Lab Manual

CSC322-Operating Systems

CUI

Department of Computer Science
Islamabad Campus

Lab Contents:

Installating Linux Distrubutions on Virtaul Machnes; Linux Command-line Interface: Basic Syntax, Navigation Commands, File & Directory Handling Commands, I/O Redirection, Controlling Access to Files, Package Management, Text-processing, Pipelining, Process Management; Writing & Compiling C++ on Linux; Process Management: Creating Child Processes, IPC with Shared-Memory & Message-Passing, Multithreading, Synchronization; Shell Scripting: Fundamentals, I/O, Variables, Operators, Conditional Statements, Looping Statements, Arrays, and Functions.

Student Outcomes (SO)

S.#	Description
1	Apply knowledge of computing fundamentals, knowledge of a computing specialization, and mathematics, science, and domain knowledge appropriate for the computing specialization to the abstraction and conceptualization of computing models from defined problems and requirements
2	Identify, formulate, research literature, and solve complex computing problems reaching substantiated conclusions using fundamental principles of mathematics, computing sciences, and relevant domain disciplines
4	Create, select, adapt and apply appropriate techniques, resources, and modern computing tools to complex computing activities, with an understanding of the limitations

Intended Learning Outcomes

Sr.#	Description	Blooms Taxonomy Learning Level	SO
CLO -5	Operate basic services and functionality of operating systems.	*Applying*	1
CLO -6	Compose Linux commands using Shell scripting.	*Applying*	1,4
CLO -7	Implement the concepts of process management.	*Applying*	2,4

Lab Assessment Policy

The lab work done by the student is evaluated using rubrics defined by the course instructor, viva-voce, project work/performance. Marks distribution is as follows:

Assignments	Lab Mid Term Exam	Lab Terminal Exam	Total
25	25	50	100

Note: Midterm and Final term exams must be computer based.

List of Labs

Lab No. 01

Installing Linux Distribution on Virtual Machine, Command-line Interface

Objective:

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

Activity Outcomes:

On completion of this lab students will be able to:

- 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) Useful Concepts

Operating System

An operating system (OS) is a program that interacts as interface between a user and ans a computer system software. It manages computer hardware, software resources, and provides common services for computer programs. Primary Goals of an Operating System include: To provide ease of use, convineance and throughput. The main functions perform by operating system can be categorized as: Process management, Resource Management, Stroge Management, Memory Management and Security Management.

Why Linux

Linux is among the most popular operating systems. The main reasons for this popularity are: Free and open source, Stable and Reliable, Secure, and Flexible.

Linux History

Linux was originally developed for ***personal computers*** based on the ***Intel x86*** architecture, but has since been ***ported*** to more ***platforms*** than any other operating system. In the early 1990s, Finnish computer science student Linus Torvalds began hacking on Minix, a small, Unix−like operating system for personal computers then used in college operating systems courses. He decided to improve the main software component underlying Minix, called the kernel, by writing his own. (The kernel is the central component of any Unix−like operating system.) OnSeptember 1991, Torvalds published the first version of this kernel on the Internet, calling it "Linux" (a play on both Minix and his own name).(7) When Torvalds published Linux, he used the copyleft software license published by the GNU Project, the GNU General Public License. Doing so made his software free to use, copy, and modify by anyone−−provided any copies or variations were kept equally free. Torvalds also invited contributions

by other programmers, and these contributions came; slowly at first but, as the Internet grew, thousands of hackers and programmers from around the globe contributed to his free software project. The Linux software was immensely extended and improved so that the Linux−based system of today is a complete, modern operating system, which can be used by programmers and non−programmers .

Popularity:

Because of the dominance of the Linux-based Android on smartphones, Linux also has the largest installed base of all general-purpose operating systems. Although Linux is used by only around 2.3 percent of desktop computers, the Chromebook, which runs the Linux kernel-based Chrome OS, dominates the US K–12 education market and represents nearly 20 percent of sub $300 notebook sales in the US. Linux is the leading operating system on servers (over 96.4% of the top 1 million web servers' operating systems are Linux), leads other big iron systems such as mainframe computers, and is the only OS used on TOP500 supercomputers (since November 2017, having gradually eliminated all competitors).

Linux also runs on embedded systems, i.e. devices whose operating system is typically built into the firmware and is highly tailored to the system. This includes routers, automation controls, smart home technology, televisions (Samsung and LG Smart TVs use Tizen and WebOS, respectively) , automobiles (for example, Tesla, Audi, Mercedes-Benz, Hyundai, and Toyota all rely on Linux), digital video recorders, video game consoles, and smartwatches. The Falcon 9's and the Dragon 2's avionics use a customized version of Linux.

Linux Distribution

Linux is open-source, free to use kernel. It is used by programmers, rganizations, profit and non-profit companies around the world to create Operating systems to suit their individual requirements.To prevent hacking attempts, many organizations keep their Linux operating systems private. Many others make their variations of Linux available publicly so the whole world can benefit at large. These versions/ types /kinds of Linux operating system are called Distributions. A list of most popular Linux distributions is given below:

Deepin	**Deepin** is a Linux desktop-oriented operating system derived from *Debian*, supporting laptops, desktops, and all-in-ones. It aims to provide a beautiful, easy-to-use, safe, and reliable operating system to global users.
MX Linux	It is one of the most popular Desktop Distributions available out there. It launched in 2006 and is now considered to be the fourth most used Operating system in the computing world.
archlinux	This Linux Distro is popular amongst Developers. It is an independently developed system. It is designed for users who go for a do-it-yourself approach.
slackware linux	Slackware aims for design stability and simplicity
redhat	Another popular enterprise based Linux Distribution is Red Hat Enterprise.It has evolved from Red Hat Linux which was discontinued in 2004. It is a commercial Distro and very popular among its clientele.

nubuntu	This is the third most popular desktop operating system after Microsoft Windows and Apple Mac OS. It is based on the Debian Linux Distribution, and it is known as its desktop environment.
	Another Linux kernel based Distro, Fedora is supported by the Fedora project, an endeavor by Red Hat. It is popular among desktop users. Its versions are known for their short life cycle.

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***.

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.

Installing Linux using Virtual Machine

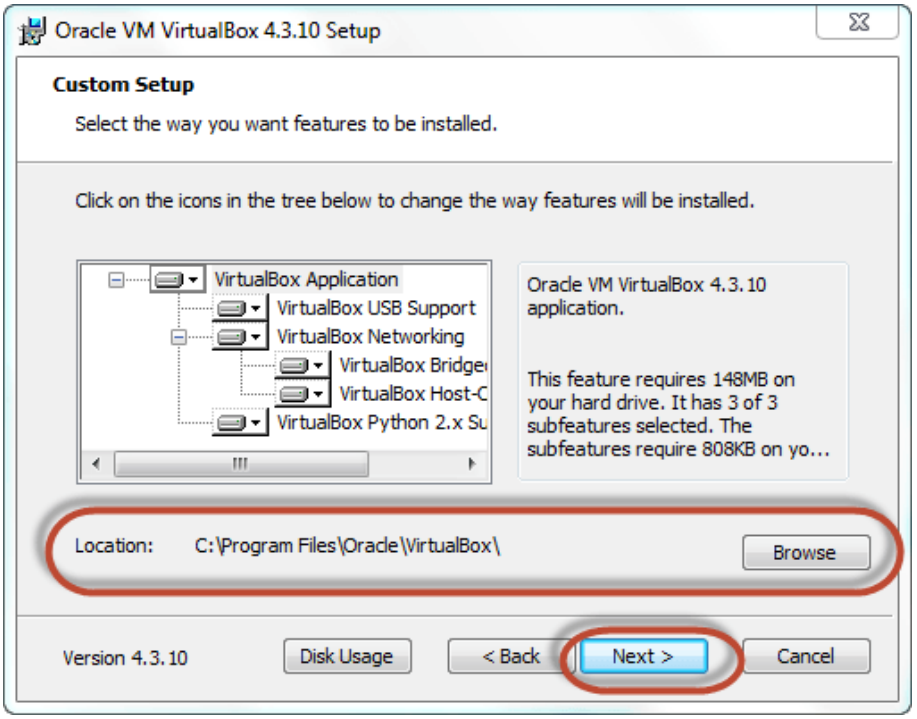
This is a popular method to install a Linux operating system. The virtual installation offers you the freedom of running on an existing OS already installed on your computer. This means if you have Windows running, then you can just run Linux with a click of a button.***Virtual machine software*** like Oracle VM can install Linux on Windows in easy steps. Let us look at them.
The following diagram shows the steps required to install Ubuntu on VM

Download and Install Virtual Box: Download Virtual box using this ***link*** Depending on your processor and OS, select the appropriate package. In our case, we have selectedwindows with AMD.

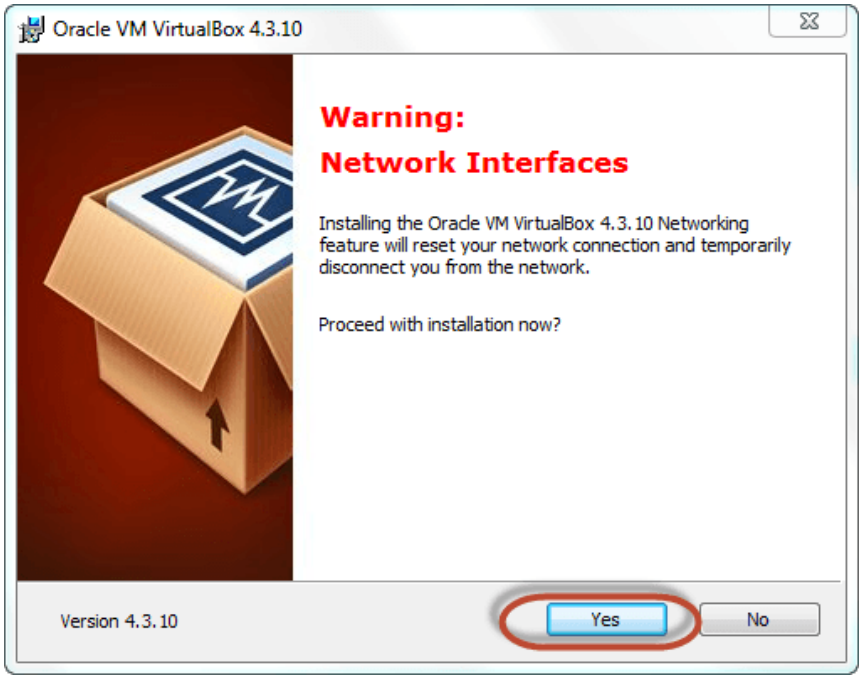
Once the download is complete, Open setup file and follow the steps below:

Click On next

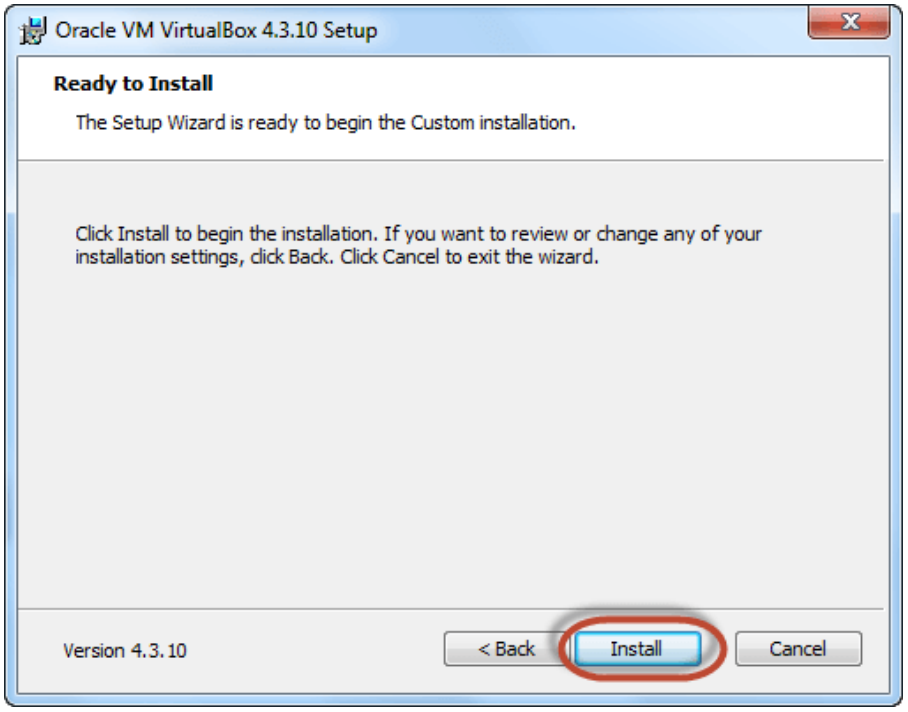
Select you're the directory to install VirtualBox and click on next



Select Desktop icon and click on next, now click on yes

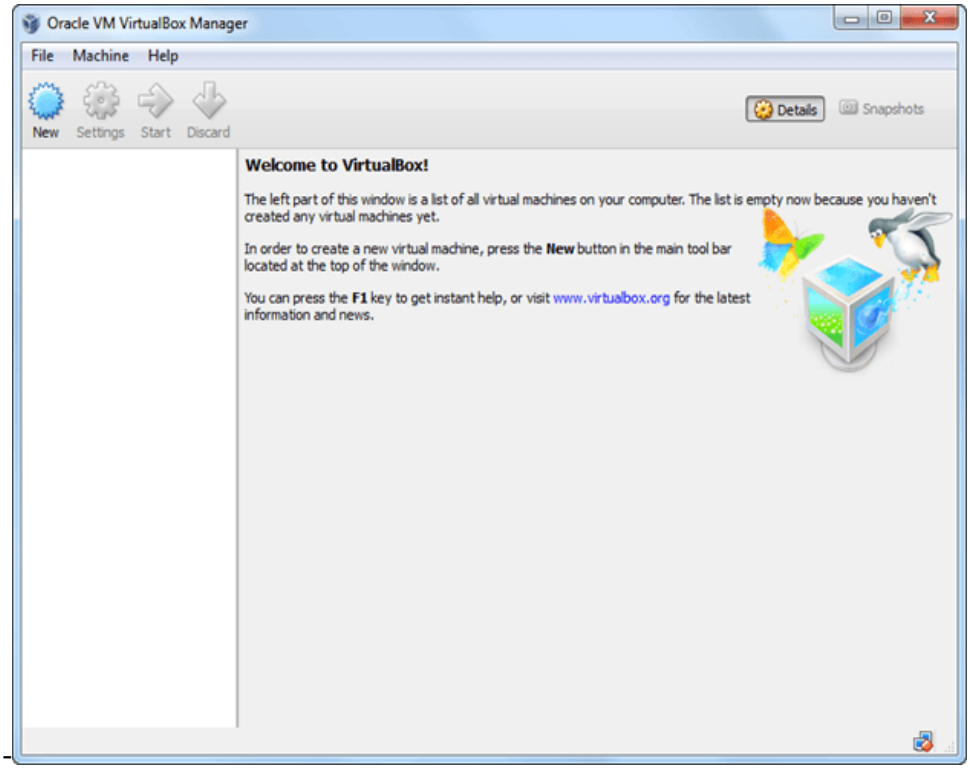


Click On install-to-install Linux on Windows.



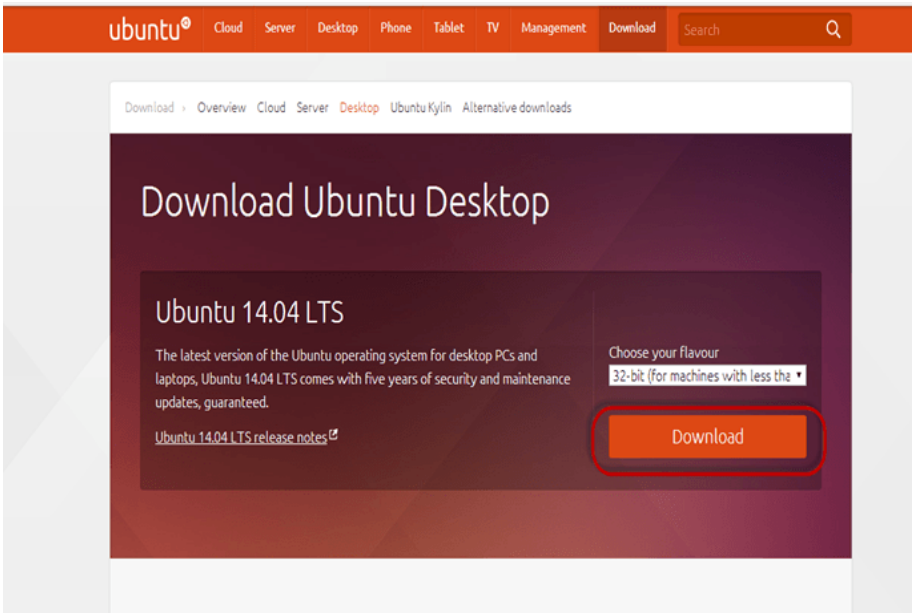
Now installation of the virtual box will start. Once complete, click on Finish Button to start Virtual Box

The virtual box dashboard looks like this



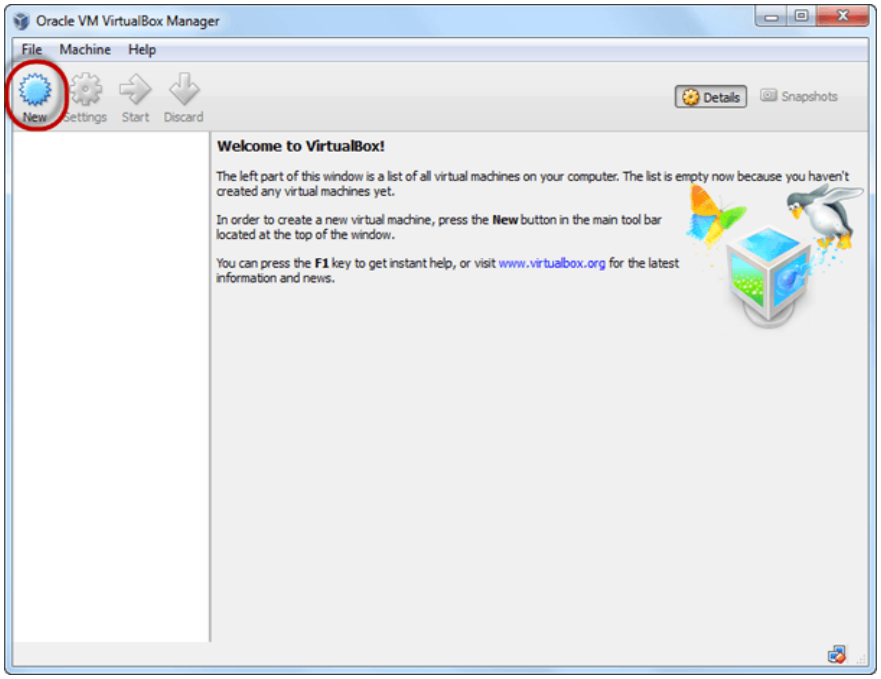
Download Ubuntu:

Visit this link to ***download*** Ubuntu.

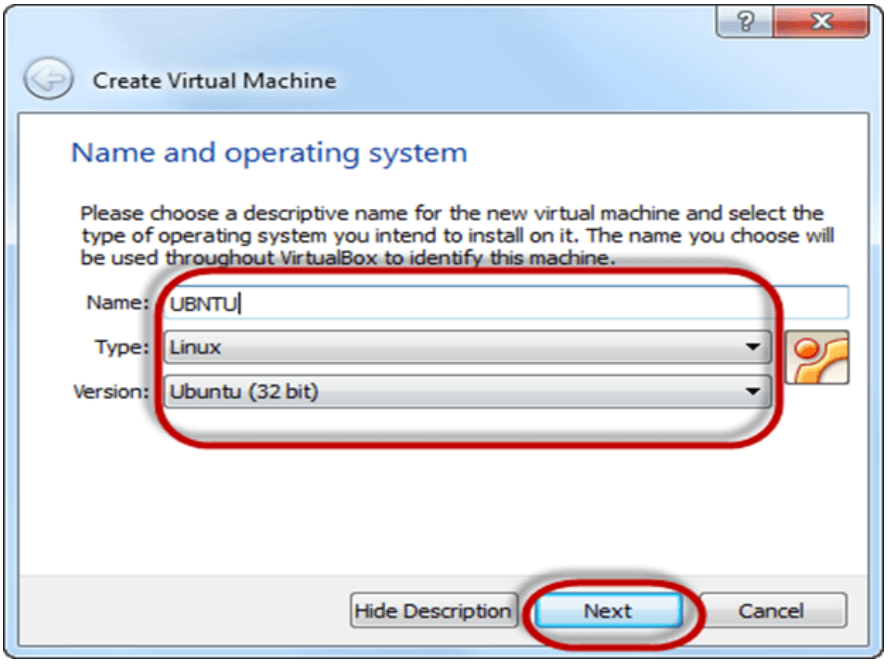


You can select 32/64-bit versions as per your choice.

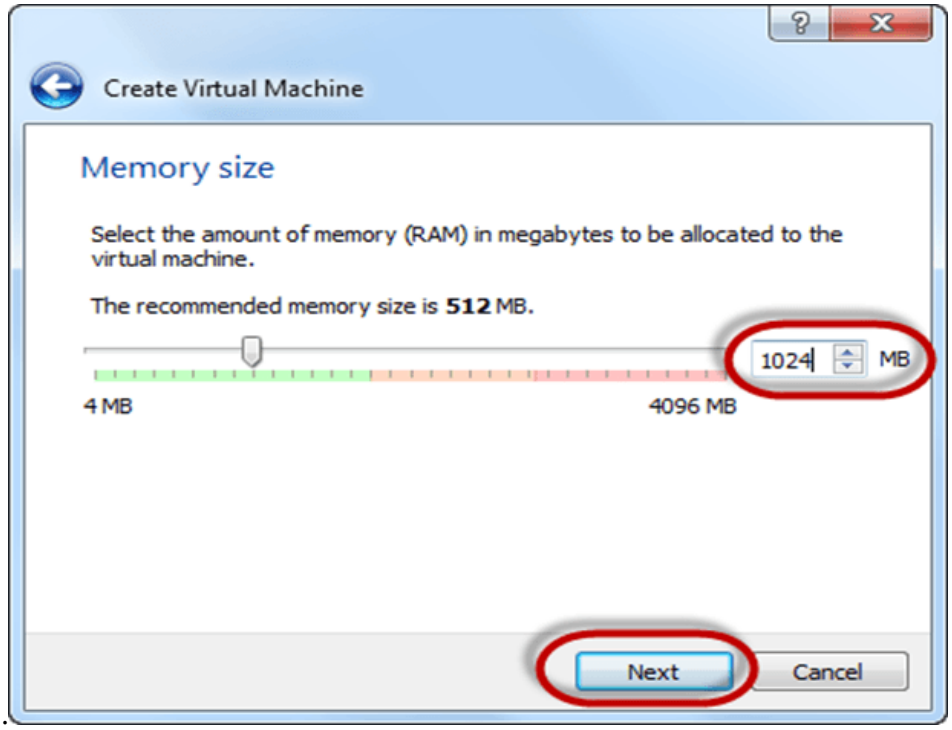
Create a Machine in Virtual Box: Open Virtual box and click on new button



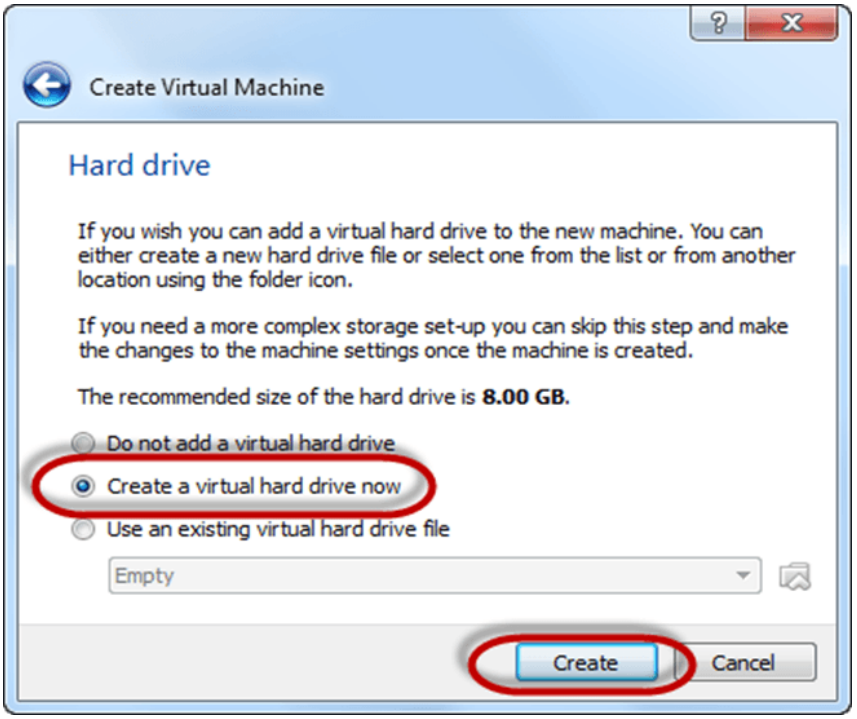
In next window, give the name of your OS which you are installing in virtual box. And select OS like ***Linux*** and version as Ubuntu 32 bit. And click on next



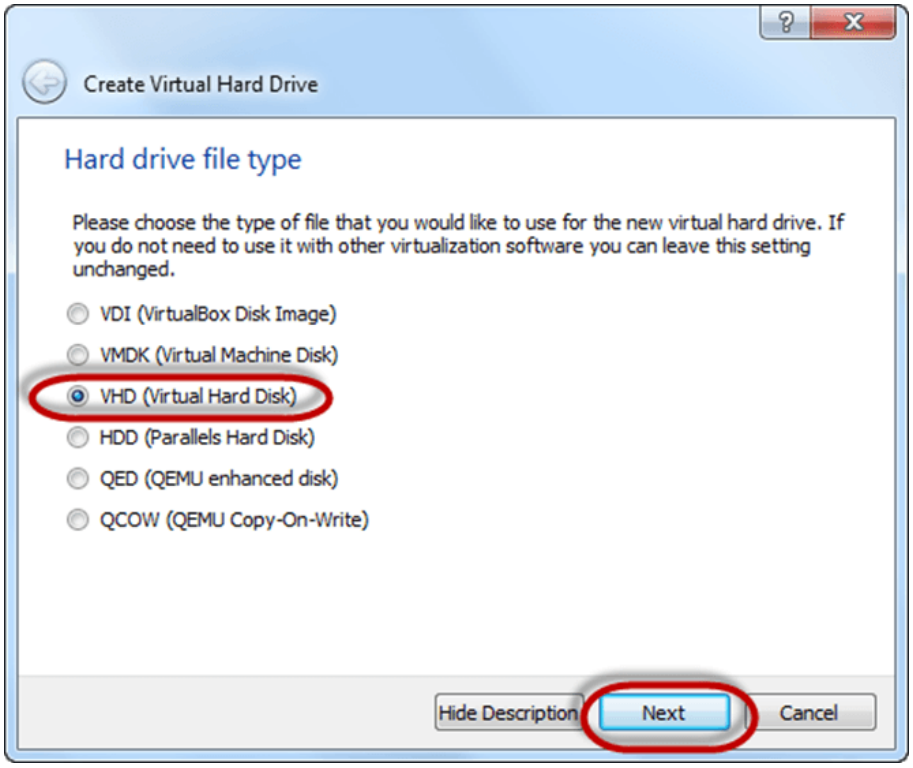
Now Allocate Ram Size To your Virtual OS. I recommended keeping 1024mb (1 GB) ram to run Ubuntu better. And click on next



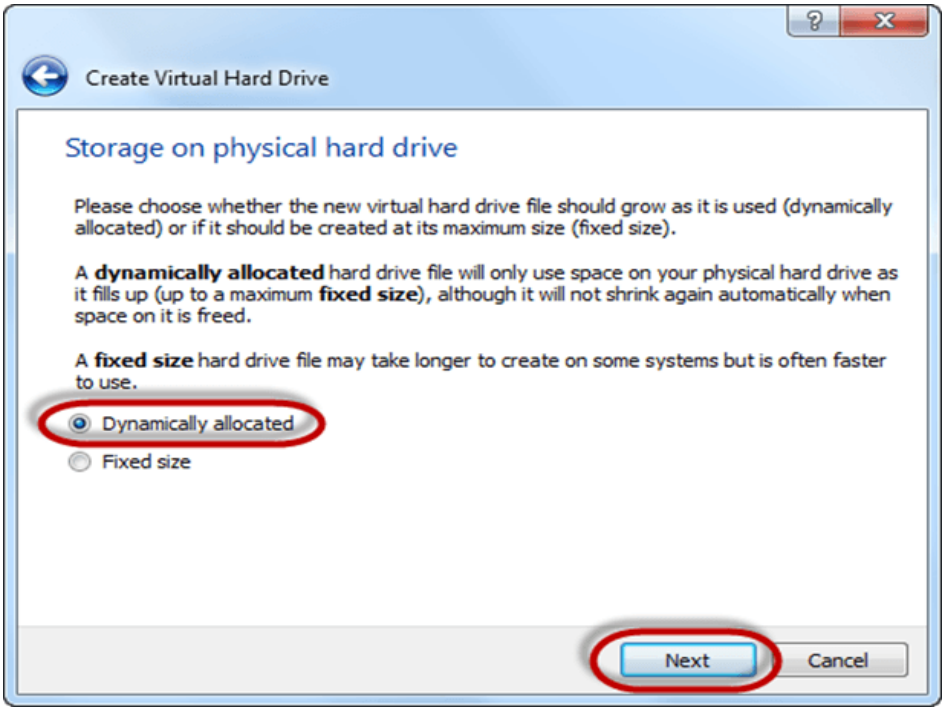
Now To run OS in virtual box we have to create virtual hard disk, click on create a virtual hard drive now and click on create button. The virtual hard disk is where the OS installation files and data/applications you create/install in this Ubuntu machine will reside



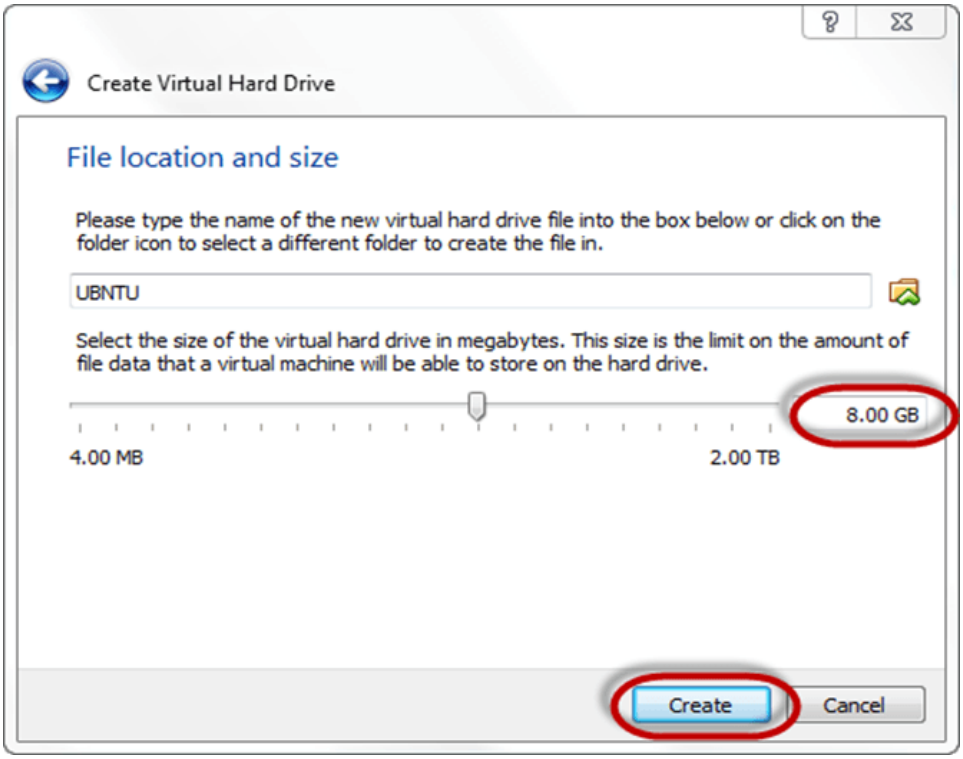
select VHD (virtual hard disk) option and click on next.



