

Lab Session # 01

OBJECT: Introduction to Linux Operating System

What is Linux?

Linux is the most popular server OS. It is distributed under an open source license. Linux is a clone of UNIX. Its functionality list is quite like UNIX. Knowing one is as good as knowing the other.

Who created Linux?

Linux is an operating system or a kernel which germinated as an idea in the mind of young and bright **Linus Torvalds** when he was a computer science student. He used to work on the **UNIX OS (proprietary software)** and thought that it needed improvements.

However, when his suggestions were rejected by the designers of UNIX, he thought of launching an OS which will be **receptive to changes, modifications suggested by its users**. So **Linus devised a Kernel** named Linux in 1991. Though he would need programs like File Manager, Document Editors, Audio - Video programs to run on it. Something as you have a cone but no ice-cream on top.

As time passed by, he collaborated with other **programmers in places like MIT** and applications for Linux started to appear. So around 1991, a working Linux operating system with some applications was officially launched, and this was the start of one of the **most loved and open-source OS options available today**.

The main advantage of Linux was that programmers were able to use the Linux Kernel to design their own custom operating systems. With time, a new range of user-friendly OS's stormed the computer world. Now, **Linux is one of the most popular and widely used Kernel**, and it is the backbone of popular operating systems like **Debian, Knoppix, Ubuntu, and Fedora**. Nevertheless, the list does not end here as there are thousands of OS's based on Linux which offer a variety of functions to the users.

Linux Kernel is normally used in combination of [GNU](#) project by Dr. Richard Stallman. All modern distributions of Linux are actually distributions of Linux/GNU

The benefits of using Linux

Linux now enjoys popularity at its prime, and it's famous among programmers as well as regular computer users around the world. Its main benefits are -

It offers a **free operating system**. You do not have to shell hundreds of dollars to get the OS like Windows!

-
- Being open-source, anyone with programming knowledge can modify it.
 - The Linux operating systems now offer **millions of programs/applications to choose from**, most of them free!

- Once you have Linux installed you no longer need an antivirus! Linux is a highly secure system. More so, there is a global development community constantly looking at ways to enhance its security. With each upgrade, the OS becomes more secure and robust
- Linux is the OS of choice for Server environments due to its stability and reliability (Mega-companies like Amazon, Facebook, and Google use Linux for their Servers). A Linux based server could run non-stop without a reboot for years on end.

Is it for me?

Users, who are new to Linux, usually shun it by falsely considering it as a difficult and technical OS to operate but, to state the truth, in the last few years Linux operating systems have become a lot more user-friendly than their counterparts like Windows, so trying them is the best way to know whether Linux suits you or not.

There are **thousands of Linux based operating systems**; most of them offer **state-of-the-art security and applications**, all of it for free!

How to Download & Install Linux (Ubuntu) in Windows

How many distributions are out there?

How many distributions are out there?



There are **hundreds of Linux operating systems or Distributions** available these days. Many of them are designed with a specific purpose in mind. For example, to run a **web server** or to run on **network switches like routers, modems**, etc.

The latest example of one of the most popular smartphone-based **Linux Distribution** is **Android**!

Many of these Distributions are built to offer **excellent personal computing**.

Here, are a few popular Linux Distributions (also called Linux Distro) -

Here, are a few popular Linux Distributions (also called Linux Distro) -

Linux Distribution	Name	Description
--------------------	------	-------------

Linux Distribution	Name	Description
	Arch	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.
	CentOS	It is one of the most used Linux Distribution for enterprise and web servers. It is a free enterprise class Operating system and is based heavily on Red Hat enterprise Distro.
	Debian	Debian is a stable and popular non-commercial Linux distribution. It is widely used as a desktop Linux Distro and is user-oriented. It strictly acts within the Linux protocols.
	Fedora	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.
	Gentoo	It is a source based Distribution which means that you need to configure the code on your system before you can install it. It is not for Linux beginners, but it is sure fun for experienced users.
	LinuxMint	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.
	OpenSUSE	It is an easy to use and a good alternative to MS Windows. It can be easily set up and can also run on small computers with obsolete configurations.
	RedHat enterprise	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.
	Slackware	Slackware is one of the oldest Linux kernel based OS's. It is another easy desktop Distribution. It aims at being a 'Unix like' OS with minimal changes to its kernel.
	Ubuntu	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.

The Best Linux Distribution!

The term best is **relative**. Each Linux distribution is built for a specific purpose-built to meet the demands of its target users.

The desktop Distributions are **available for free** on their respective websites. You might want to try them one by one till you get to know which Distribution you like the most. Each one of them offers its own unique design, **applications**, and **security**.

Installing Linux using USB stick

This is one of the easiest methods of installing Ubuntu or any distribution on your computer. Follow the steps.

Step 1) Download the .iso or the OS files on your computer from this [link](#).

Download Ubuntu Desktop

Ubuntu 16.04.3 LTS

Download the latest LTS version of Ubuntu, for desktop PCs and laptops. LTS stands for long-term support — which means five years of free security and maintenance updates, guaranteed.

[Ubuntu 16.04 LTS release notes](#)

Recommended system requirements:

- ✓ 2 GHz dual core processor or better
- ✓ 2 GB system memory
- ✓ 25 GB of free hard drive space
- ✓ Either a DVD drive or a USB port for the installer media
- ✓ Internet access is helpful

[Download](#)
[Alternative downloads and torrents >](#)

Step 2) Download free software like '[Universal USB installer](#)' to make a bootable USB stick.

Universal-USB-Installer-1.9.7.8.exe – May 2, 2017 – Changes
Update to support KDE Neon, Devuan, Vinari OS, and Ubuntu Budgie.

IMPORTANT: The Windows to Go option requires the USB be formatted NTFS with 20GB free disk space to hold the virtual disk. See [FAQ](#) for more info.

 **Download UUI**
Universal-USB-Installer-1.9.7.8.exe

[Source Code](#)

MD5: 36A6A087AD0EF0368506893D15FFCDA2

Basic Requirements | Changelog | Supported Distros | FAQ

IMPORTANT NOTE: Your USB drive must be Fat32/NTFS formatted, otherwise Syslinux will fail and your drive will NOT Boot.

Step 3) Select an Ubuntu Distribution form the dropdown to put on your USB

Select your Ubuntu iso file download in step 1.

Select the drive letter of USB to install Ubuntu and Press create button.

Universal USB Installer 1.9.7.8 Setup

Setup your Selections Page
Choose a Distro, related ISO/ZIP file and, your USB Flash Drive.

Step 1: Ubuntu Selected. Go to step 2.
Ubuntu ☐ Local iso Selected.
[Visit the Ubuntu Home Page](#)

Step 2: ubuntu-16.04.0-deskyp-amd64.iso Selected ☒ Showing *ISOs
C:\Users\Guru99\Downloads\ubuntu-16.04.0-deskyp-amd64.iso

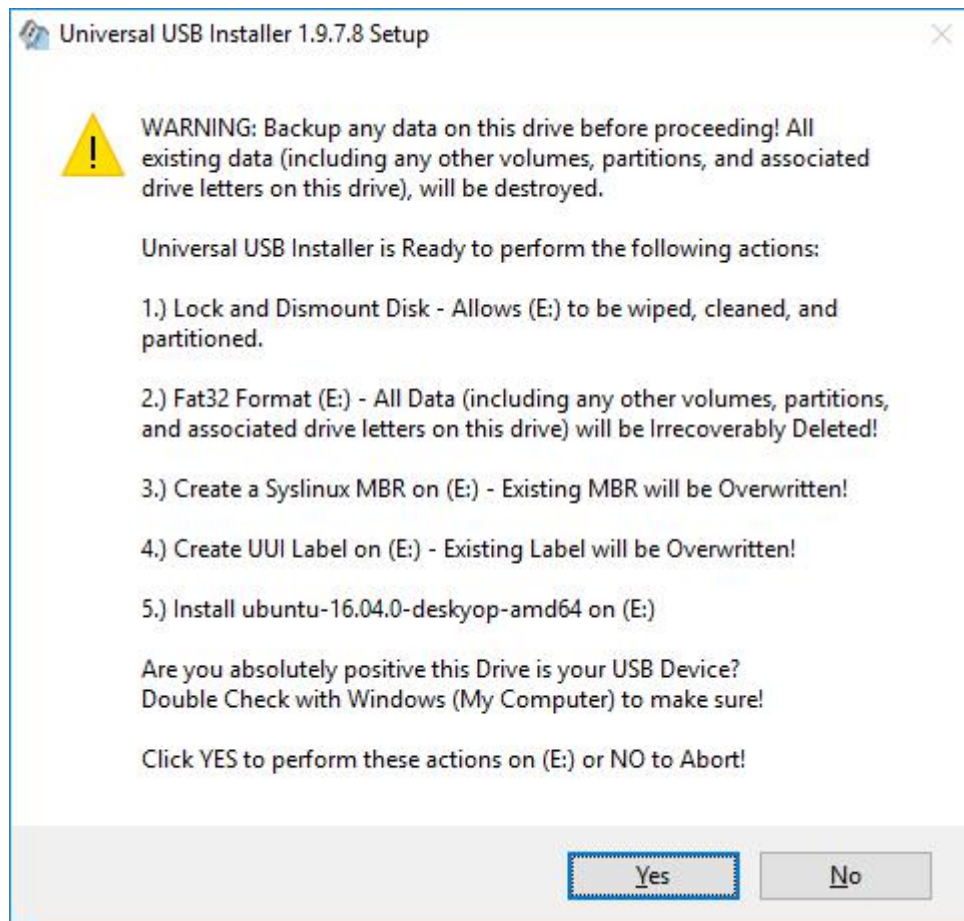
Step 3: Drive E: Selected. ☐ Show all Drives
E:\ TOSHIBA 14GB FAT32 FDD ☒ We Will Fat32 Format E:\

Step 4: Set a Persistent file size for storing changes (Optional).
0 MB

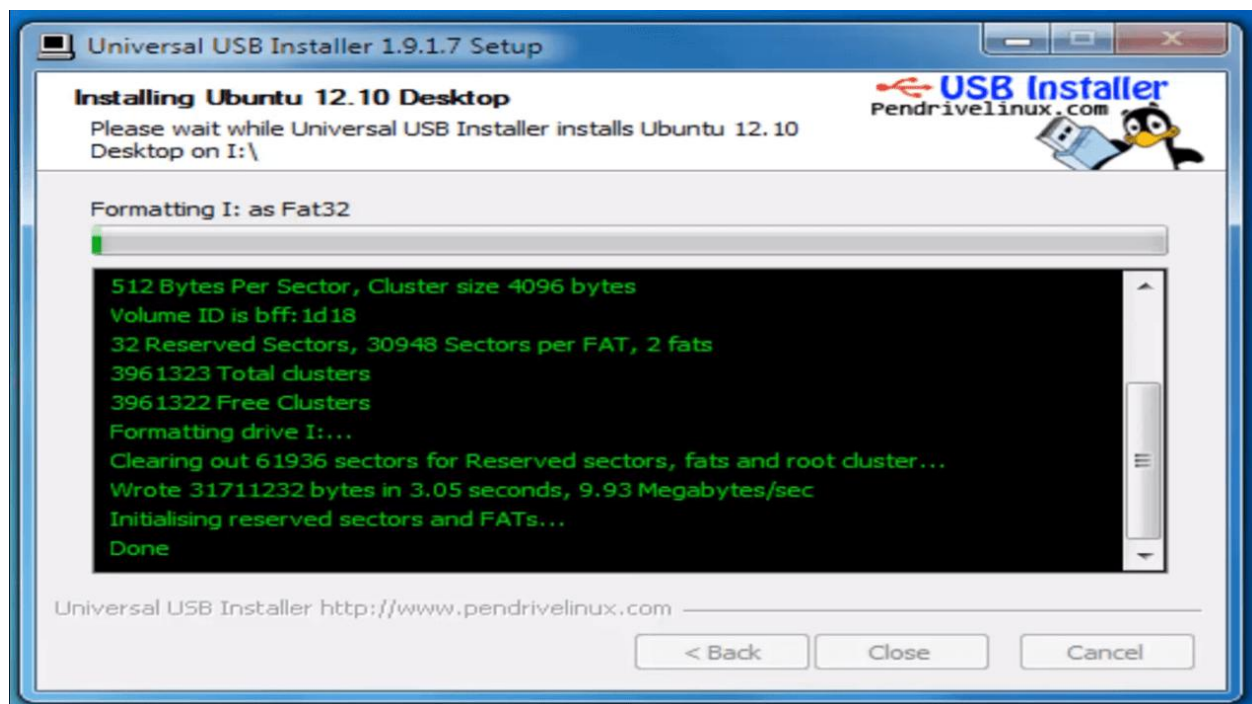
[Home Page](#) [FAQ](#) [Recommended Flash Drives](#)

Universal USB Installer <https://www.pendrivelinux.com>

Step 4) Click YES to Install Ubuntu in USB.



Step 5) After everything has been installed and configured, a small window will appear Congratulations! You now have Ubuntu on a USB stick, bootable and ready to go.



Linux vs Windows: What's the Difference?

It's time to make the big switch from your Windows or Mac OS operating system.

Mac OS uses a UNIX core. Your switch from Mac OS to Linux will be relatively smooth.

It's the Windows users who will need some adjusting. In this tutorial will introduce the Linux OS and compare it with Windows.

