OS Lab Manual

Operating Systems (COMSATS University Islamabad)

LAB MANUAL

Course: CSC322-Operating Systems

Department of Computer Science

Java Learning Procedure

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

Table of Contents

LAB # 02

Statement Purpose:

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

Activity Outcomes:

This lab teaches you the following topics:

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

Instructor Note:

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

1) Stage J (Journey)

Introduction

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

1. Navigation

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

1.1 Print Working Directory

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

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

1.2 Listing The Contents Of A Directory

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

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

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

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

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

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

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

The following options can also be used with ls command

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

1.3 Changing the Current Working Directory

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

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

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

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

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

2. Working With Directories

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

2.1. Creating a Directory

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

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

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

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

2.2. Copying Files and Directories

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

```
cp  item1  item2
```

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

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

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

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

2.3. Moving and Renaming Files and Directories

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

```
mv  item1  item2
```

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

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

Common options, used with mv command are:

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

2.4. Removing and Files and Directories

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

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

The common options, used with rm command are:

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

3. Working with Files

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

3.1 Creating and Empty Text File

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

```
touch  myfile.txt
```

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

```
cat  > myfile.txt
```

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

```
gedit myfile.txt
```

3.2 Reading the File Contents

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

```
cat  myfile.txt
```

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

```
less   myfile.txt
```

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

```
gedit   myfile.txt
```

3.3 Appending text files

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

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

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

3.4 Combining multiple text files

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

```
cat  file1  file2   file3
```

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

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

3.5 Determining File Type

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

```
file  filename
```

4. Redirecting I/O

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

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

4.1 Redirecting Standard Output

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

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

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

4.2 Redirecting Standard input

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

4.3 Pipelines

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

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

2) Stage a1 (apply)

Lab Activities:

Activity 1:

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

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

Solution:

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

Activity 2:

Perform the following tasks using Linux CLI

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

Solution:

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

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

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

4. **mkdir mydir3**
   cd mydir3

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

Activity 3:

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

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

4. View the contents of myfile

Solution:

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

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

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

4. **cat myfile**

Activity 4:

Considering the above activities, perform the following tasks

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

Solution:

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

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

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

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

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

Activity 5:

This activity is related to I/O redirection

1. Go to Desktop directory

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

Solution:

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

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

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

3) Stage V (verify)

Home Activities:

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

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

4) Stage a_2 (assess)

Lab Assignment and Viva voce

Operating Systems (LAB 3)

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

Activity Outcomes:

This lab teaches you the following topics:

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

---

1. File Permissions

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

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

1.1 id command

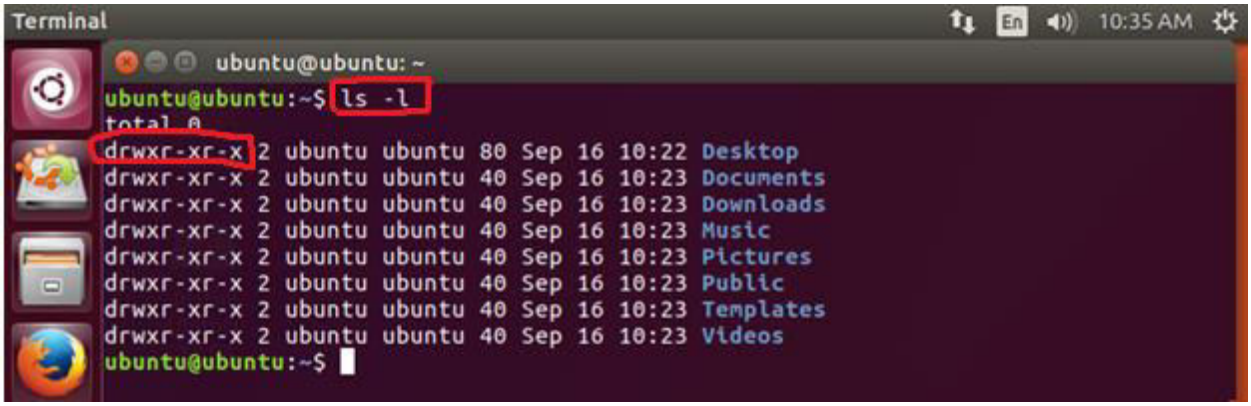
In the Linux security model, a user may own files and directories. When a user owns a file or directory, the user has control over its access. Users can, in turn, belong to a group consisting of one or more users who are given access to files and directories by their owners. In addition to granting access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

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

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

1.2 Reading, Writing, and Executing

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



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

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

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

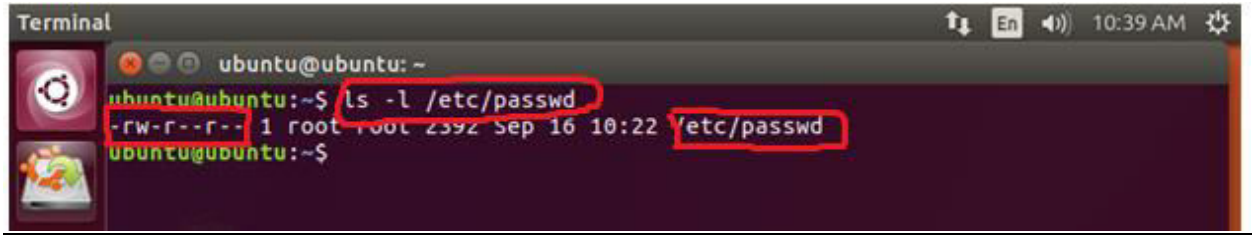
User	Group	World
rwx	rwx	rwx

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

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

1.3 Reading File Permissions

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



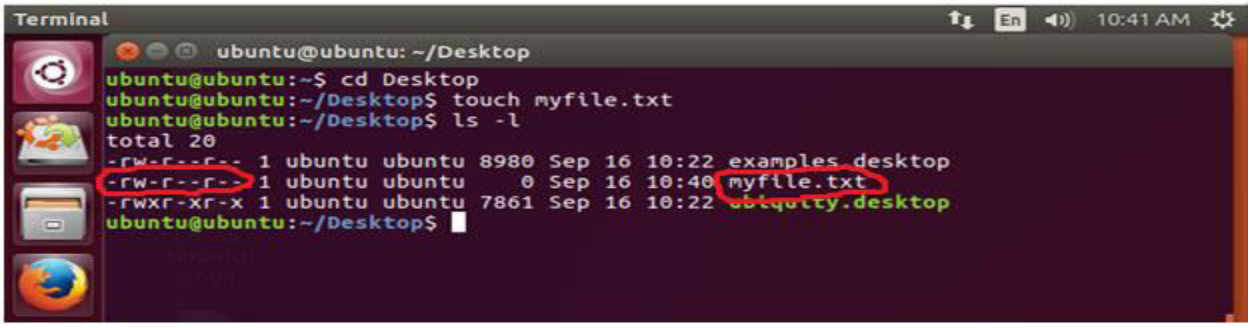
1.4 Change File Mode (Permissions)

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

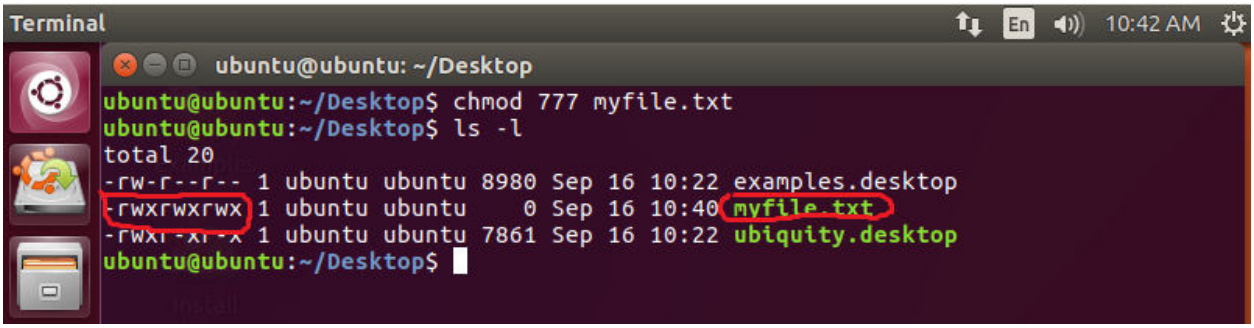
Octal	Binary	File Mode
0	000	---
1	001	--x
2	010	-w-

3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

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



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



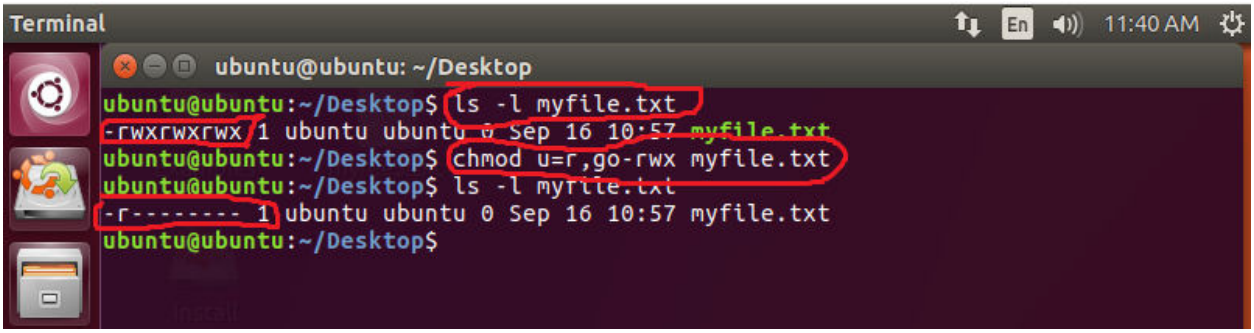
chmod also supports a symbolic notation for specifying file modes. Symbolic notation is divided into three parts: who the change will affect, which operation will be performed, and what permission will be set. To specify who is affected, a combination of the characters "u", "g", "o", and "a" is used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

Character	Meaning
u	Owner
g	Group
o	Others
a	all

If no character is specified, "all" will be assumed. The operation may be a "+" indicating that a permission is to be added, a "-" indicating that a permission is to be taken away, or a "=" indicating that only the specified permissions are to be applied and that all others are to be removed. For example:

u+x,go=rx Add execute permission for the owner and set the permissions for the group and others to read and execute. Multiple specifications may be separated by commas.

