixth floor.
elif [[ $person = todd ]]
then
        print $person is on the fifth floor.
elif [[ $person = markus ]]
then
        print $person is on the fifth floor.
else
   print "Who are you talking about?"
fi
```

## Additional Condition Tests

- In addition to the string relational operators =, !=, <, >, and the arithmetic relational operators (for strings representing integers) -lt, -le, -eq, -ge, -gt, -ne, there are other basic conditions you can test:

| | |
|---|---|
| -n *string* | string not null? |
| -z *string* | string null? |
| -a *filename* | exists? |

-f *filename* is plain file?

-d *filename* is directory?

-L *filename* is symbolic link?

-s *filename* exists and not empty?

-r *filename* read permission?

-w *filename* write permission?

-x *filename* execute permission?

-O *filename* your file?

-G *filename* your group?

*file1* -nt *file2* *file1* newer than *file2*?

*file1* -ot *file2* *file1* older than *file2*?

- Example:

```
if [[ ! -f output.file ]]; then
   print "output.file does not exist."
fi
```

- Example using return code:

```
opt="-f -d -L"
if print - $opt | grep -q -e -d
  then print "Option '-d' in list."
fi
```

## Command Substitution

- $(command) is literally replaced by the output from command (the Bourne shell syntax for this was graves: `command`).

```
$ set -A today $(date)
$ print ${today[*]}
Mon Oct 20 07:31:58 EDT 1997

$ print ${#today[*]}
6
$ print "${today[1]} ${today[2]}, ${today[5]}"
Oct 20, 1997
```

## While Statements

- Basic form:

```
while condition
do
   statements
done
```

- *condition* has the same syntax as for if statements.
- You can use break or continue or return inside a loop with the same meaning as in C.

- Example:

```
#!/usr/local/bin/ksh
# Report type of executable file anywhere in search path.
#
path=$PATH
dir=${path%%:*}
while [[ -n $path ]]; do
  if [[ -x $dir/$1 && ! -d $dir/$1 ]]; then
            file $dir/$1
            return
  fi
  path=${path#*:}
  dir=${path%%:*}
done
print "File not found."
return 1
```

# Case Selection

- The basic form of the case statement is:

```
case expression in
   pattern1 )
            statements ;;
          pattern2 )
            statements ;;
          .
          .
esac
```

- The statements corresponding to the first pattern matching the expression are executed, after which the case statement terminates.
- The *expression* is usually some variable's value.
- The *patterns* can be plain strings, or they can be Korn shell patterns using *, ?, !, [], etc. (like file-matching patterns).
- A pattern can consist of several patterns separated by | (logical or), and the pattern can also be written as (pattern).

# Sample Case

```
case $person in
   steve)
            print "He's on the sixth floor." ;;
          todd | markus)
            print "He's on the fifth floor." ;;
          *)
            print I do not know $person. ;;
esac
```

# For Loops

- A for loop can be used to iterate over all items in an array or list.
- Basic form:

```
for var [in list]
do
   statements
done
```

- If in list is omitted in a script, the list is assumed to be $* --- all the arguments to the script.
- Examples:

```
for name in Jack Jill John Jane Dick
do
  print "Next person is $name."
```

```
done

integer count=0
for arg in $@
do
  let count='count+1'
  print "argument $count: $arg"
done

for i in *.for
do
  mv $i ${i%.for}.f
  print ${i%.for}.f
done
```

## Command Line Arguments

- Arguments given to a shell script on the command line when it is invoked are available through the variable $* (a space separated list) and $@ (a list with each argument double quoted separately). Individual arguments to the shell script are referenced as $1, $2, $3, etc., and $0 is the name of the shell script itself.
- $# indicates how many arguments were passed.
- Example (print out all arguments to a shell script):

```
#!/usr/local/bin/ksh
integer i=1
for arg in $@
do
print "Argument $i is '$arg'."
(( i+=1 ))
done
return 0
```

- At a time you can have 9 arguments to be stored in the variables $1 to $9. The **shift** command moves parameters one position left i.e. $1=$2, $2=$3,…..,$9= $10^{th}$ command line argument.

## 3) Stage V (verify)

### Home Activities:

4) Write a shell script that reads 10 number from user and then prints all those numbers that are odd
5) Find prime numbers from an array

## 4) Stage $a_2$ (assess)

**Activity assessment and Viva voce at the end of lab.**

# LAB # 13

## Statement Purpose:

This lab will give you'll use command line arguments with shell scripting.

## Activity Outcomes:

The primary objective of this lab is to implement the shell scripting functions:

- Using command line arguments
- Using functions

## 1) Stage J (Journey)

## Introduction

Arguments given to a shell script on the command line when it is invoked are available through the variable $* (a space separated list) and $@ (a list with each argument double quoted separately). Individual arguments to the shell script are referenced as $1, $2, $3, etc., and $0 is the name of the shell script itself.

- $# indicates how many arguments were passed.

## Functions

- Functions are used to structure the script code.
- Functions can be defined any where in the shell script, however, these must be defined before invoking.
  Syntax: fname() {
  Statements }
- When a function is invoked, the positional parameters to the script $*, $@, $#, $1…$9 are replaced by the parameters in the function. That is how the parameters are passed to the function. When the function finishes, they are restored to their previous values.
- We can make functions return numeric values using the **return** command
- Local variables can be declared within shell functions by using the **local** keyword, the variable is then only in scope within the function.

## 2) Stage a1 (apply)

## Lab Activities:

**Example:**

- **print out all arguments to a shell script:**