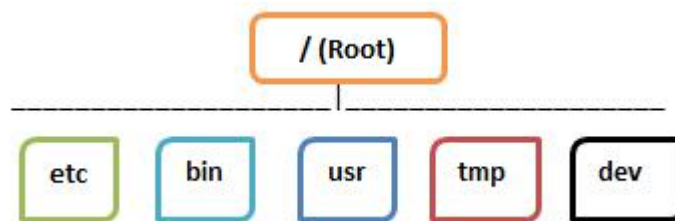
Windows Vs. Linux File System

In Microsoft Windows, files are stored in folders on different data drives like C: D: E:

But, in **Linux**, files are ordered in a tree structure starting with the root directory.

This root directory can be considered as the start of the file system, and it further branches out various other sub-directories. The root is denoted with a forward slash '/'.

A general tree file system on your UNIX may look like this.



Types of Files

In Linux and UNIX, everything is a file. Directories are files, files are files, and devices like Printer, mouse, keyboard etc. are files.

Let's look into the File types in more detail.

General Files

General Files also called as Ordinary files. They can contain image, video, program or simply text. They can be in ASCII or a Binary format. These are the most commonly used files by Linux Users.

Directory Files

These files are a warehouse for other file types. You can have a directory file within a directory (sub-directory). You can take them as 'Folders' found in Windows operating system.

Device Files:

In MS Windows, devices like Printers, CD-ROM, and hard drives are represented as drive letters like G: H:. In Linux, there are represented as files. For example, if the first SATA hard drive had three primary partitions, they would be named and numbered as `/dev/sda1`, `/dev/sda2` and `/dev/sda3`.

Note: All device files reside in the directory `/dev/`

All the above file types (including devices) have permissions, which allow a user to read, edit or execute (run) them. This is a powerful Linux/Unix feature. Access restrictions can be applied for different kinds of users, by changing permissions.

Windows Vs. Linux: Users

There are 3 types of users in Linux.

1. Regular
2. Administrative(root)
3. Service

Regular User

A regular user account is created for you when you install Ubuntu on your system. All your files and folders are stored in `/home/` which is your home directory. As a regular user, you do not have access to directories of other users.

Root User

Other than your regular account another user account called root is created at the time of installation. The root account is a **superuser** who can access restricted files, install software and has administrative privileges. Whenever you want to install software, make changes to system files or perform any administrative task on Linux; you need to log in as a root user. Otherwise, for general tasks like playing music and browsing the internet, you can use your regular account.

Service user

Linux is widely used as a Server Operating System. Services such as Apache, Squid, email, etc. have their own individual service accounts. Having service accounts increases the security of your computer. Linux can allow or deny access to various resources depending on the service.

Note:

1. You will not see service accounts in Ubuntu Desktop version.
2. Regular accounts are called standard accounts in Ubuntu Desktop

In Windows, there are 4 types of user account types.

1. Administrator
2. Standard
3. Child
4. Guest

Windows Vs. Linux: File Name Convention

In Windows, you cannot have 2 files with the same name in the same folder. While in Linux, you can have 2 files with the same name in the same directory, provided they use different cases.

Windows Vs. Linux: HOME Directory

For every user in Linux, a directory is created as **/home/**

Consider, a regular user account "Tom". He can store his personal files and directories in the directory **"/home/tom"**. He can't save files outside his user directory and does not have access to directories of other users. For instance, he cannot access directory **"/home/jerry"** of another user account "Jerry".

The concept is similar to **C:\Documents and Settings** in Windows.

When you boot the Linux operating system, your user directory (from the above example **/home/tom**) is the **default working directory**. Hence the directory **"/home/tom** is also called the **Home directory** which is a misnomer.

The working directory can be changed using some commands which we will learn later.

Windows Vs. Linux: Other Directories

In Windows, System and Program files are usually saved in C: drive. But, in Linux, you would find the system and program files in different directories. For example, the boot files are stored in the **/boot** directory, and program and software files can be found under **/bin**, device files in **/dev**. Below are important Linux Directories and a short description of what they contain.

These are most striking differences between Linux and other Operating Systems. There are more variations you will observe when switching to Linux and we will discuss them as we move along in our tutorials.

Windows Vs. Linux: Key Differences

Windows	Linux
Windows uses different data drives like C: D: E to	Unix/Linux uses a tree like a hierarchical file

stored files and folders.	system.
Windows has different drives like C: D: E	There are no drives in Linux
Hard drives, CD-ROMs, printers are considered as devices	Peripherals like hard drives, CD-ROMs, printers are also considered files in Linux/Unix
There are 4 types of user account types 1) Administrator, 2) Standard, 3) Child, 4) Guest	There are 3 types of user account types 1) Regular, 2) Root and 3) Service Account
Administrator user has all administrative privileges of computers.	Root user is the super user and has all administrative privileges.
In Windows, you cannot have 2 files with the same name in the same folder	Linux file naming convention is case sensitive. Thus, sample and SAMPLE are 2 different files in Linux/Unix operating system.
In windows, My Documents is default home directory.	For every user /home/username directory is created which is called his home directory.

Linux Command Line Tutorial: Manipulate Terminal with CD Commands

The most frequent tasks that you perform on your PC is creating, moving or deleting Files. Let's look at various options for File Management.

To manage your files, you can either use

1. Terminal (Command Line Interface - CLI)
2. File manager (Graphical User Interface -GUI)

Why learn Command Line Interface?

Even though the world is moving to GUI based systems, CLI has its specific uses and is widely used in scripting and server administration. Let's look at it some compelling uses -

- Comparatively, Commands offer more options & are flexible. Piping and stdin/stdout are immensely powerful are not available in GUI
- Some configurations in GUI are up to 5 screens deep while in a CLI it's just a single command
- Moving, renaming 1000's of the file in GUI will be time-consuming (Using Control /Shift to select multiple files), while in CLI, using regular expressions so can do the same task with a single command.
- CLI load fast and do not consume RAM compared to GUI. In crunch scenarios this matters.

Both GUI and CLI have their specific uses. For example, **in GUI, performance monitoring graphs** give **instant visual feedback** on system health, while seeing hundreds of lines of logs in CLI is an eyesore.

You must learn to use both GUI(File Manager) and CLI (Terminal)

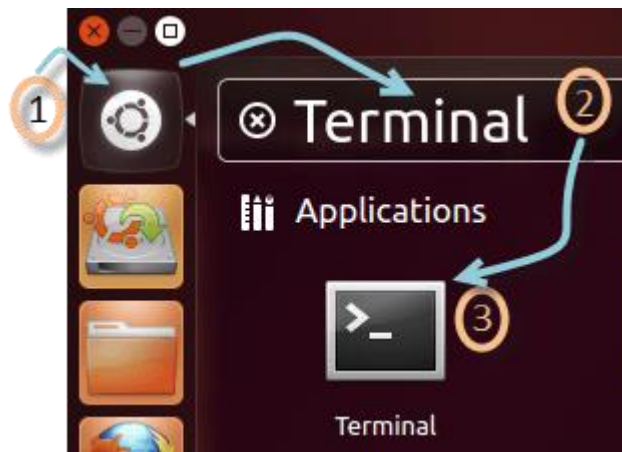
GUI of a Linux based OS is similar to any other OS. Hence, we will focus on CLI and learn some useful commands.

Launching the CLI on Ubuntu

There are 2 ways to launch the terminal.

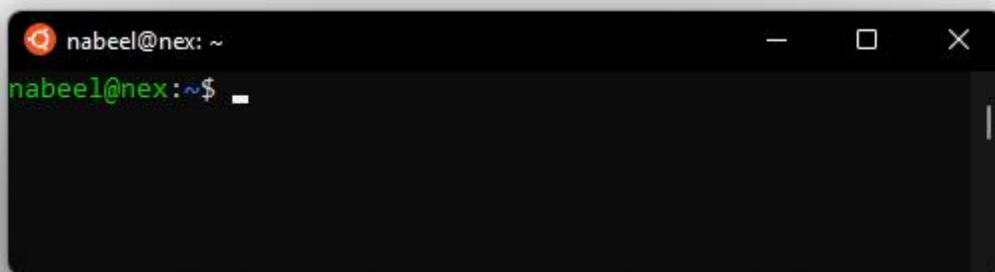
1) Go to the Dash and type terminal

1) Go to the Dash and type terminal



2) Or you can press **CTRL + Alt + T** to launch the Terminal

Once you launch the CLI (Terminal), you would find something as guru99@VirtualBox(see image) written on it.

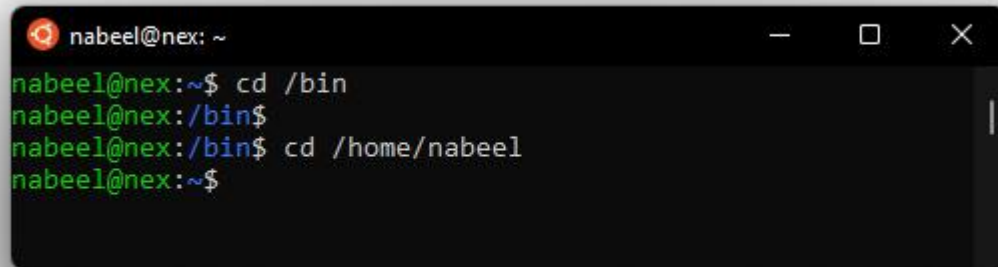


1) The first part of this line is the name of the **user** (nabeel, ubuntu, home...)

2) The second part is the computer name(nex) or the host name. The hostname helps identify a computer over the network. In a server environment, host-name becomes important.

3) The ':' is a simple separator

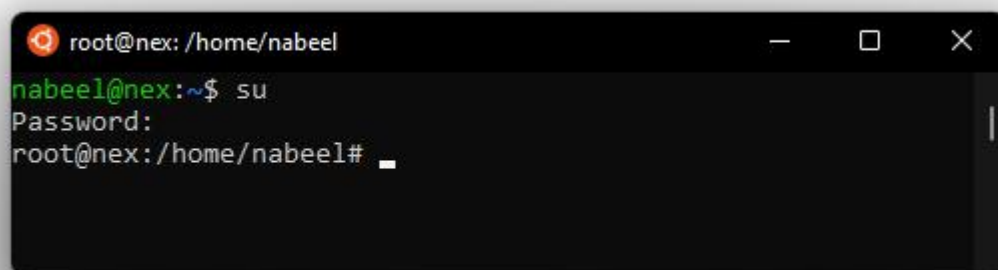
4) The tilde '~' sign shows that the user is working in the **home directory**. If you change the directory, this sign will vanish.



```
nabeel@nex: ~  
nabeel@nex:~$ cd /bin  
nabeel@nex:/bin$  
nabeel@nex:/bin$ cd /home/nabeel  
nabeel@nex:~$
```

In the above illustration, we have moved from the /home directory to /bin using the 'cd' command. The ~ sign does not display while working in /bin directory. It appears while moving back to the home directory.

5) The '\$' sign suggests that you are working as a regular user in Linux. While working as a root user, '#' is displayed.

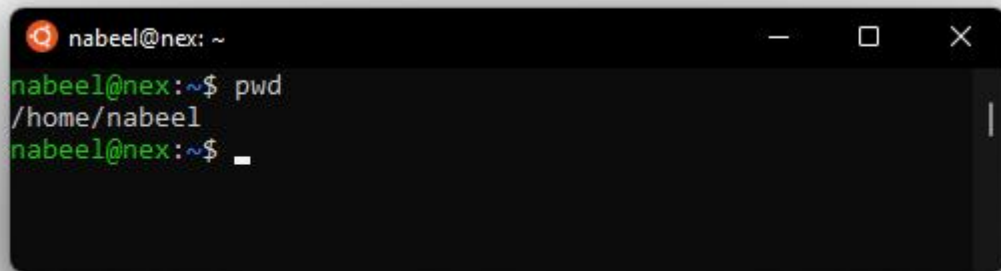


```
root@nex: /home/nabeel  
nabeel@nex:~$ su  
Password:  
root@nex:/home/nabeel#
```

Present Working Directory

The directory that you are currently browsing is called the Present working directory. You log on to the home directory when you boot your PC. If you want to determine the directory you are presently working on, use the command -

pwd

A terminal window with a dark background. The title bar shows 'nabeel@nex: ~'. The prompt is 'nabeel@nex:~\$'. The user has entered 'pwd' and the output is '/home/nabeel'. The prompt is now 'nabeel@nex:~\$' with a cursor.

```
nabeel@nex: ~
nabeel@nex:~$ pwd
/home/nabeel
nabeel@nex:~$
```

pwd command stands for **p**rint **w**orking **d**irectory

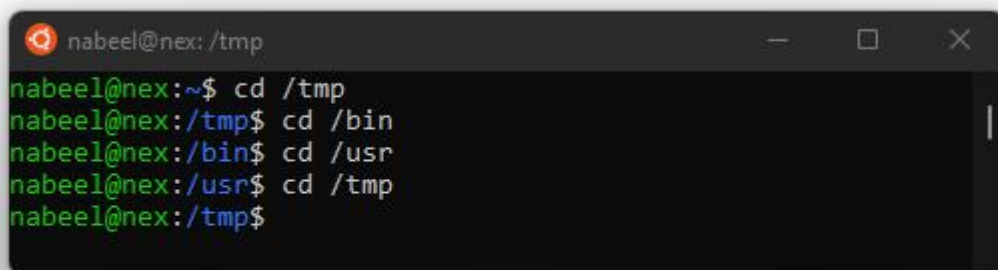
Above figure shows that /home/guru99 is the directory we are currently working on.