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

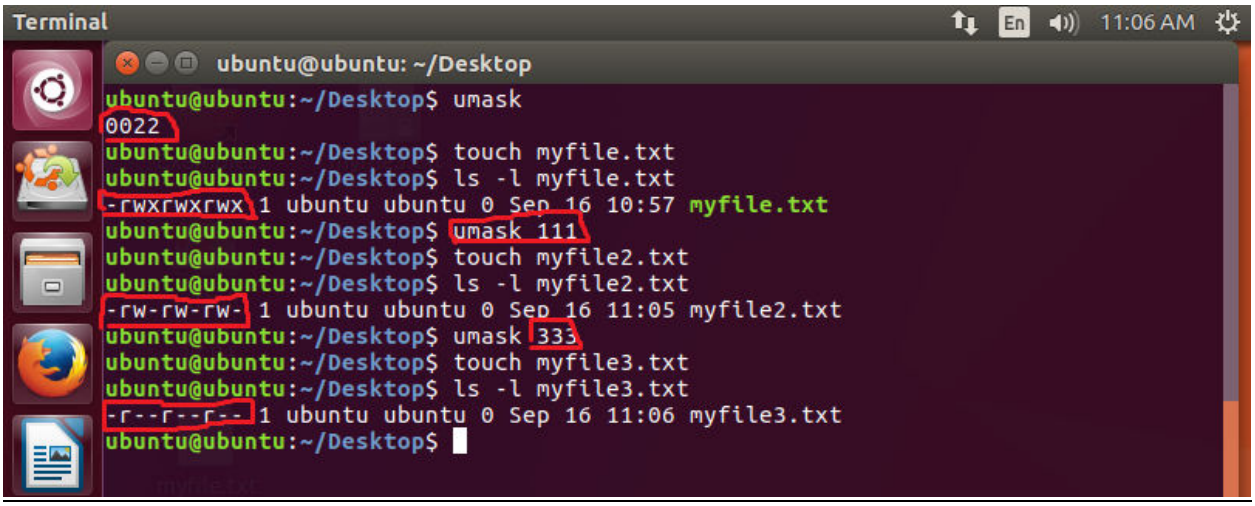


1.5 Controlling the Default Permissions

On Unix-like operating systems, the **umask** command returns, or sets, the value of the system's file mode creation mask. When user create a file or directory under Linux or UNIX, he/she creates it with a default set of permissions. In most case the system defaults may be open or relaxed for file sharing purpose.
umask command with no arguments can be used to return the current mask value. Similarly, If the **umask** command is invoked with **an octal argument**, it will directly set the bits of the mask to that argument.
The three rightmost octal digits address the "owner", "group" and "other" user classes respectively. If fewer than 4 digits are entered, leading zeros are assumed. An error will result if the argument is not a valid octal number or if it has more than 4 digits. If a fourth digit is present, the leftmost (high-order) digit addresses three additional attributes, the setuid bit, the setgid bit and the sticky bit.

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

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



1.6 Changing User Identity

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

Run A Shell with Substitute User and Group IDs

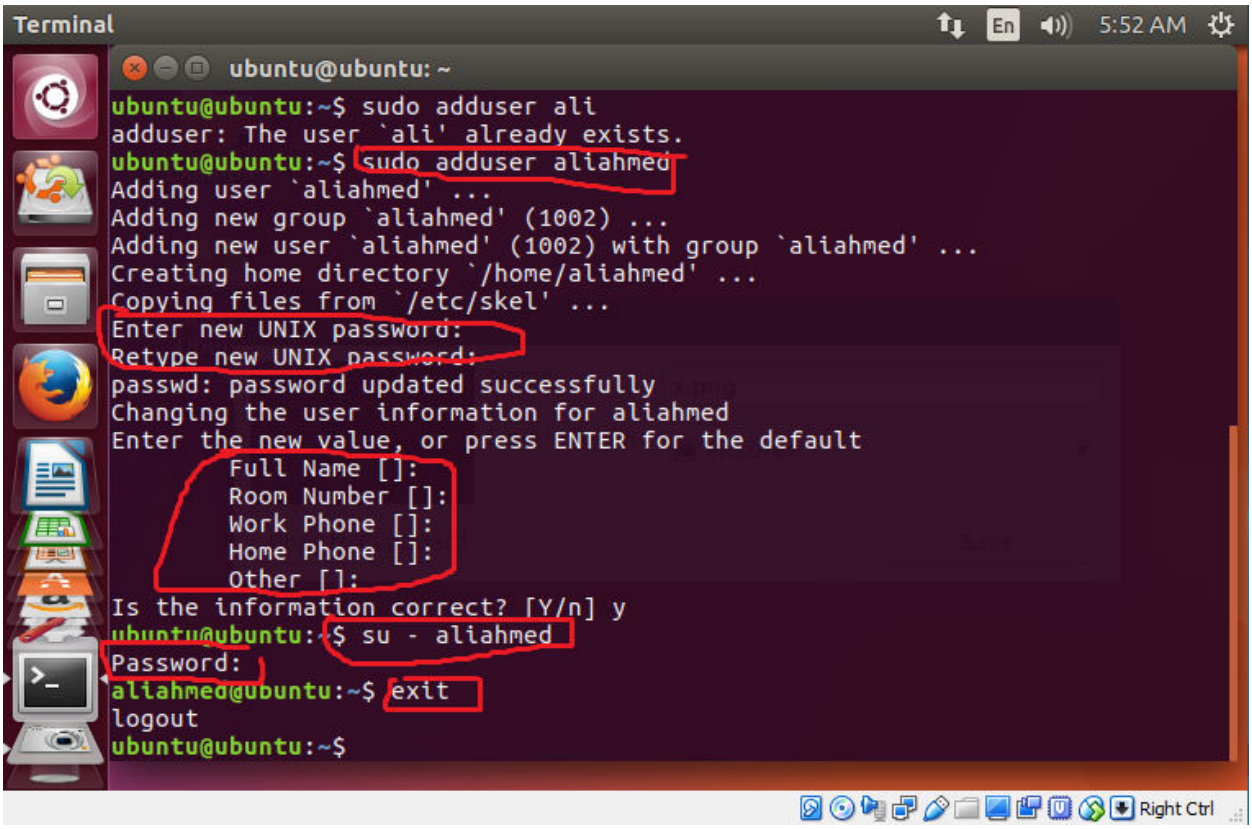
The su command is used to start a shell as another user. The command syntax looks like this:

su -l *username*

Execute A Command as Another User

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

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

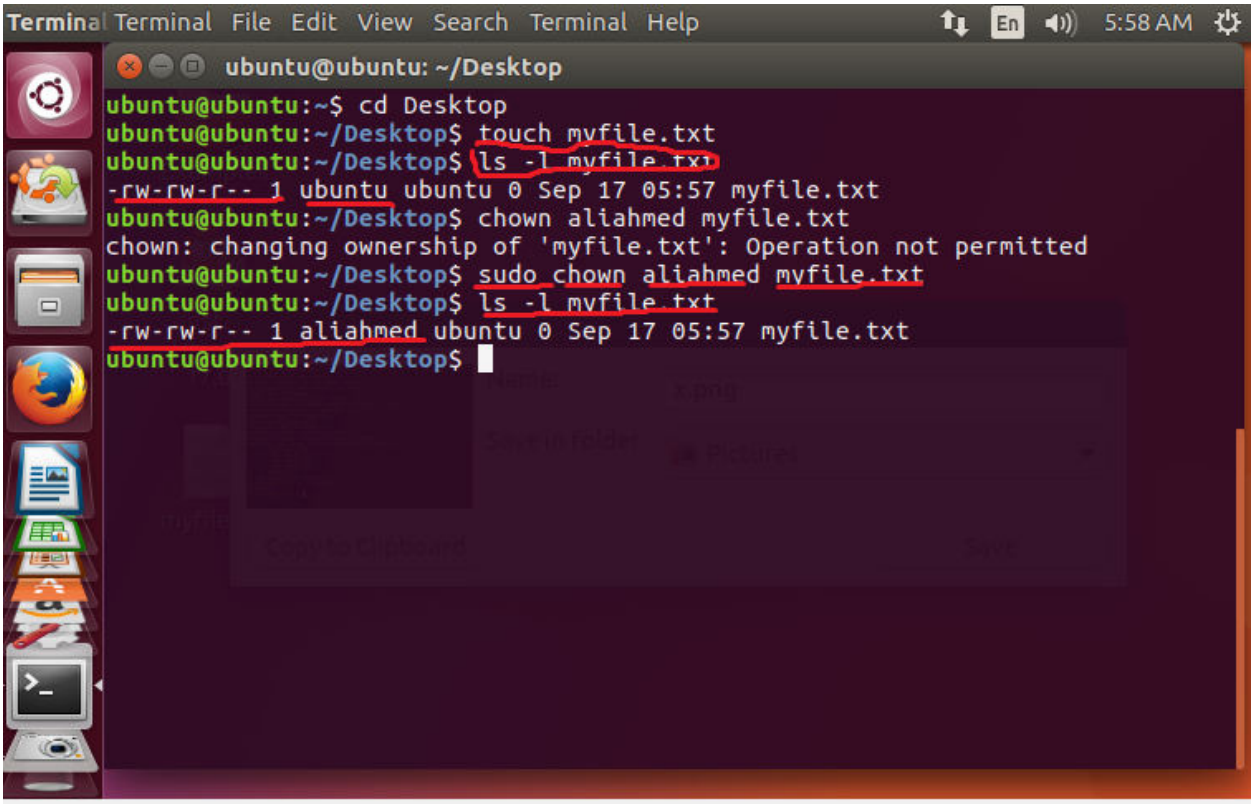


1.7 Change File Owner and Group

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

chown owner:group file/files

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



2. Package Management

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

Package Files

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

Repositories

While some software projects choose to perform their own packaging and distribution, most packages today are created by the distribution vendors and interested third parties. Packages are made available to the users of a distribution in central repositories that may contain many thousands of packages, each specially built and maintained for the distribution.