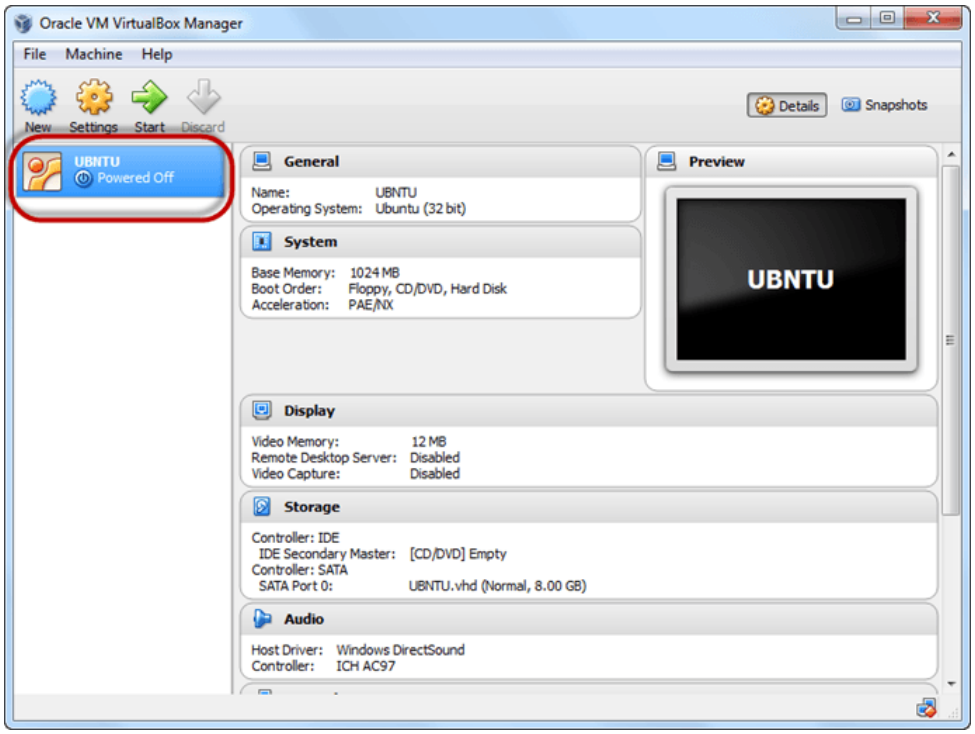
Click on dynamic allocated and click on next. This means that the size of the disk will increase dynamically as per requirement.



Allocate memory to your virtual hard drive .8GB recommended. Click on create button.



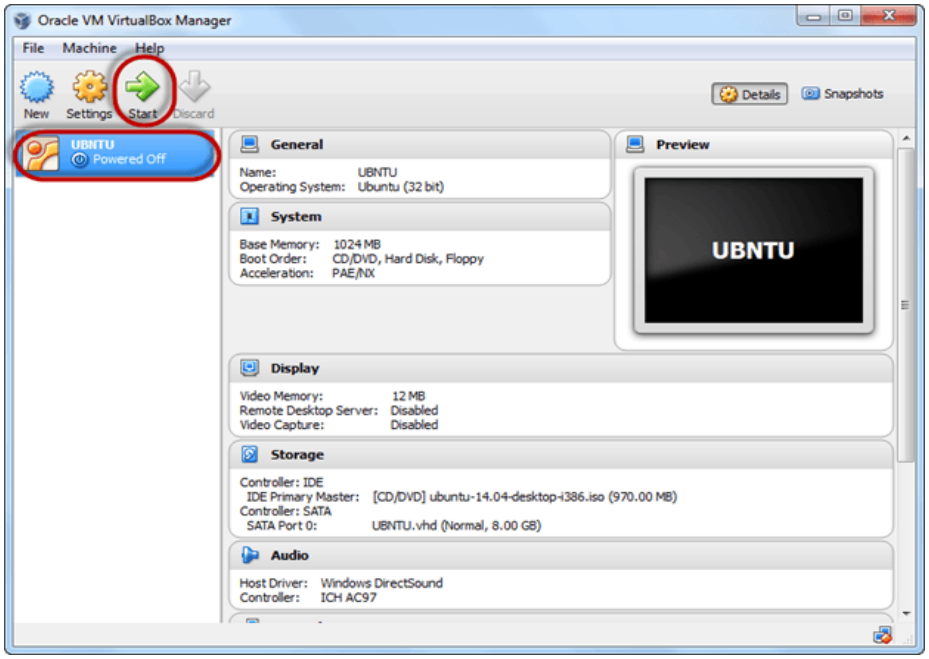
Now you can see the machine name in left panel



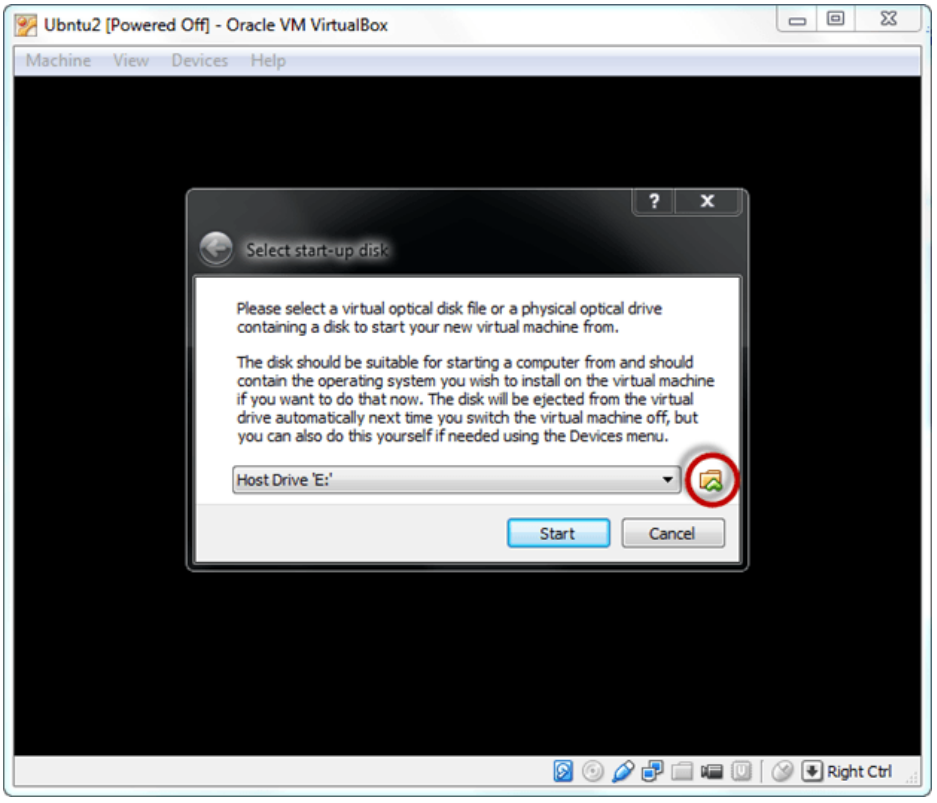
So a Machine (PC) with 8GB Hardisk, 1GB RAM is ready.

How to Install Ubuntu

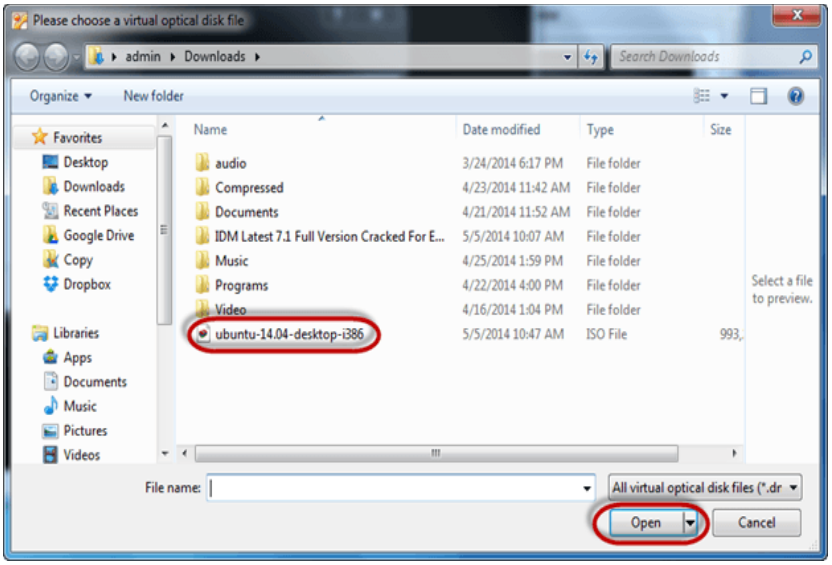
Select the Machine and Click on Start



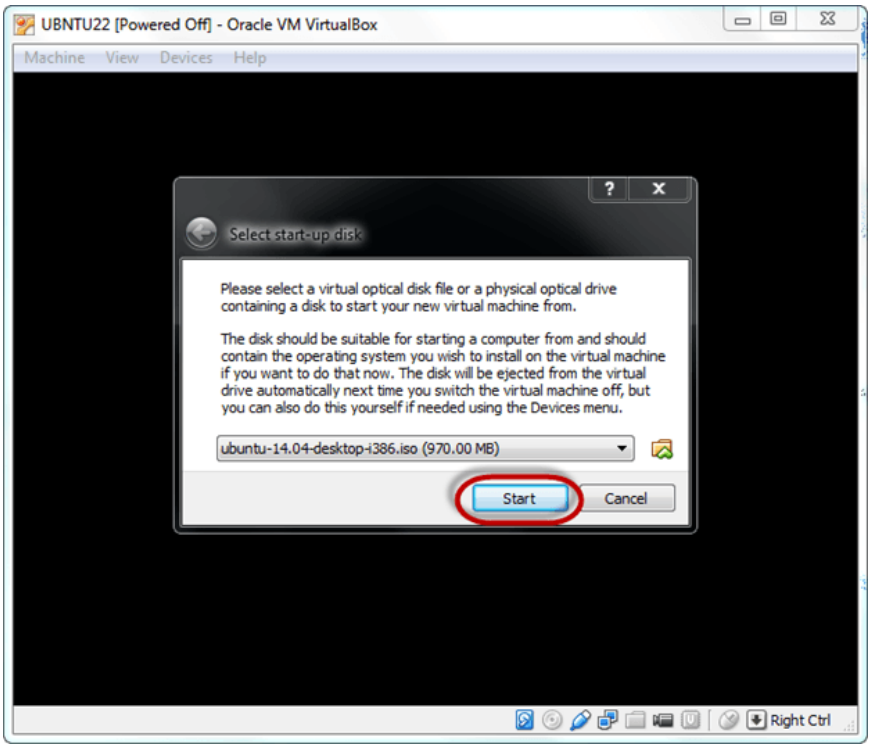
Select the Folder Option



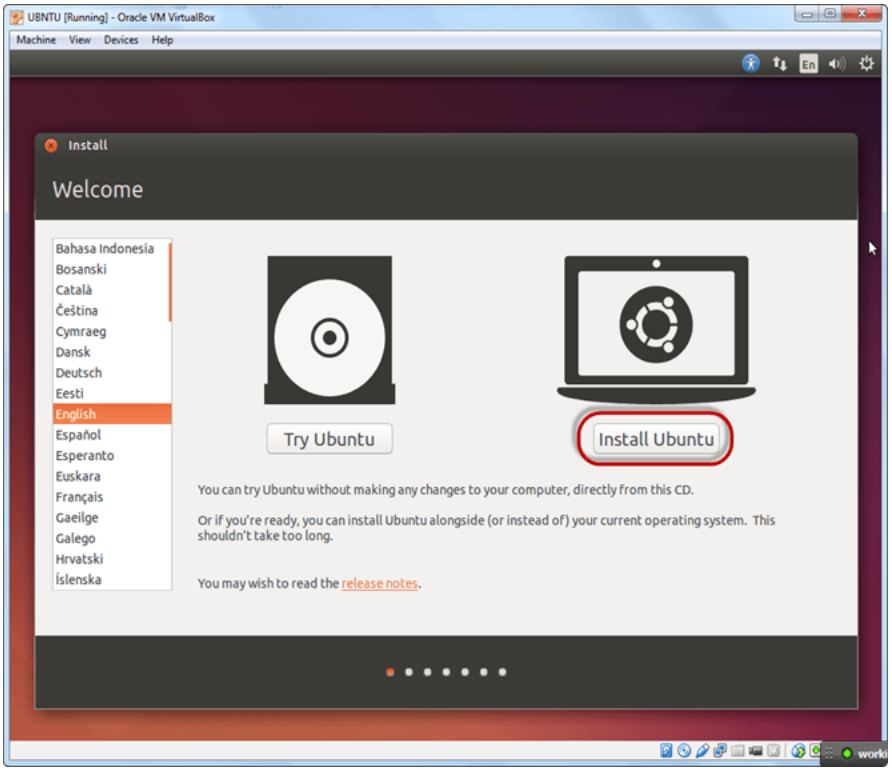
Select the Ubuntu iso file



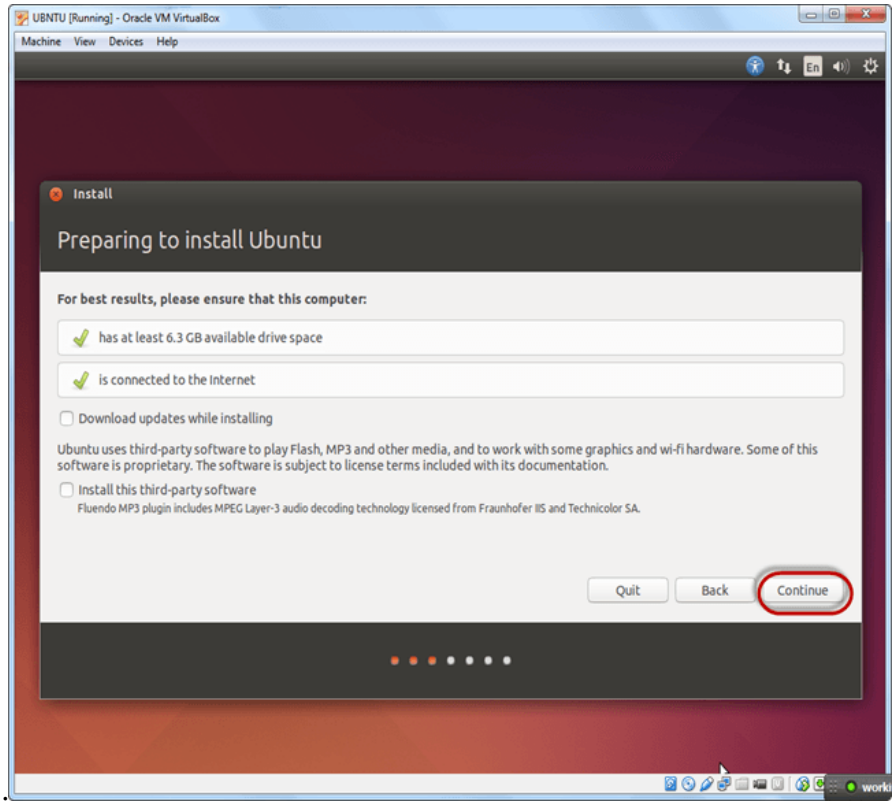
Click Startup



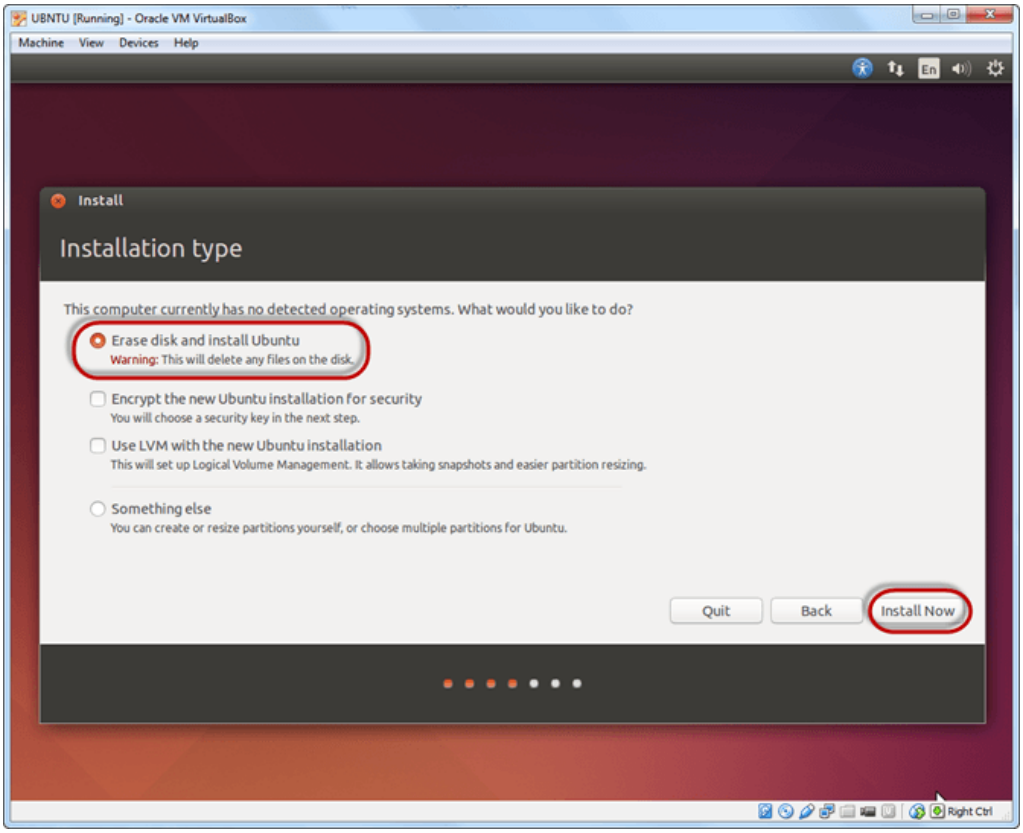
You have an option to Run Ubuntu WITHOUT installing. In this tutorial will install Ubuntu



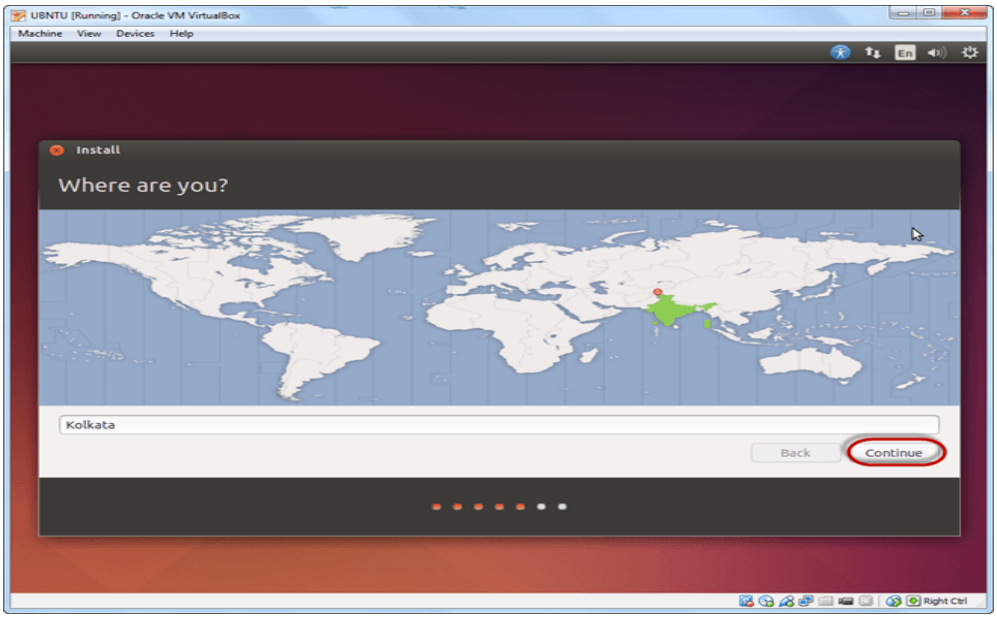
Click continue



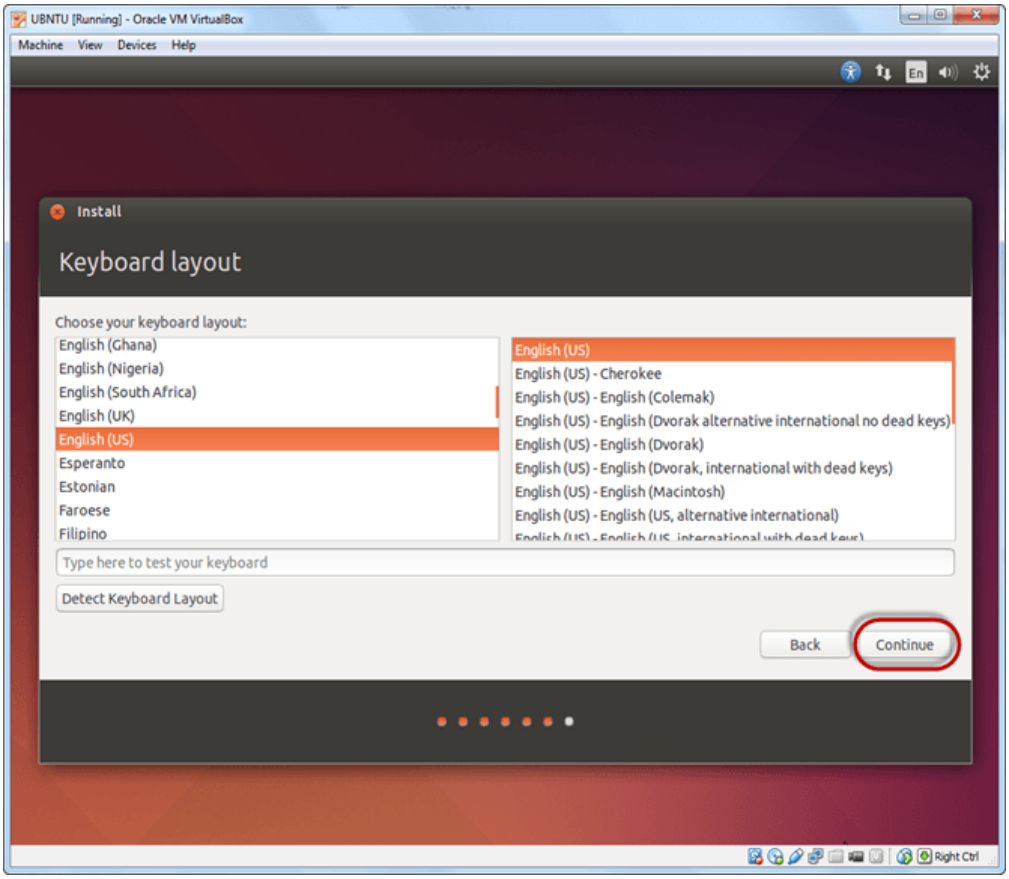
Select option to erase the disk and install Ubuntu and click on install now. This option installs Ubuntu into our virtual hard drive which is we made earlier. It will not harm your PC or Windows installation



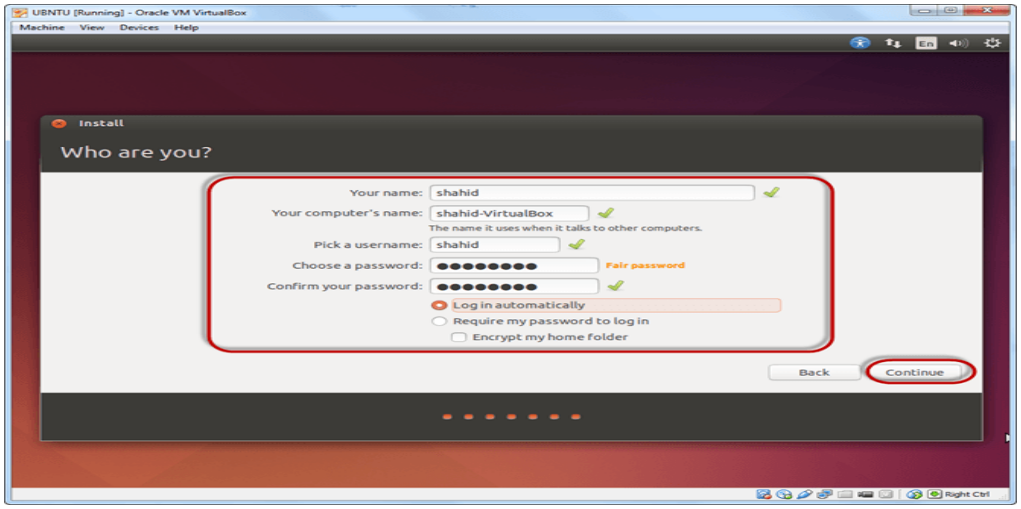
Select your location for setting up time zone, and click on continue



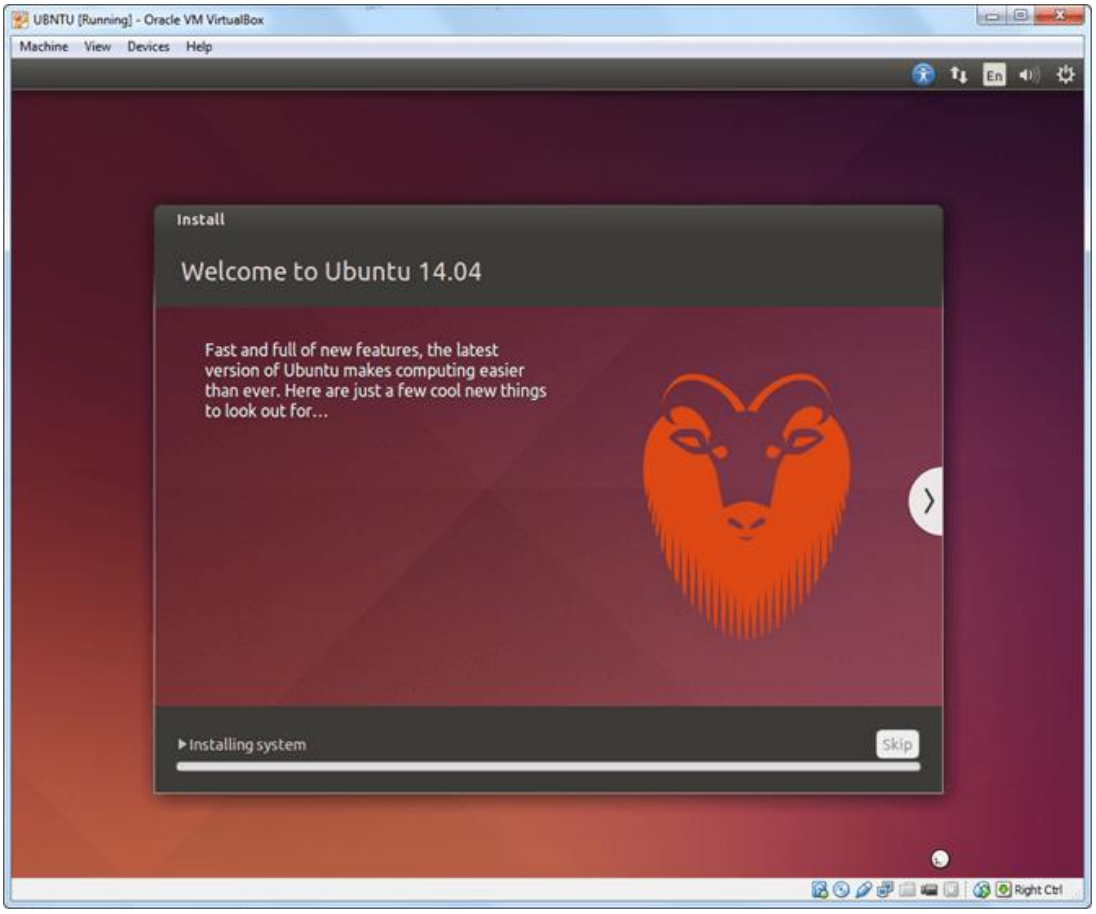
Select your keyboard layout, by default English (US) is selected but if you want to change then, you can select in the list. And click on continue



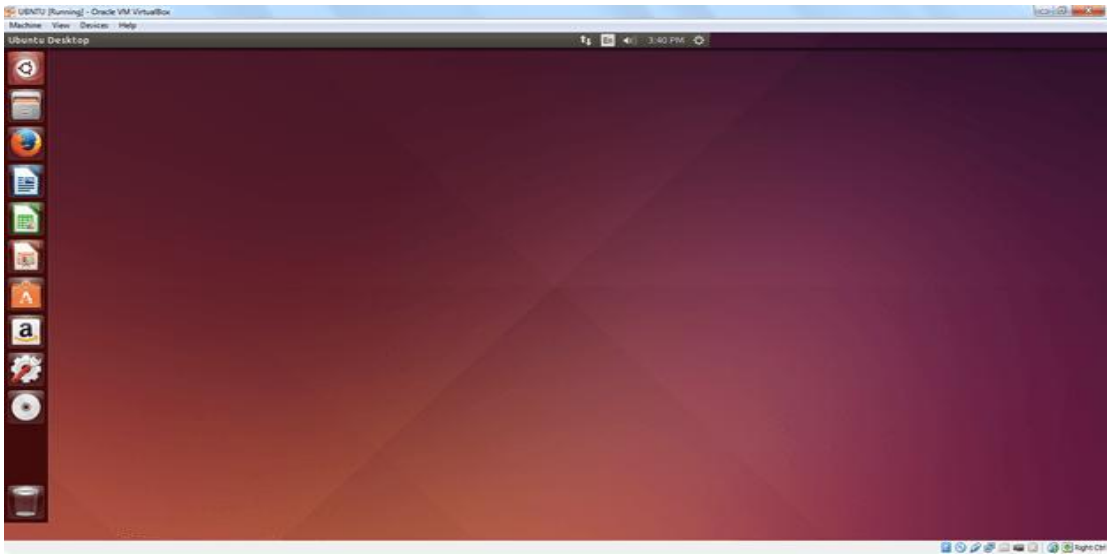
Select your username and password for your Ubuntu admin account. This information has been needed for installing any software package into Ubuntu and also for login to your OS. Fill up your details and tick on login automatically to ignore login attempt and click on continue



Installation process starts. May take up to 30 minutes. Please wait until installation process completes.



After finishing the installation, you will see Ubuntu Desktop.



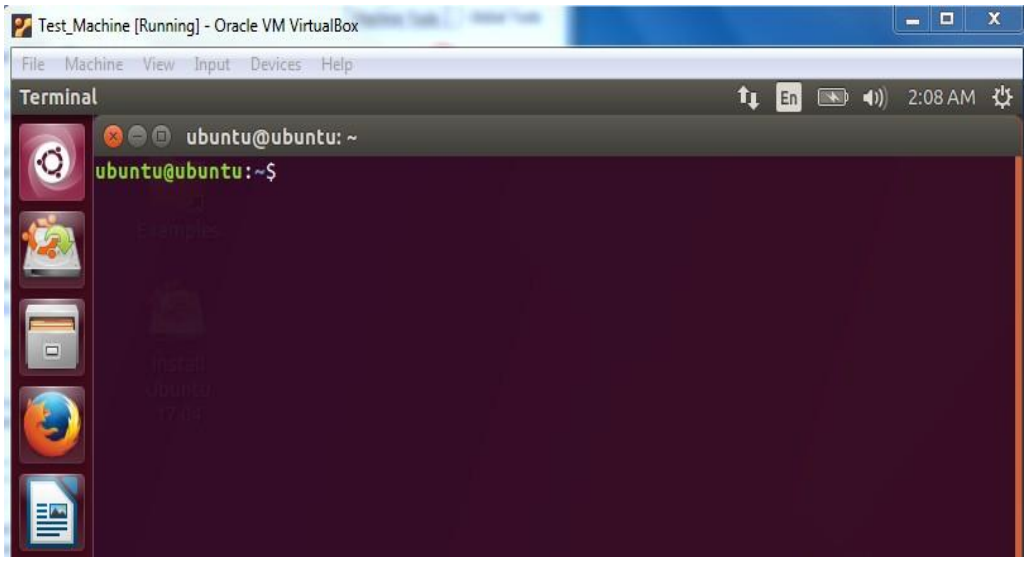
Writing Linux Commands

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.



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 c command line (i.e., the all-text display mode) and then pressing the ENTER key, which passes themto 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 -optioins  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 keyand the previous command disappears.