Changing Directories

If you want to change your current directory use the '**cd**' command.

cd /tem

Consider the following example.

A terminal window with a dark background. The title bar shows 'nabeel@nex: /tmp'. The prompt is 'nabeel@nex:~\$'. The user enters 'cd /tmp', and the prompt changes to 'nabeel@nex:/tmp\$'. Then the user enters 'cd /bin', and the prompt changes to 'nabeel@nex:/bin\$'. Then the user enters 'cd /usr', and the prompt changes to 'nabeel@nex:/usr\$'. Finally, the user enters 'cd /tmp', and the prompt changes back to 'nabeel@nex:/tmp\$'.

```
nabeel@nex:~$ cd /tmp
nabeel@nex:/tmp$ cd /bin
nabeel@nex:/bin$ cd /usr
nabeel@nex:/usr$ cd /tmp
nabeel@nex:/tmp$
```

Here, we moved from directory /tmp to /bin to /usr and then back to /tmp.

Navigating to home directory

If you want to navigate to the home directory, then type **cd**.

A terminal window with a black background and green text. The window title is 'nabeel@nex: ~'. The command history shows: 'nabeel@nex:/tmp\$', 'nabeel@nex:/tmp\$ cd', and 'nabeel@nex:~\$'.

cd

You can also use the **cd ~** command.

A terminal window with a black background and green text. The window title is 'nabeel@nex: ~'. The command history shows: 'nabeel@nex:~\$ cd /tmp', 'nabeel@nex:/tmp\$', 'nabeel@nex:/tmp\$ cd ~', and 'nabeel@nex:~\$'.

cd ~

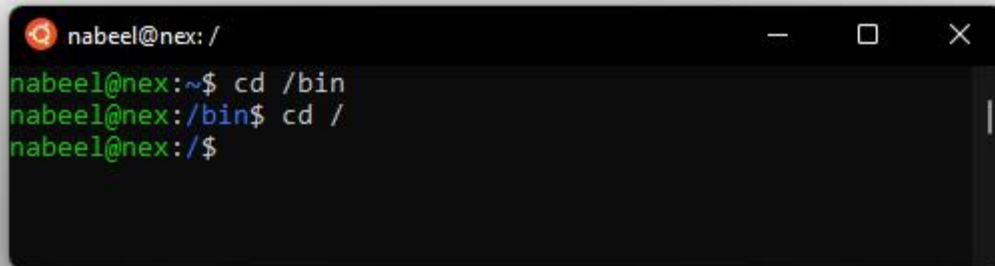
Moving to root directory

The root of the file system in Linux is denoted by '/'. Similar to 'c:\' in Windows.

Note: In Windows, you use backward slash "\" while in UNIX/Linux, forward slash is used "/"

Type 'cd /' to move to the root directory.

cd /

A terminal window with a dark background and a red logo in the top-left corner. The window title is 'nabeel@nex: /'. The terminal shows three lines of text: 'nabeel@nex:~\$ cd /bin', 'nabeel@nex:/bin\$ cd /', and 'nabeel@nex:/\$'. The prompt changes from '~\$' to '/bin\$' and then to '/\$' as the user navigates between directories.

```
nabeel@nex: /
nabeel@nex:~$ cd /bin
nabeel@nex:/bin$ cd /
nabeel@nex:/$
```

TIP: Do not forget space between **cd** and **/**. Otherwise, you will get an error.

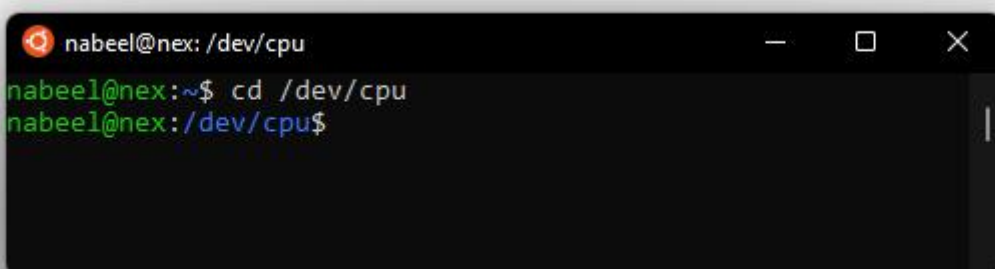
Navigating through multiple directories

You can navigate through multiple directories at the same time by specifying its complete path.

Example: If you want to move the /cpu directory under /dev, we do not need to break this operation in two parts.

Instead, we can type '/dev/cpu' to reach the directory directly.

```
cd /dev/cpu
```

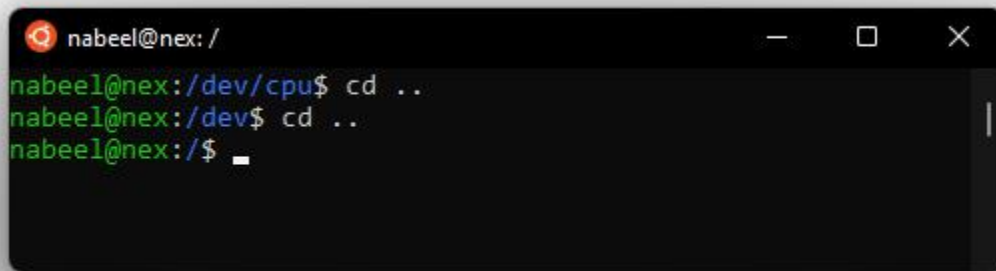
A terminal window with a dark background and a red logo in the top-left corner. The window title is 'nabeel@nex: /dev/cpu'. The terminal shows two lines of text: 'nabeel@nex:~\$ cd /dev/cpu' and 'nabeel@nex:/dev/cpu\$'. The prompt changes from '~\$' to '/dev/cpu\$' after the command is executed.

```
nabeel@nex: /dev/cpu
nabeel@nex:~$ cd /dev/cpu
nabeel@nex:/dev/cpu$
```

Moving up one directory level

For navigating up one directory level, try.

```
cd ..
```

A terminal window with a dark background and light green text. The window title is 'nabeel@nex: /'. The terminal shows three lines of commands and their output: 'nabeel@nex:/dev/cpu\$ cd ..', 'nabeel@nex:/dev\$ cd ..', and 'nabeel@nex:/\$ _'. The window has standard Linux window controls (minimize, maximize, close) in the top right corner.

```
nabeel@nex: /
nabeel@nex:/dev/cpu$ cd ..
nabeel@nex:/dev$ cd ..
nabeel@nex:/$ _
```

Here by using the 'cd ..' command, we have moved up one directory from '/dev/cpu' to '/dev'.

Then by again using the same command, we have jumped from '/dev' to '/' root directory.

Relative and Absolute Paths

A path in computing is the address of a file or folder.

Example -

In Windows

C:\documentsandsettings\user\downloads

In Linux

/home/user/downloads

There are two kinds of paths:

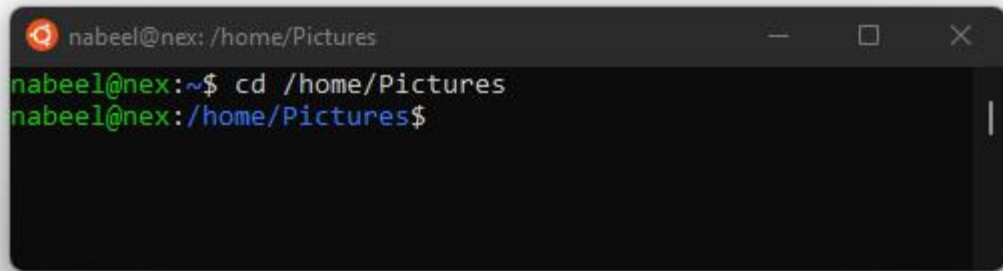
1. Absolute Path:

Let's say you have to browse the images stored in the Pictures directory of the home folder.

The absolute file path of Pictures directory **/home/Pictures**

To navigate to this directory, you can use the command.

cd /home/Pictures

A terminal window with a dark background. The title bar shows 'nabeel@nex: /home/Pictures'. The prompt is 'nabeel@nex:~\$'. The command 'cd /home/Pictures' is entered and executed. The prompt changes to 'nabeel@nex:/home/Pictures\$'.

```
nabeel@nex:~$ cd /home/Pictures
nabeel@nex:/home/Pictures$
```

This is called absolute path as you are specifying the full path to reach the file.

2. Relative Path:

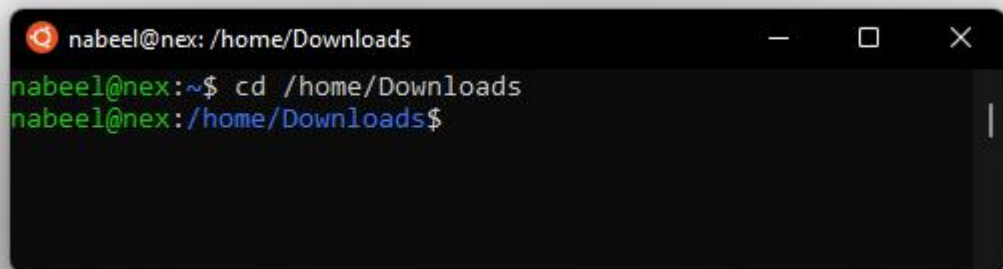
The Relative path comes in handy when you have to browse another subdirectory within a given directory.

It saves you from the effort to type complete paths all the time.

Suppose you are currently in your Home directory. You want to navigate to the Downloads directory.

You do not need to type the absolute path

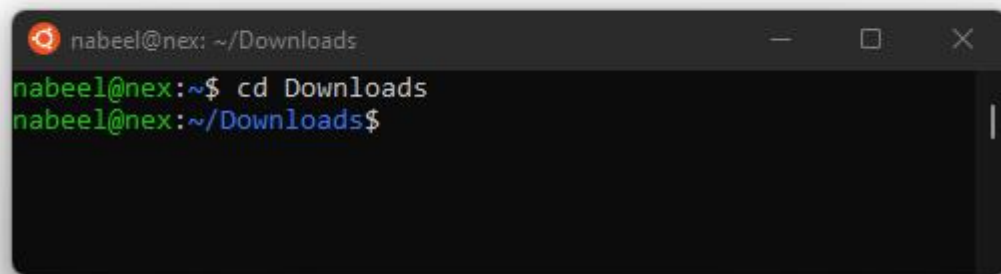
```
cd /home/guru99/Downloads
```

A terminal window with a dark background. The title bar shows 'nabeel@nex: /home/Downloads'. The prompt is 'nabeel@nex:~\$'. The command 'cd /home/Downloads' is entered and executed. The prompt changes to 'nabeel@nex:/home/Downloads\$'.

```
nabeel@nex:~$ cd /home/Downloads
nabeel@nex:/home/Downloads$
```

Instead, you can simply type '**cd Downloads**' and you would navigate to the Downloads directory as you are already present within the '**/home/guru99**' directory.

```
cd Downloads
```

A terminal window with a dark background and a light gray title bar. The title bar contains the text 'nabeel@nex: ~/Downloads' and three window control icons (minimize, maximize, close). The terminal shows two lines of text: 'nabeel@nex:~\$ cd Downloads' and 'nabeel@nex:~/Downloads\$'. The first line is in green, and the second line is in blue. The prompt 'nabeel@nex:' is in green, and the rest of the text is in blue.

```
nabeel@nex: ~/Downloads
nabeel@nex:~$ cd Downloads
nabeel@nex:~/Downloads$
```

This way you do not have to specify the complete path to reach a specific location within the same directory in the file system

Lab Session # 02

OBJECT: *Executing some of the most frequently used Linux commands*

THEORY

A Linux command is any executable file. This means that any executable file added to the system becomes a new command on the system. A Linux command is a type of file that is designed to be run, as opposed to files containing data or configuration information.

Input and Output Redirection

Linux allows to "pipe" the output from a command so that it becomes another command's input. This is done by typing two or more commands separated by the | character. The | character means "Use the output from the previous command as the input for the next command." Therefore, typing `command_1|command_2` does both commands, one after the other, before giving you the results. Another thing you can do in Linux is to send output to a file instead of the screen. To send output to a file, use the ">" symbol. There are many different reasons why you might want to do this. You might want to save a "snapshot" of a command's output as it was at a certain time, or you might want to save a command's output for further examination. You might also want to save the output from a command that takes a very long time to run, and so on.

Command Options & other parameters

You can use command options to fine-tune the actions of a Linux command. Instead of making you learn a second command, Linux lets you modify the basic, or default, actions of the command by using options.

Linux commands often use parameters that are not actual command options. These parameters, such as filenames or directories, are not preceded by a dash.

Executing a Linux Command

From the command prompt simply type the name of the command:

`$ command`

Where \$ is the prompt character for the Bourne shell.

Or if the command is not on your path type the complete path and name of the command such as:

`$ /usr/bin/command.`

Some of the frequently used Linux commands are: `su`, `pwd`, `cd`, `ls`, `more` and `less`, `find` and `grep`, `man`. (Refer to Appendix B for detail of few commands)

1. Su

Description: "su" stands for "super user". Runs a new shell under different user and group IDs. If no user is specified, the new shell will run as the root user.