Dependencies

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

High and Low-level Package Tools

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

Common Package Management Tasks

2.1 Finding a Package in a Repository

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

apt-get update
apt-cache search *search_string*

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

apt -get update
apt-get search emacs

2.2 Installing a Package from a Repository

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

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

apt-get update; apt-get install emacs

2.3 Installing a Package from a Package File

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

dpkg-install *package_file*

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

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

2.4 Removing A Package

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

apt -get remove *package_name*

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

apt -get remove emacs

2.5 Updating Packages from a Repository

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

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

apt -get update; apt-get upgrade

2.6 Upgrading a Package from a Package File

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

dpkg --install *package_file*

2.7 Listing Installed Packages

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

dpkg --list

2.8 Determining If A Package Is Installed

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

dpkg --status *package_name*

Example:
dpkg --status emacs

2.9 Displaying Information About an Installed Package

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

apt -cache show *package_name*

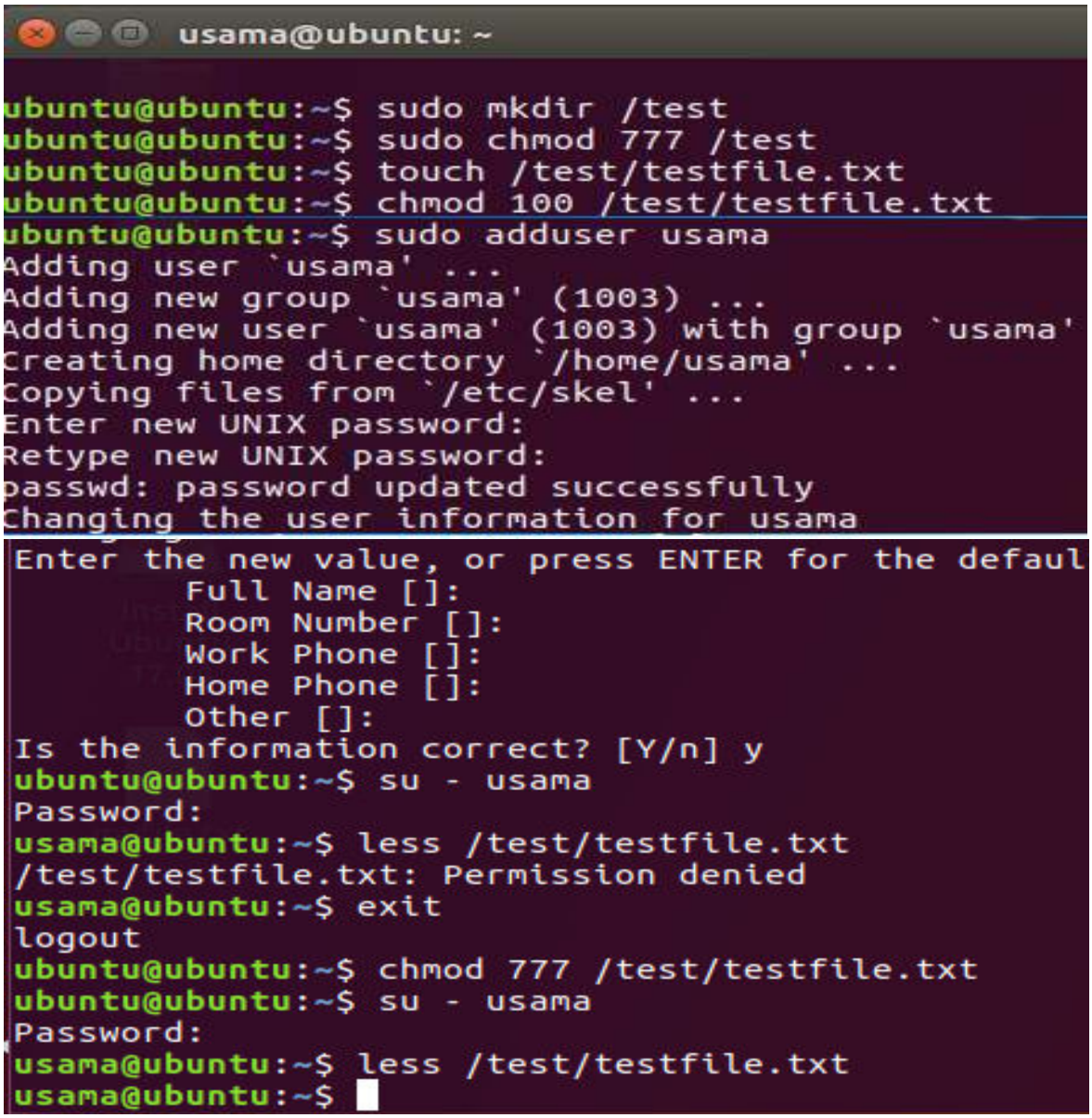
Example:
apt -cache show emacs

Lab Activities

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

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

Solution:



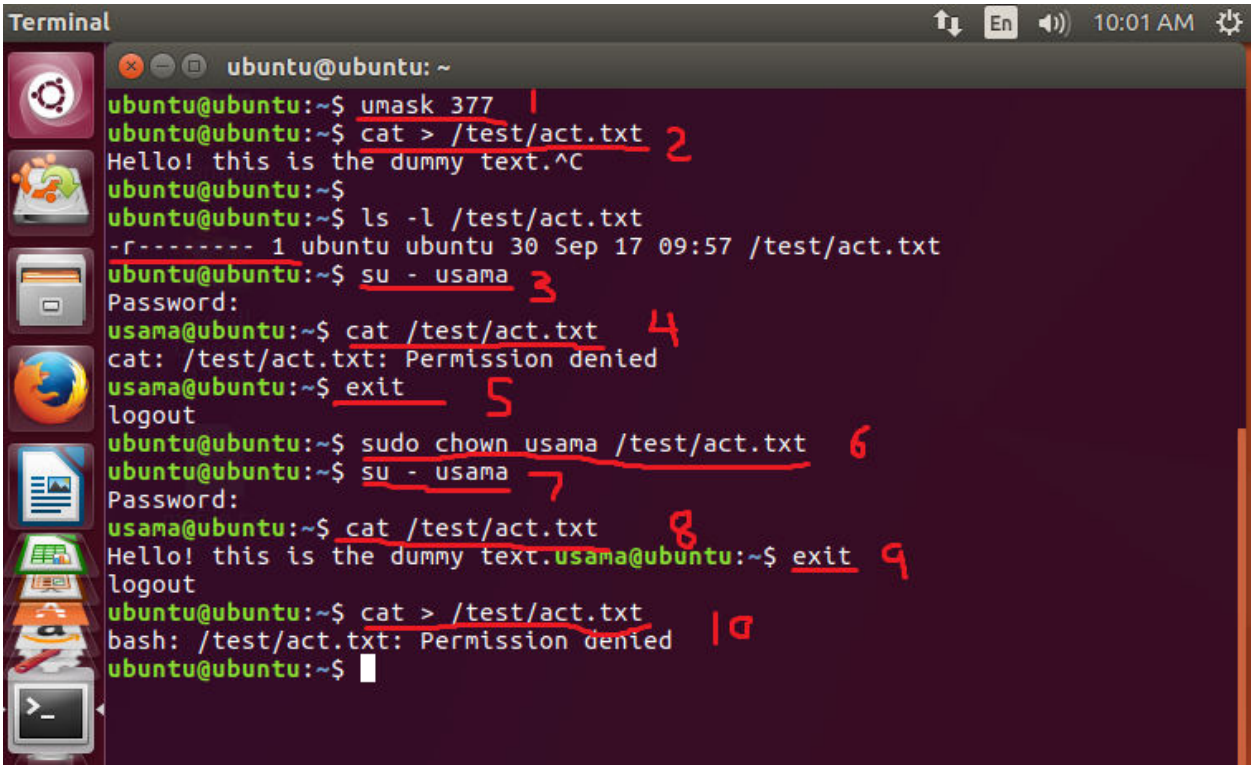
Activity 2:

Perform the following tasks

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

10. Now access the act.txt with user ubuntu

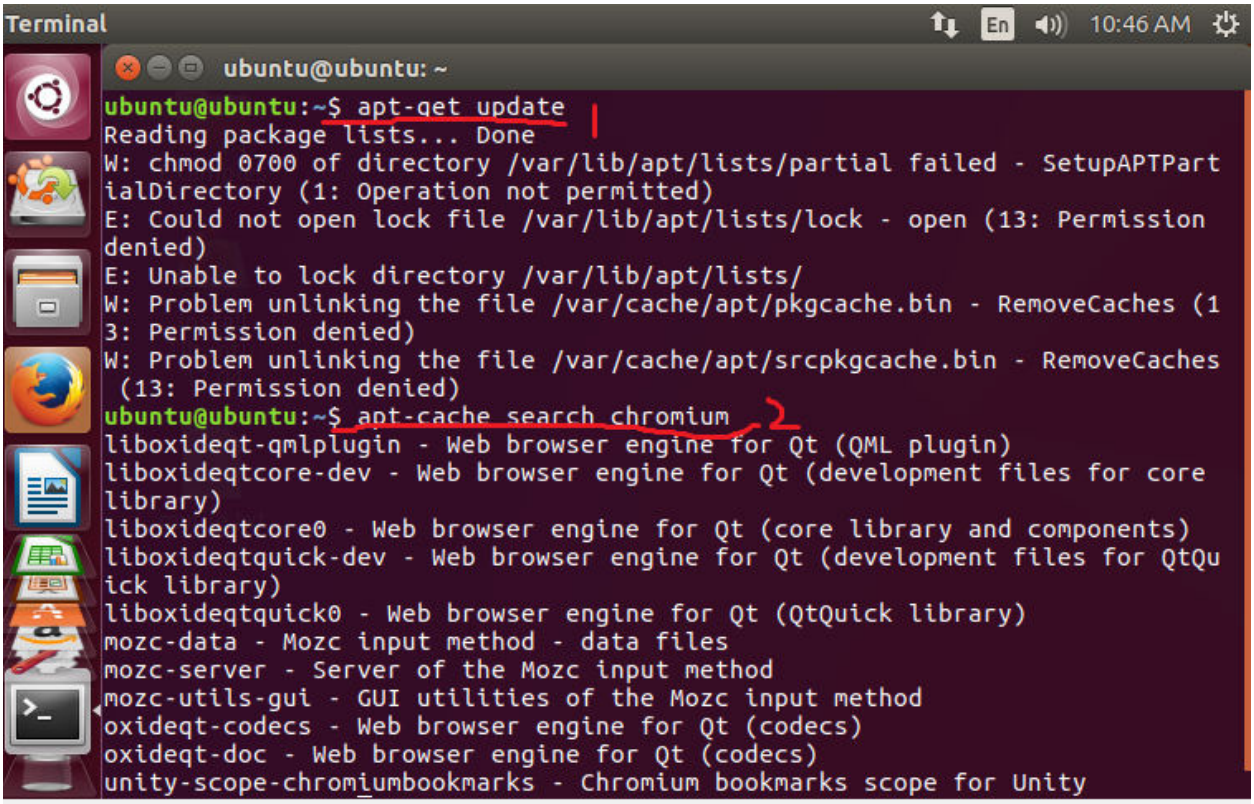
Solution:

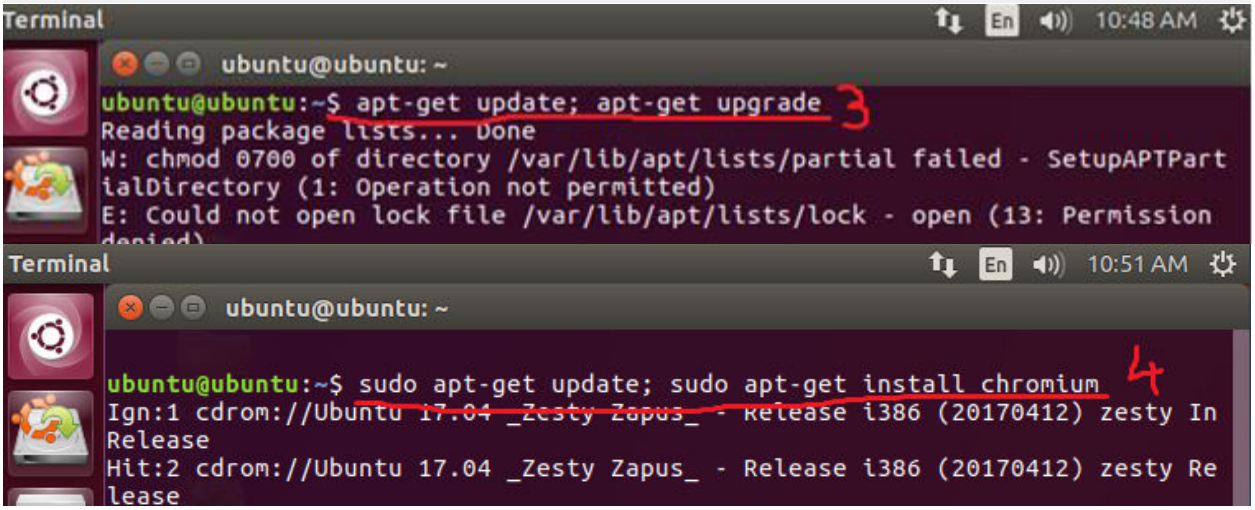


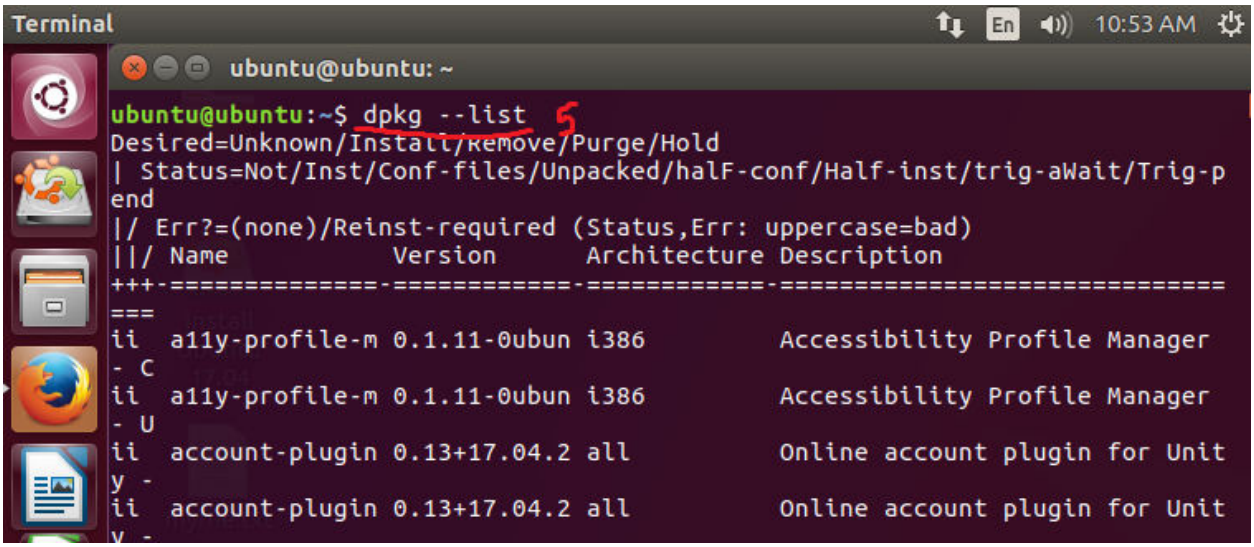
Activity 3:

Perform the following tasks

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







Operating Systems (LAB 4)

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

Activity Outcomes:

This lab teaches you the following topics:

1. An introduction to some of the most useful text-processing utilities
2. Using Linux's graphical and text-based configuration tools to manage networking, printing and date/time settings

---

Viewing File Contents With less

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

Why would we want to examine text files?

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

The less command is used like this:

```
$less <filename>
```

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

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

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

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

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

wc – Print Line, Word, and Byte Counts

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

```
$wc Myfile.txt
```

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

Pipelines

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

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

Filters

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

Sort

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

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

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

grep – Print Lines Matching A Pattern

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

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

When grep encounters a "pattern" in the file, it prints out the lines containing it. The patterns that grep can match can be very complex, but for now we will concentrate on simple text matches.
For example, find all the files in our list of programs that had the word "zip" embedded in the name:

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

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

head / tail – Print First / Last Part of Files

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

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

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

These can be used in pipelines as well:

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

Networking

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

Examining and Monitoring A Network

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

```
$ping localhost
```

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

Traceroute

The traceroute program (some systems use the similar tracepath program instead) displays a listing of all the "hops" network traffic takes to get from the local system to a specified host. For example, to see the route taken to reach specified host, we would do this:

$ **traceroute <specified host>**

netstat

The netstat program is used to examine various network settings and statistics. Through the use of its many options, we can look at a variety of features in our network setup. Using the "-ie" option, we can examine the network interfaces in our system:

$ **netstat -ie**

Using the "-r" option will display the kernel's network routing table. This shows how the network is configured to send packets from network to network:

$ **netstat -r**

Lab Activities

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

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

Solution:

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

Activity 2:
Perform the following tasks

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

Solution

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

Activity 3:
Perform the following tasks

5. Examine the network using Ping command and view the performance statistics
6. displays a listing of all the “hops” network traffic takes to get from the local system to yahoo.com
7. examine various network settings and statistics through netstat

Solution

1. ping localhost
2. traceroute yahoo.com
3. netstat -r

Operating Systems (LAB 5)

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

Activity Outcomes:

This lab teaches you the following topics:

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

---

1. Process management in Linux

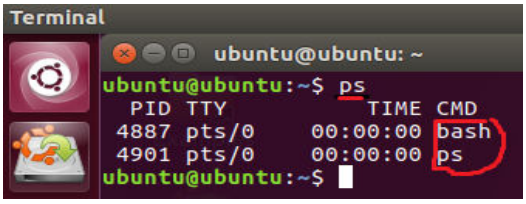
A process is the instance of a computer program that is being executed. While a computer program is a passive collection of instructions, a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the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

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

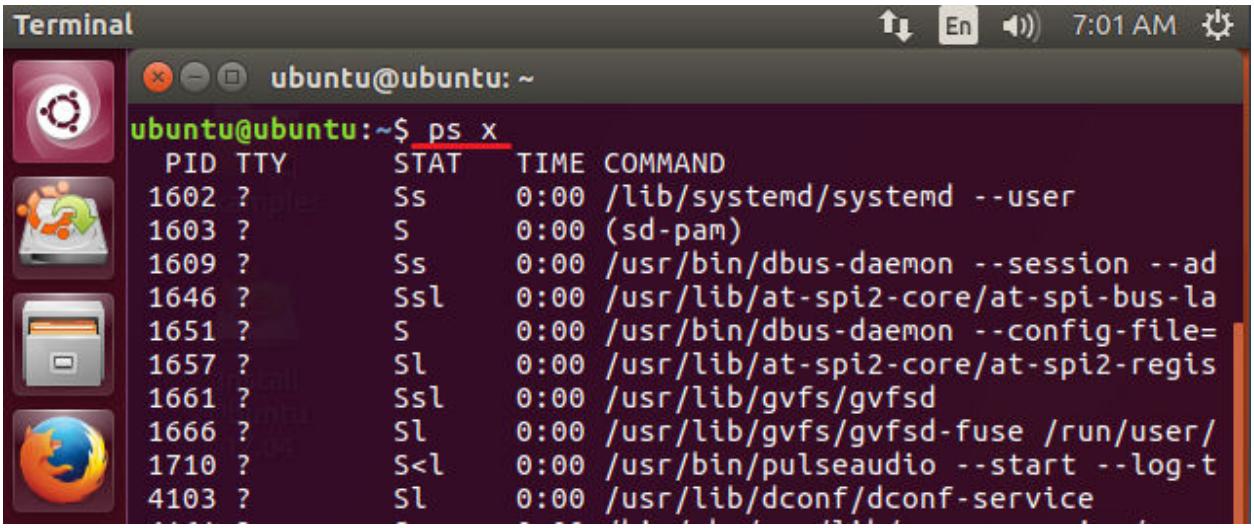
1.1 Displaying Processes in the System

1.1.1 Static view

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



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



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

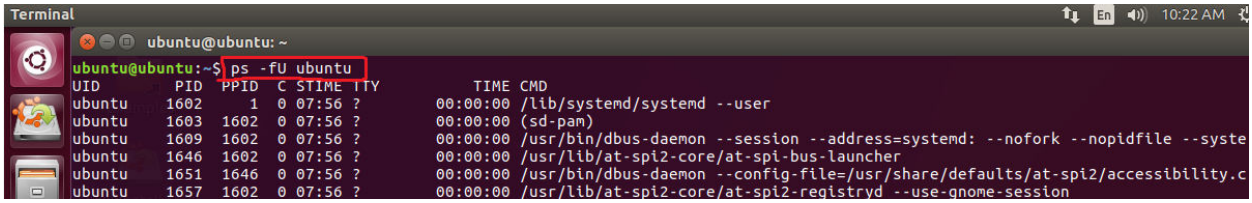
State	Meaning
R	Running
S	Sleeping or waiting for an event such as keystroke
D	Uninterruptible Sleep. Process is waiting for I/O such as a disk drive
T	Stopped
Z	A difunctional or zombie process
l	multithreaded
s	Session leader
+	Foreground process
<	A high priority process
N	A low priority process

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

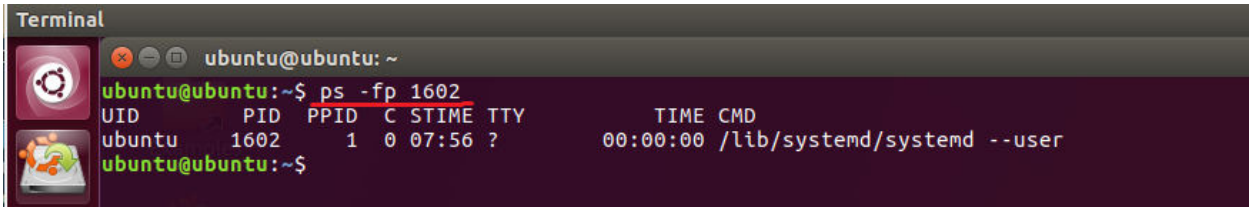
Option	Description
-A or -e	Display every active process on a Linux system
-x	Display all of the processes owned by the user
-F	Perform a full-format listing
-U	Select by real user ID or name

-u	Select by effective user ID or name
-p	Select process by pid
-r	Display only running processes
-L	Show number of threads in a process
-G	Show processes by Group