Some Basic Linux Commands

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

```
$date or $date +%ch
```

Common 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.

2) cal Command: This command is used to display the calendar of the year or the particularmonth of calendar year.

```
$cal <year> or $cal <month> <year>
```

3) who Command: It is used to display who are the users connected to our computer currently.

```
$who –option"s
```

Common Options:

H–Display the output with headers

b–Display the last booting date or time or when the system was lastly rebooted

4) whoami Command: Display the details of the current working directory.

```
$whoami
```

5) clear Command: It is used to clear the screen.

```
$clear
```

6) man Command: It help us to know about the particular command and its options & working. It is like„help" command in windows .

```
$man <command name>
```

7) df Command: is used to see the current amount of free space on your disk drives

```
$df
```

8) free Command: Likewise, to display the amount of free memory, enter the **free** command.

```
$free
```

9) exit Command: We can end a terminal session by either closing the terminal emulator window, or by enteringthe **exit** command at the shell prompt

$exit

2) Solved Lab Activities

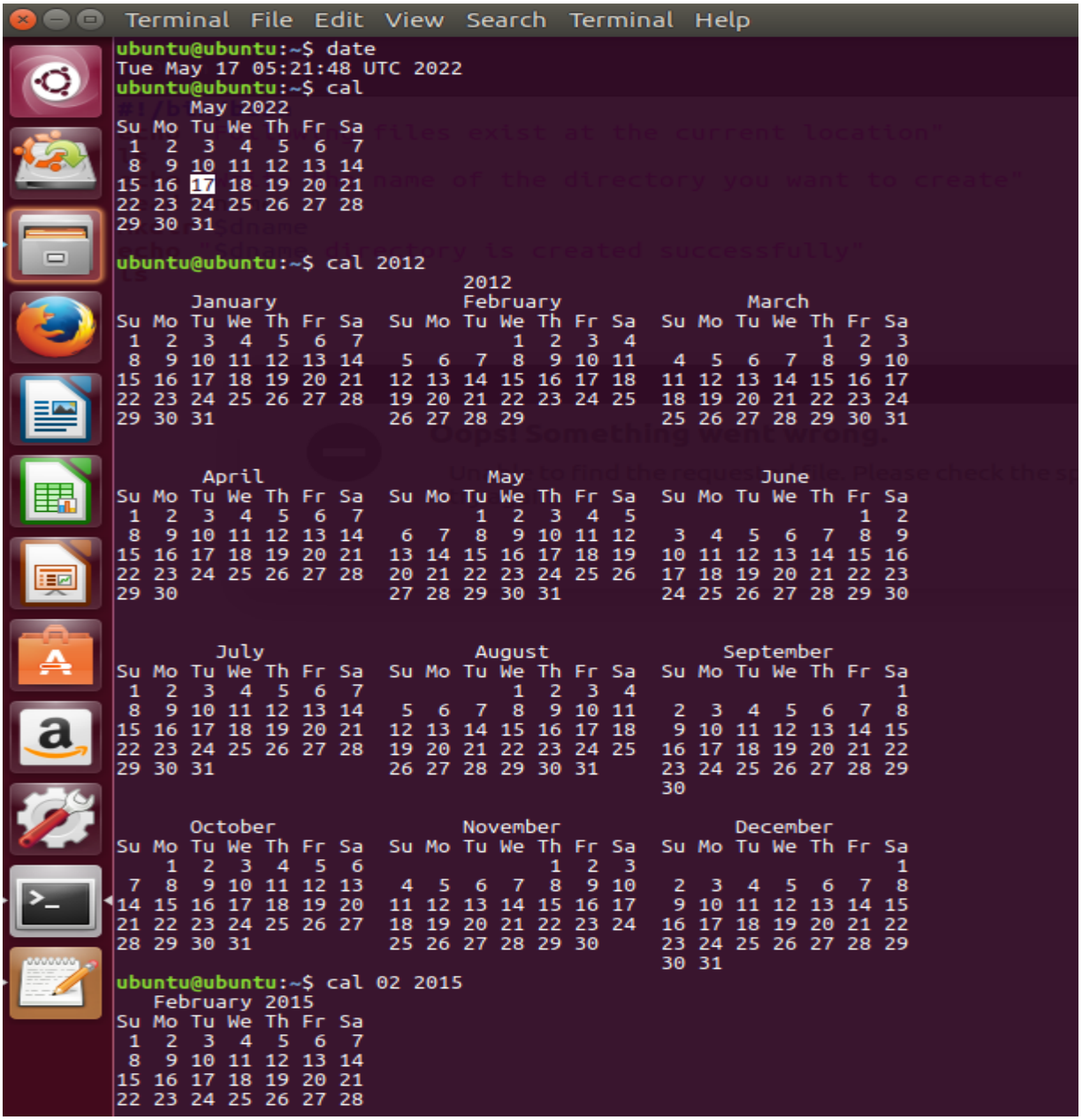
Sr.No	Allocated Time	Level of Complexity	CLO Mapping
1	10	Medium	CLO-5
2	10	Medium	CLO-5

Activity 1:

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

- ***Display the current date***
- ***Display the calendar for the current month***
- ***Display the calendar of 2012***
- ***Display the calendar of Feb 2015***

Solution:

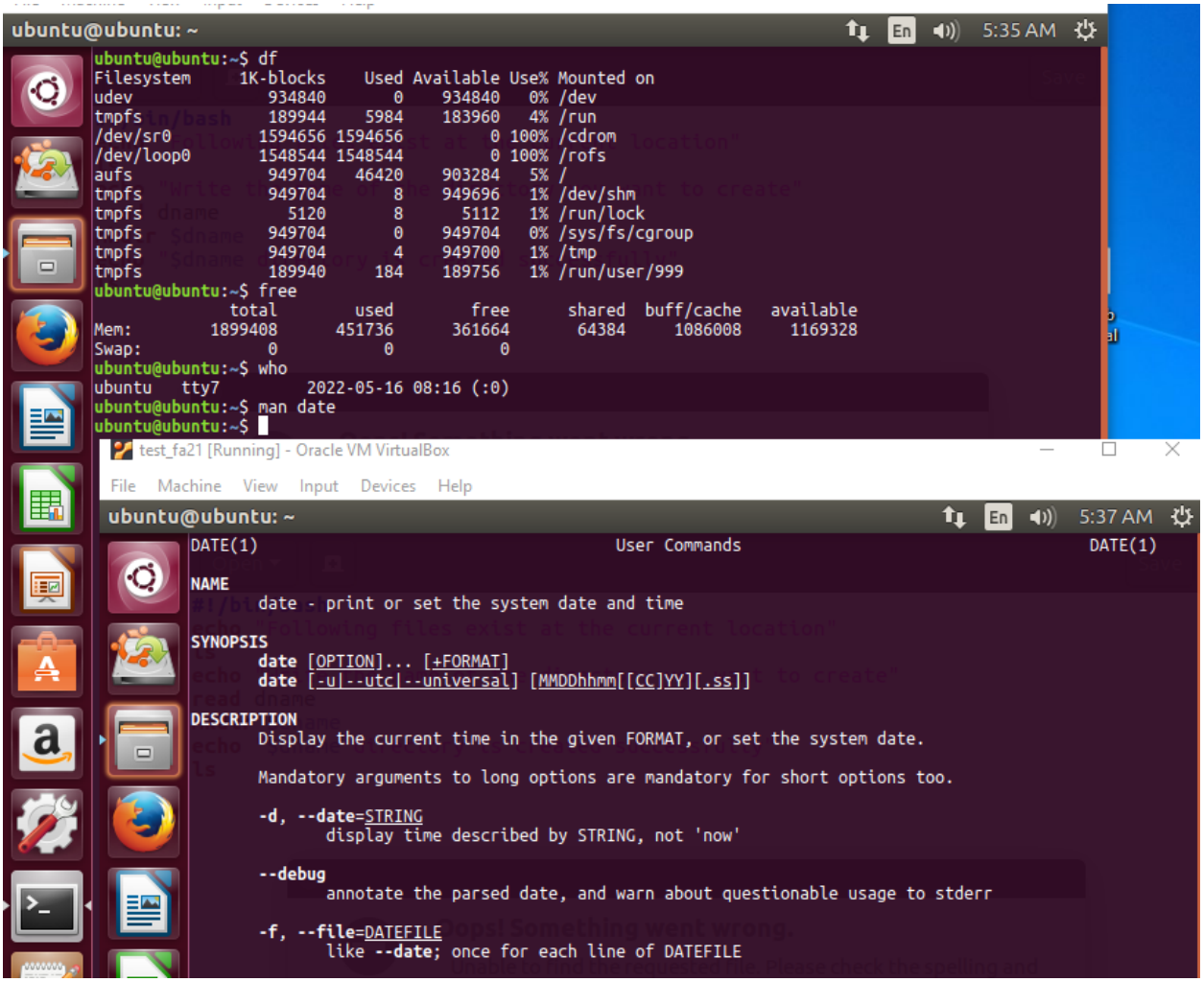


Activity 2:

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

- ***Display the amount of free storage on your machine***
- ***Display the amount of free memory on your machine***
- ***Display the user name of the current user***
- ***Open the man of date free command***

Solution:



3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 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

Lab No. 02

Working with Navigation, and File & Directory Handing Commands

Objective:

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:

On completion of this lab students will be able to:

- 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 Notes

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

1) Useful Concepts

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.

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.

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
```

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.

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)
```

Working With Directories

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

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
```

Copying Files and Directories

cp command is used to copy files and directories. The syntax to use cp command is givenbelow:

```
cp item1 item2
```

Here, item1 and item2 may be files or directories. Similarly, multiple files can also be copiedusing

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

single cp command.

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

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
```

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

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

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

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-emptydirectories 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.

Working with Files

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

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
```

```
cat  > myfile.txt
```

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

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

```
gedit myfile.txt
```

Reading the File Contents

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

```
cat  myfile.txt
```

```
less   myfile.txt
```

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

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

```
gedit   myfile.txt
```

Appending text files

cat command is also used to append a text file. Suppose we want to add some text at theend of

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

myfile.txt

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

Combining multiple text files

Using cat command, we can view the contents of multiple files. Suppose, we want to viewthe

```
cat  file1  file2   file3
```

contents of file1, file2 and file3, we can use the cat command as follows:

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
```

Determining File Type

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

```
file filename
```

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 displaysits 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 andstandard 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 whereinput comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.

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:

If we write the output of some other program to myfile.txt using > operator, its previous contents will

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

be overwritten. Now, if want to append the file instead of over-writing we can use the << operator.

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.

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) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*15*	*Medium*	*CLO-5*
2	*15*	*Medium*	*CLO-5*
3	*15*	*Medium*	*CLO-5*

Activity 1:

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

- ***Display your current directory.***
- ***Change to the /etc directory.***
- ***Go to the parent directory of the current directory.***
- ***Go to the root directory.***
- ***List the contents of the root directory.***
- ***List a long listing of the root directory.***
- ***Stay where you are, and list the contents of /etc.***
- ***Stay where you are, and list the contents of /bin and /sbin.***
- ***Stay where you are, and list the contents of ~.***
- ***List all the files (including hidden files) in your home directory.***
- ***List the files in /boot in a human readable format.***

Solution:

- pwd
- cd /etc
- cd ..
- cd /
- ls
- ls -l
- ls /etc
- ls /bin /sbin
- ls ~
- ls -al ~
- ls -lh /boot

Activity 2:

Perform the following tasks using Linux CLI

- ***Create a directory "mydir1" in Desktop Directory. Inside mydir1 create another directory "mydir2".***
- ***Change your current directory to "mydir2" using absolute path***
- ***Now, change you current directory to Documents using relative path***
- ***Create mydir3 directory in Documents directory and go into it***
- ***Now, change your current directory to mydir2 using relative path***

Solution:

- cd /home/Ubuntu/Desktop (suppose the user name is ubuntu)
  mkdir mydir1
  cd mydir1
  mkdir mydir2
- cd /home/ubuntu/Desktop/mydir1/mydir2
- cd ../../../Desktop
- mkdir mydir3
  cd mydir3
- cd ../../Desktop/mydir1/mydir2

Activity 3:

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

- ***Go to mydir3 and create an empty file myfile using cat command***
- ***Add text the text "Hello World" to myfile***
- ***Append myfile with text "Hello World again"***
- ***View the contents of myfile***

Solution:

- cd /home/Documents/mydir3 (suppose the user name is ubuntu)
  cat >myfile
- cat > myfile
  type: Hello World
  type: ctrl + d
- cat >> myfile
  type: Hello World again
  type: ctrl + d
- cat myfile

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Considering the above activities (given solved actvites section), perform the following tasks

- *move myfile to mydir1*
- *copy myfile to mydir2*
- *copy mydir2 on Desktop*
- *delete mydir1 (get confirmation before deleting)*
- *Rename myfile to mynewfile*

Task 2:

This activity is related to I/O redirection

- *Go to Desktop directory*
- *Write the long-listing of contents of Desktop on an empty file out-put-file*
- *View contents of out-put-file*

Task 3:

Considering the lab activities, perform the following tasks

- *Go to Desktop directory*
- *write the contents of mynewfile to newfile*
- *view the output of both mynewfile and newfile on screen*
- *write the combined output of mynewfile and newfile to a third file out-put-file*

Task 4:

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

Lab No. 03

Controlling Access to Files, and Managing Packages using Commands

Objective:

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

Activity Outcomes:

On completion of this lab students will be able to:

- Reading and setting file permissions.
- Setting the default file permissions.
- Performing package management tasks.

Instructor Notes

As pre-lab activity, read Chapter 09 and 14 from the book "The Linux Command Line", William E. Shotts, Jr.

1) Useful Concepts

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.

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.

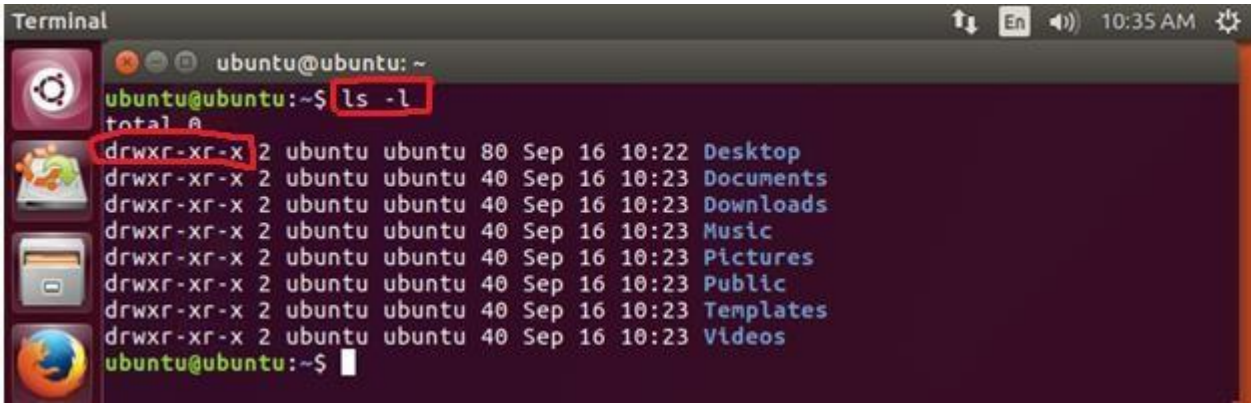
User accounts are defined in the /etc/passwd file and groups are defined in the /etc/group file. When

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.

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 isimplemented:



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

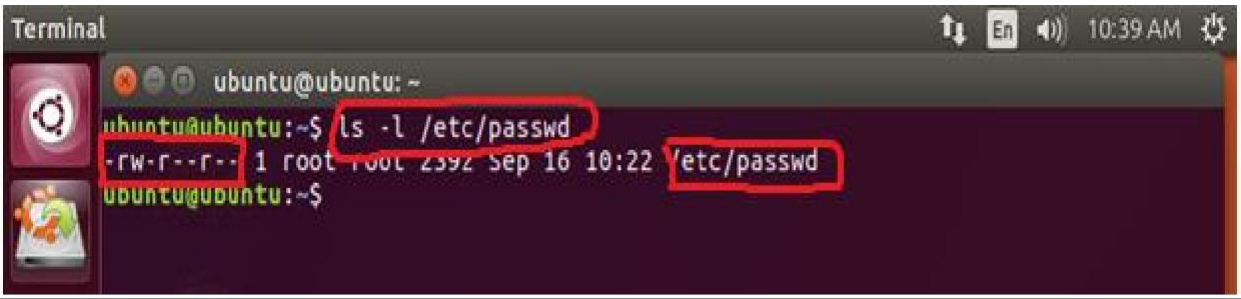
Attribute	Files	Directories
r	Allows a file to be opened and read.	Allows a directory's contents to be listed if theexecute attribute is also set.

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., cddirectory.

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

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.

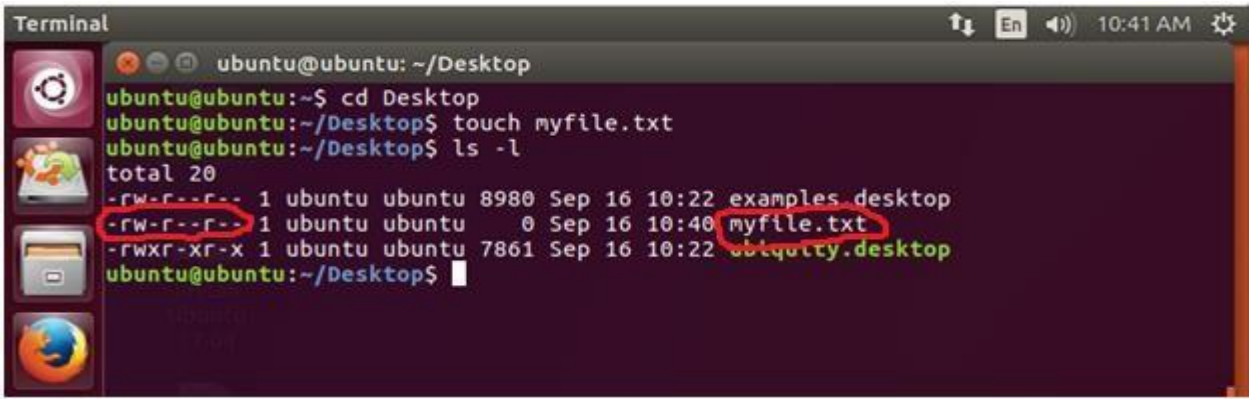


Change File Mode (Permissions)

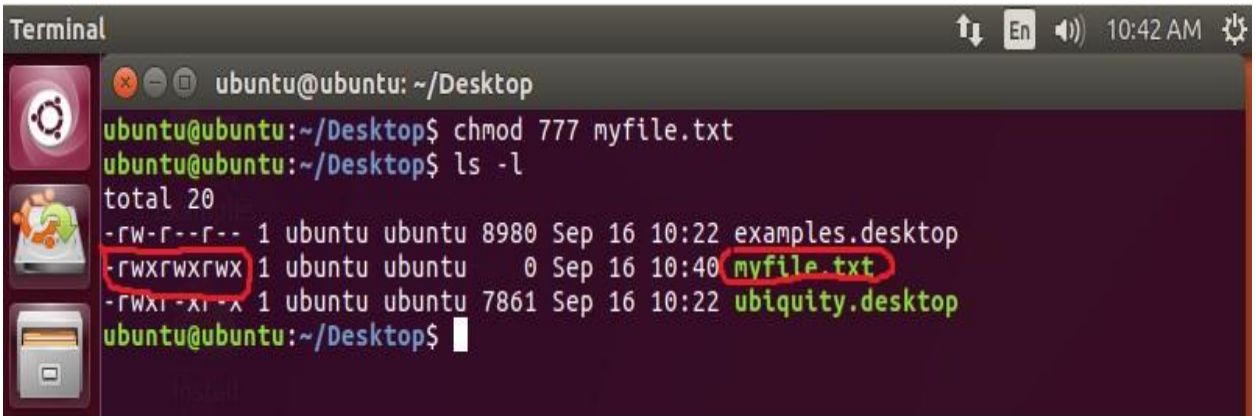
To change the mode (permissions) of a file or directory, the **chmod** command is used. Beware that only the file's owner or the super-user can change the mode of a file or directory. **chmod** supports two distinct ways of specifying mode changes: octal number representation, or symbolic representation. 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 Desktopdirectory, we create a text file **"myfile.txt"** using touch command and read its current permissions using ls command.



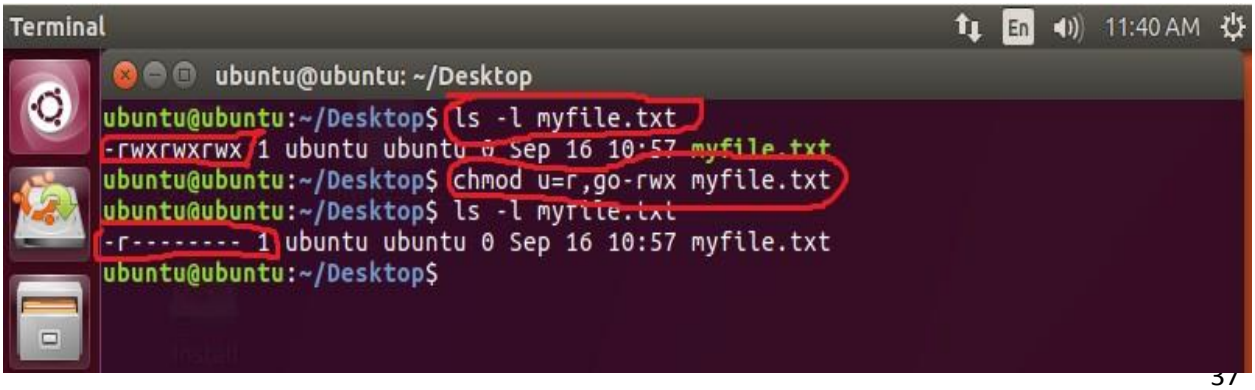
Now, we change the permission of myfile.txt and set it to 777 that is everyone can read, write and execute the file.



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



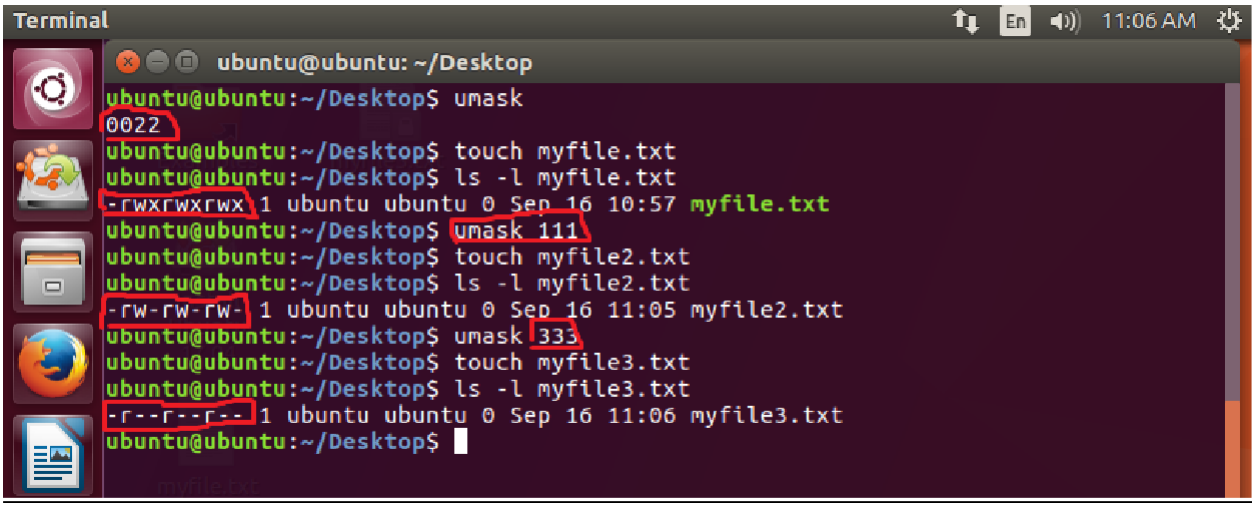
set previously, now we make it readable only to the user while the rest cannot read, write or execute the file.

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.

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.

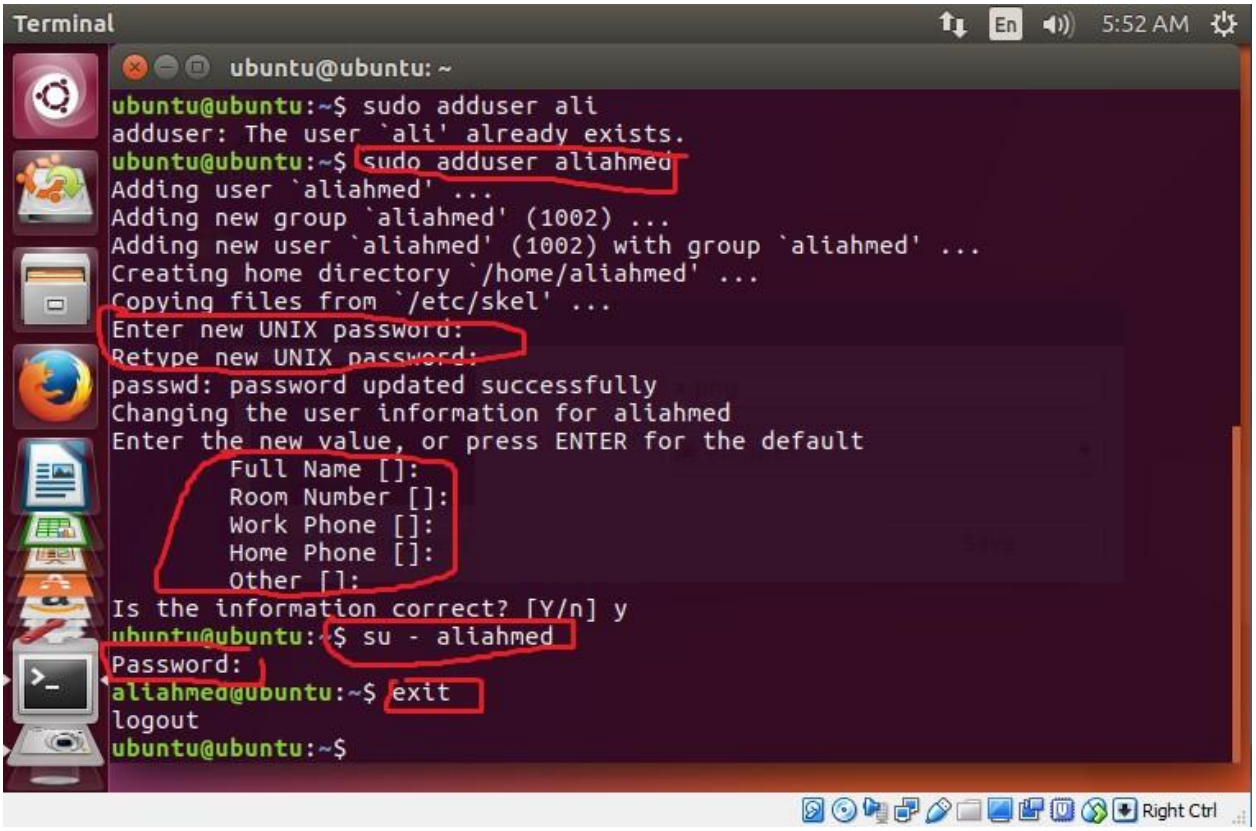
Run A Shell with Substitute User and Group IDs

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.

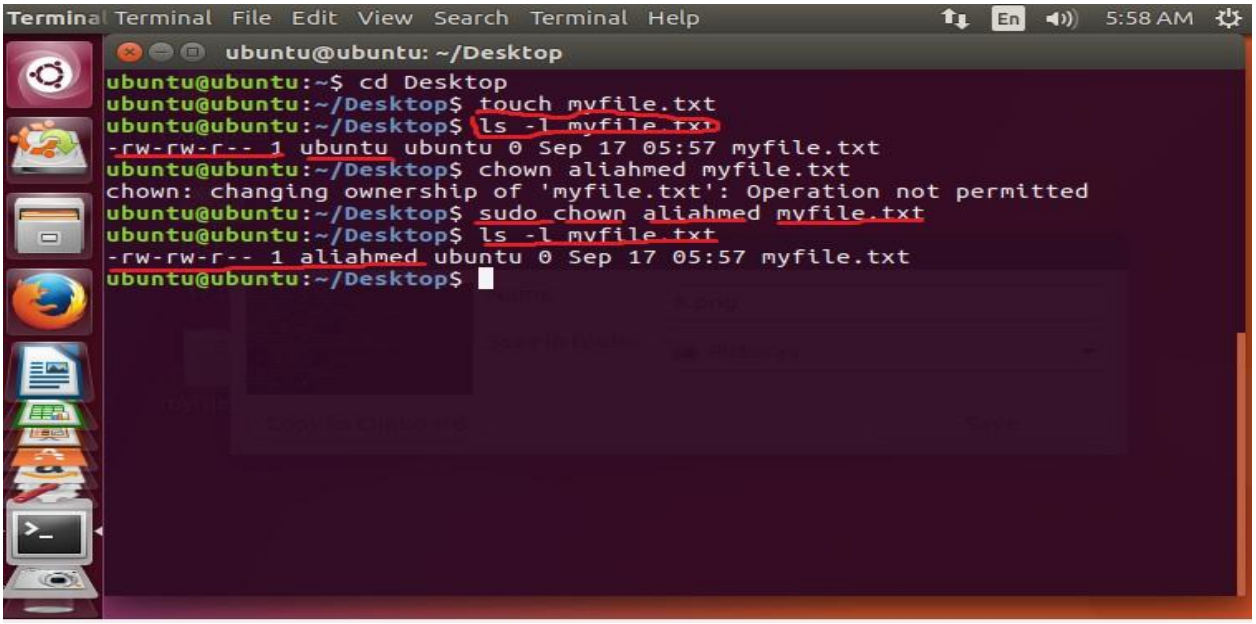


Change File Owner and Group

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.