Syntax: su [-flmp] [-c command] [-s shell] [--login] [--fast] [--preserve-environment] [--command=command] [--shell=shell] [-] [user]

(See the man pages for the description of the flags and options by using man su)

```
$ su <username>    (to become another user)  or
```

```
$ su               (to become the root user).
```

Adding a new user

```
#useradd user1
```

```
#passwd user1
```

```
<password>
```

2. pwd

Description: Displays the name of the current directory. pwd stands for present working directory. By typing this command you are informed of which directory you are currently in.

Syntax: pwd

3. cd

Description: Changes the current directory to any accessible directory on the system.

Syntax: For instance to change from /home/user1 to a subdirectory of user1 wordfiles use the following:

```
$ cd wordfiles
```

```
$ pwd
```

```
/home/user1/wordfiles
```

To change to /tmp use the following:

```
$ cd /tmp
```

```
$ pwd  
/tmp
```

4. ls

Description: Displays the listing of files and directories. If no file or directory is specified, then the current directory's contents are displayed. By default the contents are sorted alphabetically.

Syntax: To view the contents of user1 home directory use this:

```
$ ls
```

To list the contents of any directory on the system use:

```
$ ls /usr
```

(See the man pages for the description of the flags and options by using man ls)

5. more

Description: Displays one or more files screen by screen and allows for searching and jumping to an arbitrary location in the file.

Syntax: more [-dlfs] [-number] [+number] [file....]

(See the man pages for the description of the flags and options by using man more)

For example if there is a large text file called "textfile", it can be viewed a page at a time with the given command: \$ more textfile

After pressing Enter, the first screen of the file is seen with the text -More- displayed on the last line of the screen. Pressing the space bar jumps forward a full screen length, while pressing enter key moves forward one line at a time. When the end of the body of the text is reached, the command prompt appears.

To search forward through a file, enter slash followed by the word or phrase you want to search for and then press enter. The display jumps forward to the first occurrence of the word or phrase being searched for and displays the occurrence near the top of the screen. The same search can be repeated by entering n after the first search, avoiding the need to type the same word or phrase repeatedly.

6. less

Description: Displays a text file one screen at a time while allowing searching and backward scrolling.

Syntax: less [-aeEGilNrsS] file..... *(See the man pages for the description of the flags and options by using man less)*

This command is an improved version of the previous one. On addition to the functions previously described the following are some of the other actions that can be performed using less:

- Jumping directly to a line.
- Jumping directly to the beginning or end of file.
- Moving backward through the file.
- Searching backward through the file.

7. find

Description: Looks for files below the specified paths that match all the criteria indicated by the command-line options and takes any action indicated by those options. If no paths are specified, the search takes place below the current directory.

Syntax: find [path.....] [Options]

(See the man pages for the description of the flags and options by using man find)

This command can be used to search for files by name, date of creation, size and even file type. To search for files by name use the following syntax:

\$ find starting-directory parameters actions

The **starting-directory** specifies where to begin searching.

The **parameters** are where you specify the criteria by which to search. Here use

–name filename to specify the file to search for.

The actions section indicates what actions to take on found files. If –print is used then the full name and path of file is displayed.

To search for all files named myfile on your system, use this:

\$ find / -name myfile –print

(Notice that the previous command attempted to search for the entire system. To do this effectively user must be logged in as the root user).

8. grep

Description: Searches files for lines matching a specific pattern and displays the lines

Grep stands for Global Regular Expression Parser. What grep does, essentially, is find and display lines that contain a pattern that you specify. There are two basic ways to use grep.

The first use of grep is to filter the output of other commands. The general syntax is `<command> | grep <pattern>`. For instance, if we wanted to see every actively running process on the system, we would type `ps -a | grep R`. In this application, grep passes on only those lines that contain the pattern (in this case, the single letter) R. Note that if someone were running a program called Resting, it would show up even if its status were S for sleeping, because grep would match the R in Resting. An easy way around this problem is to type `grep " R "`, which explicitly tells grep to search for an R with a space on each side. You must use quotes whenever you search for a pattern that contains one or more blank spaces.

The second use of grep is to search for lines that contain a specified pattern in a specified file. The syntax here is `grep <pattern> <filename>`. Be careful. It's easy to specify the filename first and the pattern second by mistake! Again, you should be as specific as you can with the pattern to be matched, in order to avoid "false" matches.

Syntax: Assuming that you are in your home directory, then the following command searches for the word "hello" in each file in your home directory and produces the results as follows:

```
$ grep hello*
```

This command then returns one line for each occurrence of the word. The name of the file is also shown.

In general the pattern for the grep command is:

```
$ grep text-pattern file-list
```

The **text-pattern** can be a simple word or phrase or a more complicated regular expression. The use of regular expressions can be found in the man pages.

The **file-list** can take any form allowed by the shell.

To check for the contents of all files in a directory use the following:

```
$ grep text-pattern *
```

where `*` indicates that all files in the current directory should be searched.

In its simplest form, the text pattern is a single word or part of a word containing no spaces. To search for a phrase such as "is a test", enclose the pattern in quotation marks as follows:

```
$ grep "is a test" *
```

(See the man pages for the description of the flags and options by using `man grep`)

9. USING THE *man* PAGES

The man pages are manual pages provided in a standard format with most Linux software. Almost all the commands that ship with Red Hat Linux distribution include man pages. Using the man command in its most basic form, any existing man page can be read:

```
$ man command-name
```

The above displays the man page for the specified command and allows scrolling through it and searching it the same way as when using the less command to display text. If the specified man page cannot be found an error is displayed.

EXERCISES

1. Write a command to:

- List all files (and subdirectories) in the home directory.
- List all files named *chapter1* in the */work* directory.
- List all files beginning with *memo* owned by *ann*.
- Display the content of */etc/passwd* file with as many lines at a time as the last digit of your roll number.
- Search the current directory, look for filenames that don't begin with a capital letter.

- Search the system for files that were modified within the last two days.
- Recursively grep for *your-name* down a directory tree.
- List all file names containing your roll number in the end.
- List files in your home folder in human readable format.
- List all files in the current directory and their sizes; use multiple columns and mark special files.
- List the contents of directories */bin* and */etc*.
- List C source files in the current directory, showing larger file first.
- Count all files in the current directory.
- Page through *file* in “clear” mode and display prompts.

- Format *doc* to screen removing underlines.
- Create a group called *directors* and assign it a password. Add user *Ann* and user *James* to the group and assign passwords to each of them.

LAB SESSION # 3

OBJECT: File Permissions in Linux/Unix

The concept of **ownership** and **permissions** is crucial in Linux.

Ownership of Linux files

Every file and directory on your Unix/Linux system is assigned 3 types of owner, given below.

- **User**

A user is the owner of the file. By default, the person who created a file becomes its owner.

- **Group**

A user- group can contain multiple users. All users belonging to a group will have the same access permissions to the file. Suppose you have a project where a number of people require access to a file. Instead of manually assigning permissions to each user, you could add all users to a group, and assign group permission to file such that only this group members and no one else can read or modify the files.

- **Other**

Any other user who has access to a file. This person has neither created the file, nor he belongs to a usergroup who could own the file. Practically, it means everybody else. Hence, when you set the permission for others, it is also referred as set permissions for the world.

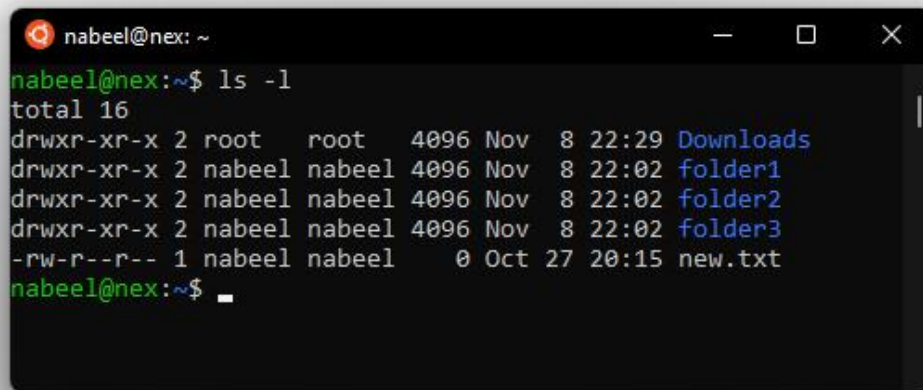
Permissions

Every file and directory in your UNIX/Linux system has following 3 permissions defined for all the 3 owners discussed above.

- **Read:** This permission give you the authority to open and read a file. Read permission on a directory gives you the ability to lists its content.
- **Write:** The write permission gives you the authority to modify the contents of a file. The write permission on a directory gives you the authority to add, remove and rename files stored in the directory. Consider a scenario where you have to write permission on file but do not have write permission on the directory where the file is stored. You will be able to modify the file contents. But you will not be able to rename, move or remove the file from the directory.
- **Execute:** In Windows, an executable program usually has an extension ".exe" and which you can easily run. In Unix/Linux, you cannot run a program unless

the execute permission is set. If the execute permission is not set, you might still be able to see/modify the program code(provided read & write permissions are set), but not run it.

ls -l on terminal gives



```
nabeel@nex: ~  
nabeel@nex:~$ ls -l  
total 16  
drwxr-xr-x 2 root  root  4096 Nov  8 22:29 Downloads  
drwxr-xr-x 2 nabeel nabeel 4096 Nov  8 22:02 folder1  
drwxr-xr-x 2 nabeel nabeel 4096 Nov  8 22:02 folder2  
drwxr-xr-x 2 nabeel nabeel 4096 Nov  8 22:02 folder3  
-rw-r--r-- 1 nabeel nabeel   0 Oct 27 20:15 new.txt  
nabeel@nex:~$
```

Here, the first '-' implies that it is a file Else, if it were a directory, 'd' would have been shown.

The characters are pretty easy to remember.

- r = read permission
- w = write permission
- x = execute permission
- - = no permission

The first part of the code is 'rw-'. This suggests that the owner 'Home' can:

- Read the file
- Write or edit the file
- He cannot execute the file since the execute bit is set to '-'.

The second part is 'rw-'. It for the user group 'Home' and group-members can:

- Read the file
- Write or edit the file

The third part is for the world which means any user. It says 'r--'. This means the user can only:

- Read the file

Changing file/directory permissions

'chmod' command is used to change the file/directory permissions

Syntax: `chmod permissions filename`

There are 2 ways to use the command -

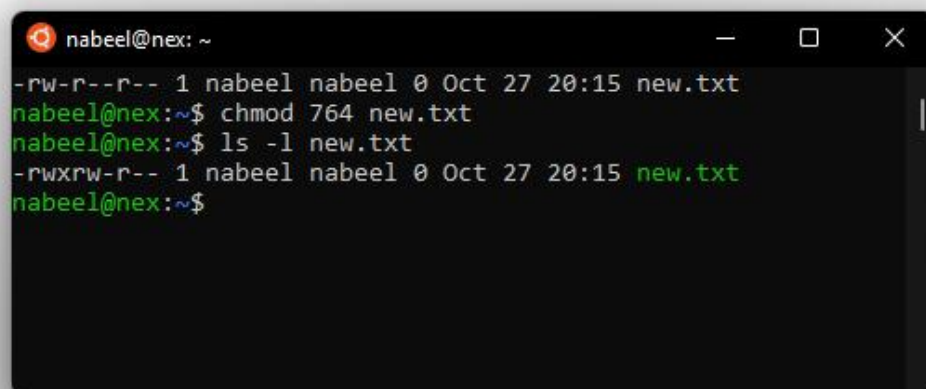
- Absolute mode
- Symbolic mode

1. Absolute(Numeric) Mode

In this mode, file permissions are not represented as characters but a three-digit octal number.

The table below gives numbers for all for permissions types.

Number	Permission Type	Symbol
0	No Permission	---
1	Execute	--x
2	Write	-w-
3	Execute + Write	-wx
4	Read	r--
5	Read + Execute	r-x
6	Read +Write	rw-
7	Read + Write +Execute	rxw



```
nabeel@nex: ~  
-rw-r--r-- 1 nabeel nabeel 0 Oct 27 20:15 new.txt  
nabeel@nex:~$ chmod 764 new.txt  
nabeel@nex:~$ ls -l new.txt  
-rwxrw-r-- 1 nabeel nabeel 0 Oct 27 20:15 new.txt  
nabeel@nex:~$
```

'764' absolute code says the following:

- Owner can read, write and execute
- Usergroup can read and write
- World can only read

This is shown as **-rwxrw-r-**

2. Symbolic Mode

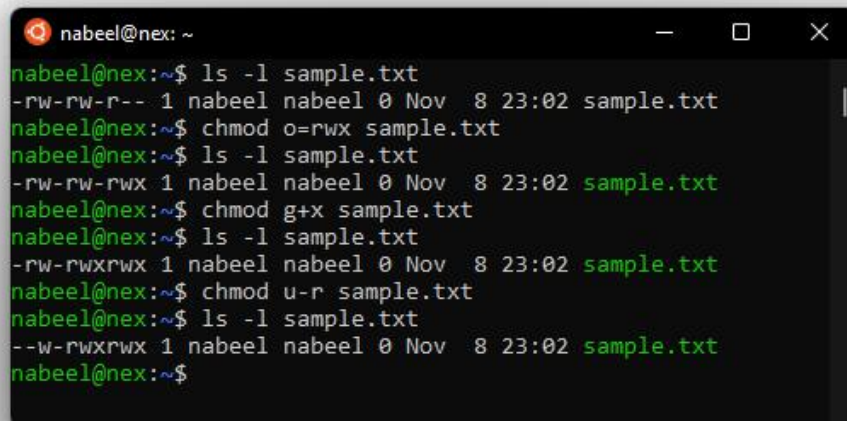
In the Absolute mode, you change permissions for all 3 owners. In the symbolic mode, you can modify permissions of a specific owner. It makes use of mathematical symbols to modify the file permissions.