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



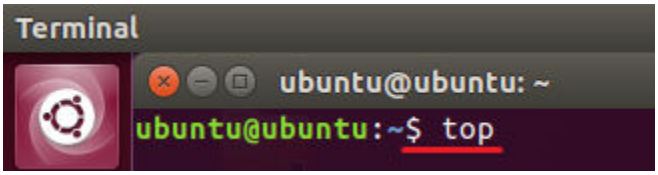
Now, we select a process with id 1602



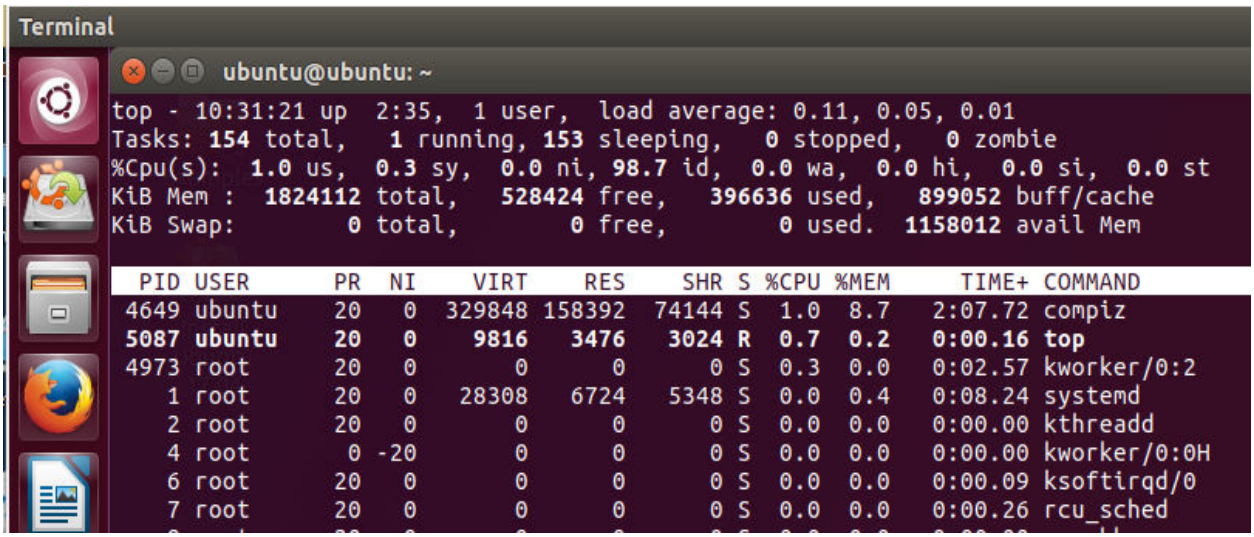
1.1.2 Dynamic view

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

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



The output of the above command is given below

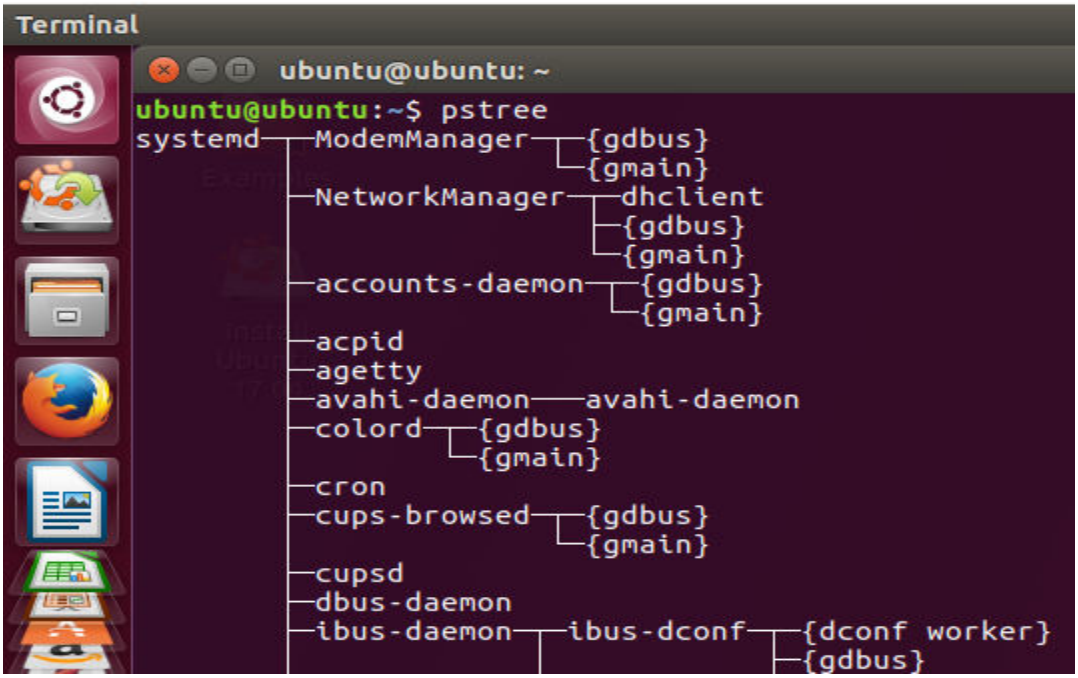


Some of the basic options available with top commands are given below

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

1.1.3 Displaying processes in Treelike structure

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

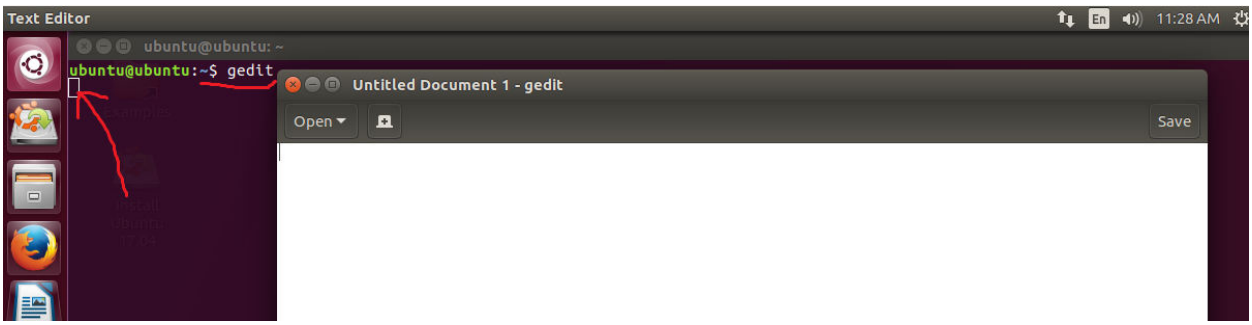


1.2 Interrupting A Process

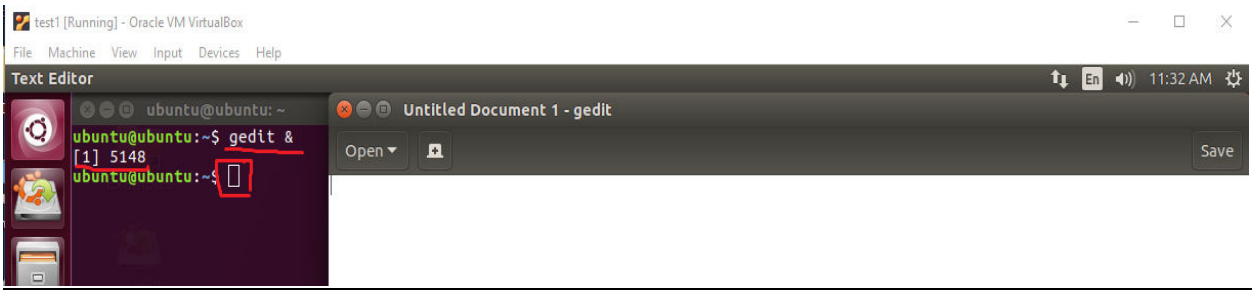
A program can be interrupted by pressing Ctrl + C. This will interrupt the given processes and stop the process. Ctrl+C essentially sends a SIGINT signal from the controlling terminal to the process, causing it to be killed.

1.3 Putting a Process in the Background

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



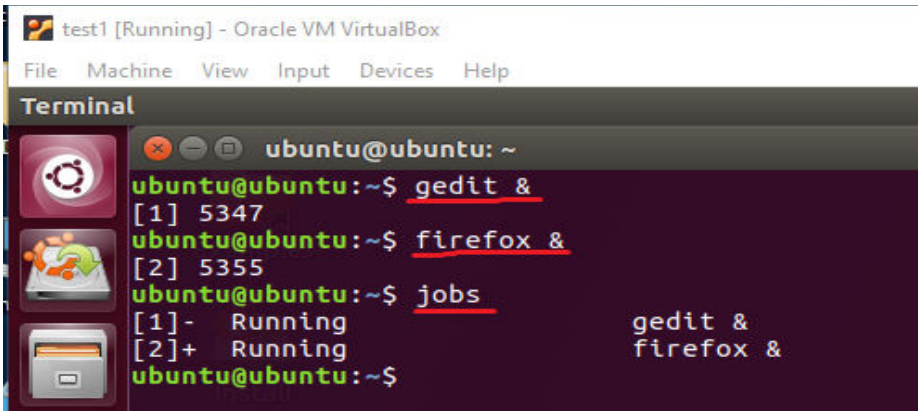
It can be noted that gedit is opened as foreground process and control does not returns to terminal unless it is closed. Now, we start the gedit again as a background process.



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

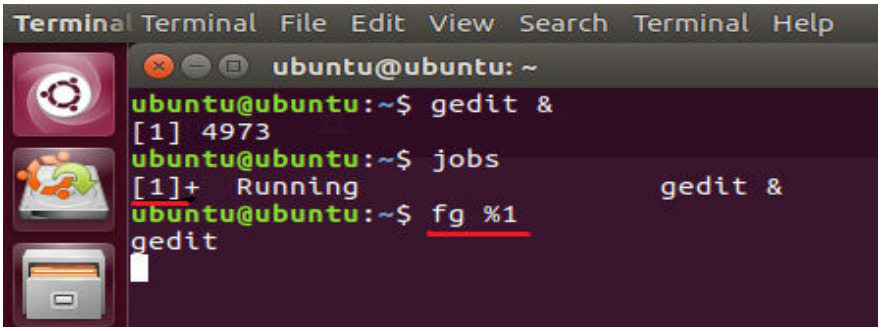
1.4 jobs command

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



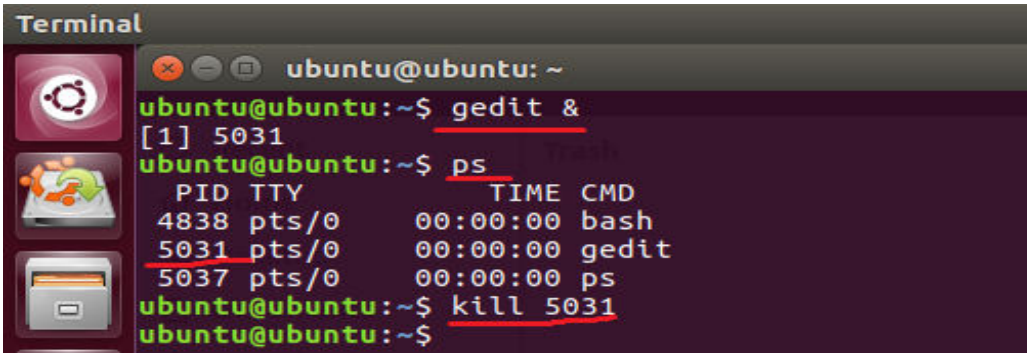
1.5 Bringing a process to the foreground

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



1.6 Killing a process

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



The kill command doesn't exactly “kill” processes, rather it sends them *signals*. Signals are one of several ways that the operating system communicates with programs. Programs, in turn,

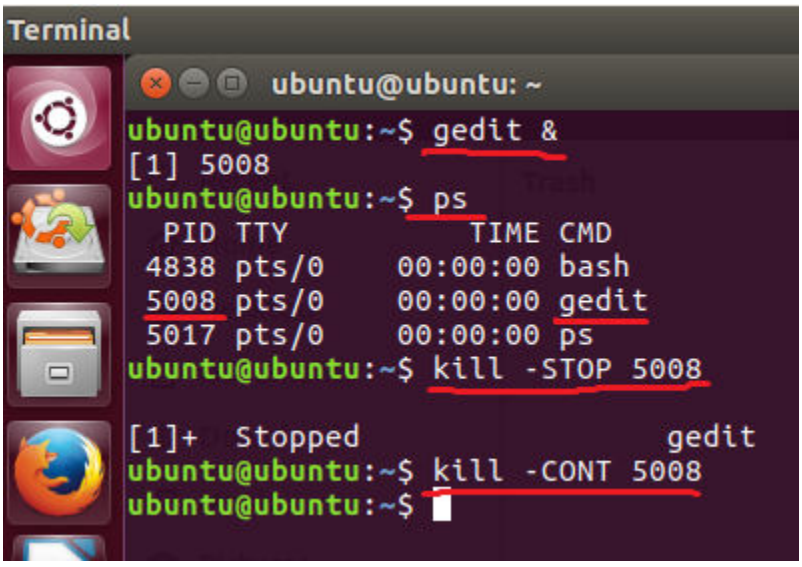
“listen” for signals and may act upon them as they are received. Following are most common signals that can be send with kill command.