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

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
```

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
```

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
```

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
```

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
```

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
```

Listing Installed Packages: These commands can be used to display a list of all the packages installed on the system:
`dpkg --list`

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`

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) Solved Lab Activities

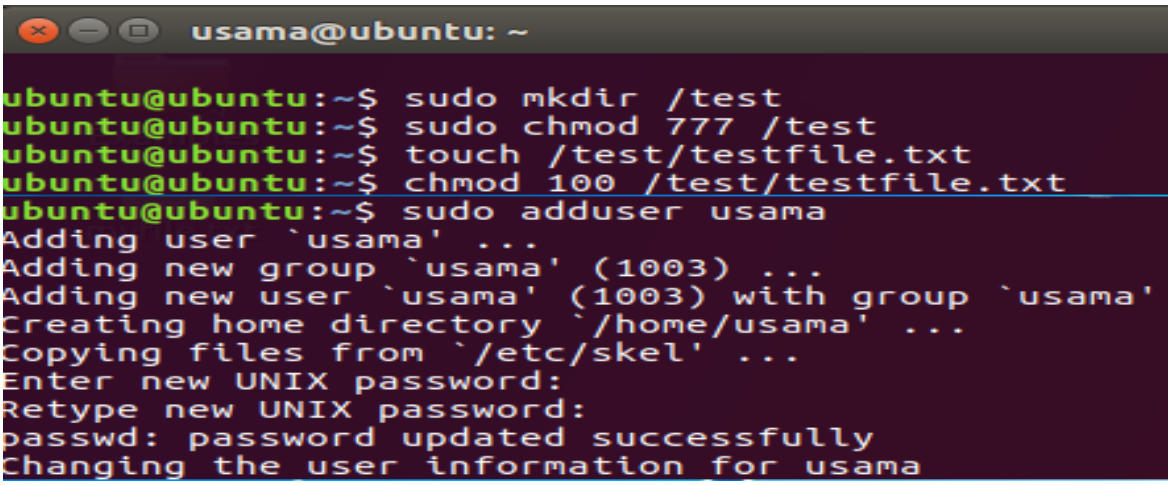
Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*15*	*Medium*	*CLO-5*
2	*15*	*Medium*	*CLO-5*
3	*15*	*Medium*	*CLO-5*

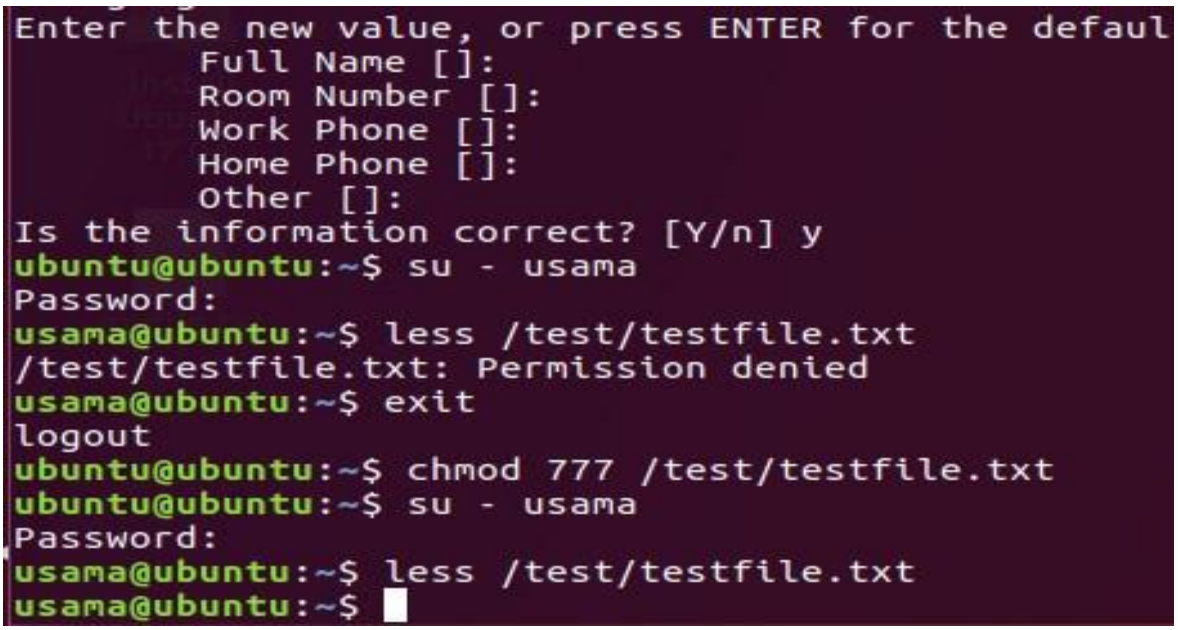
Activity 1:

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

- ***Create a new directory named test in root directory as superuser***
- ***Make this directory public for all***
- ***Create a file "testfile.txt" in /test directory***
- ***Change its permissions that no boy can write the file, but the owner can read it.***
- ***Create another user "Usama"***
- ***Run the shell with user Usama***
- ***Try to read the "testfile.txt"***
- ***Logout as Usama***
- ***Change the permission of testfile.txt so that everyone can read, write and execute it***
- ***Run shell as Usama again***
- ***Now, read the file***

Solution:



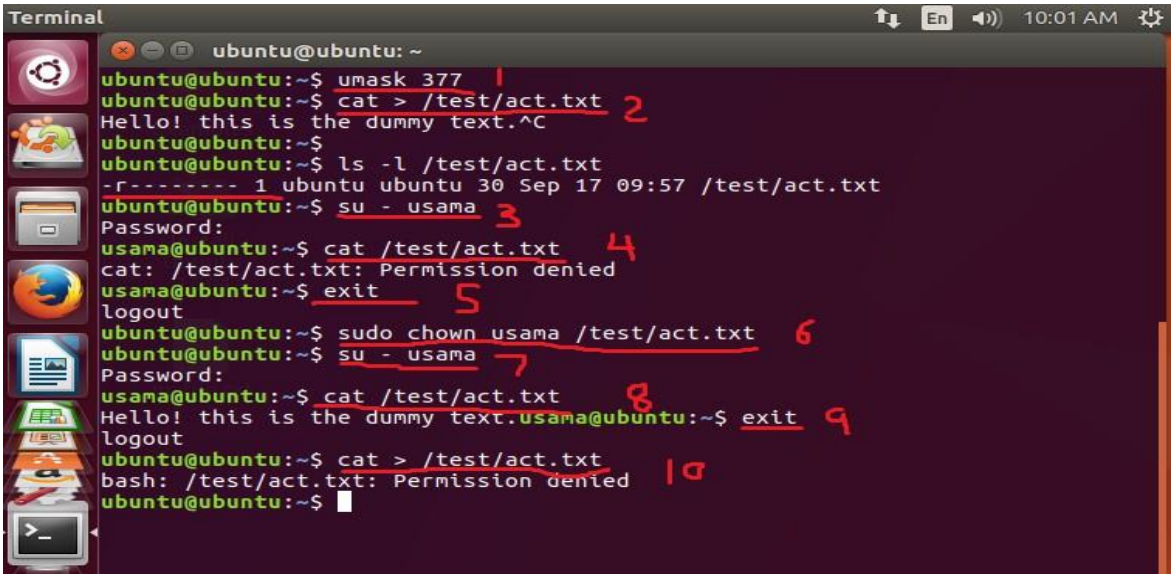


Activity 2:

Perform the following tasks

- ***Set the permission such that a newly created file is readable only to the owner***
- ***Create a text file "act.txt" in /test directory crated in previous activity***
- ***Run the shell as user "usama" (created previously)***
- ***Access the file "act.txt"***
- ***Logout as usama***
- ***Now change the ownership of the file from ubuntu to usama***
- ***Run the shell again as usama***
- ***Read the file "act.txt" using cat command***
- ***Logout as usama***
- ***Now access the act.txt with user ubuntu***

Solution:

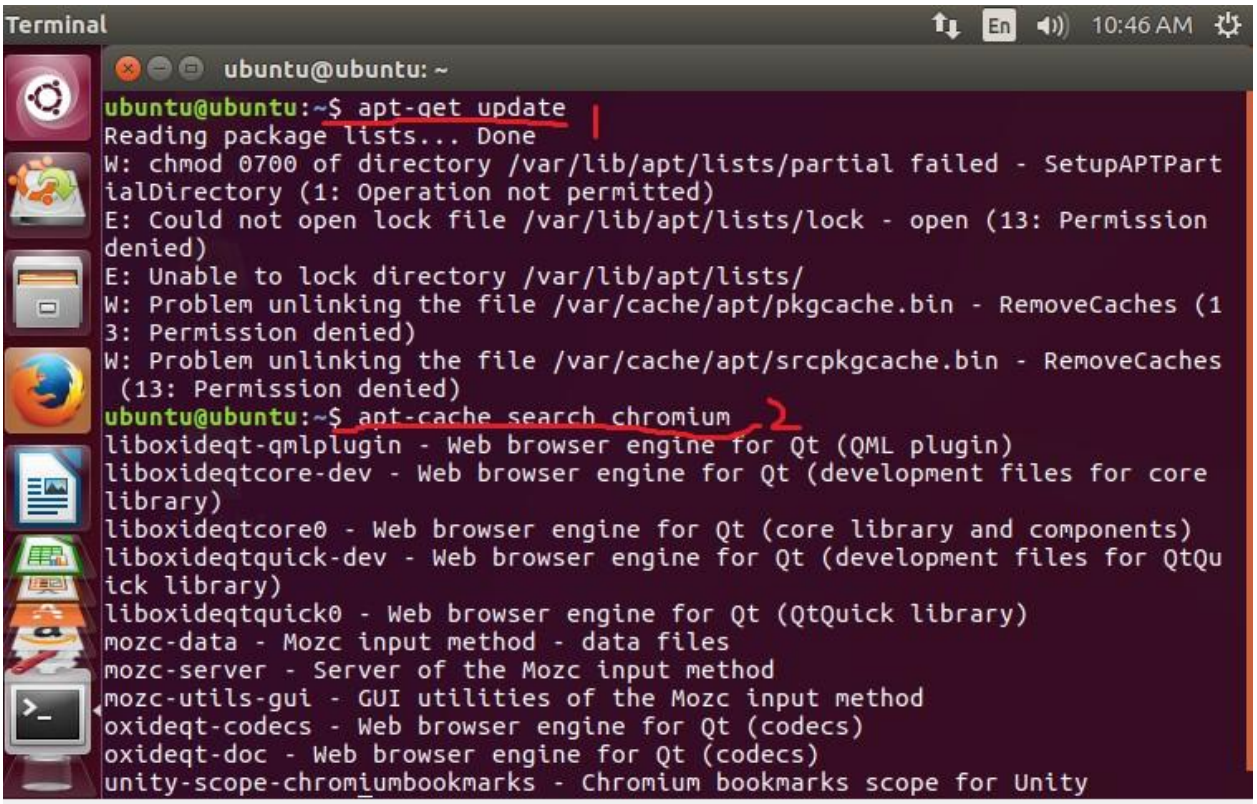


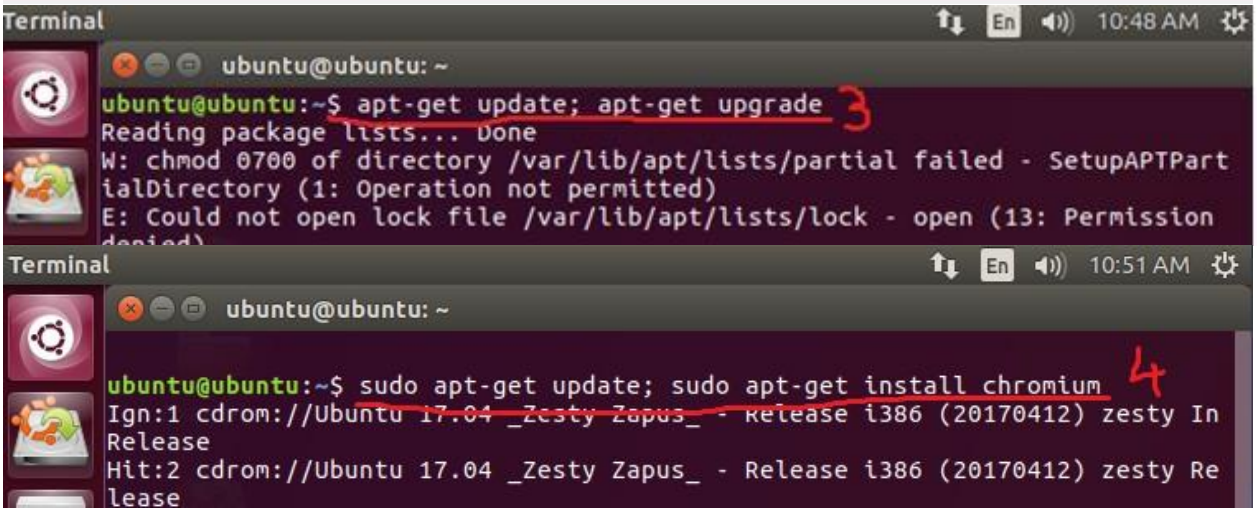
Activity 3:

Perform the following tasks

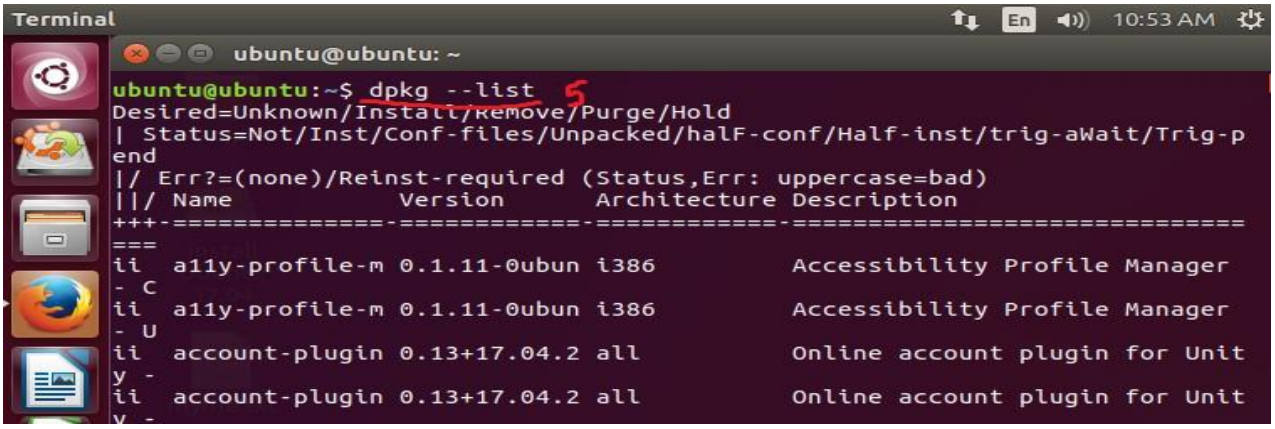
- ***search apt repository for the chromium browser***
- ***install the chromium browser using command line***
- ***write the command to update and upgrade the repository***
- ***list the software installed on your machine and write output on a file list.txt***
- ***read the list.txt file using cat command***

Solution:


Terminal
10:46 AM
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ apt-get update
Reading package lists... Done
W: chmod 0700 of directory /var/lib/apt/lists/partial failed - SetupAPTPart
ialDirectory (1: Operation not permitted)
E: Could not open lock file /var/lib/apt/lists/lock - open (13: Permission
denied)
E: Unable to lock directory /var/lib/apt/lists/
W: Problem unlinking the file /var/cache/apt/pkgcache.bin - RemoveCaches (1
3: Permission denied)
W: Problem unlinking the file /var/cache/apt/srcpkgcache.bin - RemoveCaches
(13: Permission denied)
ubuntu@ubuntu:~$ apt-cache search chromium
liboxideqt-qmlplugin - Web browser engine for Qt (QML plugin)
liboxideqtcore-dev - Web browser engine for Qt (development files for core
library)
liboxideqtcore0 - Web browser engine for Qt (core library and components)
liboxideqtquick-dev - Web browser engine for Qt (development files for QtQu
ick library)
liboxideqtquick0 - Web browser engine for Qt (QtQuick library)
mozc-data - Mozc input method - data files
mozc-server - Server of the Mozc input method
mozc-utils-gui - GUI utilities of the Mozc input method
oxideqt-codecs - Web browser engine for Qt (codecs)
oxideqt-doc - Web browser engine for Qt (codecs)
unity-scope-chromiumbookmarks - Chromium bookmarks scope for Unity


Terminal
10:48 AM
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ apt-get update; apt-get upgrade
Reading package lists... Done
W: chmod 0700 of directory /var/lib/apt/lists/partial failed - SetupAPTPart
ialDirectory (1: Operation not permitted)
E: Could not open lock file /var/lib/apt/lists/lock - open (13: Permission

Terminal
10:51 AM
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ sudo apt-get update; sudo apt-get install chromium
Ign:1 cdrom://Ubuntu 17.04 _Zesty Zapus_ - Release i386 (20170412) zesty In
Release
Hit:2 cdrom://Ubuntu 17.04 _Zesty Zapus_ - Release i386 (20170412) zesty Re
lease


Terminal
10:53 AM
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ dpkg --list
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-aWait/Trig-p
end
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name Version Architecture Description
+++-==========-==========-==========-==========
ii a11y-profile-m 0.1.11-0ubun i386 Accessibility Profile Manager
- C
ii a11y-profile-m 0.1.11-0ubun i386 Accessibility Profile Manager
- U
ii account-plugin 0.13+17.04.2 all Online account plugin for Unit
y -
ii account-plugin 0.13+17.04.2 all Online account plugin for Unit

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

You are 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: create 3 files with touch command: files1, files2, files3.
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
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)
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?

Lab No. 04

Text Processing and Pipelining

Objective:

This lab will introduce the Text Processing Tools available in bash shell along with some basic system configuration commands.

Activity Outcomes:

On completion of this lab students will be able to:

- Use some of the most useful text-processing utilities
- Write complex commands using piplines

Instructor Notes

As pre-lab activity, read Chapter 16 to 20 from the book "The Linux Command Line", William E. Shotts, Jr.

1) Useful Concepts

Text Processing Utilities

In first part of this lab, we will discuss some basic and useful utilities available in bash shell

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 storedin 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, enterthe 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 bytescontained 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 referredto 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 onsimple 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 fewlines 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
```

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 ingenerally 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 onscreen:

```
# 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: 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 rootuser:

```
# 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"
```

Synchronizing Hardware and System Clocks: Use the following syntax:

```
# hwclock --systohc OR
# hwclock -w
```

timedatectl: Display the current date and timeType the following command:

```
$ timedatectl
```

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: noRTC 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) atSun 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,
```

HH : An hour.

MM : A minute.

SS : A second, all typed in two-digit form.

YYYY: A four-digit year.

MM : A two-digit month.

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
```

Set the current time: 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: noRTC in local TZ: no
DST active: n/aLab Activities

2) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	***15***	***Low***	***CLO-5***
2	***15***	***Medium***	***CLO-5***
3	***5***	***Low***	***CLO-5***

Activity 1:

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

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

Solution:

- `touch /test/testfile.txt`
- `less testfile.txt`
- `wc testfile.txt`
- `grep testfile.txt`
- `head -n 3 testfile.txt`
- `tail -n 3 testfile.txt`

Activity 2:

Perform the following tasks

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

Solution:

- `ls /test | grep zip`
- `ls /test | head -n 1`
- `ls /test | tail -n 1`
- `ls /test /Desktop | sort | less`

Activity 3:

Perform the following tasks

- ***Examine the network using Ping command and view the performance statistics***

Solution

- `ping localhost`

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

1. *Use sort and unique command to sort a file and print unique values.*
2. *Use head and tail to print lines in a particular range in a file.*
3. *Use ls and find to list and print all lines matching a particular pattern in matching files.*
4. *Use cat, grep, tee and wc command to read the particular entry from user and store in a file and print line count.*
5. *Pipes the output from the cat (concatenate) process into the sort process to produce sorted output, and then pipes the sorted output into the uniq process to eliminate duplicate records.*

Lab No. 05

Managing Processes, and Writing, Compliling & Executing C++ on Linux

Objective:

This lab is designed to demonstarate the basic task management related activities along with an introduction to Writing C++ programs in Linux.

Activity Outcomes:

On completion of this lab students will be able to:

- How to manage processes in Linux
- Perform Compiling and Executing C++ programs using command-line in Linux

Instructor Notes

As pre-lab activity, read Chapter 10 from the book "The Linux Command Line", William E. Shotts, Jr.

1) Useful Concepts

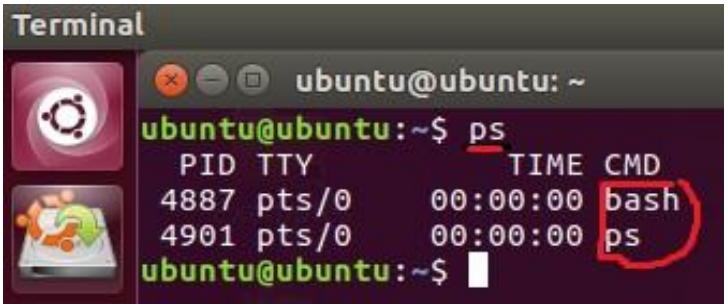
Process management in Linux

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 servicesare 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.

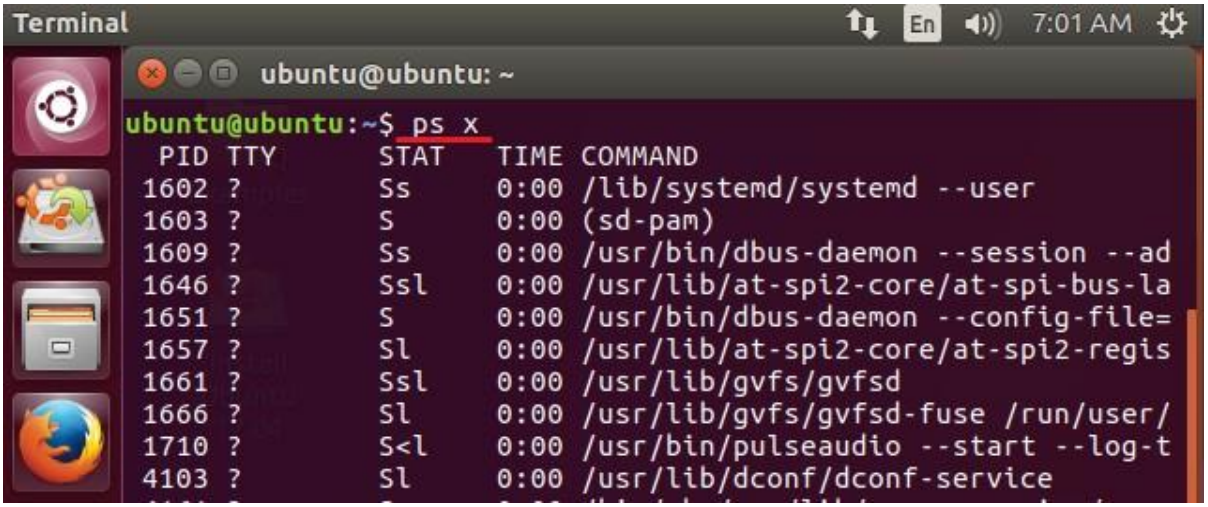
The kernel maintains information about each process to help keep things organized. For example, each process is assigned a number called a process ID or PID. PIDs are assigned in ascending order, with init always getting PID 1. The kernel also keeps track of the memory assigned to each process, as well 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.

Displaying Processes in the System

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.



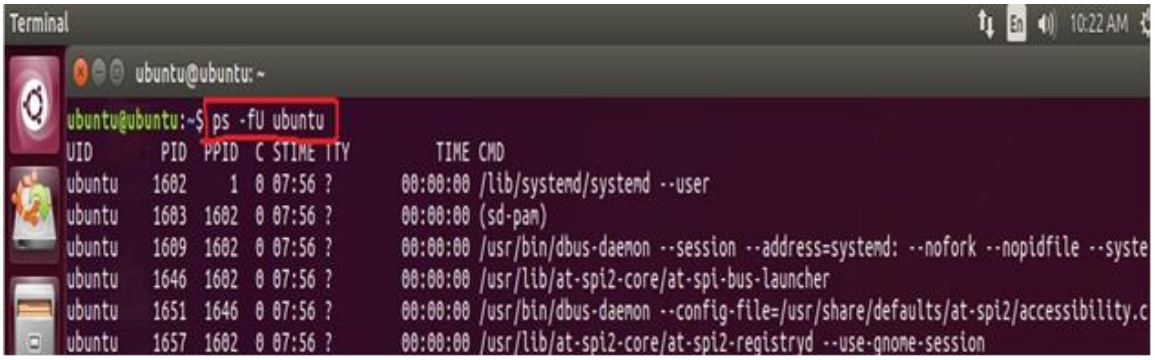
Here, a new column is also added in the output that is STAT. This column shows the currentstate of the process. The most common states are given below.

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

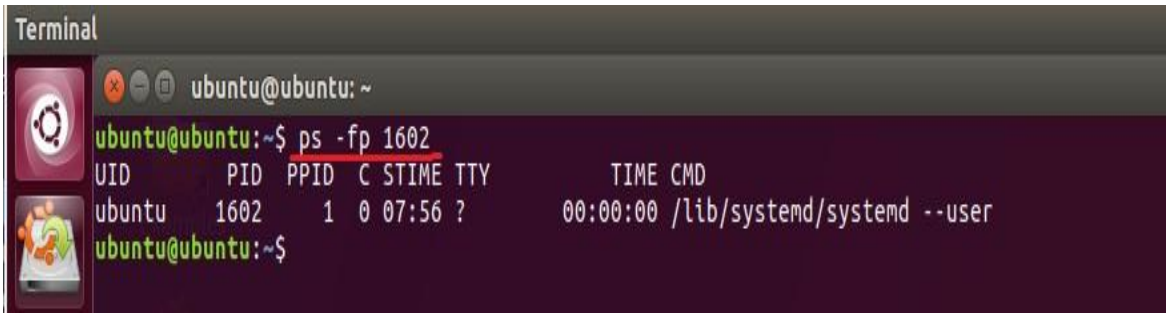
Following are the most common option that can be used with ps command.

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

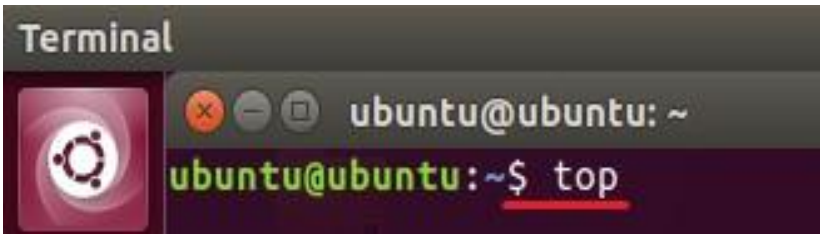
In the following example, we display processes that are related to the user ubuntu.



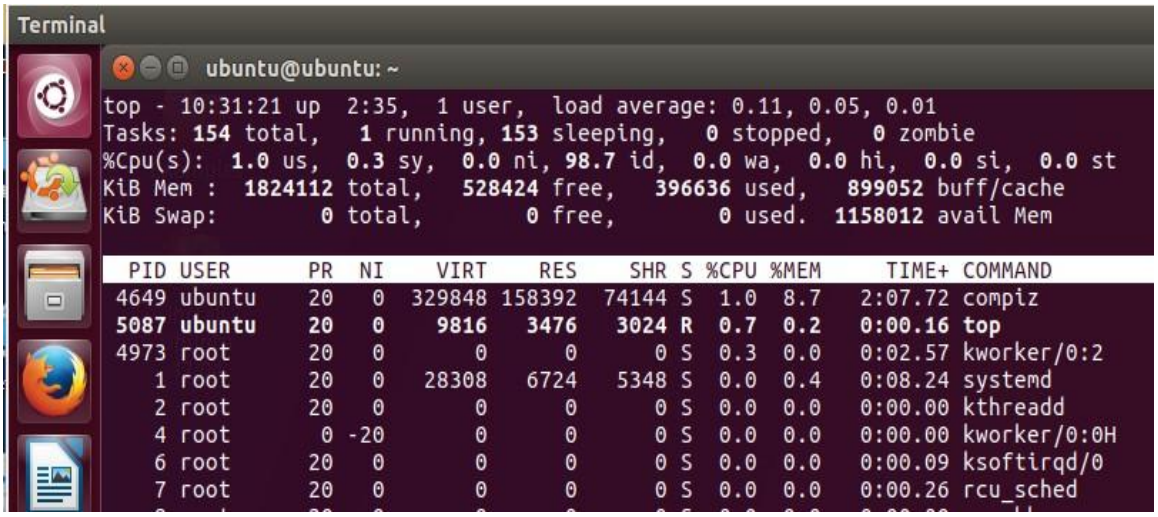
Now, we select a process with id 1602



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.



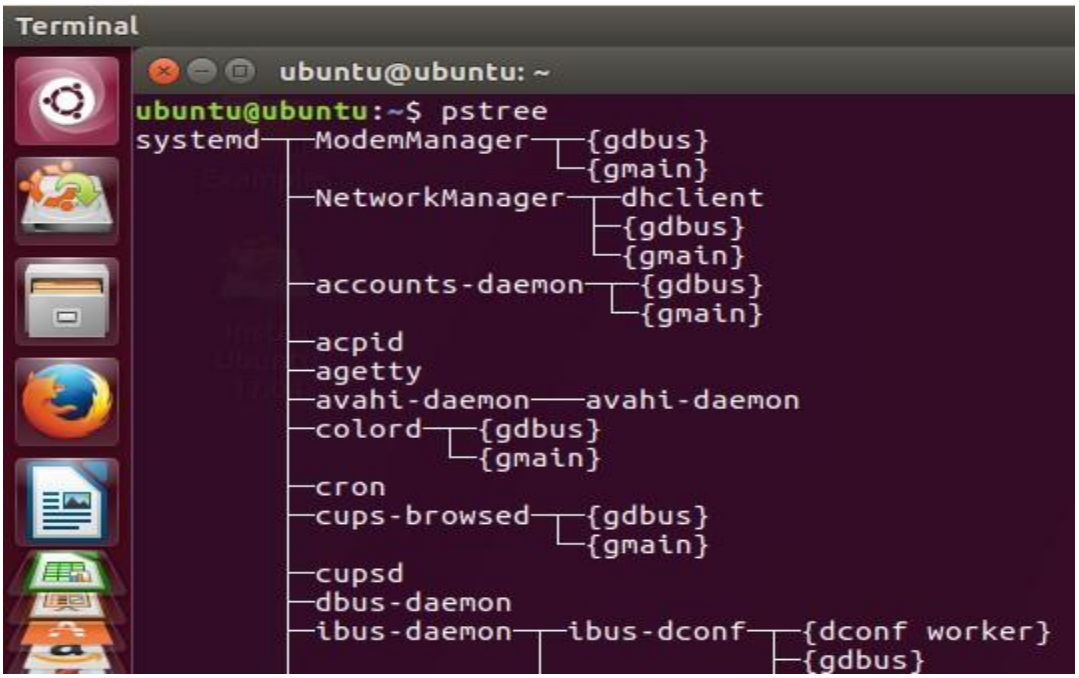
The output of the above command is given below:



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

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

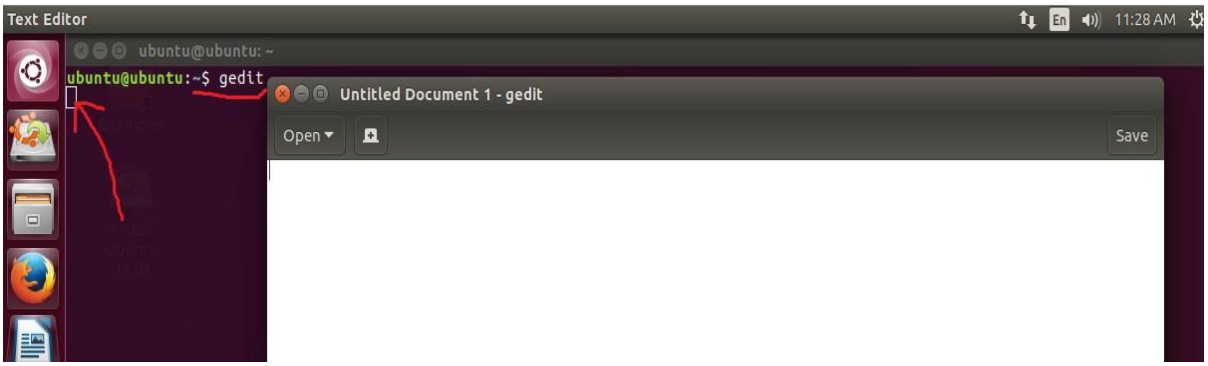


Interrupting A Process

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

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.

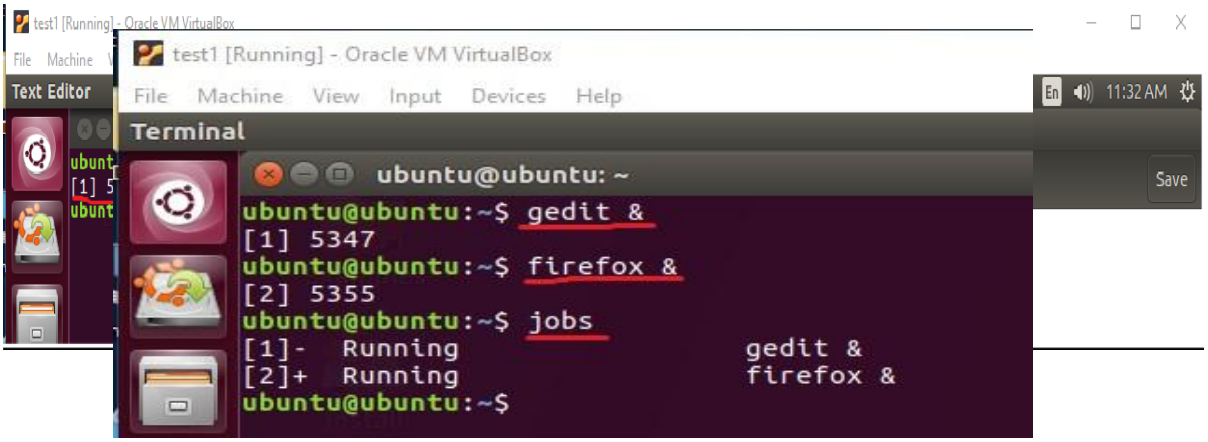


It can be noted that gedit is opened as foreground process and control does not returns to terminal unless it is closed. Now, we start the gedit again as a background process.