Operator	Description
+	Adds a permission to a file or directory
-	Removes the permission
=	Sets the permission and overrides the permissions set earlier.

The various owners are represented as -

User Denotations	
u	user/owner
g	group
o	other
a	all

We will not be using permissions in numbers like 755 but characters like rwx. Let's look into an example

A terminal window titled 'nabeel@nex: ~' with standard window controls. It shows a series of commands and their outputs for the file 'sample.txt'. The commands are: 'ls -l sample.txt', 'chmod o=rwx sample.txt', 'ls -l sample.txt', 'chmod g+x sample.txt', 'ls -l sample.txt', 'chmod u-r sample.txt', and 'ls -l sample.txt'. The permissions change from '-rw-rw-r--' to '--w-rwxrwx' through these steps.

```
nabeel@nex:~$ ls -l sample.txt
-rw-rw-r-- 1 nabeel nabeel 0 Nov  8 23:02 sample.txt
nabeel@nex:~$ chmod o=rwx sample.txt
nabeel@nex:~$ ls -l sample.txt
-rw-rw-rwx 1 nabeel nabeel 0 Nov  8 23:02 sample.txt
nabeel@nex:~$ chmod g+x sample.txt
nabeel@nex:~$ ls -l sample.txt
-rw-rwxrwx 1 nabeel nabeel 0 Nov  8 23:02 sample.txt
nabeel@nex:~$ chmod u-r sample.txt
nabeel@nex:~$ ls -l sample.txt
--w-rwxrwx 1 nabeel nabeel 0 Nov  8 23:02 sample.txt
nabeel@nex:~$
```

Changing Ownership and Group

For changing the ownership of a file/directory, you can use the following command:

chown user

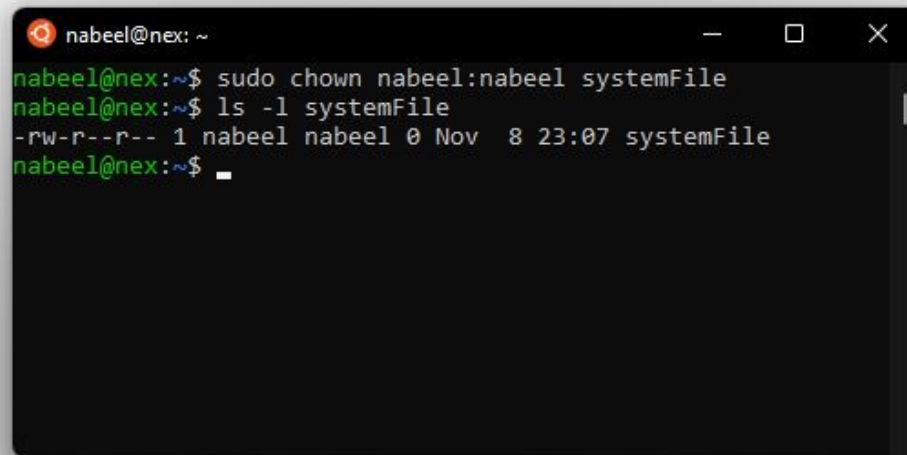
In case you want to change the user as well as group for a file or directory use the command

chown user:group filename

The 'chgrp' command can change the group ownership **chgrp group filename**

```
nabeel@nex: ~  
nabeel@nex:~$ sudo touch systemFile  
nabeel@nex:~$ ls -l  
total 16  
drwxr-xr-x 2 root    root    4096 Nov  8 22:29 Downloads  
drwxr-xr-x 2 nabeel  nabeel  4096 Nov  8 22:02 folder1  
drwxr-xr-x 2 nabeel  nabeel  4096 Nov  8 22:02 folder2  
drwxr-xr-x 2 nabeel  nabeel  4096 Nov  8 22:02 folder3  
-rwxrw---- 1 nabeel  nabeel    0 Oct 27 20:15 new.txt  
-rw-r--r-- 1 root    root      0 Nov  8 23:06 rootFile  
--w-rwxrwx 1 nabeel  nabeel    0 Nov  8 23:02 sample.txt  
-rw-r--r-- 1 root    root      0 Nov  8 23:07 systemFile  
nabeel@nex:~$
```

```
nabeel@nex: ~  
-rw-r--r-- 1 root    root      0 Nov  8 23:07 systemFile  
nabeel@nex:~$ sudo chown nabeel systemFile  
nabeel@nex:~$ ls -l  
total 16  
drwxr-xr-x 2 root    root    4096 Nov  8 22:29 Downloads  
drwxr-xr-x 2 nabeel  nabeel  4096 Nov  8 22:02 folder1  
drwxr-xr-x 2 nabeel  nabeel  4096 Nov  8 22:02 folder2  
drwxr-xr-x 2 nabeel  nabeel  4096 Nov  8 22:02 folder3  
-rwxrw---- 1 nabeel  nabeel    0 Oct 27 20:15 new.txt  
-rw-r--r-- 1 root    root      0 Nov  8 23:06 rootFile  
--w-rwxrwx 1 nabeel  nabeel    0 Nov  8 23:02 sample.txt  
-rw-r--r-- 1 nabeel  root      0 Nov  8 23:07 systemFile  
nabeel@nex:~$
```

A terminal window with a dark background and light green text. The window title bar shows 'nabeel@nex: ~' and standard window controls. The terminal displays the execution of 'sudo chown nabeel:nabeel systemFile' and 'ls -l systemFile', resulting in permissions '-rw-r--r--' for the file.

```
nabeel@nex: ~  
nabeel@nex:~$ sudo chown nabeel:nabeel systemFile  
nabeel@nex:~$ ls -l systemFile  
-rw-r--r-- 1 nabeel nabeel 0 Nov  8 23:07 systemFile  
nabeel@nex:~$
```


LAB SESSION # 4

OBJECT: Input/Output Redirection In Linux

What is Redirection?

Redirection is a feature in Linux such that when executing a command, you can change the standard input/output devices. The basic workflow of any Linux command is that it takes an input and give an output.

- The standard input (stdin) device is the keyboard.
- The standard output (stdout) device is the screen.

With redirection, the above standard input/output can be changed.

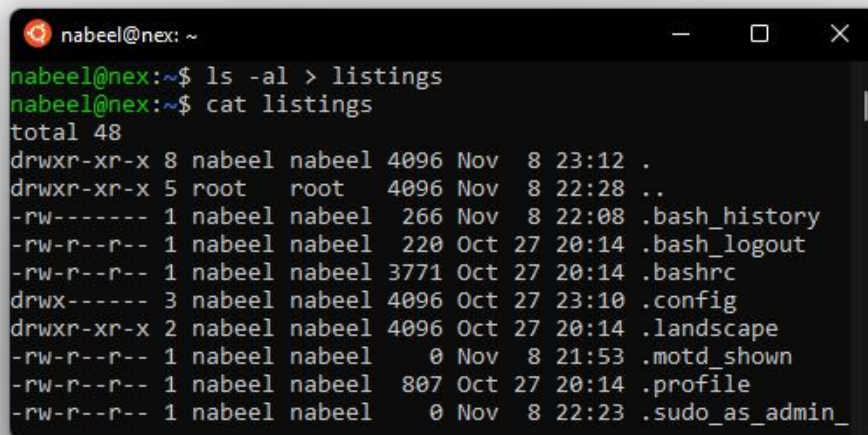
Output Redirection

The '>' symbol is used for output (STDOUT) redirection.

Example:

```
ls -al > listings
```

Here the output of command `ls -al` is re-directed to file "listings" instead of your screen.

A terminal window titled 'nabeel@nex: ~' with standard window controls. It shows the execution of 'ls -al > listings' followed by 'cat listings'. The output of 'cat listings' is displayed, showing the detailed output of 'ls -al' for the current directory, including permissions, owner, group, size, date, and file names.

```
nabeel@nex: ~  
nabeel@nex:~$ ls -al > listings  
nabeel@nex:~$ cat listings  
total 48  
drwxr-xr-x 8 nabeel nabeel 4096 Nov  8 23:12 .  
drwxr-xr-x 5 root   root   4096 Nov  8 22:28 ..  
-rw----- 1 nabeel nabeel  266 Nov  8 22:08 .bash_history  
-rw-r--r-- 1 nabeel nabeel  220 Oct 27 20:14 .bash_logout  
-rw-r--r-- 1 nabeel nabeel 3771 Oct 27 20:14 .bashrc  
drwx----- 3 nabeel nabeel 4096 Oct 27 23:10 .config  
drwxr-xr-x 2 nabeel nabeel 4096 Oct 27 20:14 .landscape  
-rw-r--r-- 1 nabeel nabeel    0 Nov  8 21:53 .motd_shown  
-rw-r--r-- 1 nabeel nabeel  807 Oct 27 20:14 .profile  
-rw-r--r-- 1 nabeel nabeel    0 Nov  8 22:23 .sudo_as_admin_
```

Note: Use the correct file name while redirecting command output to a file. If there is an existing file with the same name, the redirected command will delete the contents of that file and then it may be overwritten." If you do not want a file to be overwritten but want to add more content to an existing file, then you should use '>>' operator.

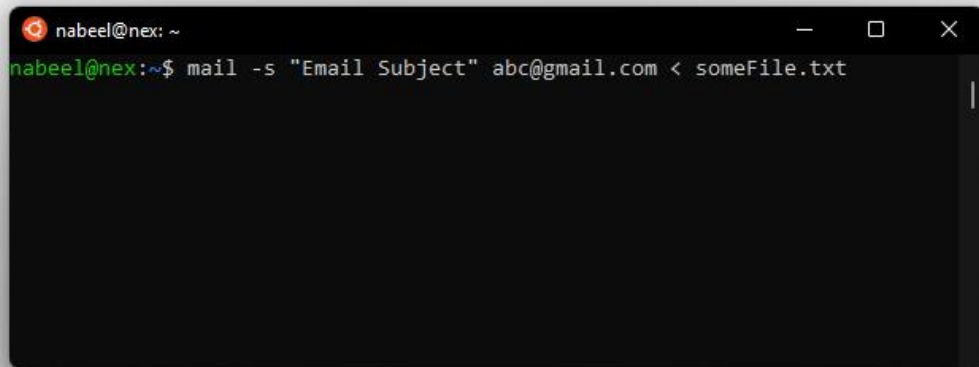
Input redirection

The '<' symbol is used for input(STDIN) redirection

Example: The mail program in Linux can help you send emails from the Terminal.

You can type the contents of the email using the standard device keyboard. But if you want to attach a File to email you can use the input re-direction operator in the following format.

mail -s "Subject" to-address < Filename



This would attach the file with the email, and it would be sent to the recipient.

The above examples were simple. Let's look at some advance re-direction techniques which make use of File Descriptors.

File Descriptors (FD)

In Linux/Unix, everything is a file. Regular file, Directories, and even Devices are files. Every File has an associated number called File Descriptor (FD).

Your screen also has a File Descriptor. When a program is executed the output is sent to File Descriptor of the screen, and you see program output on your monitor. If the output is sent to File Descriptor of the printer, the program output would have been printed.

Error Redirection

Whenever you execute a program/command at the terminal, 3 files are always open, viz., standard input, standard output, standard error. These files are always present whenever a program is run. As explained before a file descriptor, is associated with each of these files.

File	File Descriptor
Standard Input STDIN	0
Standard Output STDOUT	1

Standard Error STDERR	2
-----------------------	---

By default, error stream is displayed on the screen. Error redirection is routing the errors to a file other than the screen.

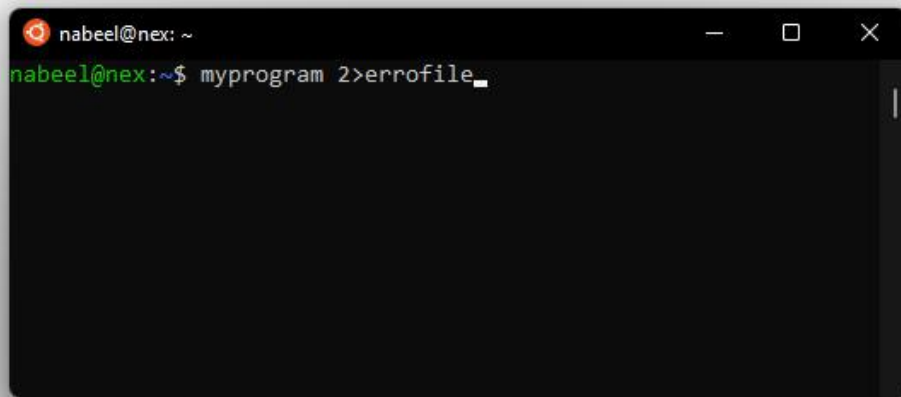
Why Error Redirection?

Error re-direction is one of the very popular features of Unix/Linux. Frequent UNIX users will reckon that many commands give you massive amounts of errors. For instance, while searching for files, one typically gets permission denied errors. These errors usually do not help the person searching for a particular file.