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

1.7 Pausing a process

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

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

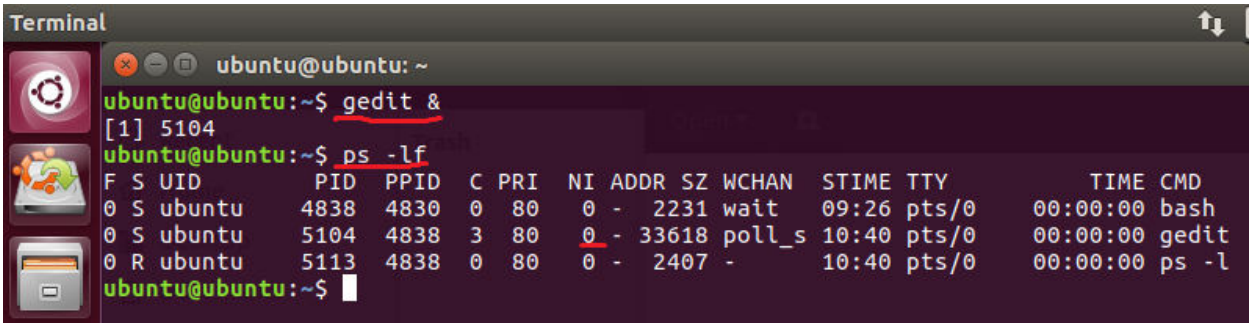


1.8 Changing process priority

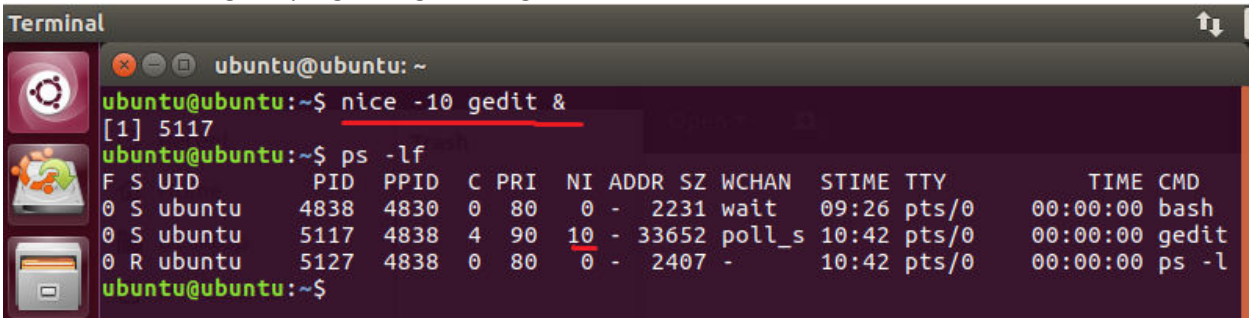
Every running process in Linux has a priority assigned to it. We can change the process priority using nice and renice utility. Nice command will launch a process with a user defined scheduling priority. Renice command will modify the scheduling priority of a running process. In Linux system priorities are 0 to 139

in which 0 to 99 for real time and 100 to 139 for users. nice value range is -20 to +19 where -20 is highest, 0 default and +19 is lowest. Relation between nice value and priority is:
PR = 20 + NI
So, the value of PR = 20 + (-20 to +19) is 0 to 39 that maps to 100-139.

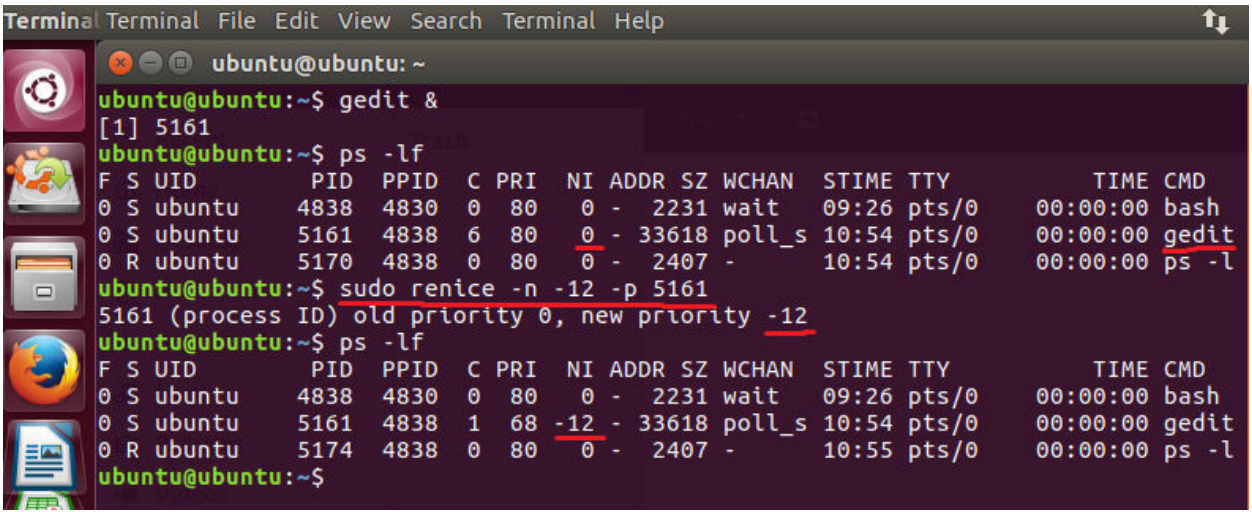
In the following example, we start the gedit program in background and see its nice value using ps command.



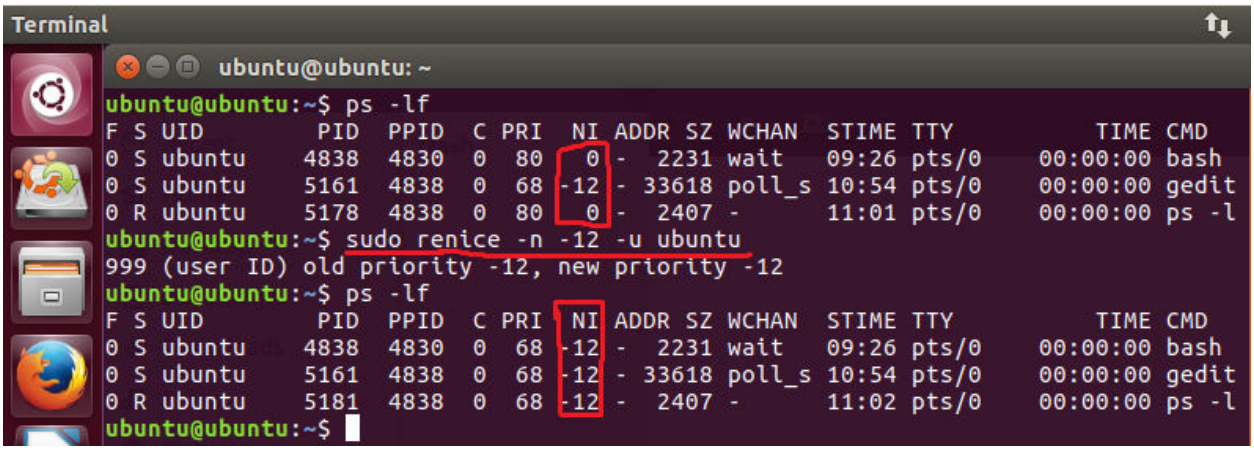
Now, we start the gedit program again using nice command.



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



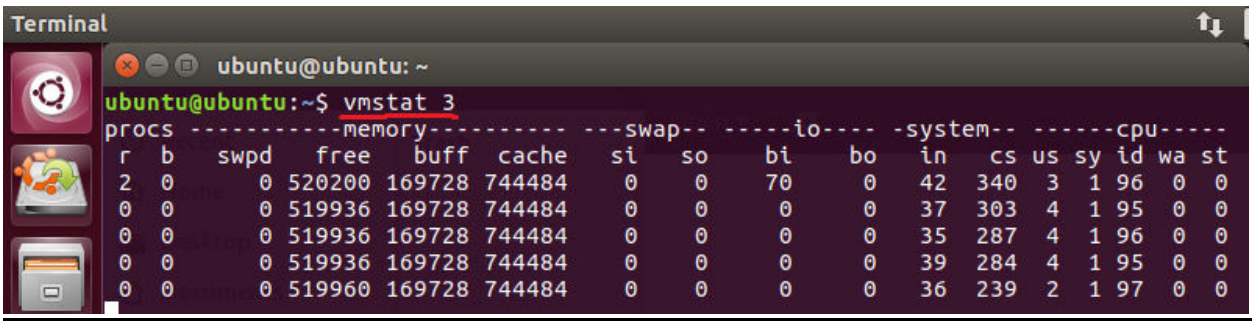
We can also use the renice command to change the priority of all of the processes belonging to a user or a group. In the following example, we change the nice value of all of the process belonging to user ubuntu.



1.9 Few more useful commands related to processes

1.9.1 vmstat command

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



1.9.2 xload command

Displays the system load over time.

1.9.3 tload command

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

2. Compiling and Executing C++ Programs

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

mnemonics, for example, CPY (for duplicate) and MOV (for move). Programs written in assembly language are processed into machine language by a program called an assembler.

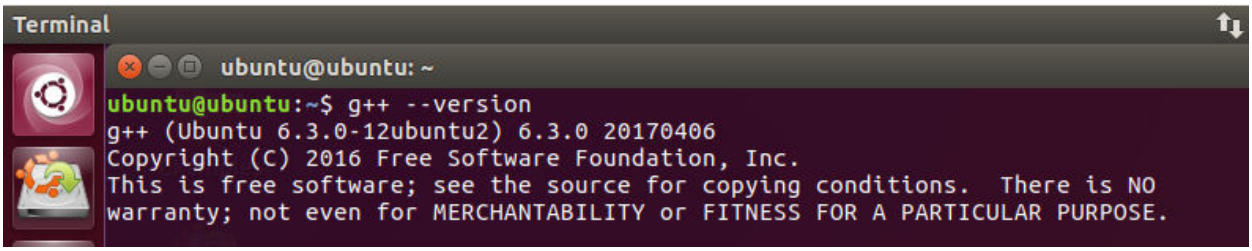
We next come to what are called high-level languages. They are called this since they enable the developer to be less worried about the details of what the processor is doing and more with taking care of the current issue. Programs written in high-level programming languages are converted into machine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