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

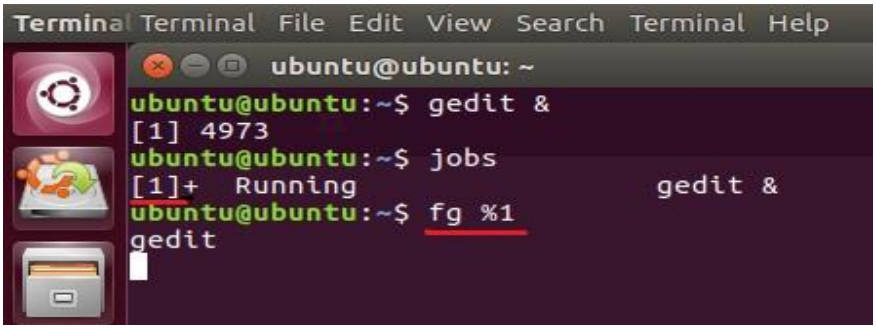
jobs command

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 jobscommand to view the running jobs.



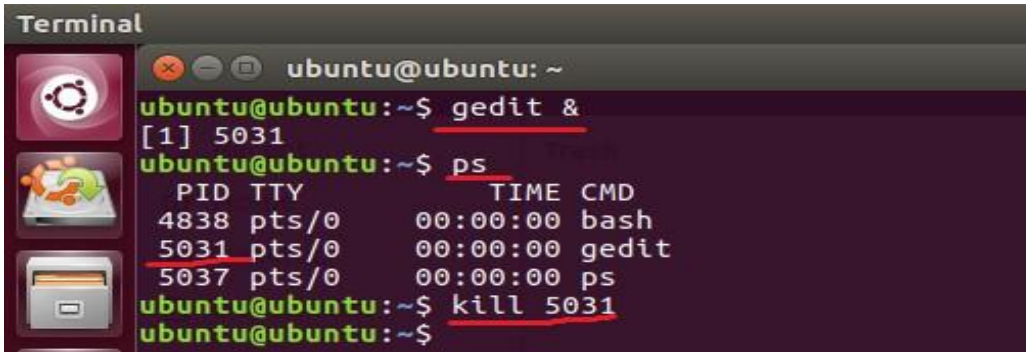
Bringing a process to the foreground

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.



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.

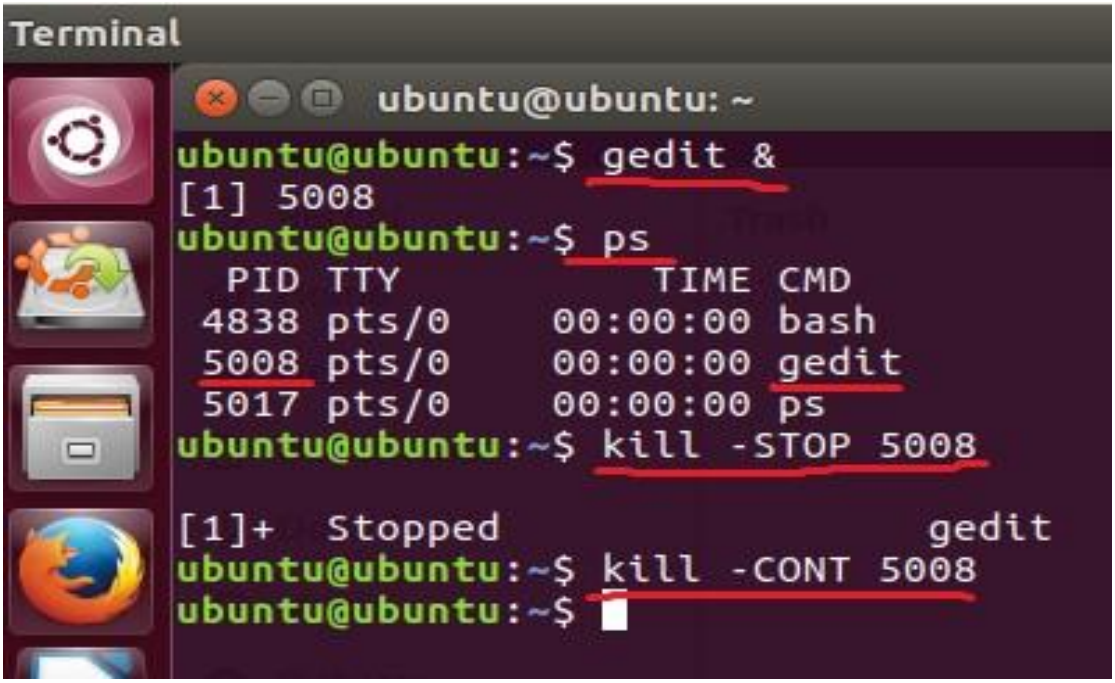
Sgnal	Meaning
INT	INT Interrupt. Performs the same function as the Ctrl-c key sent from the terminal. It will usually terminate a program.
TERM	Terminate. This is the default signal sent by the kill command. If a program is still "alive" enough to receive signals, it will terminate.
STOP	Stop. This signal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

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.

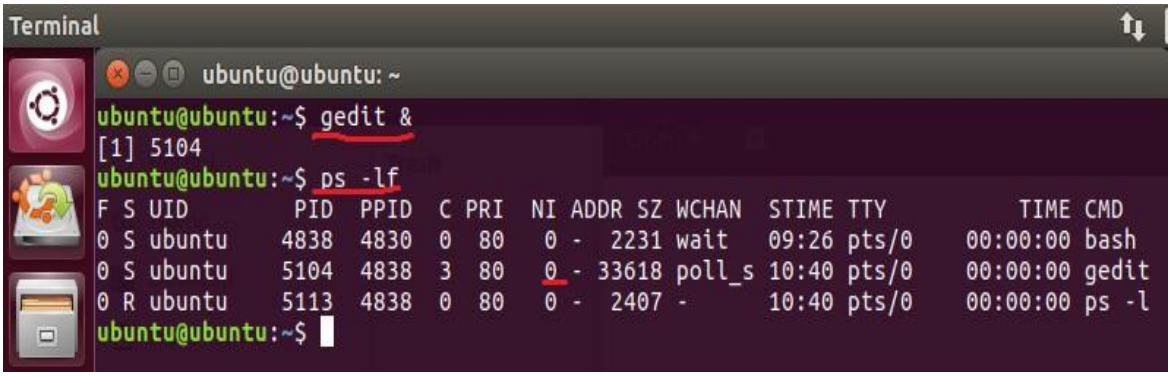
We can paus a process by using kill command with STOP option. The process id is required as an argument in kill command. In the following example, we start the gedit program in the background. Then we find the pid of gedit using ps command and then; we pause the process using kill command. Later, we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



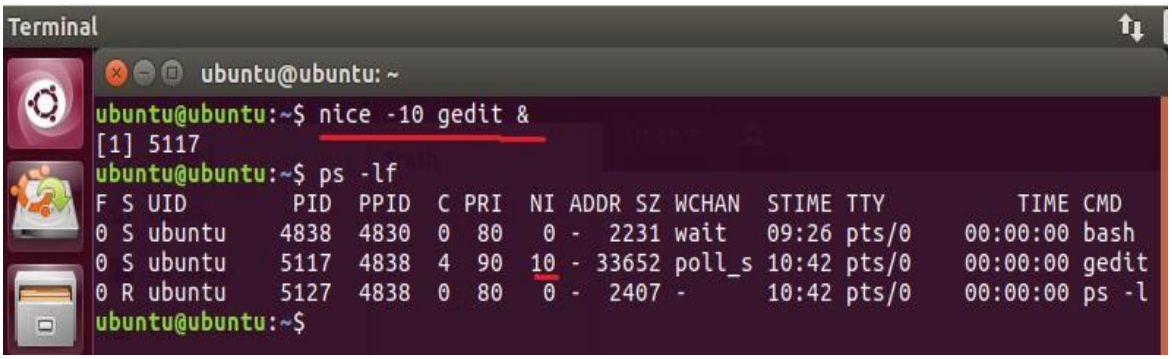
Changing process priority

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.

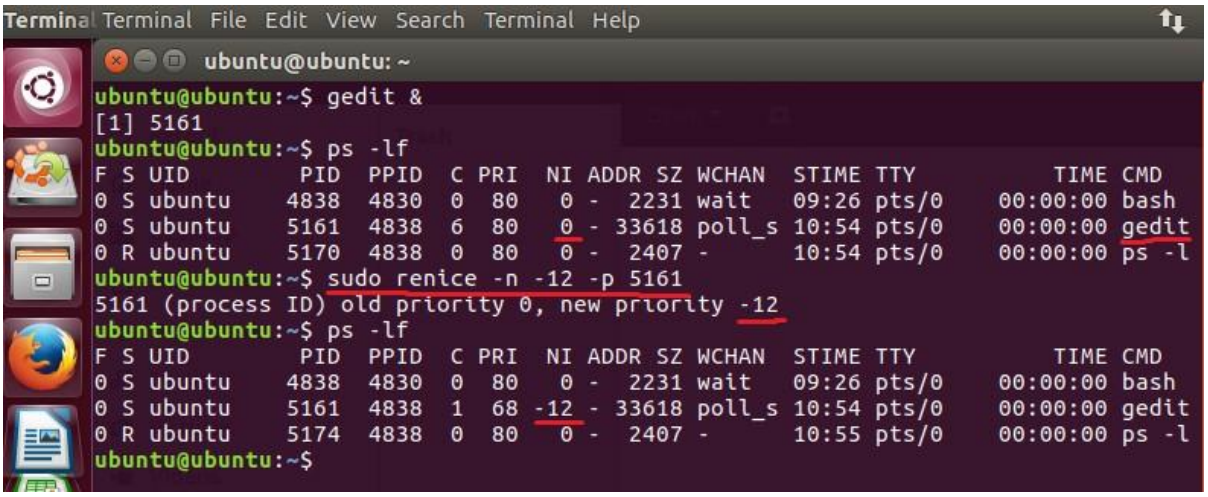
In the following example, we start the gedit program in background and see its nice value using ps command.



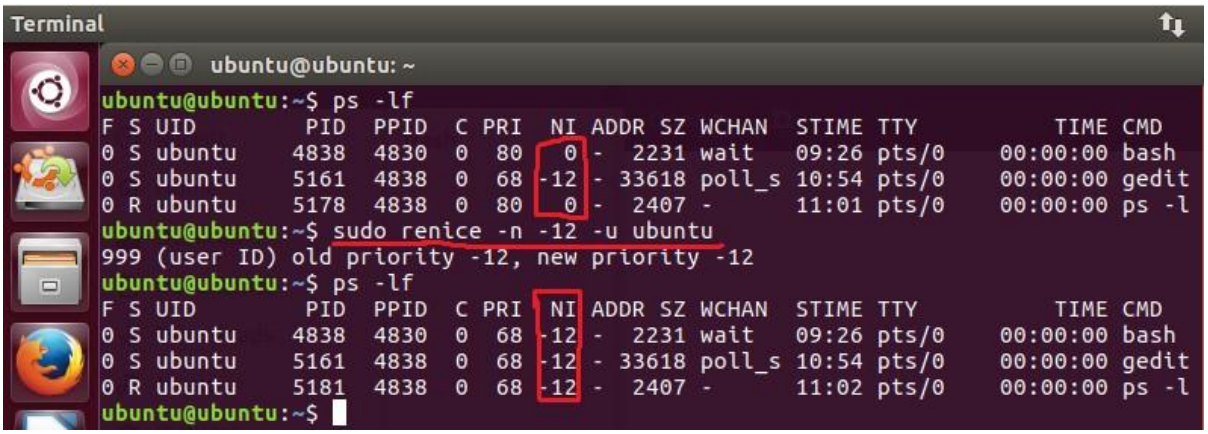
Now, we start the gedit program again using nice command.



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.



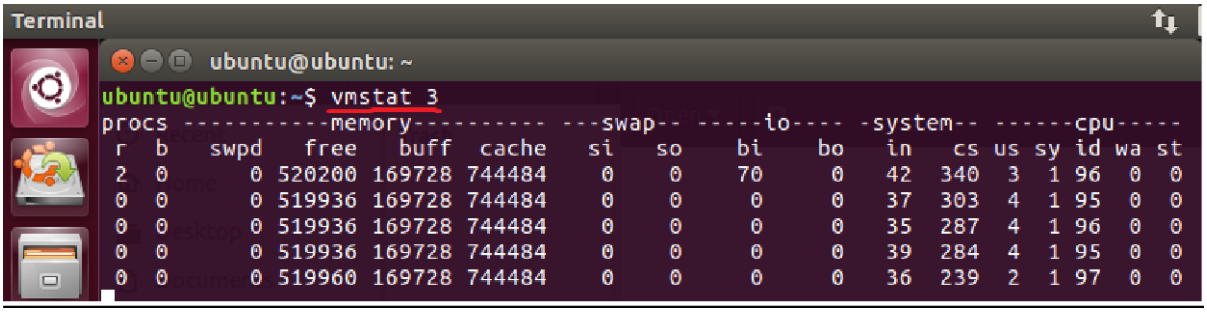
We can also use the renice command to change the priority of all of the processes belonging to a user or a group. In the following example, we change the nice value of all of the process belonging to user ubuntu.



Few more useful commands related to processes

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.



xload command

Displays the system load over time.

tload command

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

Compiling and Executing C++ Programs

Compiling is the way toward making an interpretation of source code into the local languageof 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 extremelylittle 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.

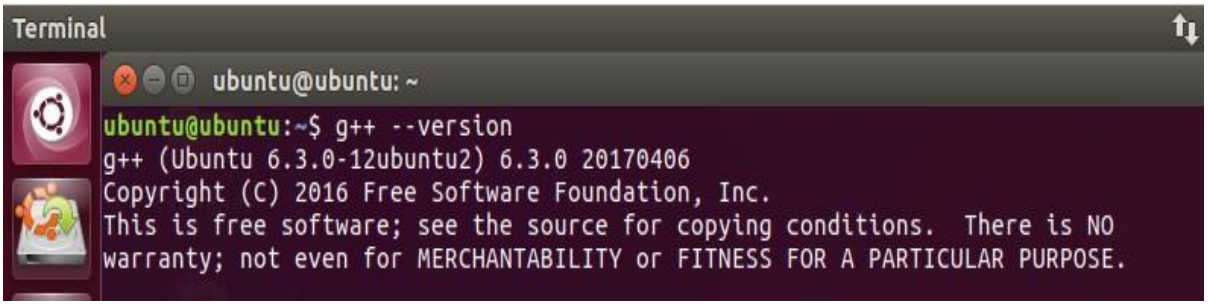
We next come to what are called high-level languages. They are called this since they enable the developer to be less worried about the details of what the processor is doing and more with taking care of the current issue. Programs written in high-level programming languages are converted into machine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

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 lotsof platform like Windows, Linux, Unix, Mac etc. Before we start programming with C++. We will needan 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.

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++
```

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.

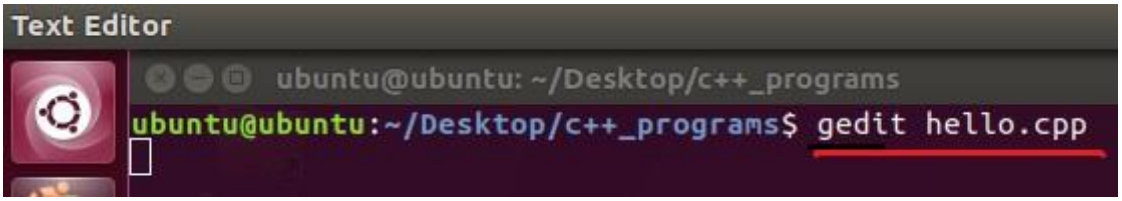
```
$ sudo g++ source-file.cpp -o executable-file
```

Running a C++ program

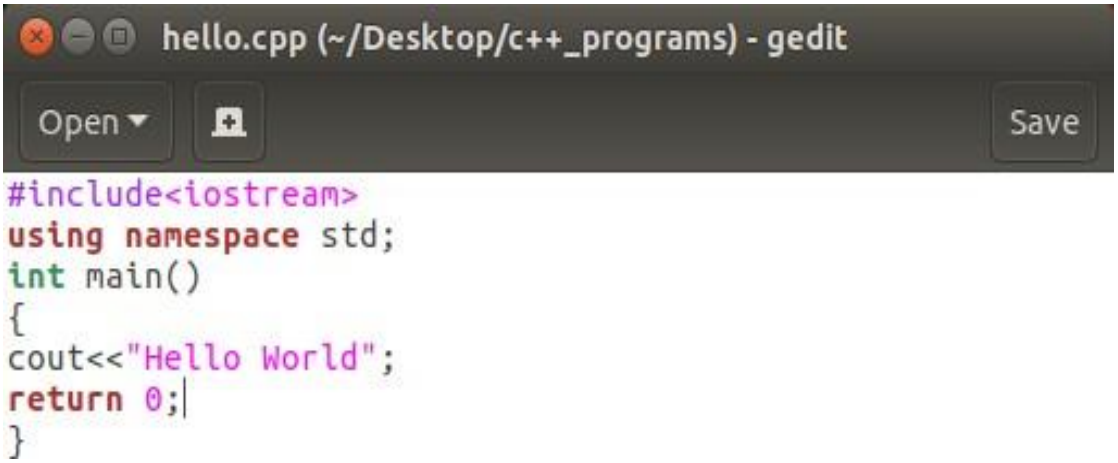
Onc a C++ program is successfully compiled, we can execute the created executable file byusing the following command:

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

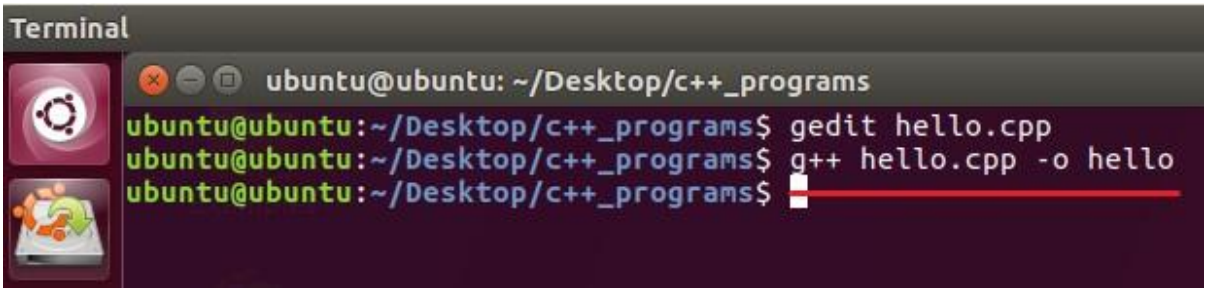
In the following example, we write a simple C++ program that displays a Hello Worldmessage and the then compile and execute this program using the above commands. To write the program, we use the gedit text editor. The source code file is saved with .cpp extension.
Open the gedit editor and pass the name of the file (hello.cpp) to be created



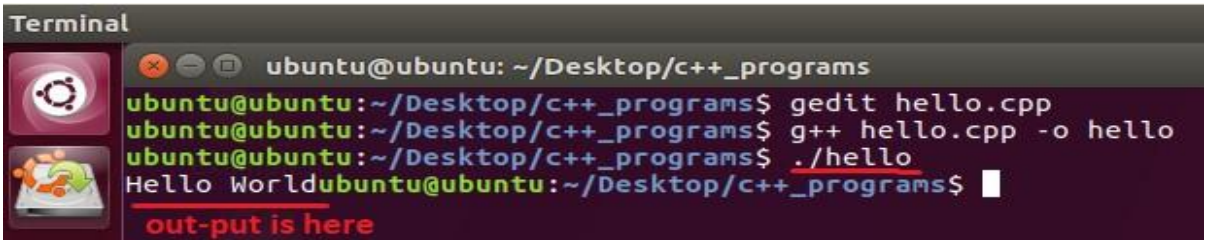
Write the following code



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



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



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.

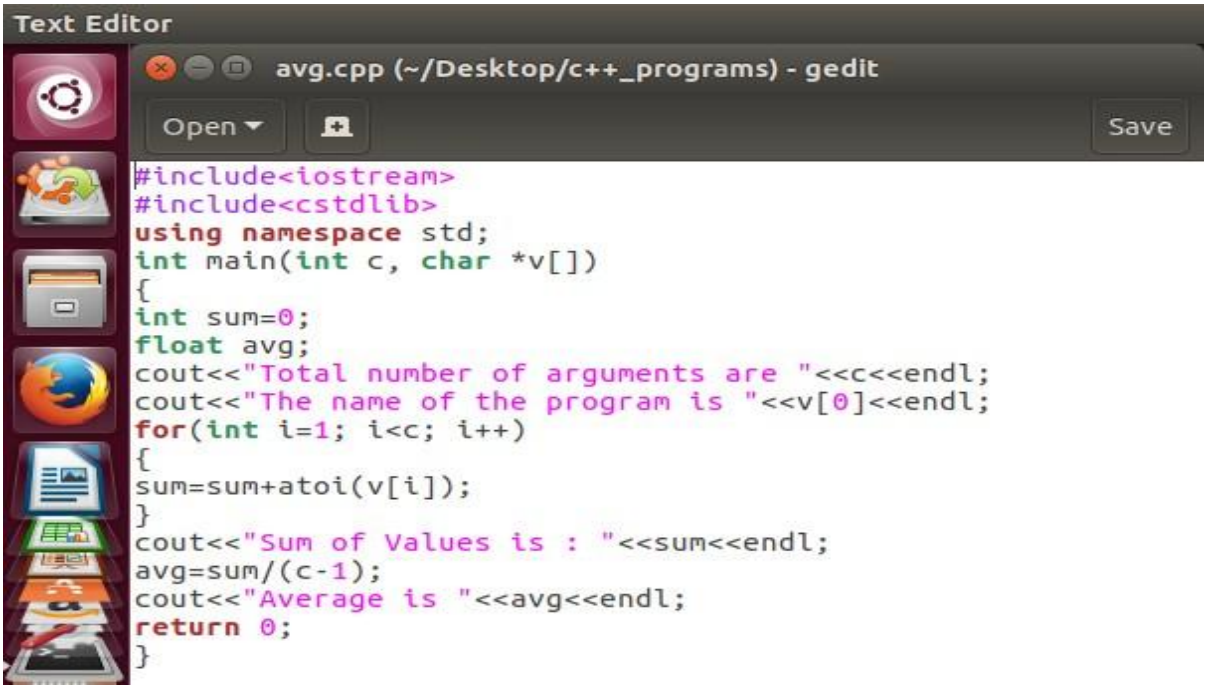
To pass command line arguments, we typically define main () with two arguments: first argument counts the number of arguments on the command line and the second is a pointer array which holds pointers of type char which points to the arguments passed to theprogram. The syntax to define the main method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

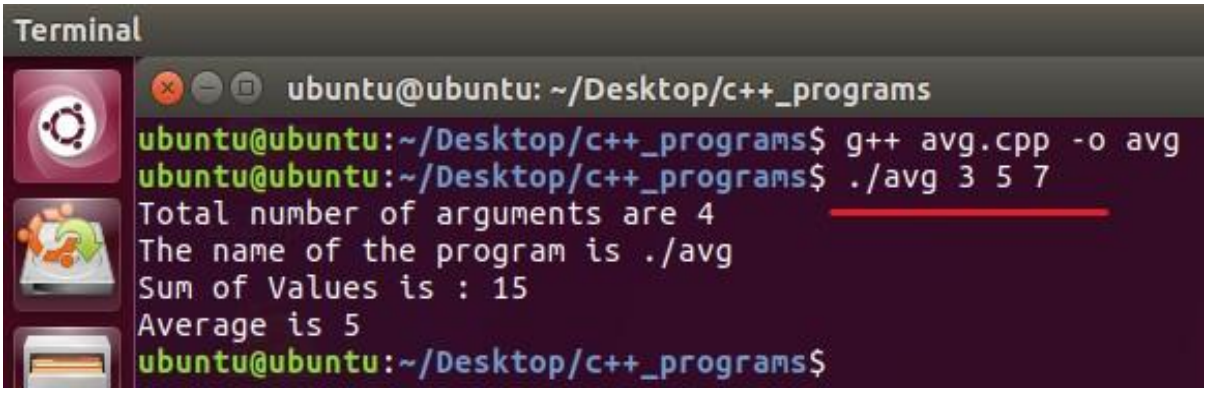
```
int main (int argc, char *argv[])
```

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) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*15*	*Low*	*CLO-5*
2	*15*	*Medium*	*CLO-5*
3	*15*	*Medium*	*CLO-7*

Activity 1:

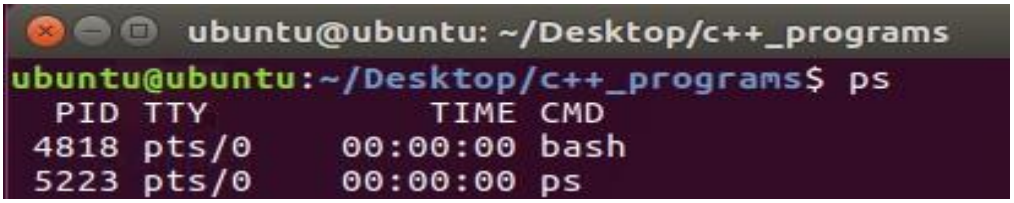
This activity is related to process monitoring in Linux

- ***Display all of the process in current shell***
- ***Display every active process in the system***
- ***Provide a full-format listing of process owned by you***
- ***Display a full-format listing of processes owned by user ubuntu***
- ***Display a process with id 1***
- ***Display the dynamic view of the current processes in the system and set the refresh interval 0.5 second***

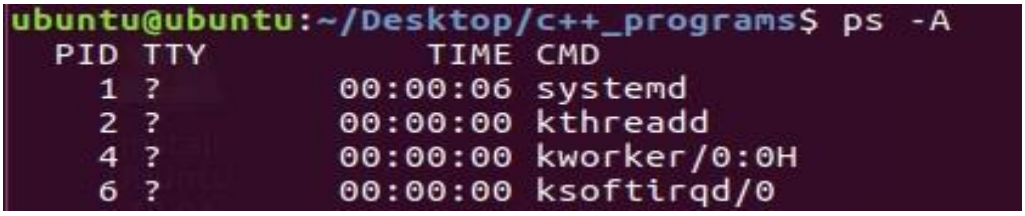
- ***Display the dynamic view of the processes owned by user with id 999 (or name ubuntu)***
- ***Start the gedit program in background and then bring it foreground***
- ***Suspend the gedit program***
- ***Resume the gedit program***

Solution:

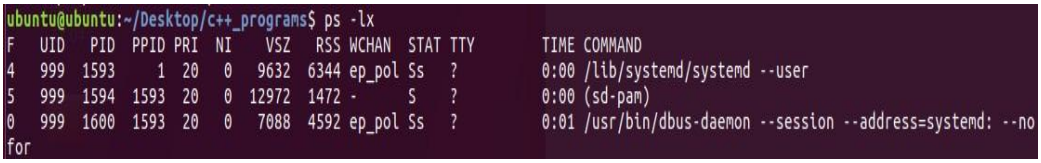
1.



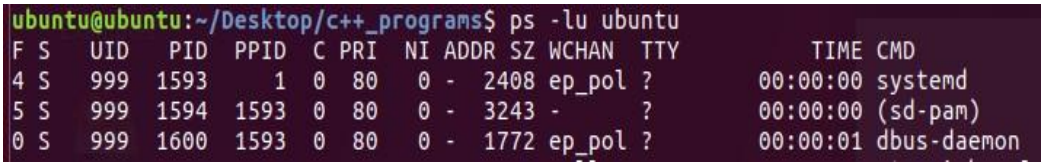
2.



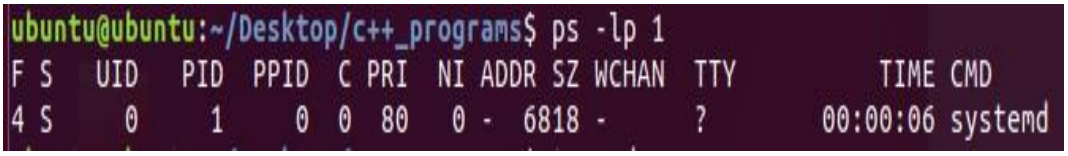
3.



4.



5.



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

- ***Start the gedit program with priority 90***
- ***Reset the priority of gedit to 65***

Solution:

1.

```
ubuntu@ubuntu: ~/Desktop/c++_programs
ubuntu@ubuntu:~/Desktop/c++_programs$ nice -n 10 gedit &
[1] 5310
ubuntu@ubuntu:~/Desktop/c++_programs$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   999  4818  4810  0  80   0 -  2231 wait   pts/0    00:00:00 bash
0 S   999  5310  4818  2  90  10 - 33685 poll_s pts/0    00:00:00 gedit
0 R   999  5320  4818  0  80   0 -  2338 -      pts/0    00:00:00 ps
```

2.

```
ubuntu@ubuntu:~/Desktop/c++_programs$ sudo renice -n -15 -p 5310
5310 (process ID) old priority 10, new priority -15
ubuntu@ubuntu:~/Desktop/c++_programs$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   999  4818  4810  0  80   0 -  2231 wait   pts/0    00:00:00 bash
0 S   999  5310  4818  0  65 -15 - 33685 poll_s pts/0    00:00:00 gedit
0 R   999  5330  4818  0  80   0 -  2338 -      pts/0    00:00:00 ps
```

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;
}
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Change the program given in Activity 3 such that it accepts the input at command-line.

Task 2:

Write a C++ program that accepts a number as input and find whether it is a palindrom or not

Task 3:

Write a program that creates an array of size 1000 and fills this array with random numbers between 2 and 100; and then it finds how many of these numbers are prime.

Lab No. 06
Using Fork, Exec, Wait & Exit System-Calls for Creating Chile Processes

Objective:

This lab describes how a program can create, terminate, and control children processes using system calls.

Activity Outcomes:

On completion of this lab students will be able to:

- Write programs that uses process creation system call fork ( )
- Use exec system call
- Use wait system call
- Use sleep system call

Instructor Notes

As pre-lab activity, read the content from the following (or some other) internet source: https://www.geeksforgeeks.org

1) Useful Concepts

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]
```

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.

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.