```
#!/usr/local/bin/ksh
integer i=1
for arg in $@
do
print "Argument $i is '$arg'."
(( i+=1 ))
done
return 0
```

- At a time you can have 9 arguments to be stored in the variables $1 to $9. The **shift** command moves parameters one position left i.e. $1=$2, $2=$3,…..,$9= 10th command line argument.

### Activity 1:

**Using function:**

```
myvar=global
    f1()
            { local myvar=”local”
                    echo f1 is called and the value of myvar is $myvar
                    echo $#  arguments are passed to it and their values are $@
            }
    echo Script started
    echo $# arguments are passed to the script which are: $*
    f1 arg1 arg2 arg3
    echo the value of global variable myvar is $myvar
    echo after calling the function f1, the arguments in the script are $*
    f2()
            {       echo f2 is called, the value of myvar is $myvar
                    while true ; do
                            echo –n Are you ok(yes or no)?
                            read  x
                                    case “$x”  in
                                     y | yes)  return 0;;
                                     n | no)  return 1;;
                                    *) echo “Answer: yes or no”
                            esac
                    done
            }
    if  [ f2 –eq 1]
            echo thanks God
    else
            echo Get well soon
    fi
    exit 0
```

## 3) Stage V (verify)

### Home Activities:

- Write a shell script that reads 10 number from user and then prints all those numbers that are odd
- Find prime numbers from an array

## 4) Stage $a_2$ (assess)

**Activity assessment and Viva voce at the end of lab.**

# LAB # 14

## Statement of Purpose:

This lab will introduce the basic concept of file arrays in shell scripting

## Activity Outcomes:

This lab teaches you the following topics:

1. Creating arrays
2. Accessing arrays
3. Performing different operations such as deleting, sorting on arrays

# 1) Stage J (Journey)

## What Are Arrays?

Arrays are variables that hold more than one value at a time. Arrays are organized like a table. Let's consider a spreadsheet as an example. A spreadsheet acts like a *two-dimensional array*. It has both rows and columns, and an individual cell in the spreadsheet can be located according to its row and column address. An array behaves the same way. An array has cells, which are called *elements*, and each element contains data. An individual array element is accessed using an address called an *index* or *subscript*.
Most programming languages support *multidimensional arrays*. A spreadsheet is an example of a multidimensional array with two dimensions, width and height. Many languages support arrays with an arbitrary number of dimensions, though two- and three-dimensional arrays are probably the most commonly used.
Arrays in bash are limited to a single dimension. We can think of them as a spreadsheet with a single column. Even with this limitation, there are many applications for them. Array support first appeared in bash version 2. The original Unix shell program, sh, did not support arrays at all.

## Creating an Array

Array variables are named just like other bash variables, and are created automatically when they are accessed. Here is an example:

```
[me@linuxbox ~]$ a[1]=foo
[me@linuxbox ~]$ echo ${a[1]}
foo
```

An array can also be created with the declare command:

```
[me@linuxbox ~]$ declare -a a
```

Using the -a option, this example of declare creates the array a.

## Assigning and Accessing Values to an Array

Values may be assigned in one of two ways. Single values may be assigned using the following syntax:
name[subscript]=value

where name is the name of the array and subscript is an integer (or arithmetic expression) greater than or equal to zero. Note that the first element of an array is subscript zero, not one. value is a string or integer assigned to the array element.
Multiple values may be assigned using the following syntax:

*name*=(*value1 value2* ...)

where *name* is the name of the array and *value...* are values assigned sequentially to elements of the array, starting with element zero. For example, if we wanted to assign abbreviated days of the week to the array days, we could do this:

```
[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)
```

It is also possible to assign values to a specific element by specifying a subscript for each value:

```
[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu
```

The array elements can be accessed using array name and the index of the element to be accessed. Index.

## Array Operations

There are many common array operations. Such things as deleting arrays, determining their size, sorting, etc. have many applications in scripting.

### Outputting the Entire Contents of An Array

The subscripts * and @ can be used to access every element in an array. As with positional parameters, the @ notation is the more useful of the two. Here is a demonstration:

```
[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")
[me@linuxbox ~]$ for i in ${animals[*]}; do echo $i; done
a
dog
a
cat
a
fish
```

### Determining the Number of Array Elements

Using parameter expansion, we can determine the number of elements in an array in much the same way as finding the length of a string. Here is an example:

```
[me@linuxbox ~]$ a[100]=foo
[me@linuxbox ~]$ echo ${#a[@]}  # number of array elements
1
[me@linuxbox ~]$ echo ${#a[100]}  # length of element 100
3
```

### Adding Elements to the End of an Array

Knowing the number of elements in an array is no help if we need to append values to the end of an array, since the values returned by the * and @ notations do not tell us the maximum array index in use. Fortunately, the shell provides us with a solution. By using the += assignment operator, we can automatically append values to the end of an array. Here, we assign three values to the array foo, and then append three more.

```
[me@linuxbox ~]$ foo=(a b c)
[me@linuxbox ~]$ echo ${foo[@]}
a b c
[me@linuxbox ~]$ foo+=(d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

**Sorting an Array**

Just as with spreadsheets, it is often necessary to sort the values in a column of data. The shell has no direct way of doing this, but it's not hard to do with a little coding:

```
#!/bin/bash

# array-sort : Sort an array

a=(f e d c b a)

echo "Original array: ${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))
echo "Sorted array:   ${a_sorted[@]}"
```

**Deleting an Array**

To delete an array, use the unset command:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

```
[me@linuxbox ~]$ unset foo
[me@linuxbox ~]$ echo ${foo[@]}

[me@linuxbox ~]$
```

## 2) Stage a1 (apply)

## Activity 1

Write a shell script that

1. Prints all elements of an array
2. Prints first element of an array
3. Prints an element at a specific index

4. Prints elements in a range
5. Prints size of an array
6. Search a specific element in an array
7. Search and replace specific elements

# Solution:

```
#! /bin/bash
# To declare static Array
arr=(Ali ahemd 1 umar asad hassan)

# To print all elements of array
echo ${arr[@]}   # Ali ahemd 1 umar asad hassan
echo ${arr[*]}   # Ali ahemd 1 umar asad hassan
echo ${arr[@]:0} # Ali ahemd 1 umar asad hassan
echo ${arr[*]:0} # Ali ahemd 1 umar asad hassan

# To print first element
echo ${arr[0]}   # Ali
echo ${arr}      # Ali

# To print particular element
echo ${arr[3]}   # umar
echo ${arr[1]}   # ahmed

# To print elements from a particular index
echo ${arr[@]:0} # Ali ahemd 1 umar asad hassan
echo ${arr[@]:1} # ahemd 1 umar asad hassan
echo ${arr[@]:2} # 1 umar asad hassan
echo ${arr[0]:1} # li

# To print elements in range
echo ${arr[@]:1:4} # ahemd 1 umar asad hassan
echo ${arr[@]:2:3} # 1 umar asad
echo ${arr[0]:1:3} # li

# Length of Particular element
echo ${#arr[0]}  # 3
echo ${#arr}     # 7

# Size of an Array
echo ${#arr[@]} # 6
echo ${#arr[*]}  # 6

# Search in Array
echo ${arr[@]/*[aA]*/} # 1

# Replacing Substring Temporary
echo ${arr[@]//a/A}
echo ${arr[@]}
echo ${arr[0]//r/R}
```

## Activity 2

Write a shell script that

- That asks the user to enter total number of array elements
- Reads input from keyboard and then stores it in array
- Prints the array elements

## Solution:

```
# !/bin/bash
# To input array at run
# time by using while-loop

# echo -n is used to print
# message without new line
echo -n "Enter the Total numbers :"
read n
echo "Enter numbers :"
i=0

# Read upto the size of
# given array starting from
# index, i=0
while [ $i -lt $n ]
do
        # To input from user
        read a[$i]

        # Increment the i = i + 1
        i=`expr $i + 1`
done

# To print array values
# starting from index, i=0
echo "Output :"
i=0

while [ $i -lt $n ]
do
        echo ${a[$i]}

        # To increment index
        # by 1, i=i+1
        i=`expr $i + 1`
done
```

# 4) Stage V (verify)

## Home Activities:

5) Write a shell script that reads 10 number from user and then prints all those numbers that are odd

6) Find prime numbers from an array

## 4) Stage $a_2$ (assess)

**Activity assessment and Viva voce at the end of lab.**