While executing shell scripts, you often do NOT want error messages cluttering up the normal program output. The solution is to re-direct the error messages to a file.

Example 1

```
$ myprogram 2>errorsfile
```



Above we are executing a program names myprogram.

The file descriptor for standard error is 2.

Using "2>" we re-direct the error output to a file named "errorfile"

Thus, program output is not cluttered with errors.

Example 2

Here is another example which uses cat statement -

```
Cat filename.ext 2>error.log
```

```
nabeel@nex: ~  
nabeel@nex:~$ #showing the content of a file that does not exist  
nabeel@nex:~$  
nabeel@nex:~$  
nabeel@nex:~$ cat someFile.txt  
cat: someFile.txt: No such file or directory  
nabeel@nex:~$
```

```
nabeel@nex: ~  
nabeel@nex:~$  
nabeel@nex:~$ cat someFile.txt  
cat: someFile.txt: No such file or directory  
nabeel@nex:~$  
nabeel@nex:~$  
nabeel@nex:~$ #we are getting the error here, now redirecting it ot a file  
nabeel@nex:~$ cat someFile.txt 2>error.log  
nabeel@nex:~$
```

```
nabeel@nex: ~  
nabeel@nex:~$ #we are getting the error here, now redirecting it ot a file  
nabeel@nex:~$ cat someFile.txt 2>error.log  
nabeel@nex:~$  
nabeel@nex:~$ #checking the content of error.log file  
nabeel@nex:~$  
nabeel@nex:~$ cat error.log  
cat: someFile.txt: No such file or directory  
nabeel@nex:~$
```


LAB SESSION # 5

OBJECT: Linux/Unix Process Management

What is a Process?

An instance of a program is called a Process. In simple terms, any command that you give to your Linux machine starts a new process.

- Having multiple processes for the same program is possible.

Types of Processes:

- Foreground Processes: They run on the screen and need input from the user. For example, Office Programs
- Background Processes: They run in the background and usually do not need user input. For example, Antivirus.

1. Bg/Fg

Running a Foreground Process

To start a foreground process, you can either run it from the dashboard, or you can run it from the terminal. When using the Terminal, you will have to wait, until the foreground process runs.

Running a Background process

If you start a foreground program/process from the terminal, then you cannot work on the terminal, till the program is up and running. Particular, data-intensive tasks take lots of processing power and may even take hours to complete. You do not want your terminal to be held up for such a long time. To avoid such a situation, you can run the program and send it to the background so that terminal remains available to you. Let's learn how to do this -

Example:

1. Launch 'banshee' music player
2. Stop it with the 'ctrl +z' command
3. Continue it with the 'fg' utility.

2. Top

This utility tells the user about all the running processes on the Linux machine.

```
nabeel@nex: ~
nabeel@nex:~$ top
top - 18:29:59 up 28 min,  0 users,  load average: 0.00, 0.00, 0.00
Tasks:  5 total,   1 running,  4 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,   0.0 sy,   0.0 ni,100.0 id,   0.0 wa,   0.0 hi,   0.0 si,   0.0 st
MiB Mem :  5897.3 total,  5747.0 free,   67.4 used,   82.9 buff/cache
MiB Swap:  2048.0 total,  2048.0 free,    0.0 used.  5667.2 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+  COMMAND
    1 root        20   0   1804   1188 1104 S   0.0   0.0   0:00.26 init
   142 root        20   0   1816    88    0 S   0.0   0.0   0:00.00 init
   143 root        20   0   1816   100    0 S   0.0   0.0   0:00.35 init
   144 nabeel       20   0 10032  5052 3348 S   0.0   0.1   0:00.31 bash
   176 nabeel       20   0 10864  3672 3160 R   0.0   0.1   0:00.00 top
```

Press 'q' on the keyboard to move out of the process display.
The terminology follows:

Field	Description	Example 1	Example 2
PID	The process ID of each task	1525	961
User	The username of task owner	Home	Root
PR	Priority Can be 20(highest) or -20(lowest)	20	20
NI	The nice value of a task	0	0
VIRT	Virtual memory used (kb)	1775	75972
RES	Physical memory used (kb)	100	51
SHR	Shared memory used (kb)	28	7952
S	Status There are five types: 'D' = uninterruptible sleep 'R' = running 'S' = sleeping 'T' = traced or stopped 'Z' = zombie	S	R
%CPU	% of CPU time	1.7	1.0
%MEM	Physical memory used	10	5.1

Field	Description	Example 1	Example 2
TIME+	Total CPU time	5:05.34	2:23.42
Command	Command name	Photoshop.exe	Xorg

3. PS

This command stands for 'Process Status'. It is similar to the "Task Manager" that pops up in a Windows Machine when we use Cntrl+Alt+Del. This command is similar to 'top' command but the information displayed is different.

To check all the processes running under a user, use the command -

ps ux

```

nabeel@nex:~$ ps ux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
nabeel    144  0.0  0.0  10032   5052 pts/0    Ss   18:18   0:00 -bash
nabeel    178  0.0  0.0  10620   3388 pts/0    R+   18:30   0:00 ps ux
nabeel@nex:~$

```

You can also check the process status of a single process, use the syntax -

ps PID

```

nabeel@nex:~$ ps 144
  PID TTY      STAT   TIME COMMAND
   144 pts/0    Ss      0:00 -bash
nabeel@nex:~$

```

4. Kill

This command **terminates running processes** on a Linux machine.

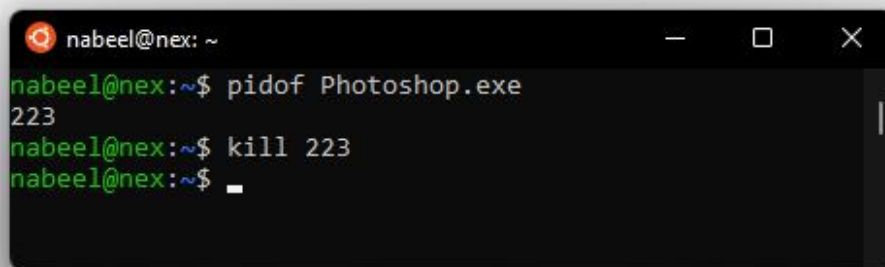
To use these utilities you need to know the PID (process id) of the process you want to kill

Syntax - **kill PID**

To find the PID of a process simply type

pidof Process name

Let us try it with an example.

A terminal window with a dark background and light green text. The window title is 'nabeel@nex: ~'. The terminal shows the following commands and output: 'nabeel@nex:~\$ pidof Photoshop.exe' followed by '223' on the next line, then 'nabeel@nex:~\$ kill 223', and finally 'nabeel@nex:~\$' followed by a cursor. The window has standard Linux window controls (minimize, maximize, close) in the top right corner.

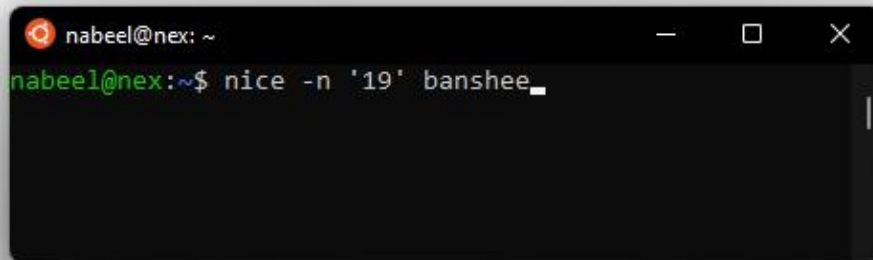
```
nabeel@nex: ~
nabeel@nex:~$ pidof Photoshop.exe
223
nabeel@nex:~$ kill 223
nabeel@nex:~$
```

5. NICE

Linux can run a lot of processes at a time, which can slow down the speed of some high priority processes and result in poor performance. To avoid this, you can tell your machine to prioritize processes as per your requirements. This priority is called Niceness in Linux, and it has a value between -20 to 19. The lower the Niceness index, the higher would be a priority given to that task. The default value of all the processes is 0.

To start a process with a niceness value other than the default value use the following syntax

nice -n 'Nice value' process name

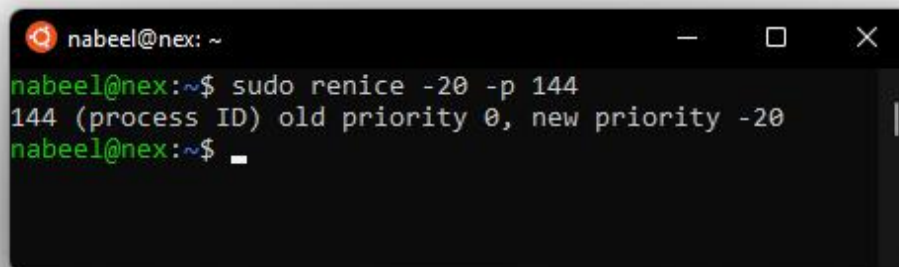


```
nabeel@nex: ~  
nabeel@nex:~$ nice -n '19' banshee_
```

If there is some process already running on the system, then you can 'Renice' its value using syntax.

renice 'nice value' -p 'PID'

To change Niceness, you can use the 'top' command to determine the PID (process id) and its Nice value. Later use the renice command to change the value. Let us understand this by an example.



```
nabeel@nex: ~  
nabeel@nex:~$ sudo renice -20 -p 144  
144 (process ID) old priority 0, new priority -20  
nabeel@nex:~$
```

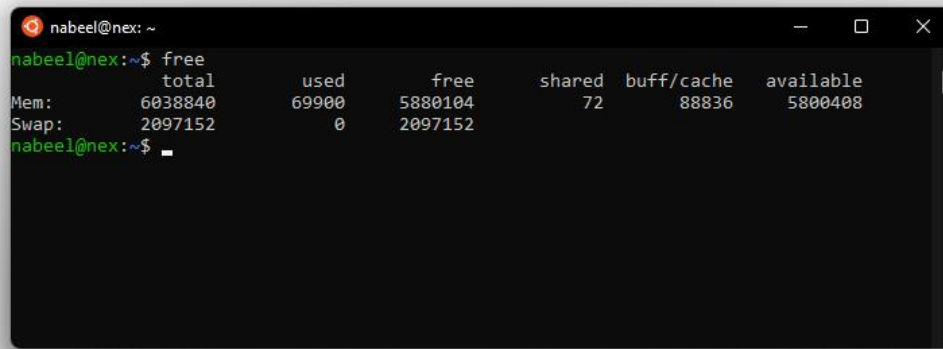
6. DF

This utility reports the free disk space(Hard Disk) on all the file systems. If you want the information in a readable format, then use the command

'df-h'

7. Free

This command shows the free and used memory (RAM) on the Linux system.

A terminal window with a dark background and light green text. The window title is 'nabeel@nex: ~'. The prompt is 'nabeel@nex:~\$'. The command 'free' has been executed, displaying a table of memory usage statistics. The table has columns: total, used, free, shared, buff/cache, and available. The rows are for Mem (Memory) and Swap. The output shows that memory is mostly free, with a small amount used and shared. Swap space is also mostly free.

```
nabeel@nex: ~
nabeel@nex:~$ free
              total        used        free      shared  buff/cache   available
Mem:           6038840        69900       5880104          72       88836     5800408
Swap:          2097152           0       2097152
```

You can use the arguments
free -m to display output in MB
free -g to display output in GB

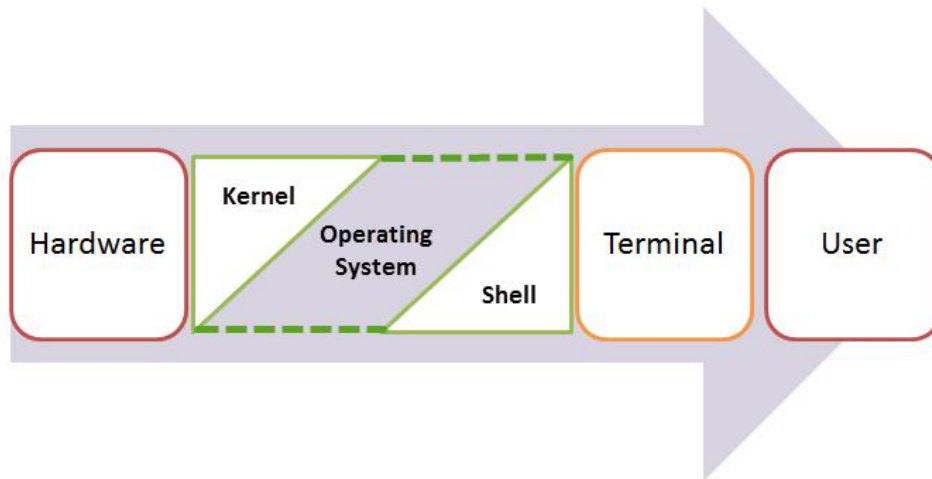
LAB SESSION # 6

OBJECT: Shell Scripting

What is a Shell?

An Operating is made of many components, but its two prime components are -

- Kernel
- Shell



A **Kernel** is at the nucleus of a computer. It makes the communication between the hardware and software possible.

While the Kernel is the innermost part of an operating system, a shell is the outermost one. As it wraps around the delicate interior of an Operating system protecting it from accidental damage, hence it is named **Shell**.

A shell is software that provides an interface for an operating system's users to provide access to the kernel's services. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation.

A shell that is accessed through command-line-interface called **terminal**, takes input from user in the form of commands, processes it, and then gives an output. It is therefore referred to as the “command interpreter”. It is the interface through which a user can run commands, programs, and shell scripts.