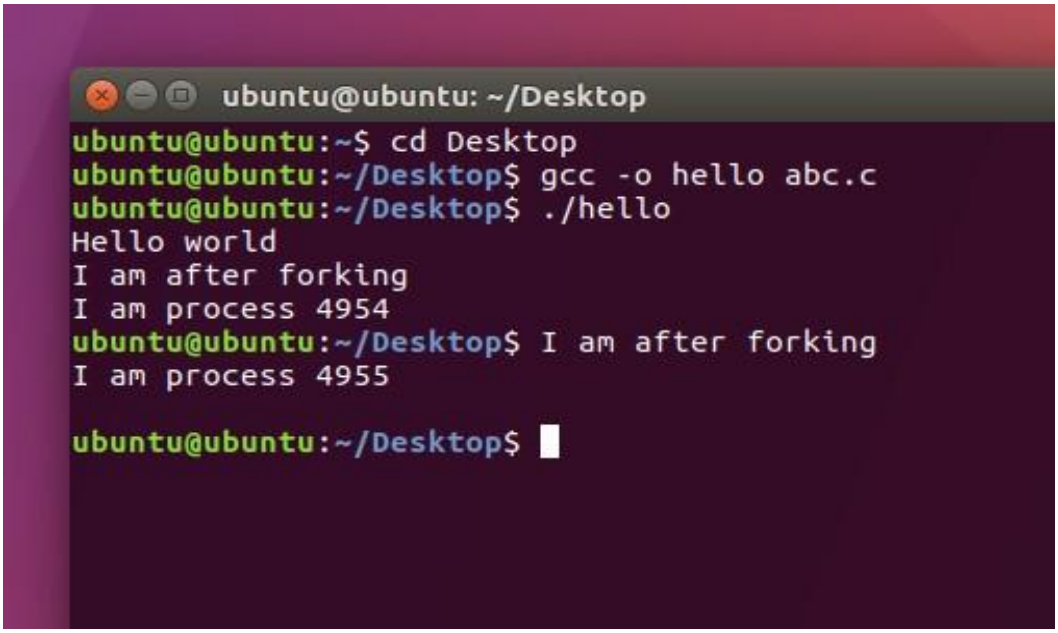
Example 1 – Single fork() :

```
  #include <stdio.h>
  #include <unistd.h> /* contains fork
  prototype */ int main(void)
  {
  printf("Hello orld!\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 thenext statement.

Note that:

- There is no guarantee which process will print I am a process first.
- The child process begins execution at the statement immediately after the fork, not at the beginning of the program.
- 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:

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.

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"

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.

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 orunusual occurrence.

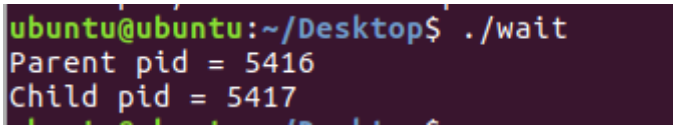
Example:

// 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:



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.

Ecec ( )

The exec family of functions replaces the current running process with a new process. It can beused 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 youspecify 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: 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;
}
```

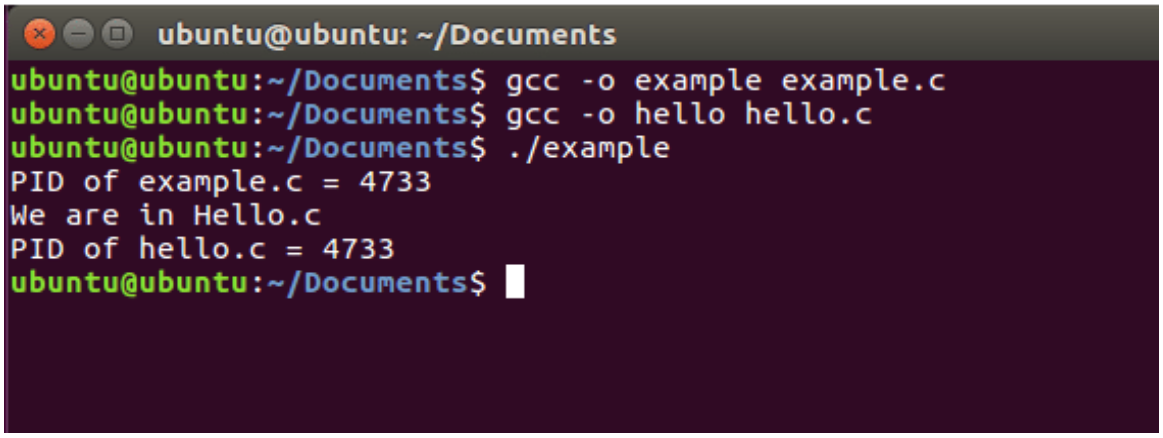
hello.c

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) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*25*	*Medium*	*CLO-7*
2	*25*	*Medium*	*CLO-7*

Activity 1:

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

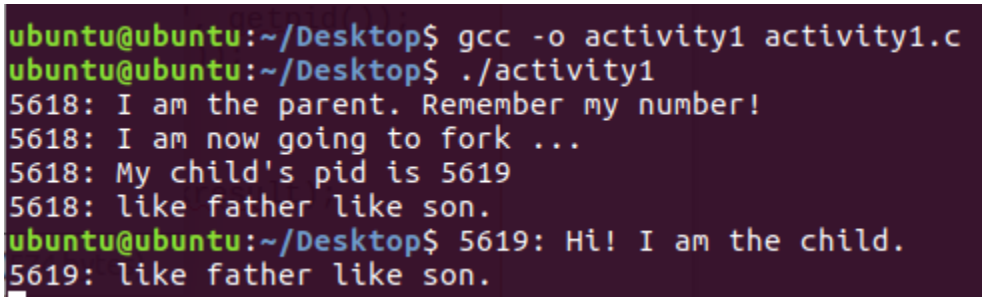
- ***Print something and Check id of the parent process***
- ***Create a child process and print child process id in parent process***
- ***Create a child process and print child process id in child process***

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;
}
```

Out-put



Activity 2:

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.

Solution:

```
#include <stdio.h>
int 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()) ;
return 0;
}
```

Output:

Activity 3:

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;
}
```

Out-put:

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

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write a C++ program that creates an array of size 1000 and populates it with random integers between 1 and 100. Now, it creates two child processes. The first child process finds how many prime numbers are there among first 500 number while the second child process finds the number of prime numbers among the remaining 500 numbers.

Lab No. 07
Creating Multithreaded Applications

Objective
The objective of this lab is to familiarize students with the implementation of inter-process communication using shared memory and message passing.

Activity Outcomes
On completion of this lab students will be able to

- Write cooperating processes that share data using shared memory
- Write cooperating processes that share data using message passing

Instructor Notes
As pre-lab activity, read the content from the following (or some other) internet source: https://www.geeksforgeeks.org

1) Useful Concepts

Cooperating Processes

A process can be of two type: Independent or Cooperating. An independent process is not affected by the execution of other processes while a cooperating process can affect or 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. Cooperating processes need to share data with each other. 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 cooperation between them. The most common methods to share data between processes include: Shared Memory and Message passing.

Shared Memory
Shared memory is a common way to share data between processes. In shared memory, the communicating processes request the OS to declare some memory that can be shared between them. OS declares this shared memory in user space and it is under the control of user processes. Once the shared memory is declared then it is attached with all of the communicating processes. Now, the writer process can write data in the shared memory that can be read by the reader processes.
Communication using shared memory takes place in the following steps

Writer Process	**Reader Process**
1. Declare shared memory if it is not already declared 2. Attach shared memory with the process 3. Write data in shared memory 4. Detach shared memory from the processes	1. Declare shared memory if it is not already declared 2. Attach shared memory with the process 3. Read data from shared memory 4. Detach shared memory from the processes

	5. Destroy shared memory if shared data is no more required

Now, we discuss the routines that can be used to implement shared memory.

Routine	Description
ftok( )	To generate a unique key
shmget(key, memory_size, shmflag)	Used to declare shared memory. Upon successful completion, *shmget()* returns an identifier for the shared memory segment.
*shmat(int shm_id, void *shmaddr, int shmflag)*	Before you can use a shared memory segment, you have to attach yourself to it using *shmat()*.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(void *shmaddr)*	Used to detach the process from the shared memory
shmctl(int shm_id, IPC_RMID, NULL)	It is used to destroy the shared memory

Message Passing

Message passing is another way to share data between processes. Communication between processes takes place in form of messages. This communication remains under the control of OS. OS that support message passing, provide facility/Services to send and receive messages. Inter-process communication using message passing takes place in the following steps:

Sender Process	**Receiver Process**
1. Request the OS to declare message queue if it is not already declared 2. Send message in the queue	1. Request the OS to declare message queue if it is not already declared 2. Receive message from the queue 3. Destroy message queue if message/shared data is no more required

Now, we discuss the routines that can be used to implement shared memory.

Routine	Description
ftok( )	To generate a unique key
msgget(key, shmflag)	Used to declare message queue. Upon successful completion, *shmget()* returns an identifier for the shared memory segment.
*msgsnd(int msg_id, void *msgaddr, size_t msgsize, int msgflag)*	Used to send message in the message queue
*msgsnd(int msg_id, void *msgaddr, size_t msgsize, int msgtype, int msgflag)*	Used to receive message from message queue
msgctl(int msg_id, IPC_RMID, NULL)	It is used to destroy the message queue

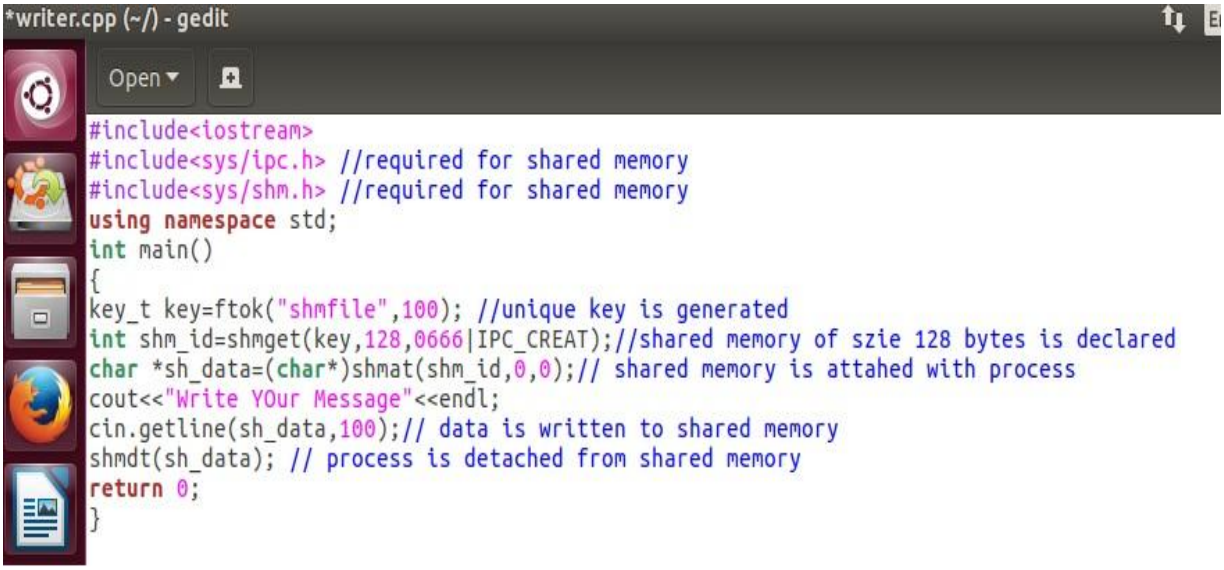
2) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*25*	*Medium*	*CLO-7*
2	*25*	*Medium*	*CLO-7*

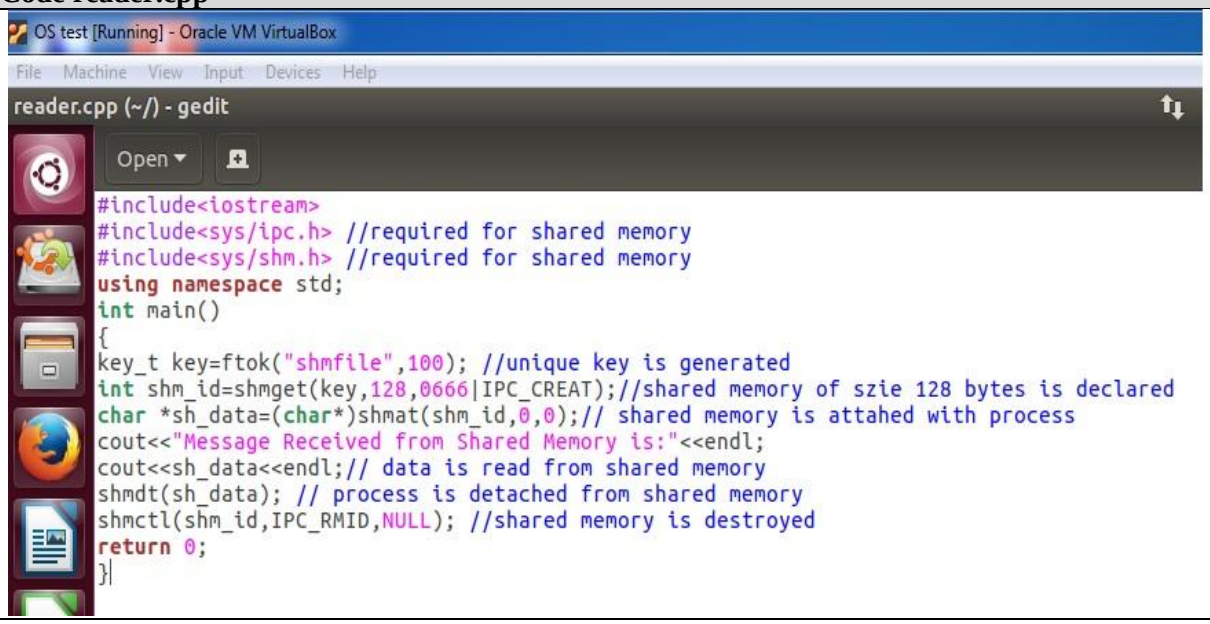
Activity 1:

In this activity, we implement the shared memory. We have written two programs. The first program is writer.cpp. This program declares a shared memory and writes data in it. The second program; reader.cpp reads the data from shared memory written by the writer.

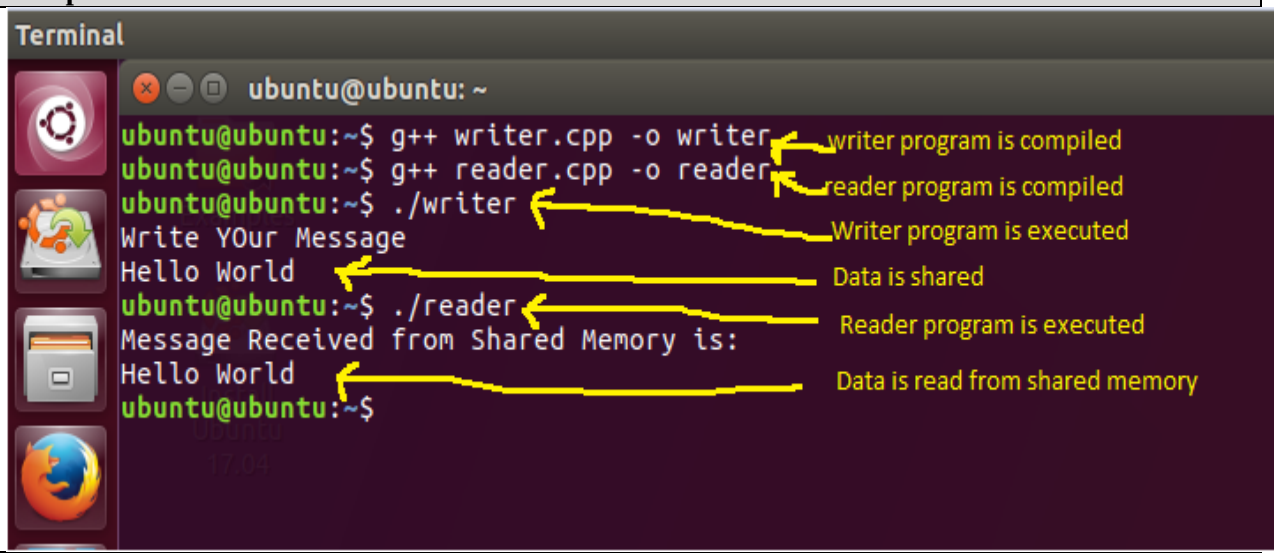
Code writer.cpp



Code reader.cpp



Out-put



Activity 2:

In this activity, we show how the IPC is implemented using the message passing. First, we have written a program sender.cpp that declares a message queue and sends a message in it. The second program receiver.cpp receives the message from the message queue.

Code sender.cpp

der.cpp (~/) - gedit

Open

sender.cpp

receiv

```
#include<iostream>
#include<sys/ipc.h> //required for IPC
#include<sys/msg.h> //required for message passing
using namespace std;
struct message  // structure for message is defined
{
long message_type;
char message_text[100];
};
int main()
{
message msg; // variable of type message is declared
msg.message_type=1;
cout<<"Please Write Your Message"<<endl;
cin.getline(msg.message_text, 100); //message is typed
key_t key=ftok("shmfile",100); //unique key is generated
int msg_id=msgget(key,0666|IPC_CREAT);//message queue is declared
msgsnd(msg_id,&msg,sizeof(msg.message_text),0);// mesage is sent to message queue
return 0;
}
```

Code receiver.cpp

receiver.cpp (~/) - gedit

Open

```
#include<iostream>
#include<sys/ipc.h> //required for IPC
#include<sys/msg.h> //required for message passing
using namespace std;
struct message  // structure for message is defined
{
long message_type;
char message_text[100];
};
int main()
{
message msg; // variable of type message is declared
key_t key=ftok("shmfile",100); //unique key is generated
int msg_id=msgget(key,0666|IPC_CREAT);//message queue is declared
msgrcv(msg_id,&msg,sizeof(msg.message_text),1,0);// mesage is received from message queue
cout<<"Message Received From Message Queue is;"<<endl;
cout<<msg.message_text<<endl;// message is displayed
return 0;
}
```

Out-put

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write a C++ program that creates 3 child processes. Each process creates a random number between 0 and 10. Now, each of the child process takes a value between 0 and 10 from user and compares it with the random number. If the user's guess is equal to the random number then user is declared winner. This process continues as long as the user wins. In the end, each child process shares the number of turns taken by each user to win. The parent process then decides the winner i.e. the user that takes minimum number of turns to win.

Lab No. 08
Creating Multithreaded Applications

Objective
This lab is designed to familiarize the students with the implementation of multithreading.

Activity Outcomes:
On completion of this lab students will be able to

- Implement simple multithreaded applications

Instructor Notes
As pre-lab activity, read the content from the following (or some other) internet source:
https://www.geeksforgeeks.org/multithreading-in-cpp/

1) Useful Concepts

Multithreading

A thread is a path of execution within a process. A process can contain multiple threads. A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. Parallelism is the feature that allows your computer to run two or more programs from same application simultaneously. As a result, our applications are executed in less time and become more interactive. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.

We can implement parallel applications in two ways: process-based and thread-based. Process-based parallelism is achieved through the simultaneous execution of more than one processes from the same application while thread-based parallelism deals with the simultaneous execution of pieces of the same processes. Generally, thread-based parallelism is considered efficient than process-based parallelisms.

Pthread Library

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.

Creating Threads

We can use the following routine to create a POSIX thread

```
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 is as follows:

Parameter	Description
Thread	An unique identifier for the new thread returned by the subroutine.
Attr	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.
start_routine	The C++ routine that the thread will execute once it is created.
Arg	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.

It is to be noted that global variables can be accessed from all threads while variables defined within a thread are not accessible to other threads.

Displaying Thread ID

We can use the pthread_self( ) routine to display the id of the current thread.

Terminating Threads

Following routine terminates a POSIX thread

```
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.

Joining and Detaching Threads

Following routines are used for joining threads

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

The pthread_join subroutine blocks the c

alling thread until the specified threadid thread terminates.

Note: To compile a program that uses pthread library, we need to use the –pthread option. For example, to compile a program test.cpp we can write the following command

```
g++  test.cpp –o  test  -pthread
```

2) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*20*	*Medium*	*CLO-7*
2	*25*	*Medium*	*CLO-7*

Activity 1:

The following 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.

Solution:

Code

```
#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);
```

Out-put

```
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:

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

Solution:

Code

```
#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);
    }
```

Out-put

```
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.
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write C++ program that

- *Declares an array of size 1000 and populates it with random numbers between 1 and 100. Then it finds factorial of these numbers and checks how many of these prime numbers are. Also find the time taken by the system to perform these tasks.*
- *Now, create two thread such that the first thread process finds the factorial of these numbers (given in the above array) while the other finds how many of these are prime. Now, calculate the time taken by these children processes to perform these tasks.*
- *Now, compare which approach performs better*

Lab No. 10
Synchronization: Two-Process Solutions, MUTEX, and Semaphore

Objective
This lab is designed to implement the solutions to the critical-solution problem.

Activity Outcomes:
On completion of this lab students will be able to

- Implement two process solutions to critical-section problem
- Solve the CS problem using MUTEX and Semaphore

Instructor Notes
As pre-lab activity, read the content from the following (or some other) internet source: https://www.geeksforgeeks.org

1) Useful Concepts

The Critical-Section Problem

Cooperating processes or threads share some data with each other. If two or more threads or processes access and manipulate the shared data concurrently then this may result in data inconsistency. To avoid such data inconsistencies, we need to make it sure that threads/processes must be synchronized and if one thread/process is manipulating the shared data then no other thread/process should be allowed to access that data.

Each cooperating thread/process has some segments of critical code that is the segment of code where shared data is accessed and manipulated. These segment of codes are called critical-sections. We need to make it sure that if one thread/process is executing its critical section then no other process should be allowed to execute its critical section. Designing solutions to ensure this; is called the CS problem. We can define the CS problem as:

The critical section problem is used to design a protocol followed by a group of processes, so that when one process has entered its critical section, no other process is allowed to execute in its critical section.

The general structure of a solution to critical-section problem is:

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Two Process Solutions to CS problem

A simple solution:

First, we discuss a simple solution. This solution uses a variable turn. The value of turn decides, whose turn it is to enter in CS. The solution is given below:

```
Code for process i
do {
     while (turn == j); // Entry
Code
      critical section
       turn = j;  //Exit code
     remainder section
   } while (true);
```

```
Code for process j
do {
     while (turn == i); // Entry
Code
      critical section
       turn = i;  //Exit code
     remainder section
   } while (true);
```

This simple solution ensure mutual exclusion and bounded waiting conditions but it fail to ensure progress condition.

Peterson's Solution:

Peterson's solution satisfies all the conditions for a good solution. It is also a two process solution. The two processes share two variables: int turn and Boolean flag[2]. The variable turn indicates whose turn it is to enter the critical section while the flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready. The pseudo code of the Peterson's solution is given below:

```
Code for process i
do
{
       flag[ i ] = true;
       turn = j;
       while (flag[ j ] && turn =
= j);
      critical section
       flag[ i ] = false;
      remainder section
} while (true);
```

```
Code for process j
do
{
       flag[ j] = true;
       turn = i;
       while (flag[ i ] && turn =
= i);
      critical section
       flag[ j ] = false;
      remainder section
} while (true); while (true);
```

MUTEX Lock

MUTEX lock is software based solution to CS problem and is applicable on n threads/processes. 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.

A typical sequence in the use of a MUTEX is as follows:

- Create and initialize a MUTEX variable

- Several threads attempt to lock the MUTEX
- Only one succeeds and that thread owns the MUTEX
- The owner thread performs some set of actions
- The owner unlocks the MUTEX
- Another thread acquires the MUTEX and repeats the process
- Finally the MUTEX is destroyed.

The routines to perform these tasks are given below:

```
pthread_mutex_init(pthread_mutex_t  var, pthread_mutexattr_t  attr)
// to initialize MUTEX variable
```

```
pthread_mutex_lock ( pthread_mutex_t var )     // to lock CS
```

```
pthread_mutex_unlock ( pthread_mutex_t var)    // to unlock CS
```

```
pthread_mutex_destroy(pthread_mutex_t  var)  // to destroy MUTEX
variable
```

Semaphore

Semaphore is another synchronization tool that can be used to solve several synchronization problems. Semaphore is an integer variable but it can be accessed only through two functions which are wait( ) and signal( ).

```
Pseudo code for wait function

wait(S) {

    while (S <= 0)

       ; // busy wait

    S - -;

}
```

```
Pseudo code for signal function

signal (S) {

    S++;

}
```

```
Pseudo code for solution to CS problem using semaphore

do {

    wait(s)  //entry code

       critical section

    signal(S)  // exit code

      remainder section

 } while (true);
```

The following routines are used to implement POSIX semaphore

```
#include <semaphore.h>    // header-file
```

```
sem_t   // semaphore data type
```

```
int sem_init(sem_t *sem, int pshared, unsigned value);   // to
initialize of semaphore
```

```
sem_wait()   // wait function
sem_post()   // signal function
int sem_destroy(sem_t *sem);  // to destroy semaphore
```

2) Solved Lab Activities

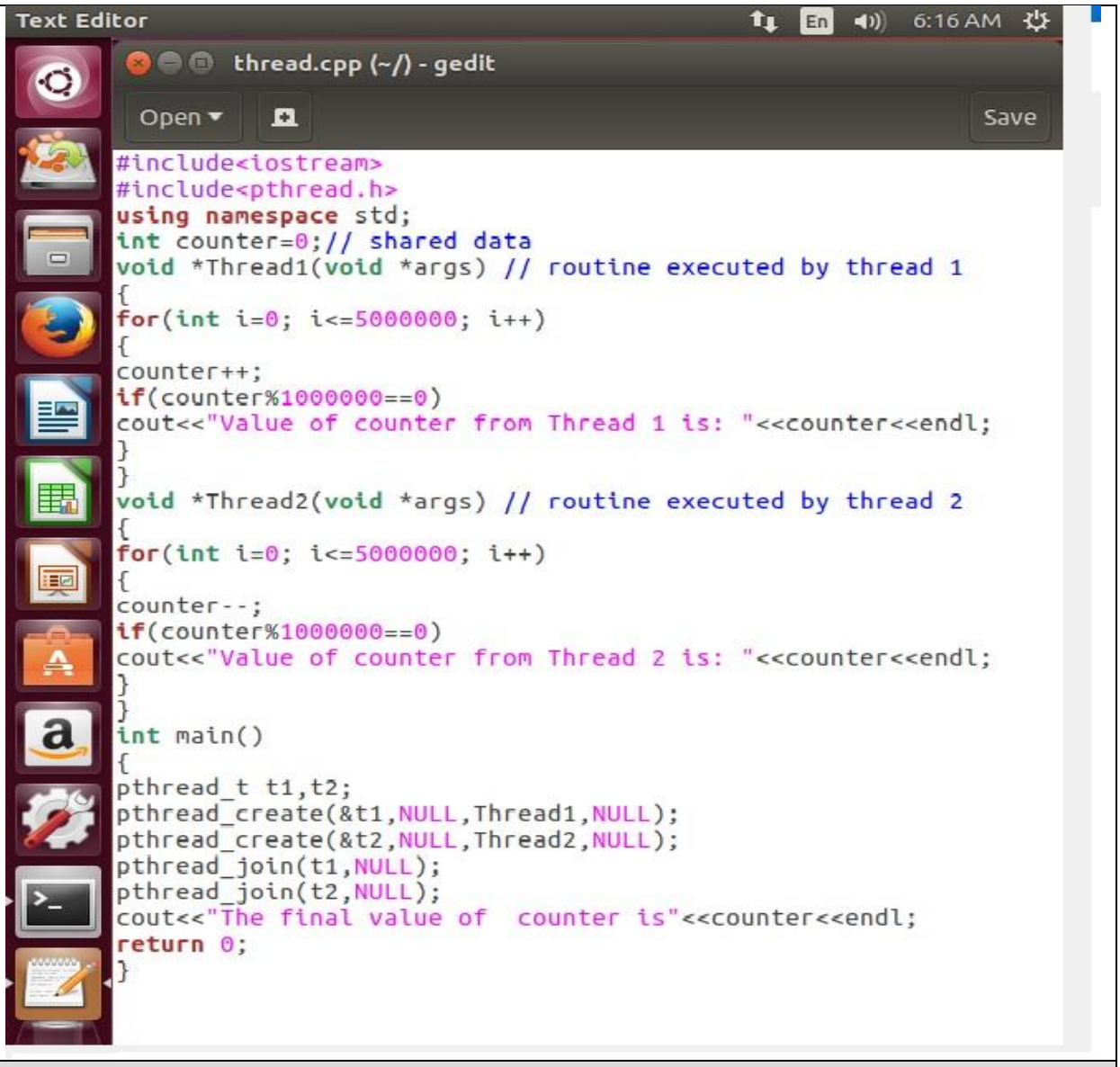
Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*20*	*Medium*	*CLO-7*
2	*10*	*Medium*	*CLO-7*
3	*15*	*Medium*	*CLO-7*
4	*15*	*Medium*	*CLO-7*

Activity 1:

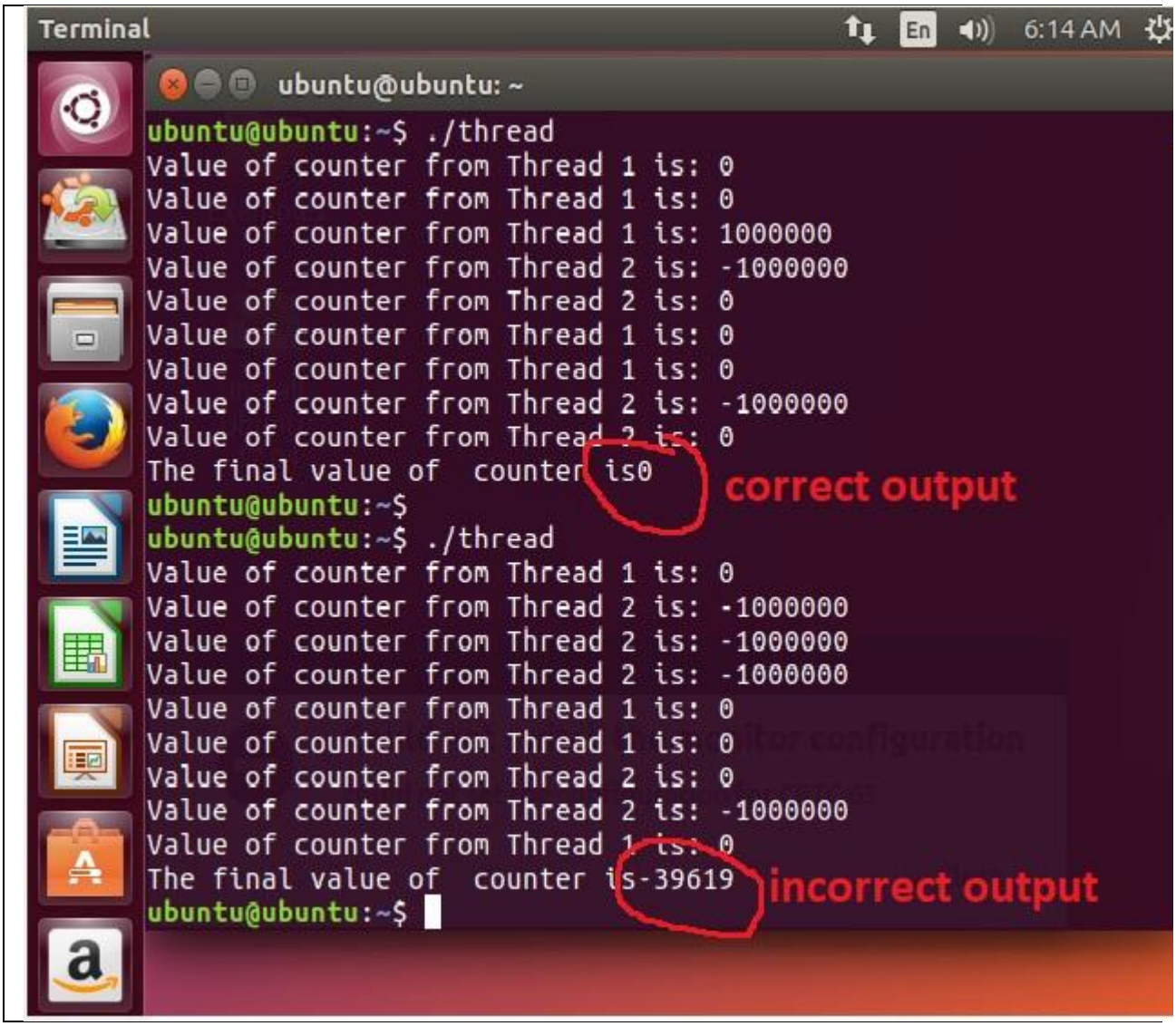
In this activity, we identify how the concurrent access to shared data may result in data inconsistency. We create two threads that manipulate a shared variable counter concurrently. As a result, sometimes we get correct output while sometimes the output is not c

Solution:

Code


Text Editor
En
6:16 AM
thread.cpp (~/) - gedit
Open
Save
#include<iostream>
#include<pthread.h>
using namespace std;
int counter=0;// shared data
void *Thread1(void *args) // routine executed by thread 1
{
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of counter from Thread 1 is: "<<counter<<endl;
}
}
void *Thread2(void *args) // routine executed by thread 2
{
for(int i=0; i<=5000000; i++)
{
counter--;
if(counter%1000000==0)
cout<<"Value of counter from Thread 2 is: "<<counter<<endl;
}
}
int main()
{
pthread_t t1,t2;
pthread_create(&t1,NULL,Thread1,NULL);
pthread_create(&t2,NULL,Thread2,NULL);
pthread_join(t1,NULL);
pthread_join(t2,NULL);
cout<<"The final value of counter is"<<counter<<endl;
return 0;
}

Out-put


Terminal
En
6:14 AM
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./thread
Value of counter from Thread 1 is: 0
Value of counter from Thread 1 is: 0
Value of counter from Thread 1 is: 1000000
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 2 is: 0
Value of counter from Thread 1 is: 0
Value of counter from Thread 1 is: 0
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 2 is: 0
The final value of counter is0
correct output
ubuntu@ubuntu:~$
ubuntu@ubuntu:~$ ./thread
Value of counter from Thread 1 is: 0
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 1 is: 0
Value of counter from Thread 1 is: 0
Value of counter from Thread 2 is: 0
Value of counter from Thread 2 is: -1000000
Value of counter from Thread 1 is: 0
The final value of counter is-39619
incorrect output
ubuntu@ubuntu:~$