2.1 C++ compiler for Linux

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

2.1.1 Installing g++ compiler

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



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

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

2.2 Compiling C++ program

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

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

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

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

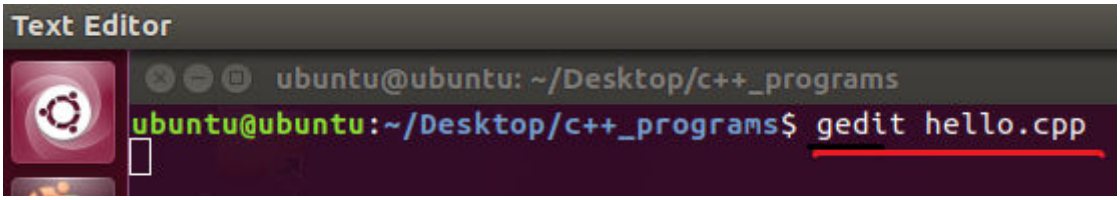
2.3 Running a C++ program

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

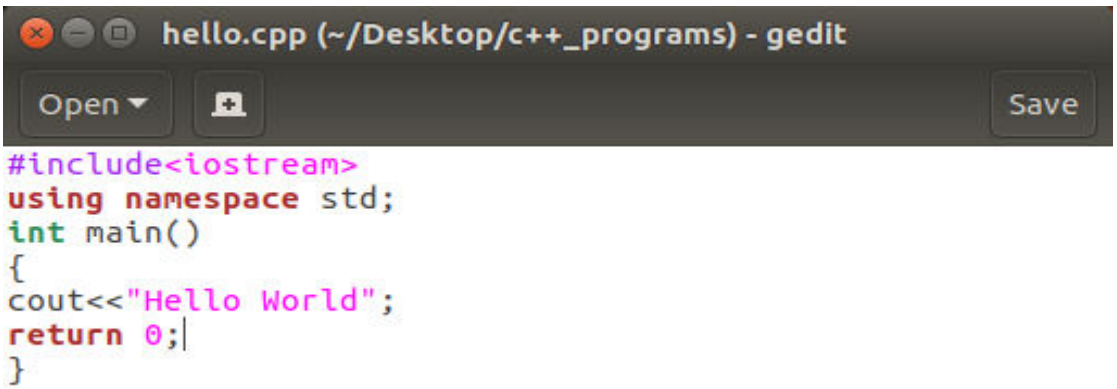
```
$ ./executable-file
```

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

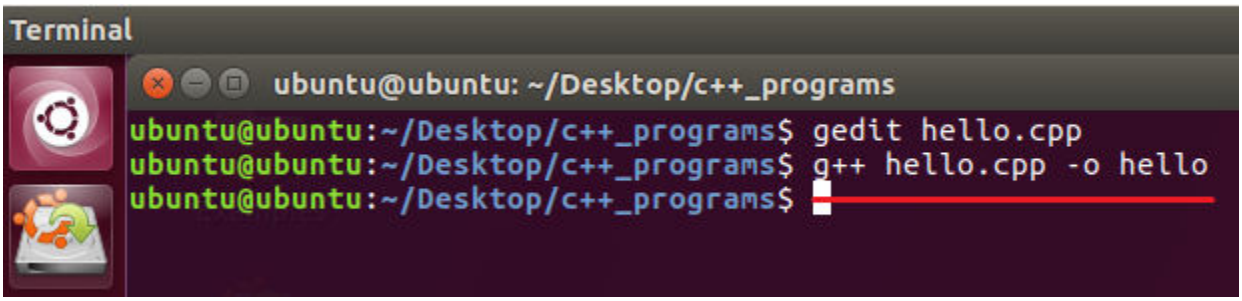
Open the gedit editor and pass the name of the file (hello.cpp) to be created



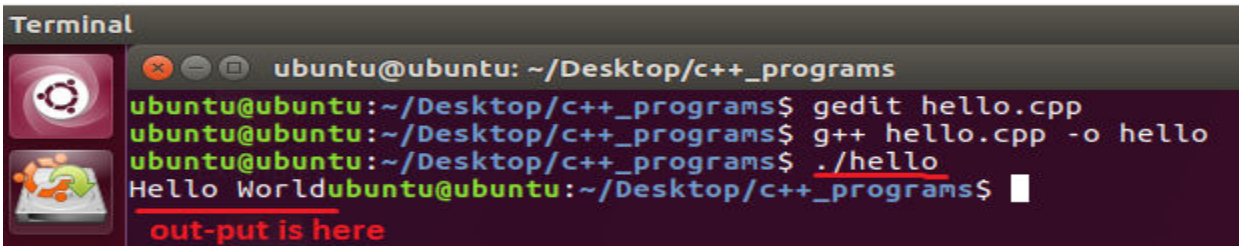
Write the following code



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



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



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

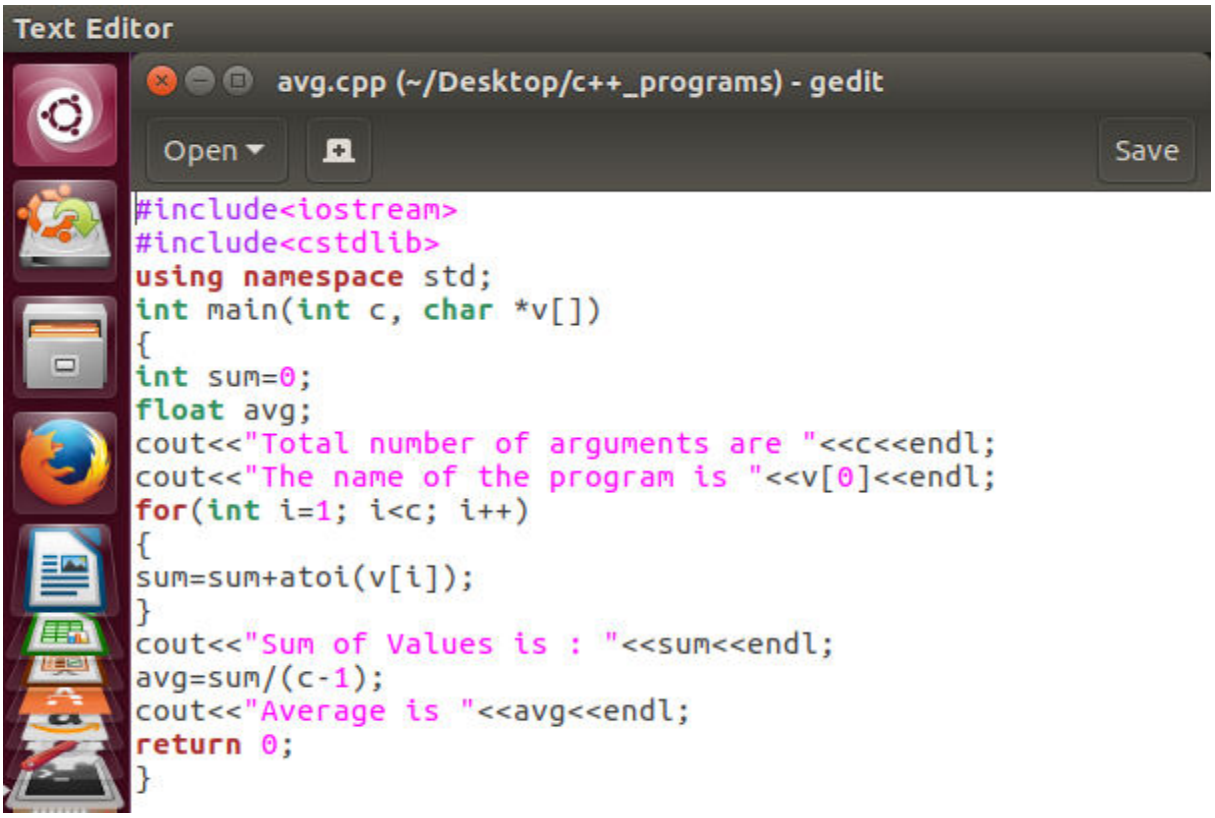
Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C++ programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main () method.

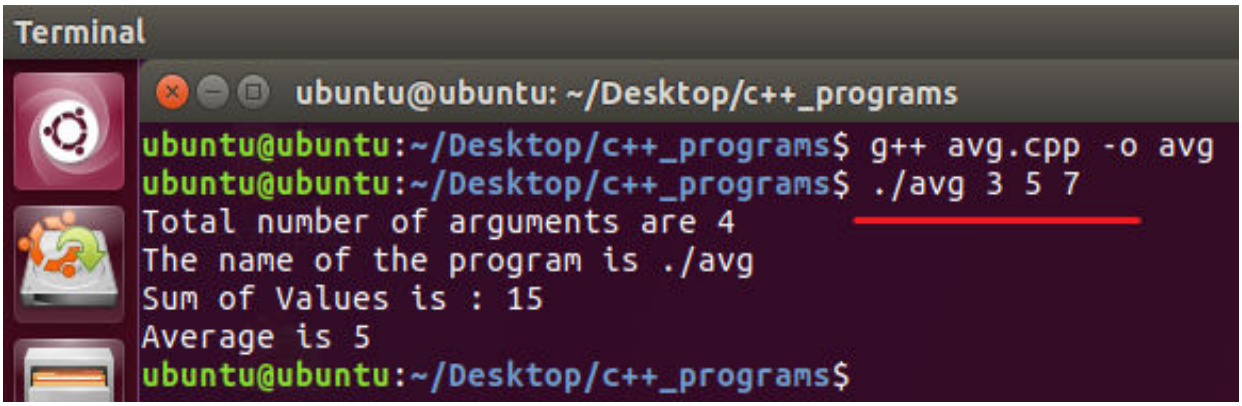
To pass command line arguments, we typically define main () with two arguments: first argument counts the number of arguments on the command line and the second is a pointer array which holds pointers of type char which points to the arguments passed to the program. The syntax to define the main method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

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

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

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





Lab Activities

Activity 1:

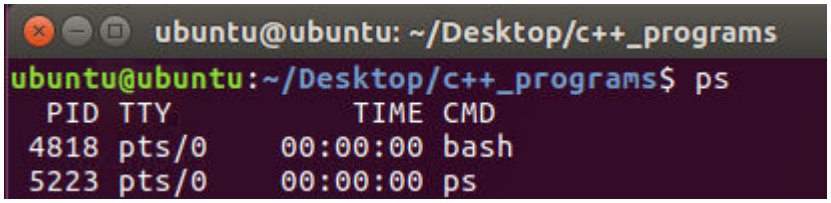
This activity is related to process monitoring in Linux

1. Display all of the process in current shell
2. Display every active process in the system
3. Provide a full-format listing of process owned by you

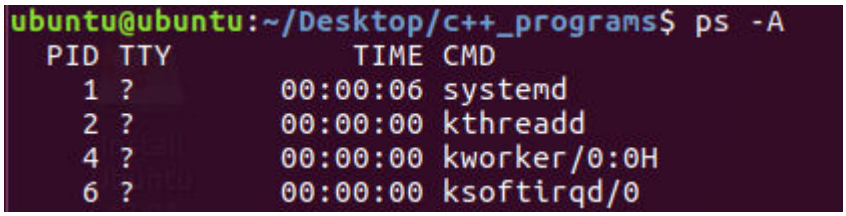
4. Display a full-format listing of processes owned by user ubuntu
5. Display a process with id 1
6. Display the dynamic view of the current processes in the system and set the refresh interval 0.5 second
7. Display the dynamic view of the processes owned by user with id 999 (or name ubuntu)
8. Start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

Solution:

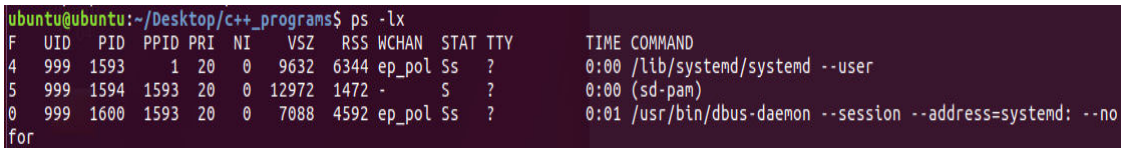
1.