The shell is much more than just a command interpreter, it is also a programming language of its own with complete programming language constructs such as variables, conditional execution, loops, functions and many more.

Types of Shell

There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions. They are divided into two major types.

- **Bourne shell** – If you are using a Bourne-type shell, the **\$** character is the default prompt.

The Bourne Shell has the following subcategories –

- Bourne shell (sh)
 - Korn shell (ksh)
 - Bourne Again shell (**bash**) -- *Default shell for most Linux distributions*
 - POSIX shell (sh)
-
- **C shell** – If you are using a C-type shell, the **%** character is the default prompt. The different C-type shells follow –
 - C shell (csh)
 - TENEX/TOPS C shell (tcsh)

What is Shell Scripting?

Shell scripting is writing a series of command for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script, which can be stored as file and executed anytime.

A shell script or shell program has its own syntax like other programming languages. It allows user to define variables, assign various values, and so on. Complex programs may contain conditional statements, loops, and functions.

Let us understand the steps in creating a Shell Script –

1. **Create a file** using a vi editor(or any other editor). Name script file with extension **.sh**
2. Start the script with **#!/bin/sh**
3. Write some code.
4. Save the script file as **filename.sh**
5. For executing the script type **bash filename.sh**

The bang line, in step 2, tells the kernel that a specific shell or language is to be used to interpret the contents of the file. This is the line which is written at the beginning of every program. It starts with the bang character (**#!**) and goes for example like this:

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

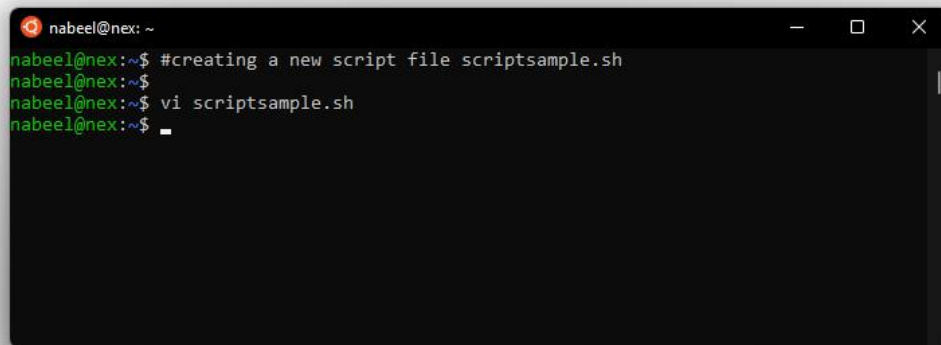
This is the link to the Bourne again shell(bash). The instructions written after this bang line will be interpreted using the above named shell.

Let's create a small script -

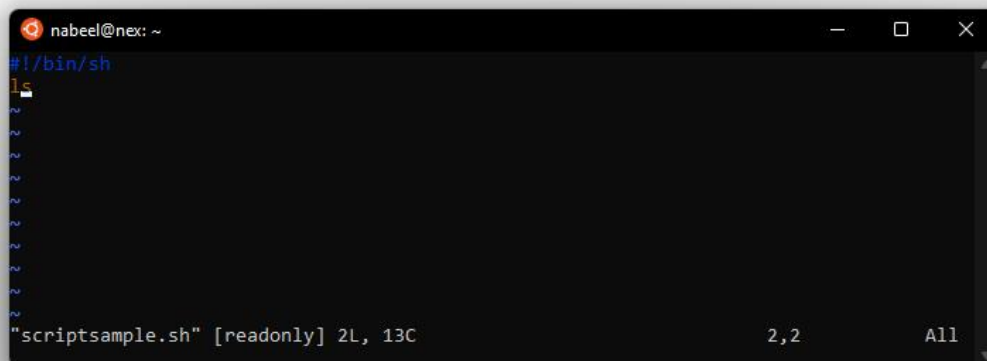
```
#!/bin/sh
```

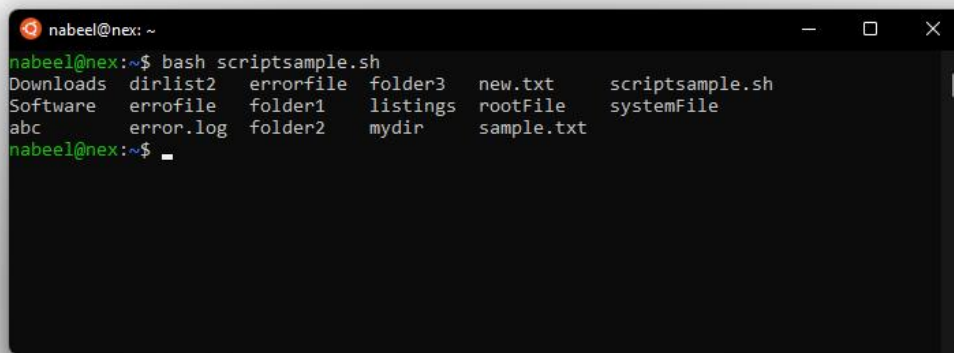
```
ls
```

Let's see the steps to create it -

A terminal window titled 'nabeel@nex: ~' showing the following commands and output:

```
nabeel@nex:~$ #creating a new script file scriptsample.sh
nabeel@nex:~$
nabeel@nex:~$ vi scriptsample.sh
nabeel@nex:~$
```

A terminal window titled 'nabeel@nex: ~' showing the execution of the script. The first line is the shebang `#!/bin/sh`. The second line is `ls`, which has been executed, resulting in a vertical list of tilde characters (~) representing the contents of the current directory. At the bottom of the window, the status bar shows: `"scriptsample.sh" [readonly] 2L, 13C` on the left, `2,2` in the center, and `All` on the right.

A terminal window titled 'nabeel@nex: ~' showing the execution of a script. The prompt is 'nabeel@nex:~\$' followed by 'bash scriptsample.sh'. The output is a 3x6 grid of files and folders: Downloads, Software, abc, dirlist2, errofile, error.log, errorfile, folder1, folder2, folder3, listings, mydir, new.txt, rootFile, sample.txt, scriptsample.sh, and systemFile. The prompt returns to 'nabeel@nex:~\$'.

Command 'ls' is executed when we execute the scrip **sample.sh** file.

Commenting is important in any program. In Shell programming, the syntax to add a comment is

#comment

What are Shell Variables?

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

Defining Variables: Variables are defined as follows –

Syntax: **variable_name=variable_value**

Example: **DEPARTMENT="Computer Science"**

Accessing Values: To access the value stored in a variable, prefix its name with the dollar sign (\$)

Example: **echo \$DEPARTMENT**

Unsetting Variables: Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks.

Syntax: **unset variable_name**

Example: **unset NAME**

the following creates a shell variable and then prints it:

variable="Hello"

echo \$variable

unset variable

echo \$variavle

Below is a small script which will use a variable. Type it, execute it, and understand it.

```
#!/bin/sh
echo "what is your name?"
read name
echo "How do you do, $name?"
read remark
echo "I am $remark too!"
```

Positional parameters

The shell has knowledge of a special kind of variable called a positional parameter. Positional parameters are used to refer to the parameters that were passed to a shell program on the command line or a shell function by the shell script that invoked the function. When you run a shell program that requires or supports a number of command-line options, each of these options is stored into a positional parameter. The first parameter is stored into a variable named **1**, the second parameter is stored into a variable named **2**, and so forth. To access the values stored in these variables, you must precede the variable name with a dollar sign (\$) just as you do with variables you define.

The following shell program expects to be invoked with two parameters. The program takes the two parameters and prints the second parameter that was typed on the command line first and the first parameter that was typed on the command line second.

```
# program reverse, prints the command line parameters out in reverse order
echo "$2" "$1"
```

If you invoked this program by entering:

reverse hello there

The program would return the following output:

there hello

Built-in Variables

These are special variables that Linux provides that can be used to make decisions in a program. Their values cannot be modified. The following table lists these variables and gives a brief description of what each is used for.

Variable	Usage
\$0	Stores the first word of the entered command (the name of the shell program).
\$n	These variables correspond to the arguments with which a script was

	invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is \$1, the second argument is \$2, and so on).
\$#	Stores the number of command-line arguments that were passed to the shell program.
\$?	Stores the exit value of the last command that was executed.
\$*	Stores all the arguments that were entered on the command line (\$1 \$2 ...).
"\$@"	Stores all the arguments that were entered on the command line, individually quoted ("\$1" "\$2" ...).

LAB SESSION # 7

OBJECT: Arrays for Shell

Following is the simplest method of creating an array variable. This helps assign a value to one of its indices.

```
array_name[index]=value  
NAME[0]="Nabeel"  
NAME[1]="Qadir"  
NAME[2]="Mahnaz"  
NAME[3]="Ayan"  
NAME[4]="Daisy"
```

```
array_name=(value1 ... valuen)
```

Accessing Array Values

After you have set any array variable, you access it as follows –

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

You can access all the items in an array in one of the following ways –

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

```
#!/bin/sh
```

```
NAME[0]="Nabeel"  
NAME[1]="Qadir"  
NAME[2]="Mahnaz"  
NAME[3]="Ayan"  
NAME[4]="Daisy"  
echo "First Index: ${NAME[0]}"  
echo "Second Index: ${NAME[1]}"  
echo "Complete Array First Method: ${NAME[*]}"  
echo "Complete Array Second Method: ${NAME[@]}"
```


LAB SESSION # 8

OBJECT: Shell Basic Operators

There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) in this chapter.

We will now discuss the following operators –

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

```
#!/bin/sh
```

```
val=`expr 2 + 2`  
echo "Total value : $val"
```

The following points need to be considered while adding –

- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between ‘`, called the backtick.

Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then –

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr \$a + \$b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr \$a - \$b` will give -10

* (Multiplication)	Multiplies values on either side of the operator	`expr \$a * \$b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr \$b / \$a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr \$b % \$a` will give 0
= (Assignment)	Assigns right operand in left operand	a = \$b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[\$a == \$b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[\$a != \$b] would return true.

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example [\$a == \$b] is correct whereas, [\$a==\$b] is incorrect.

All the arithmetical calculations are done using long integers.

Example

Here is an example which uses all the arithmetic operators –

```
#!/bin/sh
```

```
a=10
```

```
b=20
```

```
val=`expr $a + $b`  
echo "a + b : $val"
```

```
val=`expr $a - $b`  
echo "a - b : $val"
```

```
val=`expr $a \* $b`  
echo "a * b : $val"
```

```
val=`expr $b / $a`  
echo "b / a : $val"
```

```
val=`expr $b % $a`
```

```
echo "b % a : $val"
```

```
if [ $a == $b ]  
then  
    echo "a is equal to b"  
fi
```

```
if [ $a != $b ]  
then  
    echo "a is not equal to b"  
fi
```

The above script will produce the following result –

```
a + b : 30  
a - b : -10  
a * b : 200  
b / a : 2  
b % a : 0  
a is not equal to b
```

The following points need to be considered when using the Arithmetic Operators –

- There must be spaces between the operators and the expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- Complete expression should be enclosed between ‘`’`, called the inverted commas.
- You should use `\` on the `*` symbol for multiplication.
- **if...then...fi** statement is a decision-making statement which has been explained in the next section.

Shell Relational Operators

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

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	[\$a -ne \$b] is true.

	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.

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, [\$a <= \$b] is correct whereas, [\$a <= \$b] is incorrect.

Example

Here is an example which uses all the relational operators –

```
#!/bin/sh
```

```
a=10
```

```
b=20
```

```
if [ $a -eq $b ]
```

```
then
```

```
    echo "$a -eq $b : a is equal to b"
```

```
else
```

```
    echo "$a -eq $b: a is not equal to b"
```

```
fi
```

```
if [ $a -ne $b ]
```

```
then
```

```
    echo "$a -ne $b: a is not equal to b"
```

```
else
```

```
    echo "$a -ne $b : a is equal to b"
```

```
fi
```

```
if [ $a -gt $b ]
```

```

then
    echo "$a -gt $b: a is greater than b"
else
    echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
    echo "$a -lt $b: a is less than b"
else
    echo "$a -lt $b: a is not less than b"
fi

if [ $a -ge $b ]
then
    echo "$a -ge $b: a is greater or equal to b"
else
    echo "$a -ge $b: a is not greater or equal to b"
fi

if [ $a -le $b ]
then
    echo "$a -le $b: a is less or equal to b"
else
    echo "$a -le $b: a is not less or equal to b"
fi

```

The above script will generate the following result –

```

10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or equal to b

```

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 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 condition becomes true.	[\$a -lt 20 -o \$b -gt 100] 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.

Example

Here is an example which uses all the Boolean operators –

```
#!/bin/sh
```

```
a=10
```

```
b=20
```

```
if [ $a != $b ]
```

```
then
```

```
    echo "$a != $b : a is not equal to b"
```

```
else
```

```
    echo "$a != $b: a is equal to b"
```

```
fi
```

```
if [ $a -lt 100 -a $b -gt 15 ]
```

```
then
```

```
    echo "$a -lt 100 -a $b -gt 15 : returns true"
```

```
else
```

```
    echo "$a -lt 100 -a $b -gt 15 : returns false"
```

```
fi
```

```
if [ $a -lt 100 -o $b -gt 100 ]
```

```
then
```

```
    echo "$a -lt 100 -o $b -gt 100 : returns true"
```

```
else
```

```
    echo "$a -lt 100 -o $b -gt 100 : returns false"
```

```
fi
```

```
if [ $a -lt 5 -o $b -gt 100 ]
```

```
then
```

```
    echo "$a -lt 100 -o $b -gt 100 : returns true"
```

```
else
```

```
    echo "$a -lt 100 -o $b -gt 100 : returns false"
```

```
fi
```

The above script will generate the following result –

10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false

String Operators

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then –

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[\$a = \$b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[\$a != \$b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z \$a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n \$a] is not false.
str	Checks if str is not the empty string; if it is empty, then it returns false.	[\$a] is not false.