Activity 2:

Now, we implement the simple solution in the code written in Activity 1 to protect the CS

Solution:

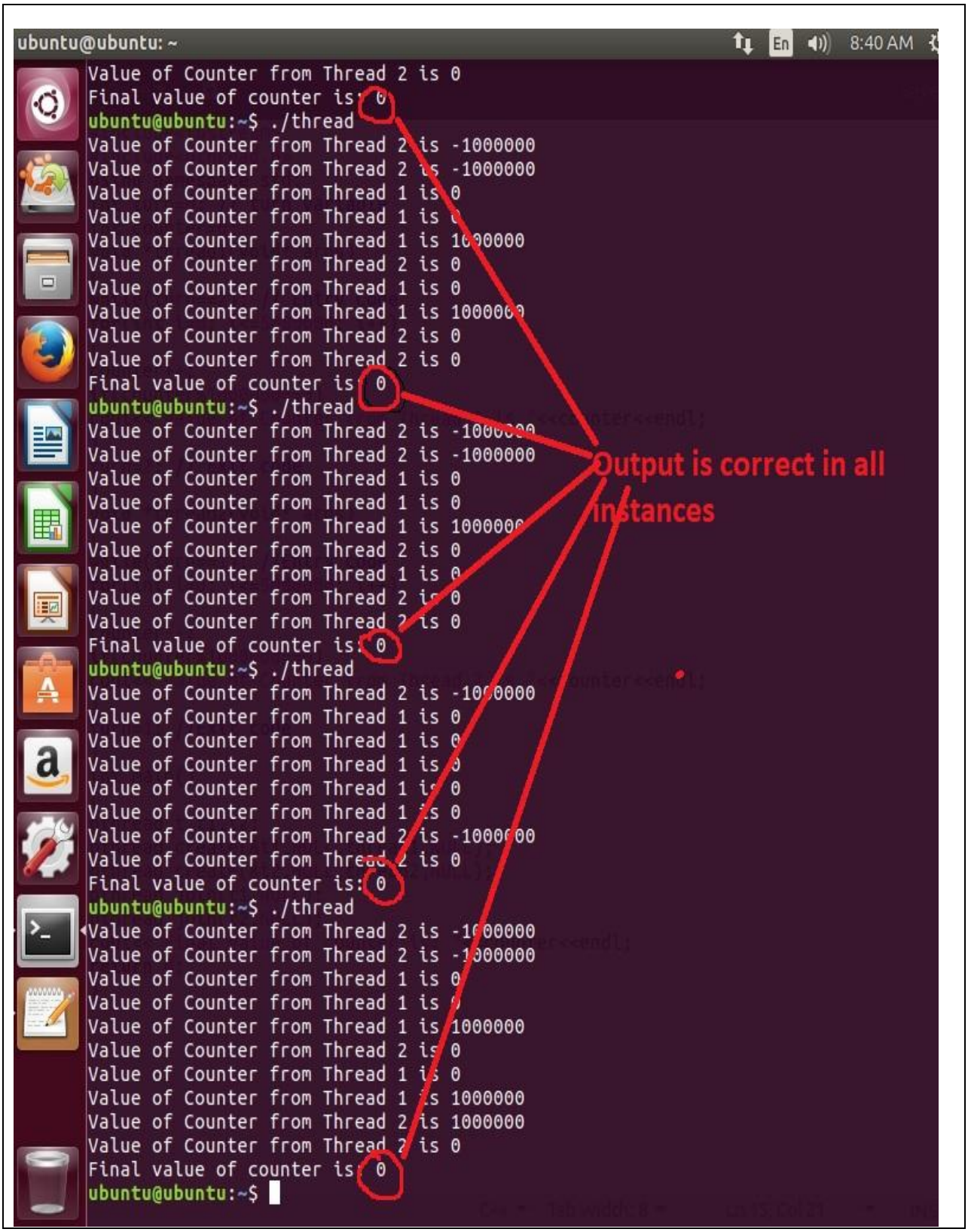
Code

thread.cpp (~/) - gedit

Open

```
#include<iostream>
#include<pthread.h>
using namespace std;
int turn=0; // turn variable
int counter=0;
void *thread1(void* args)
{
while(turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
turn=2; // Exit code
}
void *thread2(void* args)
{
while(turn==1); //Entry code
for(int i=0; i<=5000000; i++)
{
counter--;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 2 is "<<counter<<endl;
}
turn=1; //Exit code
}
int main()
{
pthread_t t1,t2;
pthread_create(&t1,NULL,thread1,NULL);
pthread_create(&t2,NULL,thread2,NULL);
pthread_join(t1,NULL);
pthread_join(t2,NULL);
cout<<"Final value of counter is: "<<counter<<endl;
return 0;
}
```

Out-put


ubuntu@ubuntu: ~
En
8:40 AM
Value of Counter from Thread 2 is 0
Final value of counter is: 0
ubuntu@ubuntu:~$ ./thread
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 1000000
Value of Counter from Thread 2 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 1000000
Value of Counter from Thread 2 is 0
Value of Counter from Thread 2 is 0
Final value of counter is: 0
ubuntu@ubuntu:~$ ./thread
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 1000000
Value of Counter from Thread 2 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 2 is 0
Value of Counter from Thread 2 is 0
Final value of counter is: 0
ubuntu@ubuntu:~$ ./thread
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is 0
Final value of counter is: 0
ubuntu@ubuntu:~$ ./thread
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 1000000
Value of Counter from Thread 2 is 0
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is 1000000
Value of Counter from Thread 2 is 1000000
Value of Counter from Thread 2 is 0
Final value of counter is: 0
ubuntu@ubuntu:~$
Output is correct in all instances

Activity 3:

In this activity, we show that how the progress in not satisfied in simple solution

Solution:

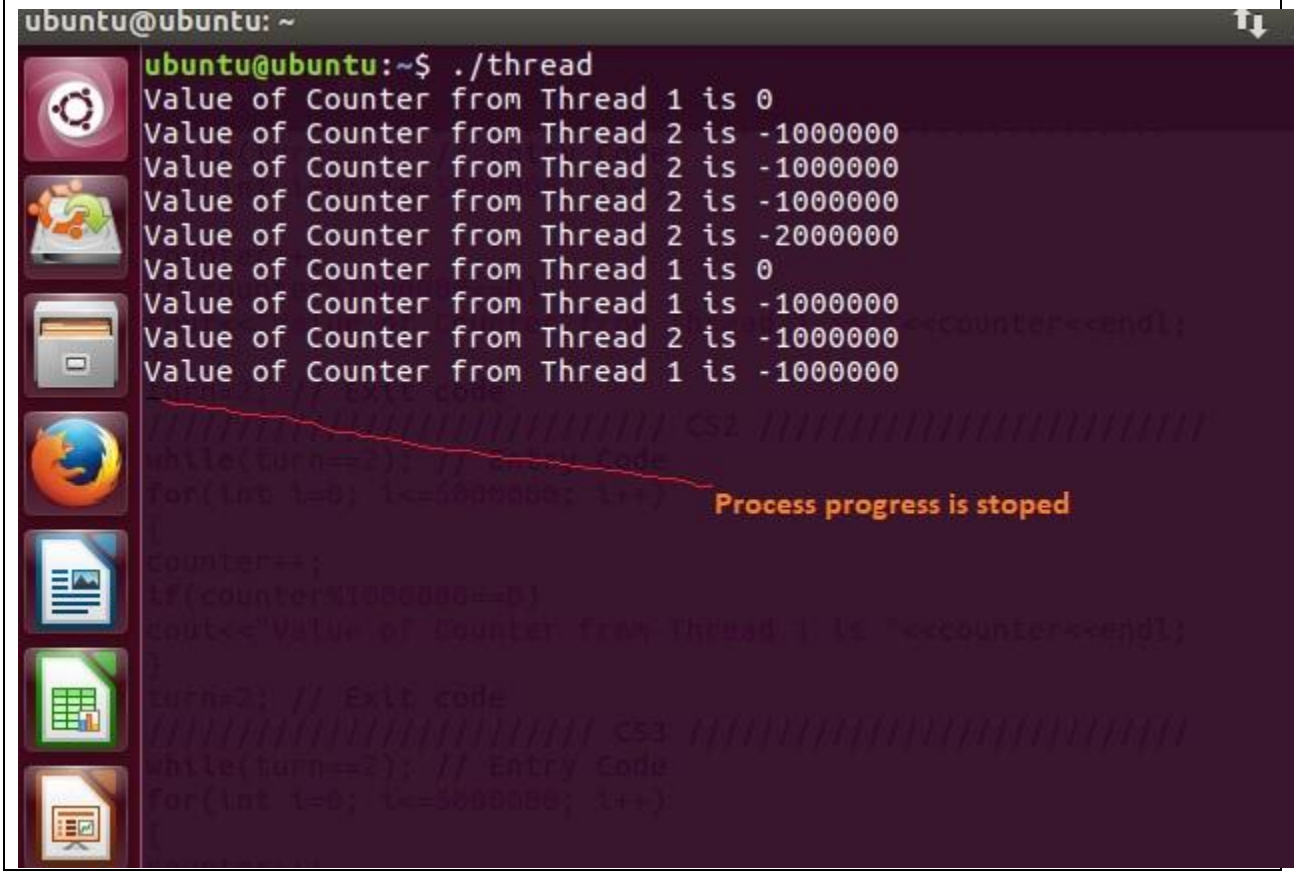
Code

thread.cpp (~/) - gedit

Open

```
#include<iostream>
#include<pthread.h>
using namespace std;
int turn=0; // turn variable
int counter=0;
void *thread1(void* args)
{
//////////////////////////  CS1 ////////////////////////
while(turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
turn=2; // Exit code
////////////////////////////// CS2 ////////////////////////
while(turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
turn=2; // Exit code
//////////////////////////// CS3 ////////////////////////////
while(turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
turn=2; // Exit code
}
void *thread2(void* args)
{
while(turn==1); //Entry code
for(int i=0; i<=5000000; i++)
{
counter--;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 2 is "<<counter<<endl;
}
turn=1; //Exit code
}
int main()
{
pthread_t t1,t2;
pthread_create(&t1,NULL,thread1,NULL);
pthread_create(&t2,NULL,thread2,NULL);
pthread_join(t1,NULL);
pthread_join(t2,NULL);
cout<<"Final value of counter is: "<<counter<<endl;
return 0;
}
```

Out-put


ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./thread
Value of Counter from Thread 1 is 0
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 2 is -2000000
Value of Counter from Thread 1 is 0
Value of Counter from Thread 1 is -1000000
Value of Counter from Thread 2 is -1000000
Value of Counter from Thread 1 is -1000000
Process progress is stoped

Activity 4:

In this activity, we implement the Peterson's solution and show that progress condition is satisfied.

Solution:

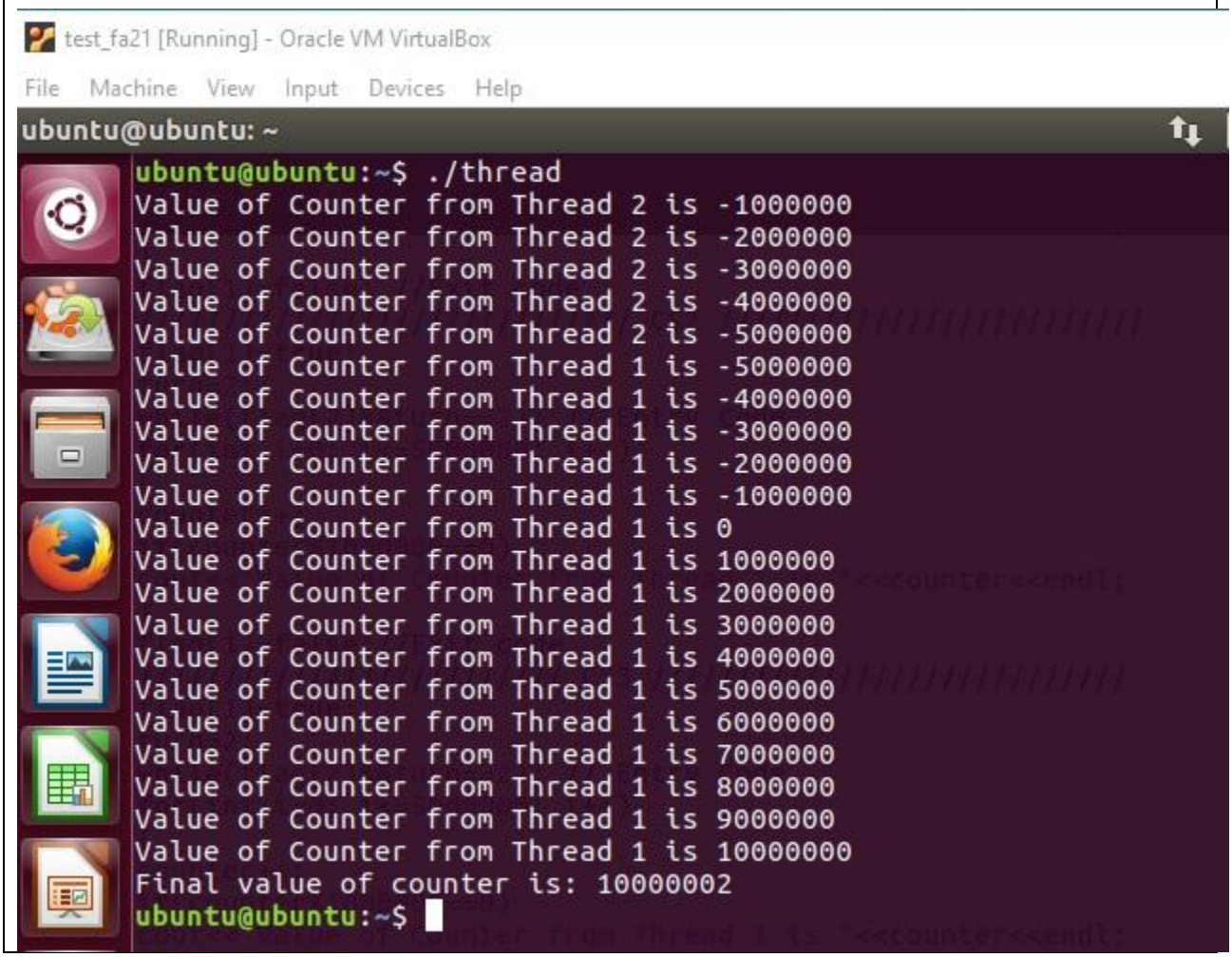
Code

thread.cpp (~/) - gedit

Open

```
#include<iostream>
#include<pthread.h>
using namespace std;
bool flag[3]={false,false,false}; // flag array
int turn=0; // turn variable
int counter=0;
void *thread1(void* args)
{
//////////////////////////  CS1 ////////////////////////
flag[1]=true;
turn=2;
while(flag[2] && turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
flag[1]=false; //Exit code
////////////////////////////// CS2 ////////////////////////
flag[1]=true;
turn=2;
while(flag[2]&&turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
flag[1]=false;//Exit code
////////////////////////// CS3 ////////////////////////////
flag[1]=true;
turn=2;
while(flag[2]&&turn==2); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter++;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 1 is "<<counter<<endl;
}
flag[1]=false;//Exit code
}
void *thread2(void* args)
{
flag[2]=true;
turn=1;
while(flag[1]&&turn==1); // Entry Code
for(int i=0; i<=5000000; i++)
{
counter--;
if(counter%1000000==0)
cout<<"Value of Counter from Thread 2 is "<<counter<<endl;
}
flag[2]=false; //Exit code
}
int main()
{
pthread_t t1,t2;
pthread_create(&t1,NULL,thread1,NULL);
pthread_create(&t2,NULL,thread2,NULL);
pthread_join(t1,NULL);
pthread_join(t2,NULL);
cout<<"Final value of counter is: "<<counter<<endl;
return 0;
}
```

Out-put



3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write the code for the problem given in Activity 3 and protect the CS's using the MUTEX lock

Task 2:

Write the code for the problem given in Activity 3 and protect the CS's using the Semaphore

Lab No. 11
Writing & Executing Shell Scripts, I/O, Variables, and Operators

Objective
This lab introduces the fundamental concepts of shell scripting along with its basic constructs.

Activity Outcomes:
On completion of this lab students will be able to:

- Understand the process of writing and executing shell scripts
- Use input/output commands
- Use variables and operators

Instructor Notes
As pre-lab activity, read Chapter 24, 26, 28 & 34 from the book "The Linux Command Line", William E. Shotts, Jr.

1) Useful Concepts

Introduction to Shell
Shell is an Interface between the user and the operating system. It is simply a program that 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. Other examples of Linux shells include: Korn, C Shell, tcsh, bash, zsh, etc.

Shell Script
In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line. Shell scripts are the equivalent of batch fi les in MS-DOS, and can contain long lists of commands, complex flow control, arithmetic evaluations, user-defined variables, user-defined functions, and sophisticated condition testing. The basic advantage of shell scripting includes:

- Shell script can take input from user, file and output them on screen.
- Useful to create our own commands.
- Save lots of time.
- To automate some task of day today life.
- System Administration part can be also automated.

Writing and Executing Shell Script
We need to perform following steps to create and execute shell script

1. **Write Script:** Shell scripts are ordinary text files. We can use any text editor such as gedit to write shell script

2. **Make the shell script executable:** next step is to make the script executable. We can do it using the chmod command. For example if the script file is saved with the name test then me can make it executable in the following way:

```
sudo  chmod  777  test
```

3. **Place the script at some suitable place:** The shell automatically searches certain directories for executable files when no explicit pathname is specified. For maximum convenience, we will place our scripts in these directories. For example, we can move the executable file test to /usr/bin directory using the following command:

```
sudo  mv test /usr/bin
```

Now, we can execute the test file just like normal commands.

Writing First Shell Script:

Now, we write a simple Shell script to demonstrate the above mention steps. This script just shows a *"Hello World"* message on the terminal screen. Open the gedit and write the following code.

```
1 #!/bin/bash
2 #this is the first script
3 echo "Hello World"
```

The first line of the code tells about the shell for which we are writing the script. The second line is a comment while the third line displays the message on the terminal screen. After writing this code save it with the name test. Now, make it executable using the chmod command and move it to /usr/bin directory. Now, we can run this script by just writing test on the command line.

Shell Variables

Shell variables can be categorized as system-defined, parameter or user-defined. The details are given below:

1. **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 executable
   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

$# – The number of parameters passed
$$ – The PID of the parent process i.e. shell
$? – exit status of last command run

2. **Parameter variables** contain the values of the parameters passed to a shell script.
   - $0 Path of the program
   - $1, $2 … Store the parameters given to the script
   - $* A list of all parameters, separated by the character defined in IFS
   - $@ Same as $* except the parameters are separated by space character
3. **User defined** variables are subject to the following rules:
   - Variable names can begin with a letter or an underscore and can contain an arbitrary number of letters, digits and underscores
   - No arbitrary upper limit to the variables you can define or use
   - It is not necessary to declare a variable before using it
   - Variables are of loosely typed
   - A variable retains its value from the time it is set – whether explicitly by you or implicitly by the shell – until its value is changed or the shell exits
   - To retrieve the value, precede the variable name with a **dollar sign ($)**
   - 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.

Input/Out-put Commands

echo- The echo command is used to display a line of text/string on standard output or a file. We use quotation marks to display text/string. To display the value of a variable, we need to place $ symbol before the variable name.

```
echo "The value of x is $x"
```

read command- The read built-in command is used to read a single line of standard input i.e. keyboard. This command can be used to read keyboard input or, when redirection is employed, a line of data from a file. The syntax of read command is:

```
read  -options  arguments
```

Where options is one or more of the available options listed below and variable is the name of one or more variables used to hold the input value. If no variable name is supplied, the shell variable REPLY contains the line of data.

```
#! /bin/bash
echo "Please Enter a number"
read  num
echo "You Entered $num"
```

Common options for read command are:

Option	Description
-a	*array* Assign the input to *array*, starting with index zero.
-n num	Read *num* characters of input, rather than an entire line.
-p prompt	-p *prompt* Display a prompt for input using the string *prompt*.
-t second	Timeout. Terminate input after *seconds*. read returns a non-zero exit status if an input times out
-u fd	Use input from file descriptor *fd*, rather than standard input

Operators

An operator is a symbol that usually represents an action or process. There various types of operators supported by bash shell.
Arithmetic Operators- Arithmetic operators are used to perform arithmetic operations on variables. Following arithmetic operators are available in bash shell

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Integer Division
%	Remainder
**	Exponentiation

Writing Arithmetic Expressions- We use arithmetic expansion to perform arithmetic operations. It can be done using the following syntax

```
$((Arithmetic Expression))
```

In the following example, we perform some basic arithmetic operations

```
#! /bin/bash
echo "Please Enter First Number"
read  num1
echo "Please Enter the Second Number"
read  num2
echo "The sum is $(($num1+$num2))"
echo "The multiplication is $(($num1*$num2))"
echo "Square of sum of number is $(($(($num1+$num2))**2))"
```

We can also use expr command to write arithmetic expressions. The syntax is as given below:

```
echo 'expr $num1 + $num2'
```

Note: There must be a space between operator and operands.

Assignment Operators

Assignment operators are used to assign values to variables. The following assignment operators are available in bash shell

Operator	Operation
=	Simple assignment. Assigns value to parameter. Example: a=$b Note: there should be no space between = operator and operands
parameter+=value	Addition. Equivalent to parameter = parameter + value.
parameter-=value	Subtraction. Equivalent to parameter = parameter - value.
parameter*=value	Multiplication. Equivalent to parameter = parameter * value.
parameter/=value	Integer Division. Equivalent to parameter = parameter / value.
parameter%=value	Remainder. Equivalent to parameter = parameter % value.
parameter++	Post increment
Parameter--	Post decrement
++parameter	Pre increment
-- parameter	Post increment

Comparison Operators

These operators are used to compare the values of variables. Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

Logical Operators

These operators are used to perform logical operations. Suppose the values of variable a and b are 10 and 20 respectively then:

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true, then the	[$a -lt 20 -o $b -gt 100]

	condition becomes true.	is true.
-a	This is logical AND. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

File Operators

We have several operators that can be used to test file properties. Assume a variable file holds an existing file name "test" the size of which is 100 bytes and has read, write and execute permission on.

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

2) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	***10***	***Low***	***CLO-6***
2	***10***	***Low***	***CLO-6***
3	***10***	***Low***	***CLO-6***
4	***10***	***Low***	***CLO-6***
5	***10***	***Low***	***CLO-6***

Activity 1:

Write a shell script that gets two numbers from user and perform basic arithmetic operations (+, -, *, /, %, **) on these numbers.

Solution:

Code

*scripts (/usr/bin) - gedit

Open

```
#!/bin/bash
echo "Please Enter the First Number"
read num1
echo "Please Enter the Second Number"
read num2
s=$(($num1+$num2))  #addition
echo "Sum of $num1 and $num2 is $s"
d=$(($num1-$num2))  #subtraction
echo "Dfiference of $num1 and $num2 is $d"
m=$(($num1*$num2))  #multiplication
echo "Product of $num1 and $num2 is $m"
div=$(($num1/$num2))  #division
echo "Division of $num1 and $num2 is $div"
r=$(($num1%$num2))  #remainder
echo "Remainder of $num1 and $num2 is $r"
e=$(($num1**$num2)) #exponentiation
echo "$num1 raise to power $num2 is $e"
```

Out-put

test_fa21 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal File Edit View Search Terminal Help

En 9:53 A

```
ubuntu@ubuntu:~$ sudo chmod 777 scripts
ubuntu@ubuntu:~$ sudo mv scripts /usr/bin
ubuntu@ubuntu:~$ scripts
Please Enter the First Number
18
Please Enter the Second Number
6
Sum of 18 and 6 is 24
Dfiference of 18 and 6 is 12
Product of 18 and 6 is 108
Division of 18 and 6 is 3
Remainder of 18 and 6 is 0
18 raise to power 6 is 34012224
[2]+  Done                    gedit /usr/bin/scripts
ubuntu@ubuntu:~$
```

Activity 2:

Write a shell script that gets two numbers at the command line. First it displays the path of the script and perform basic arithmetic operations (+, -, *, /, %, **) on these numbers.

Solution:

Code

scripts (/usr/bin) - gedit

Open

```
#!/bin/bash
echo "Path of the Script is $0"
num1=$1
num2=$2
s=$(($num1+$num2))  #addition
echo "Sum of $num1 and $num2 is $s"
d=$(($num1-$num2))  #subtraction
echo "Dfiference of $num1 and $num2 is $d"
m=$(($num1*$num2))  #multiplication
echo "Product of $num1 and $num2 is $m"
div=$(($num1/$num2))  #division
echo "Division of $num1 and $num2 is $div"
r=$(($num1%$num2))  #remainder
echo "Remainder of $num1 and $num2 is $r"
e=$(($num1**$num2)) #exponentiation
echo "$num1 raise to power $num2 is $e"
```

Out-put

ubuntu@ubuntu: ~

```
ubuntu@ubuntu:~$ scripts 18 6
Path of the Script is /usr/bin/scripts
Sum of 18 and 6 is 24
Dfiference of 18 and 6 is 12
Product of 18 and 6 is 108
Division of 18 and 6 is 3
Remainder of 18 and 6 is 0
18 raise to power 6 is 34012224
ubuntu@ubuntu:~$
```

Activity 3:

Write a shell script that creates a long list of all directories exists at the current location

Solution:

Code

scripts (/usr/bin) - gedit

Open

```
#!/bin/bash
echo "Following Files Exist at the Current Location
ls -l
```

Out-put

ubuntu@ubuntu: ~

```
ubuntu@ubuntu:~$ scripts
Following Files Exist at the Current Location
total 16
drwxr-xr-x 2 ubuntu ubuntu   80 May 16 08:15 Desktop
drwxr-xr-x 2 ubuntu ubuntu   40 May 16 08:16 Documents
drwxr-xr-x 2 ubuntu ubuntu   40 May 16 08:16 Downloads
drwxr-xr-x 2 ubuntu ubuntu   40 May 16 08:16 Music
drwxr-xr-x 2 ubuntu ubuntu   40 May 16 08:16 Pictures
drwxr-xr-x 2 ubuntu ubuntu   40 May 16 08:16 Public
-rw-r--r-- 1 ubuntu ubuntu  303 May 16 09:52 scripts
drwxr-xr-x 2 ubuntu ubuntu   40 May 16 08:16 Templates
-rwxr-xr-x 1 ubuntu ubuntu 8088 May 16 09:23 thread
-rw-r--r-- 1 ubuntu ubuntu 1467 May 16 09:23 thread.cpp
drwxr-xr-x 2 ubuntu ubuntu   40 May 16 08:16 Videos
ubuntu@ubuntu:~$
```

Activity 4:

Write a shell script that asks the user to enter the file name and deletes that file. After deleting the file it shows a success message and also displays the list of the remaining files

Solution:

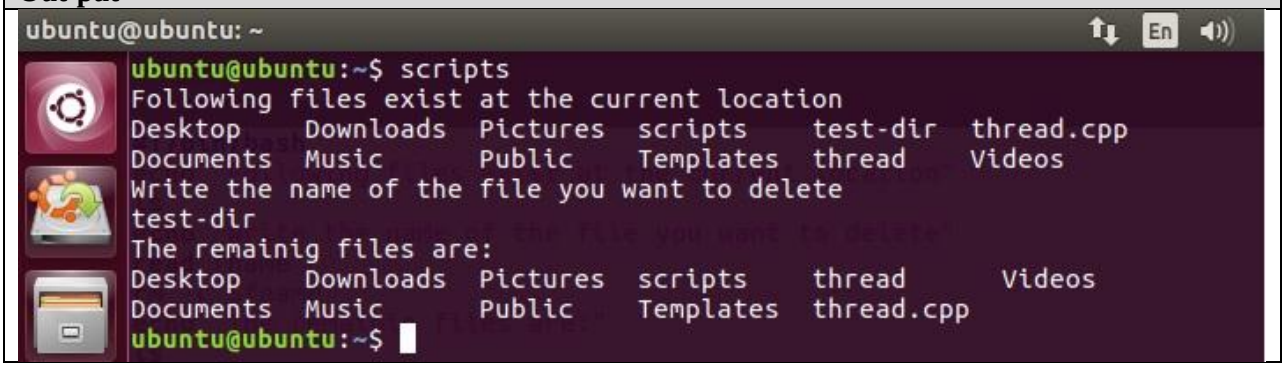
Code

scripts (/usr/bin) - gedit

Open

```
#!/bin/bash
echo "Following files exist at the current location"
ls
echo "Write the name of the file you want to delete"
read fname
rm -r $fname
echo "The remainig files are:"
ls
```

Out-put

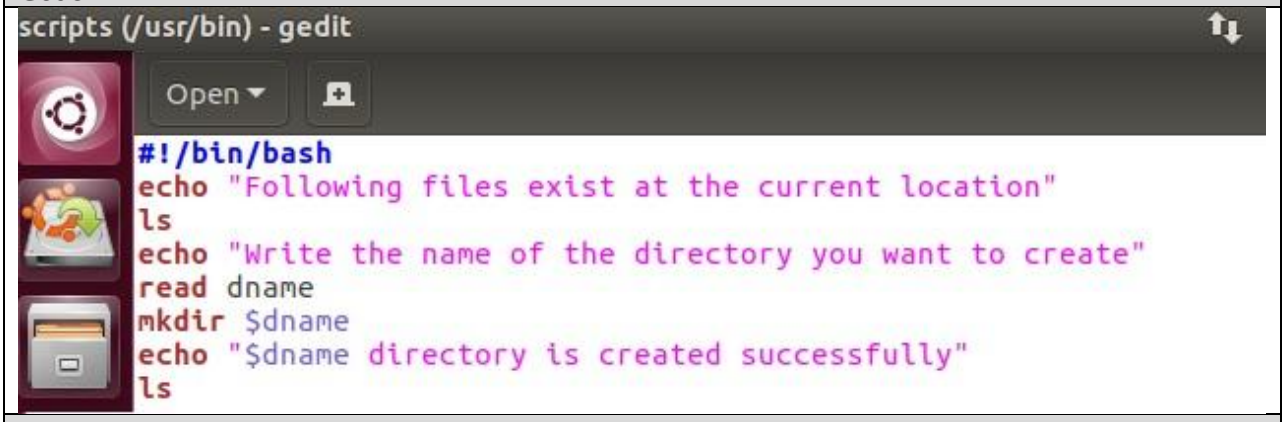


Activity 5:

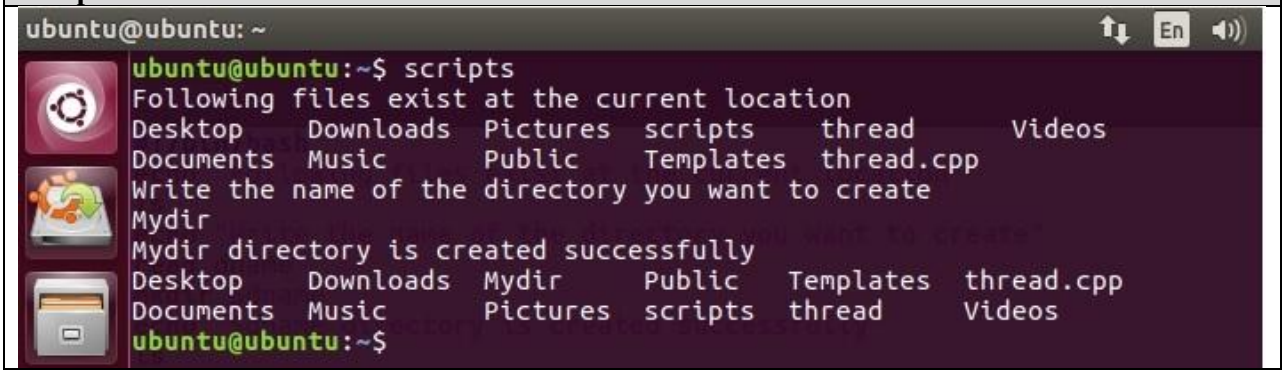
Write a shell script that modifies the basic mkdir command as: first it show the list of existing files on the current location. Then it asks the users to enter the name of the directory; and then creates that directory and display a success message.

Solution:

Code



Out-put



3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write a shell script that asks the user to enter the name of the directory and the destination; and then creates a directory at the given destination. Also display a success message along with the list of directories only.