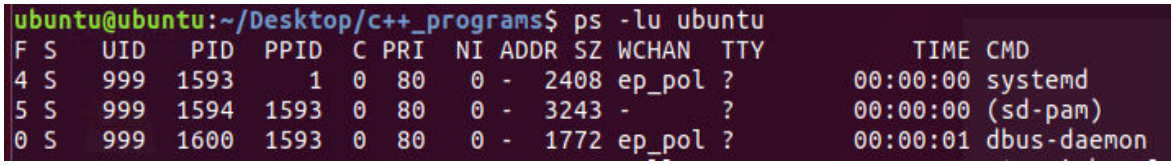
2.



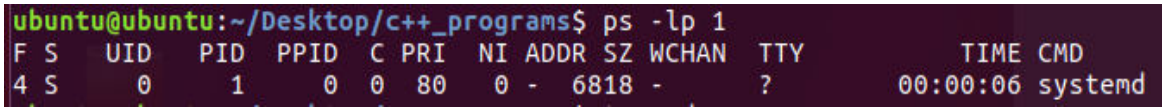
3.



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

1. Start the gedit program with priority 90
2. 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;
}
```

Activity 4:

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

Solution:

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

LAB MANUAL

Course: CSC322-Operating Systems

Department of Computer Science

Java Learning Procedure

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

Table of Contents

LAB # 06

Statement Purpose:

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

Activity Outcomes:

This lab teaches you the following topics:

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

Instructor Note:

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

1) Stage J (Journey)

1. Process Creation Concepts

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

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

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

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

ps [-option]

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

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

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

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

Process Identification:

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

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

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

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

Function: pid_t getpid (void)

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

Function: pid_t getppid (void)

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

1.1 fork () | Process Creation

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

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

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

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

Example 1 – Single fork() :

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

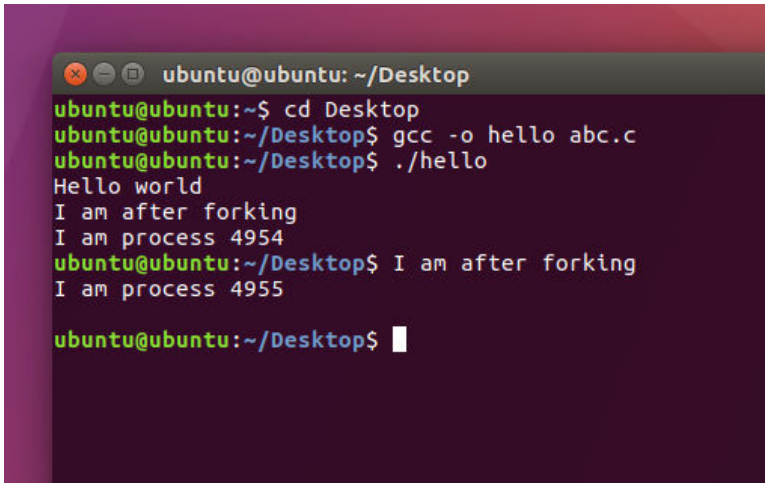
Write this program and run using the following commands:

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

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

./hello //run the code

Output:



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

Note that:

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

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

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

Example 2:

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

Executing the above code will give the following output:

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

This programs give Ids of both parent and child process.

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

Example 3 – Multiple Fork() :

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

Output:

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

Example 4:

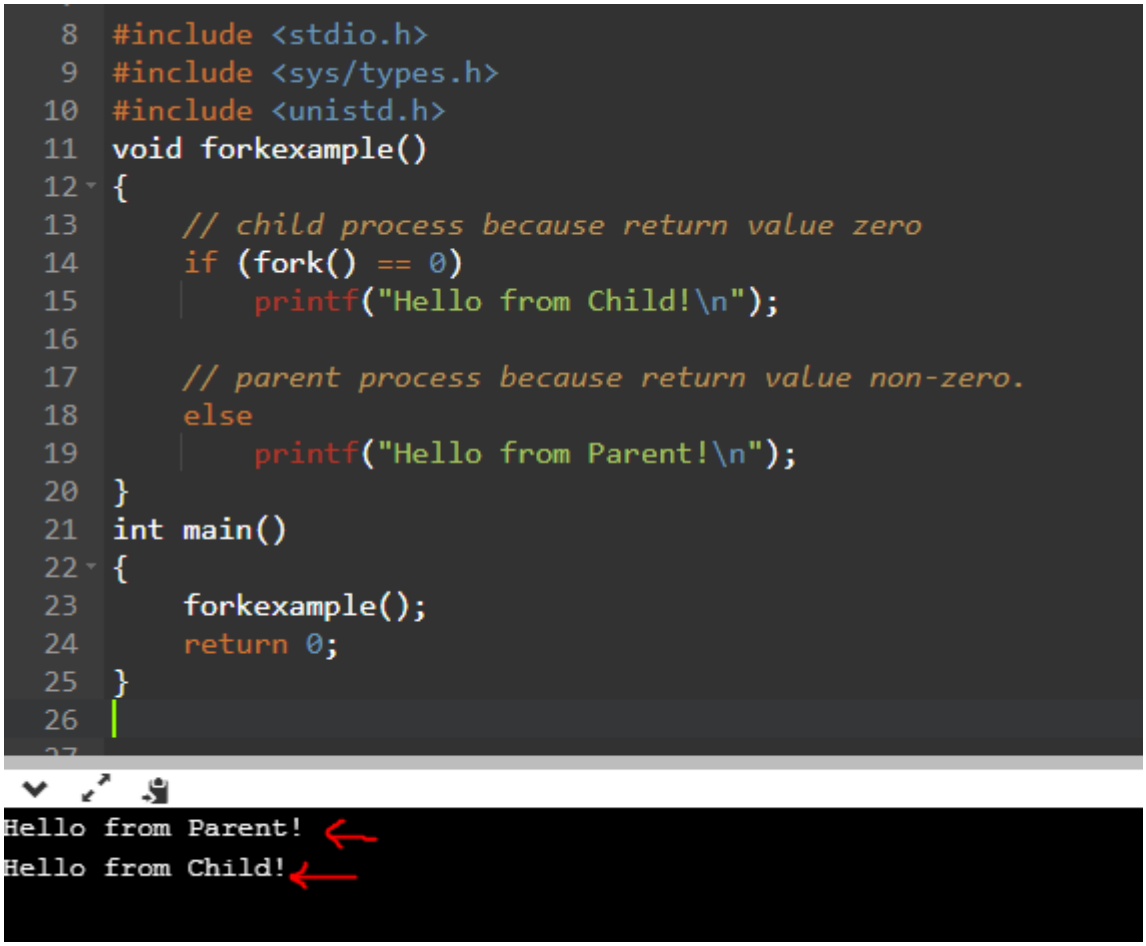
Using fork ( ) inside a function

Predict the output and verify after executing the following code

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

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

Output:



1.2 Wait () | Process Completion

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

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

1. wait ():

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

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

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

2. Exit ():

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

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

Example 1:

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

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

Output:

Example 2:

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

Output:

3. Sleep ( ):

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

Orphan Process:

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

Zombie Process:

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

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

4. Ecec ( ):

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

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

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

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

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

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

example.c

CODE:

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

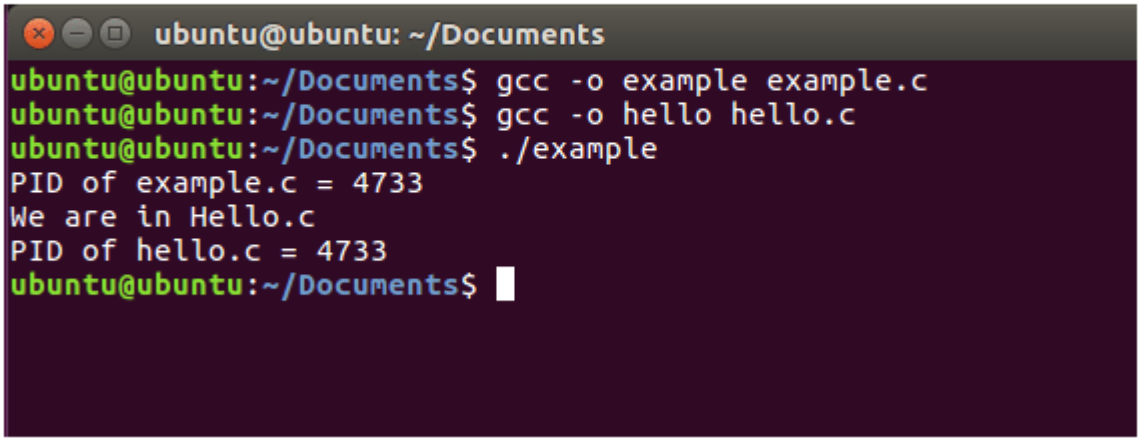
hello.c

CODE:

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

Output:

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



2) Stage a1 (apply)

Lab Activities:

Activity 1:

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

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

Solution:

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

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

Activity 2:

1. Create a process and make it an orphan.

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

Steps to create an orphan process:

1. print something and get its pid and ppid

2. create a child process

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

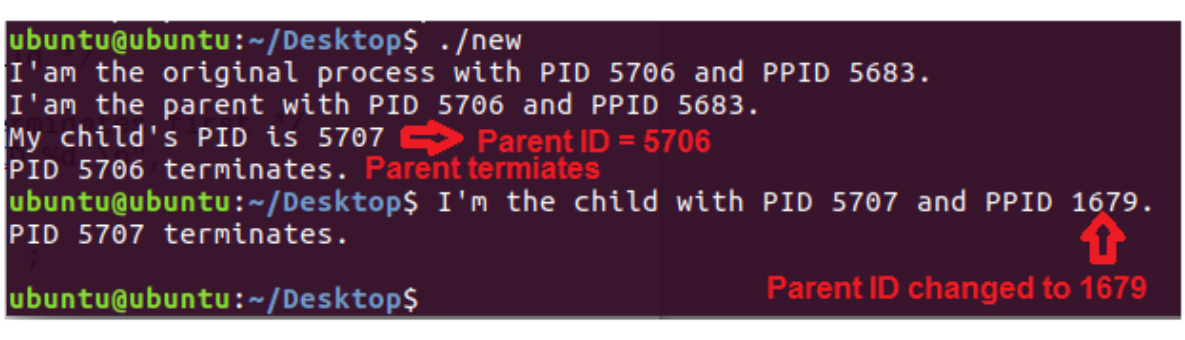
4. Make child sleep for 5 seconds

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

Solution:

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

Output:



Activity 3:

Create a process and make it a Zombie.

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

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

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

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

Solution:

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

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

Activity 4:

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

Solution:

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

Output:

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

Activity 5:

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

Solution:

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

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

Output:

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

3) Stage V (verify)

Home Activities:

Practice fork, exec, sleep and wait at home.

4) Stage a_2 (assess)

Activity assessment and Viva voce at the end of lab.