Example

Here is an example which uses all the string operators –

[Live Demo](#)

```
#!/bin/sh
```

```
a="abc"
```

```
b="efg"
```

```
if [ $a = $b ]
```

```
then
```

```
    echo "$a = $b : a is equal to b"
```

```
else
```

```
    echo "$a = $b: a is not equal to b"
```

```
fi
```

```
if [ $a != $b ]
```

```
then
```

```
    echo "$a != $b : a is not equal to b"
```

```
else
```

```
    echo "$a != $b: a is equal to b"
```



```

fi

if [ -z $a ]
then
    echo "-z $a : string length is zero"
else
    echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
    echo "-n $a : string length is not zero"
else
    echo "-n $a : string length is zero"
fi

if [ $a ]
then
    echo "$a : string is not empty"
else
    echo "$a : string is empty"
fi

```

The above script will generate the following result –

```

abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty

```

File Test Operators

We have a few operators that can be used to test various properties associated with a Unix file.

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	[-c \$file] is false.

the condition becomes true.

-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.

Example

The following example uses all the **file test** operators –

Assume a variable file holds an existing file name **"/home/testuser/test.sh"** the size of which is 100 bytes and has **read**, **write** and **execute** permission –

```
#!/bin/sh
```

```
file="/home/testuser/test.sh"
```

```
if [ -r $file ]
```

```
then
```

```
    echo "File has read access"
```

```
else
```

```
    echo "File does not have read access"
```

```
fi
```


LAB SESSION # 9

OBJECT: Decision making (if - else)

Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here –

- The **if...else** statement
- The **case...esac** statement

The if...else statements

nix Shell supports following forms of **if...else** statement –

- [if...fi statement](#)
- [if...else...fi statement](#)
- [if...elif...else...fi statement](#)

Syntax

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

Example

```
#!/bin/sh
```

```
a=10
b=20
```

```
if [ $a == $b ]
then
    echo "a is equal to b"
fi
```

```
if [ $a != $b ]
then
    echo "a is not equal to b"
fi
```

Syntax

```
if [ expression ]
then
    Statement(s) to be executed if expression is true
else
    Statement(s) to be executed if expression is not true
fi
```

Example

The above example can also be written using the *if...else* statement as follows –

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
    echo "a is equal to b"
else
    echo "a is not equal to b"
fi
```

Example

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
    echo "a is equal to b"
elif [ $a -gt $b ]
then
    echo "a is greater than b"
elif [ $a -lt $b ]
then
    echo "a is less than b"
else
    echo "None of the condition met"
fi
```


LAB SESSION # 10

OBJECT: Shell Loops

The **while** loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax

```
while command
do
    Statement(s) to be executed if command is true
done
```

Here the Shell *command* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *command* is *false* then no statement will be executed and the program will jump to the next line after the done statement.

Example

Here is a simple example that uses the **while** loop to display the numbers zero to nine –

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
    echo $a
    a=`expr $a + 1`
done
```

The **for** loop operates on lists of items. It repeats a set of commands for every item in a list.

Syntax

```
for var in word1 word2 ... wordN
do
    Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable *var* is set to the next word in the list of words, word1 to wordN.

Example

Here is a simple example that uses the **for** loop to span through the given list of numbers –

[Live Demo](#)

```
#!/bin/sh

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

Following is the example to display all the files starting with **.bash** and available in your home. We will execute this script from my root –

```
#!/bin/sh

for FILE in $HOME/.bash*
do
    echo $FILE
done
```

Until Loop

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax

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

Here the Shell *command* is evaluated. If the resulting value is *false*, given *statement(s)* are executed. If the *command* is *true* then no statement will be executed and the program jumps to the next line after the done statement.

Example

Here is a simple example that uses the until loop to display the numbers zero to nine –

```
#!/bin/sh

a=0

until [ ! $a -lt 10 ]
do
    echo $a
    a=`expr $a + 1`
done
```

The break statement

Example

Here is a simple example which shows that loop terminates as soon as **a** becomes 5 –

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
    echo $a
    if [ $a -eq 5 ]
    then
        break
    fi
    a=`expr $a + 1`
done
```

The continue statement

Example

The following loop makes use of the **continue** statement which returns from the continue statement and starts processing the next statement –

[Live Demo](#)

```
#!/bin/sh
```

```
NUMS="1 2 3 4 5 6 7"
```

```
for NUM in $NUMS
```

```
do
```

```
    Q=`expr $NUM % 2`
```

```
    if [ $Q -eq 0 ]
```

```
    then
```

```
        echo "Number is an even number!!"
```

```
        continue
```

```
    fi
```

```
    echo "Found odd number"
```

```
done
```


LAB SESSION # 11

OBJECT: Demonstrate multi-threading using POSIX thread (Pthread) Library

POSIX Introduction:

The POSIX thread libraries are a standard based thread API for C/C++. It allows one to spawn a new

concurrent process flow. It is most effective on multi-processor or multi-core systems where the process

flow can be scheduled to run on another processor thus gaining speed through parallel or distributed

processing. Threads require less overhead than “forking” or “spawning” a new process because the

system does not initialize a new system virtual memory space and environment for the process. While

most effective on multiprocessor systems, gains also found on uniprocessor systems which exploit latency

in I/O and other system functions which may halt process execution. (One thread may execute while

another is waiting for I/O or some other system latency). All threads within a process share the same

address space. A thread is created by defining a function and its arguments which will be processed in the

thread. The purpose of using the POSIX thread library in your software is to execute software faster.

Thread Basic:

- Thread operations include thread creation, termination, synchronization (joins, blocking),

scheduling, data management and process interaction.

- A thread does not maintain a list of created threads, nor does it know the thread that created it.

- All threads within a process share the same address space.

- Threads in the same process share:

- o Process instructions
- o Most data
- o open files (descriptors)
- o signals and signal handlers
- o current working directory
- o User and group id
- Each thread has a unique:
 - o Thread ID
 - o set of registers, stack pointer
 - o stack for local variables, return addresses
 - o signal mask
 - o priority
- o Return value: errno, ...pthread functions return "0" if OK.

The Pthreads API:

The current POSIX standard is defined only for the C language. For portability, the pthread.h header file should

be included in each source file using the pthreads library. As naming conventions; all identifiers in the

pthreads library begin with pthread_. The subroutines which comprise the pthreads API can be informally grouped into four major groups:

1. Thread Management: Routines that work directly on threads - creating, detaching, joining, etc.

They also include functions to set/query thread attributes (joinable, scheduling etc.)

2. Mutexes: Routines that deal with synchronization, called a "mutex", which is an abbreviation for

"mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking

mutexes. These are supplemented by mutex attribute functions that set or modify attributes

associated with mutexes.

3. Condition variables: Routines that address communications between threads that share a mutex.

Based upon programmer specified conditions. This group includes functions to create, destroy,

wait and signal based upon specified variable values. Functions to set/query condition variable

attributes are also included.

4. Synchronization: Routines that manage read/write locks and barriers.

Creating & Terminating Threads:

Routines:

- o `pthread_create (thread,attr,start_routine,arg)`

- o `pthread_exit (status)`

- o `pthread_cancel (thread)`

- o `pthread_attr_init (attr)`

- o `pthread_attr_destroy (attr)`

Creating Threads:

Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by

the programmer.

`pthread_create (thread,attr,start_routine,arg)` creates a new thread and makes it executable.

This routine can be called any number of times from anywhere within your code.

`pthread_create` arguments:

- o `thread`: An opaque, unique identifier for the new thread returned by the subroutine.

- o `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.

- o `start_routine`: the C routine that the thread will execute once it is created.

- o `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.

Thread Attributes:

- By default, a thread is created with certain attributes. Some of these attributes can be changed

by the programmer via the thread attribute object.

- `pthread_attr_init` and `pthread_attr_destroy` are used to initialize/destroy the thread attribute object.
- Other routines are then used to query/set specific attributes in the thread attribute object.

Attributes include: Detached or joinable state, Scheduling policy, Scheduling parameters,

Scheduling contention scope, Stack size, Stack address, Stack guard (overflow) size.

Terminating Threads:

There are several ways in which a thread may be terminated:

- o The thread returns normally from its starting routine. Its work is done.
- o The thread makes a call to the `pthread_exit()` subroutine - whether its work is done or not.
- o The thread is canceled by another thread via the `pthread_cancel()` routine.
- o The entire process is terminated due to making a call to either the `exec()` or `exit()`.
- o If `main()` finishes first, without calling `pthread_exit()` explicitly itself.

Discussion on `pthread_exit()`:

- The `pthread_exit()` routine allows the programmer to specify an optional termination status parameter. This optional parameter is typically returned to threads "joining" the terminated thread.
- In subroutines that execute to completion normally, you can often dispense with calling `pthread_exit()` - unless, of course, you want to pass the optional status code back.
- Cleanup: the `pthread_exit()` routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

Discussion on calling `pthread_exit()` from `main()`:

- There is a definite problem if main () finishes before the threads it spawned if you don't

call pthread_exit() explicitly. All of the threads it created will terminate because main ()

is done and no longer exists to support the threads.

- By having main () explicitly call pthread_exit() as the last thing it does, main() will block

and be kept alive to support the threads it created until they are done.

Compiling Threaded Programs

- For GNU linux: gcc -pthread filename.c

EXERCISES

a) Practice following simple code that creates 5 threads with the pthread_create() routine.

Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit(). Attach the snapshot of output window.

```
#include <pthread.h>

#include <stdio.h>

#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid;

    tid = (long)threadid;

    printf("Hello World! It's me, thread #%ld!\n", tid);

    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];

    int rc;

    long t;

    for(t=0; t<NUM_THREADS; t++){
```

```

printf("In main: creating thread %ld\n", t);
rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
if (rc){
printf("ERROR; return code from pthread_create() is %d\n", rc);
exit(-1);
}
}

/* Last thing that main() should do */
pthread_exit(NULL);
}

```

b) Write a piece of code that demonstrates the thread argument passing in `pthread_create()`

method. The code should explain how to pass a simple integer to each thread. The calling thread

uses a unique data structure for each thread, ensuring that each thread's argument remains intact

throughout the program. Also attach the print out of code and snapshot of output window.

code fragment demonstrates how to pass a simple integer to each thread.

```

long taskids[NUM_THREADS];
for(t=0; t<NUM_THREADS; t++)
{
taskids[t] = t;
printf("Creating thread %ld\n", t);
rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
...
}

```


LAB SESSION # 12

OBJECT: Simulation of First Fit and Best Fit Memory Allocation Algorithms

using C Language

THEORY

Memory management is the act of managing computer memory at the system level. The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed. This is critical to any advanced computer system where more than a single process might be underway at any time.

The memory manager is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory.

Since primary memory can be space-multiplexed, the memory manager can allocate a portion of primary memory to "each process" for its own use. Describing the strategies below:

1. First Fit

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

Advantage: Fastest algorithm because it searches as little as possible.

Disadvantage: The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

2. Best Fit

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

Advantage: Memory utilization is much better than first fit as it searches the smallest free partition first available.

Disadvantage: It is slower and may even tend to fill up memory with tiny useless holes.

3. Worst fit

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

Advantage: Reduces the rate of production of small gaps.

Disadvantage: If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

EXERCISES

1. Simulate the First Fit Memory Allocation Algorithm in C following the given steps. Also attach the print out of code and snapshot of output window. For compilation of C code, refer to steps discussed in Lab Session # 08.
 - a) Declare structures 'hole' and 'process' to hold information about set of holes and processes respectively.
 - b) Get number of holes, say 'nh' and the size of each hole.
 - c) Get number of processes, say 'np'. Also get the memory requirements for each process.
 - d) Allocate processes to holes, by examining each hole as follows:
 - i. If hole size > process size then mark process as allocated to that hole and decrement hole size by process size.
 - ii. Otherwise check the next from the set of holes.
 - e) Print the list of process and their allocated holes or unallocated status.
 - f) Print the list of holes, their actual and current availability.
 - g) Stop
2. Simulate the Best Fit Memory Allocation Algorithm in C following the given steps. Also attach the print out of code and snapshot of output window. For compilation of C code, refer to steps discussed in Lab Session # 08.
 - a) Declare structures 'hole' and 'process' to hold information about set of holes and processes respectively.
 - b) Get number of holes, say 'nh' and the size of each hole.
 - c) Get number of processes, say 'np'. Also get the memory requirements for each process.
 - d) Allocate processes to holes, by examining each hole as follows:
 - i. Sort the holes according to their sizes in ascending order.

- ii. If hole size $>$ process size then mark process as allocated to that hole and decrement hole size by process size.
- iii. Otherwise check the next from the set of sorted holes.
- e) Print the list of process and their allocated holes or unallocated status.
- f) Print the list of holes, their actual and current availability.

Stop