Task 2:

In this task you have to re-write the cp command in such a way that your script asks the users to enter the path of files to be copied and the destination; and then copies all of the files at the destination. Also display a success message along with the list of directories only.

Task 3:

Write a shell script that takes a file path and a search string as input; and displays all of the lines that contain that string.

Task 4:

Write a shell script that takes a list of text files as input and merges all those files into a single file; and displays its contents in the less application.

Lab No. 12
Writing Shell Scripts using Conditional-Statements, and Loops

Objective
The objective of this lab is to familiarize the students with the use of conditional and looping statements in shell scripting.

Activity Outcomes:
On completion of this lab students will be able to

- Write shell scripts that uses conditional statements
- Use looping statements available in bash shell

Instructor Notes
As pre-lab activity, read Chapter 27 &31 from the book "The Linux Command Line", William E. Shotts, Jr.

1) Useful Concepts

Conditional Statements

A conditional statement tells a program to execute an action depending on whether a condition is true or false. It is often represented as an if-then or if-then-else statement.

if Statement
if statement is provided in many variant in bash shell. The most commonly used syntax is as given below:

```
if  [ condition ]
      then
          commands
fi
```

Note that there must be a space between condition and both opening and closing brackets. The syntax of else-if command is

```
if  [ condition ]
then
     commands
elif [ condition ]
```

```
then
            commands
.
.
.
else
              commands
fi
```

The following example shows the use of if statement. In this example, we take a number as input from user and check whether it is even or odd.

```
#! /bin/bash
echo "Please Enter a Number"
read  num
if [ $((num%2)) -eq 0 ]
then
           echo "$num is an even
number"
else
            echo "$num is an odd
number"
fi
```

Exit Status

Commands issue a value to the system when they terminate, called an exit status. This value, which is an integer in the range of 0 to 255, indicates the success or failure of the command's execution. By convention, a value of zero indicates success and any other value indicates failure. The exit status of a command is saved in a system variable $?.

Other syntax for if statement

Recent versions of bash include a compound command that acts as an enhanced replacement for test. It uses the following syntax:

```
if  [[ condition ]]
then
             commands
fi
```

Similarly, (( )) syntax can also be used which is designed for arithmetic expressions.

Case statement

The bash case statement is generally used to simplify complex conditionals when you have multiple different choices. Using the case statement instead of nested if statements will help you make your bash scripts more readable and easier to maintain. The syntax of case statement is:

```
case    EXPRESSION
in
Pattern-1)

Commands
                ;;
Pattern-2)

Commands
               ;;
          .
          .
          .
Pattern-n)

Commands
              ;;
*)

Commands
              ;;
esac
```

Following are some examples of some valid patterns for the case statement.

Pattern	Description
a)	Matches if *word* equals "a".
[[:alpha:]]	Matches if *word* is a single alphabetic character.
???)	Matches if *word* is exactly three characters long
***.txt)**	Matches if *word* ends with the characters ".txt".
***)**	Matches any value of *word*. It is good practice to include this as the last pattern in a case command, to catch any values of *word* that did not match a previous pattern; that is, to catch any possible invalid values.

Looping Statements

A program loop is a series of statements that executes for a specified number of repetitions or until specified conditions are met. While, until and for are the common looping statement provided by bash shell.

For Loop

The original syntax of for loop is:

```
for     variable    in
values
do
        statements
```

```
done
```

Following example shows the working of for loop

```
#!/bin/bash
for     i   in  A B C
D E
do
        echo $i
done
```

Out-Put
A
B
C
D
E

We can give a range of values as {0..9}. bash shell also supports a C like syntax of for loop which is given below:

```
for  (( Expression;  Expression2; Expression))
do
            statements
done
```

While Loop

The syntax of while loop is as given below

```
While (( Condition ))
do
             statements
done
```

Following example shows the working of while loop

```
#!/bin/bash
count=1
while [[ $count -le 5 ]]
do
    echo $count
    count=$((count + 1))
done
```

Out-Put
1
2
3
4
5

Breaking the while loop: bash provides two built-in commands that can be used to control program flow inside loops. The break command immediately terminates a loop, and program control resumes with the next statement following the loop. The continue command causes the remainder of the loop to be skipped, and program control resumes with the next iteration of the loop.

Until Loop

The until command is much like while, except instead of exiting a loop when a nonzero exit status is encountered, it does the opposite. An until loop continues until it receives a zero exit status i.e. the condition becomes true.

```
#!/bin/bash
count=1
until [[ $count -gt 5
]]
do
  echo $count
```

Out-Put
1
2
3
4

count=$((count + 1)) done	**5**

2) Solved Lab Activities

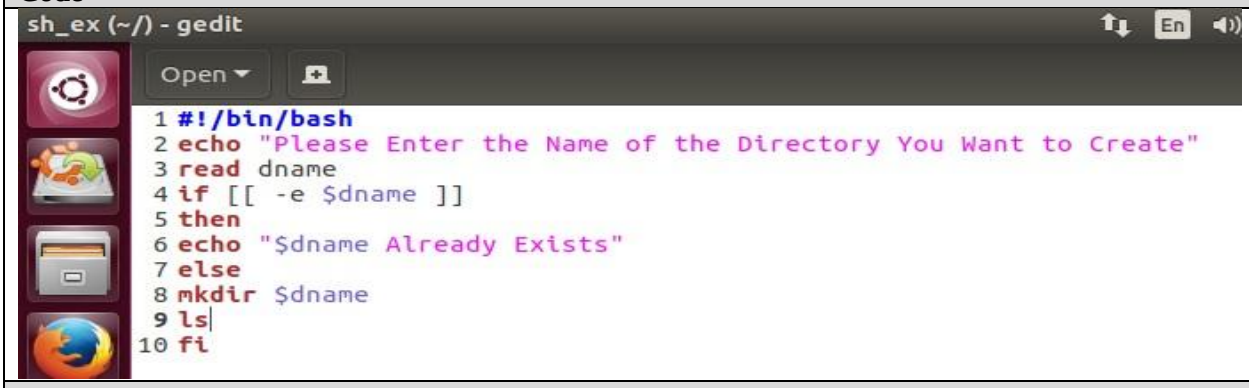
Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*10*	*Low*	*CLO-6*
2	*15*	*Medium*	*CLO-6*
3	*15*	*High*	*CLO-6*

Activity 1:

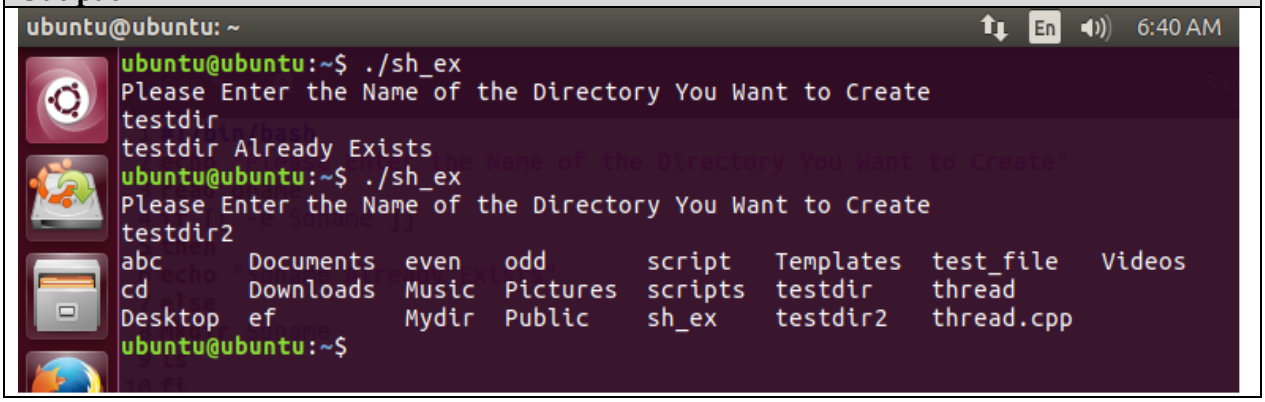
Write a shell script that modifies the mkdir command as: first it takes the directory name from the user as input and checks if a directory with same name exists then shows an error message otherwise creates the directory and show the success message.

Solution:

Code



Out-put



Activity 2:

Write a shell script that takes a word as input and finds whether it is a single alphabet, ABC followed by a digit, of length 3, ends with .txt or it is something else.

Solution:

Code	**Out-put**
#!/bin/bash read -p "enter word > " case $REPLY in [[:alpha:]]) echo "is a single alphabetic character." ;;	**Out-Put** **1** **2** **3**

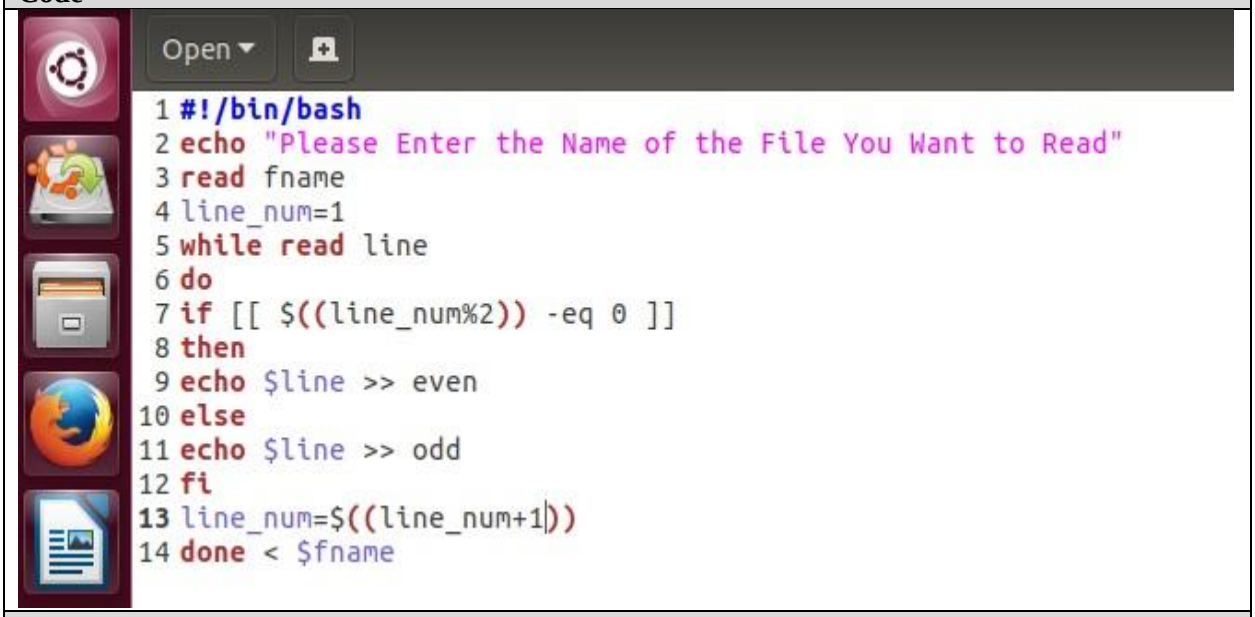
[ABC][0-9]) echo "is A, B, or C followed by a digit." ;; ???) echo "is three characters long." ;; *.txt) echo "is a word ending in '.txt'" ;; *) echo "is something else." ;; esac	**4** **5**

Activity 3:

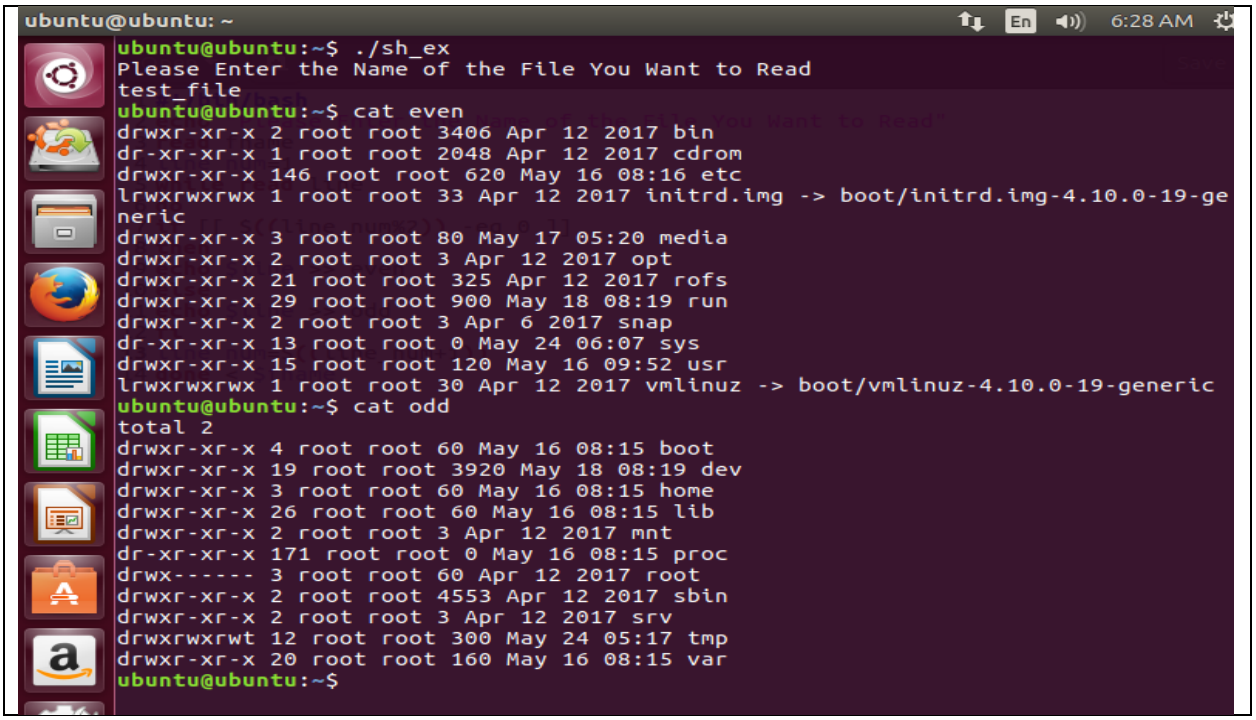
Write a shell script that, given a filename as the argument will write the even numbered line to a file with name even file and odd numbered lines in a text file called odd file.

Solution:

Code



Output



3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write a shell script that asks the user to enter the marks for three subjects and calculates the GPA for each subject along with the CGPA.

Task 2:

Write a menu-driven shell script that gives four options A, B, C and Q to the user to select one of them. If user enters A then it displays the host-name and uptime, if user enters B then it gives information about disk and memory space, if user enter C then it gives information about home space utilization and if user enters Q then it quits the program.

Task 3:

Write a shell script that, given a filename as the argument will count vowels, blank spaces, characters, number of line and symbols.

Lab No. 13
Using Arrays, and Functions in Shell Scripts

Objective
This lab is designed to introduce the usage of arrays and functions in shell scripting.

Activity Outcomes:
On completion of this lab students will be able to:

- Write shell scripts using array
- Write functions

Instructor Notes
As pre-lab activity, read Chapter 35 &31 from the book "The Linux Command Line", William E. Shotts, Jr.

1) Useful Concepts

Arrays
Arrays are variables that hold more than one value at a time. Arrays are organized like a table. Arrays in bash are limited to a single dimension.

Creating an Array

Array variables are named just like other bash variables, and are created automatically when they are accessed. Here is an example:

```
#!/bin/bash
a[1]=5
echo "${a[1]}"
```

Out-put

```
5
```

We can also use the declare command to declare an array. The syntax is given below

```
declare  -a   array_name
```

Adding Values to an Array
New values can be added to an array using the following syntax

```
array_name[index]=value
```

To add multiple values, we use the following syntax

```
array_name=(value1  value2 …. )
```

These values are assigned sequentially to elements of the array, starting with element zero. It is also possible to assign values to a specific element by specifying a subscript for each value:

```
array_name=([index]=value1  [index]=value2 …. )
```

Accessing Array Elements

Array elements can be accessed as follows

```
array_name=([index]=value1  [index]=value2 …. )
```

Operations on Arrays

Out-putting Entire Array: by using * or @ as index, we can output an entire array.

```
#!/bin/bash
animals=("a dog" "a cat" "a fish")
for i in ${animals[*]}
do echo $i;
done
```

Out-put
a dog
a cat
a fish

Note: we can also use ${animals[@]} instead of ${animals[*]}. If we use “” marks i.e. “${animals[@]}” then contents are displayed on single line

Determining the Number of Array Elements: we can find the total number of elements in an array by using following

```
${#array_name[@]}
```

While the length of an element can be found as

```
${#array_name[index]}
```

The following example shows the usage of these

```
#!/bin/bash
a[100]=foo
echo ${#a[@]}       # number of array elements
echo ${#a[100]}     # length of element 100
```

Out-put
1
3

Finding the Index Used by an Array: As bash allows arrays to contain “gaps” in the assignment of subscripts, it is sometimes useful to determine which elements actually exist. This can be done with a parameter expansion using the following forms:

```
${!array_name[@]}  or  ${#array_name[*]}
```

The following example shows the usage of this

```
#!/bin/bash
foo=([2]=a [4]=b [6]=c)
for i in "${!foo[@]}"
do
    echo $i
done
```

Out-put
2
4
6

Adding Elements to the End of an Array:

```
#!/bin/bash
foo=(a b c)
echo ${foo[@]}
 foo+=(d e f)
echo ${foo[@]}
```

Out-put
a b c

a b c d e f

Sorting an Array:

```
#!/bin/bash
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[@]}"
```

Out-put
Original array: f e d c b a
Sorted array: a b c d e f

Deleting an Array:

```
#!/bin/bash
foo=(a b c d e f)
echo ${foo[@]}
unset foo
echo ${foo[@]}
```

Note: to delete a specific index, we can use unset 'foo[index]'

Out-put
a b c d e f

Writing Functions

A Bash function is essentially a set of commands that can be called multiple times. The purpose of a function is to help you make your bash scripts more readable and to avoid writing the same code repeatedly. Compared to most programming languages, Bash functions are somewhat limited.

The syntax for declaring a bash function is straightforward. Functions may be declared in two different formats:

```
function-name( ){
                Commands
                 }
Or
function function-name( ){
                        Commands
                       }
```

Functions can be called by name.

```
#!/bin/bash
hello_world () {
   echo 'hello, world'
}
hello_world
```

Out-put
hello world

We can define local variables within the function using the **local** keyword. To return a value, we can use return statement. Following example shows the use of return command.

```
#!/bin/bash
my_function () {
  echo "hello world"
  return 55
}

my_function
```

Out-put
hello world
55

`echo $?`	

Arguments can be passed to functions in the following way.

`#!/bin/bash`
`greeting ()`
`{`
`  echo "Hello $1"`
`}`
 	
`greeting "Ali"`
	Out-put
	Hello Ali

2) Solved Lab Activities

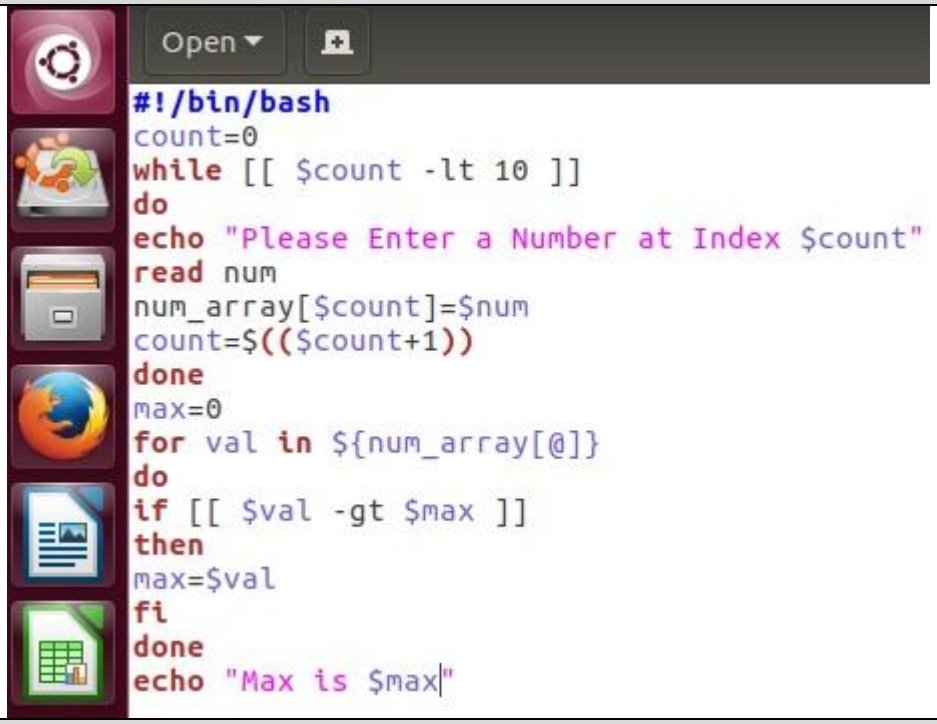
Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	***20***	***Medium***	***CLO-6***
2	***25***	***Medium***	***CLO-6***
3	***25***	***Medium***	***CLO-6***

Activity 1:

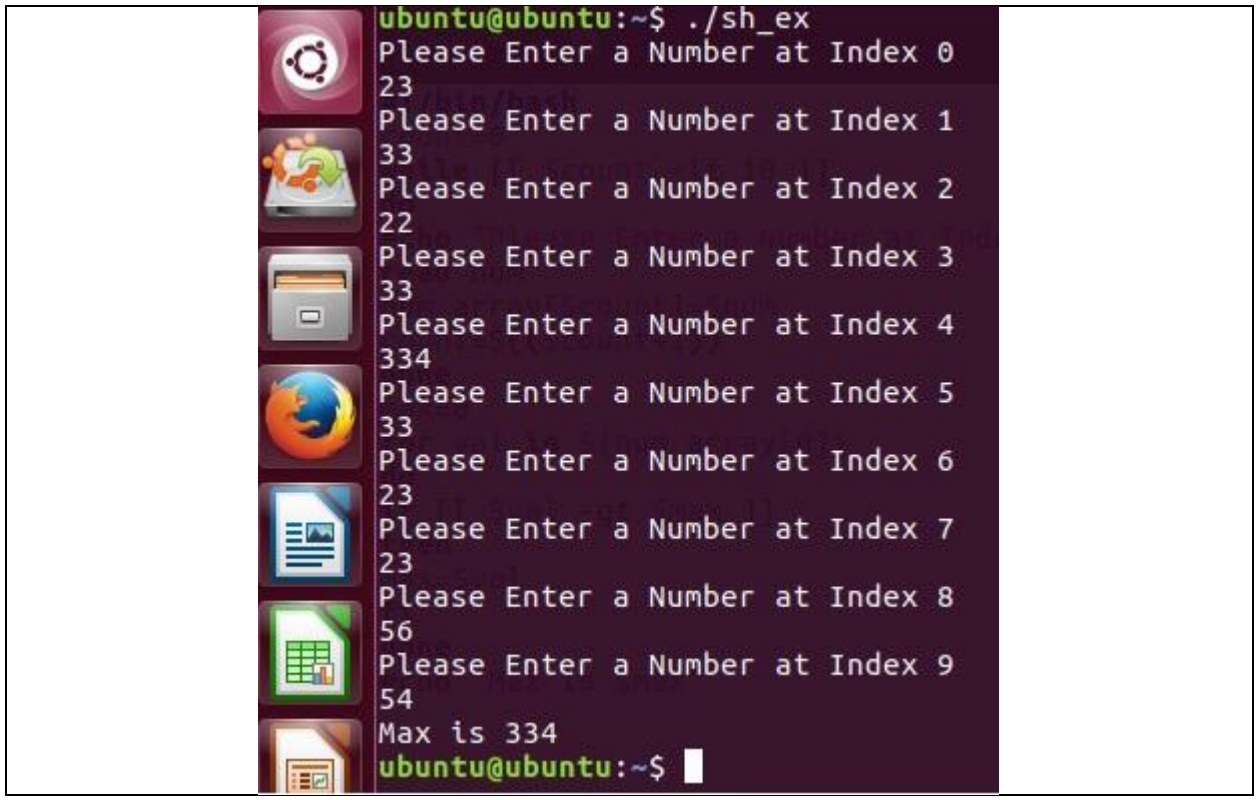
Write a shell script that accepts 10 numbers from user as input and finds: maximum, minimum, odd/even of them.

Solution:

Code



Out-put


ubuntu@ubuntu:~$ ./sh_ex
Please Enter a Number at Index 0
23
Please Enter a Number at Index 1
33
Please Enter a Number at Index 2
22
Please Enter a Number at Index 3
33
Please Enter a Number at Index 4
334
Please Enter a Number at Index 5
33
Please Enter a Number at Index 6
23
Please Enter a Number at Index 7
23
Please Enter a Number at Index 8
56
Please Enter a Number at Index 9
54
Max is 334
ubuntu@ubuntu:~$

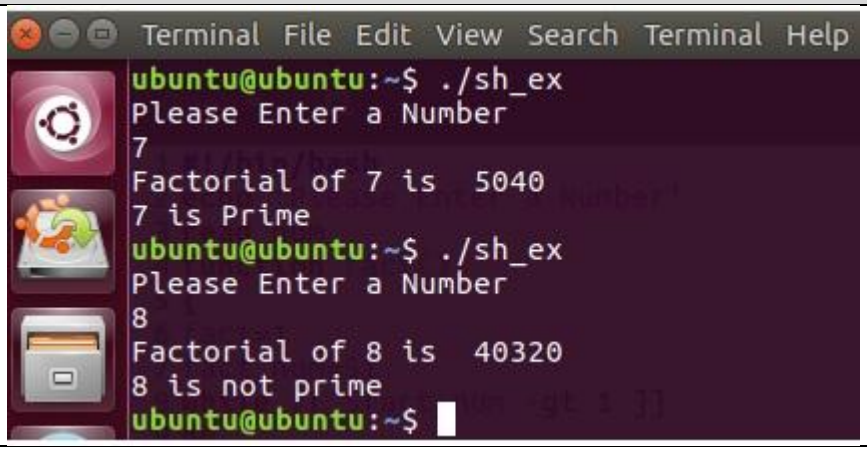
Activity 2:

Write a shell script that contains two function fact( ) and is_prime( ). fact( ) function finds the factorial of a number while is_prime( ) checks whether a number is prime or not. Your script should take an integer as input from user and pass this number to fact( ) and is_prime () functions as argument.

Solution:

Code

Out-put

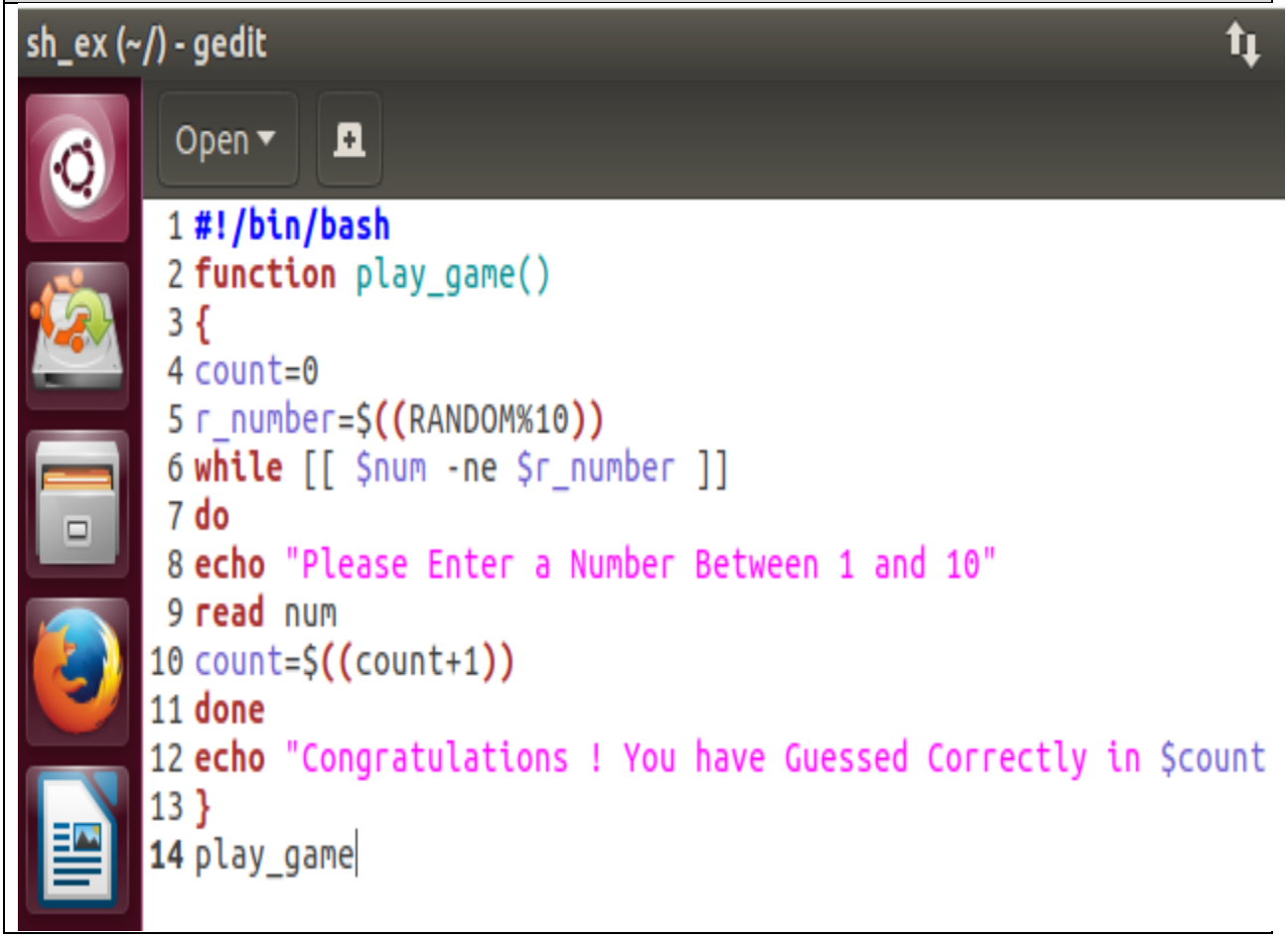


Activity 3:

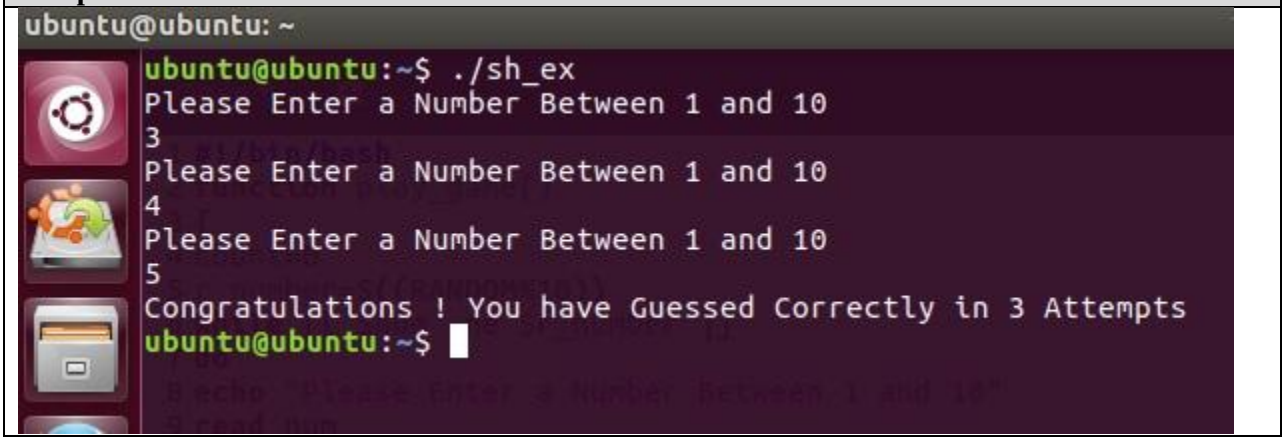
Write a shell script that contains a function play_game. This function generates a random number between 1 and 10; and keeps on asking the user to guess the number as long as user enters a number which is equal to the random number. In the end, the total number of attempts made by the user to enter correct guess is displayed.

Soltuion:

Code



Out-put



3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write a function that finds the sum of digits in an integer

Task 2:

Write a function that finds whether an integer is a palindrome or not

Task 3:

Write a function that writes an integer in reverse (for example writes 123 as 321)