# Subject: Unix and Shell Programming

**Course Code: BCAC601 Semester: 6**

# Module 1

**Introduction to UNIX**

Unix is a computer Operating System which is capable of handling activities from multiple users at the same time. The development of Unix started around 1969 at AT&T Bell Labs by Ken Thompson and Dennis Ritchie.

**What is Unix?**

The Unix operating system is a set of programs that act as a link between the computer and the user.

The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the operating system or the kernel.

Users communicate with the kernel through a program known as the shell. The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

Unix was originally developed in 1969 by a group of AT&T employees Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna at Bell Labs.

There are various Unix variants available in the market. Solaris Unix, AIX, HP Unix and BSD are a few examples. Linux is also a flavor of Unix which is freely available.

Several people can use a Unix computer at the same time; hence Unix is called a multiuser system.

A user can also run multiple programs at the same time; hence Unix is a multitasking environment.

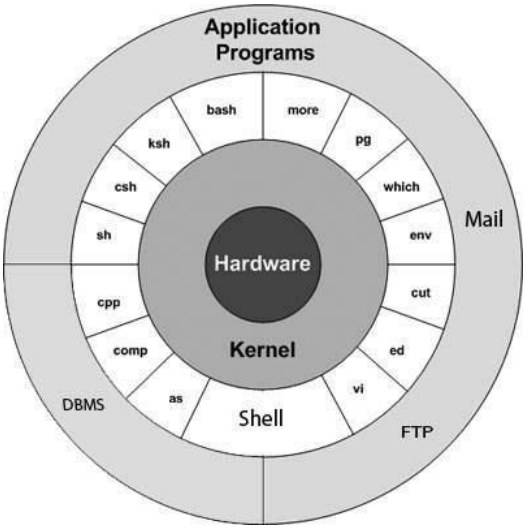
**Difference between Linux and Unix**

| Comparison | Linux | Unix |
|---|---|---|
| Definition | It is an open-source operating system which is *freely available to everyone*. | It is an operating system which *can be only used by its copyrighters*. |
| Examples | It has different distros like Ubuntu, Redhat, Fedora, etc | IBM AIX, HP-UX and Sun Solaris. |
| Users | Nowadays, Linux is in great demand. Anyone can use Linux whether a home user, developer or a student. | It was developed mainly for servers, workstations and mainframes. |
| Usage | Linux is used everywhere from servers, PC, smartphones, tablets to mainframes and | It is used in servers, workstations and PCs. |

| | | |
|---|---|---|
| | supercomputers. | |
| Cost | Linux is freely distributed, downloaded, and distributed through magazines also. And priced distros of Linux are also cheaper than Windows. | Unix copyright vendors decide different costs for their respective Unix Operating systems. |
| Development | As it is open source, it is developed by sharing and collaboration of codes by world-wide developers. | Unix was developed by AT&T Labs, various commercial vendors and non-profit organizations. |
| Manufacturer | Linux kernel is developed by the community of developers from different parts of the world. Although the father of Linux, Linus Torvalds oversees things. | Unix has three distributions IBM AIX, HP-UX and Sun Solaris. Apple also uses Unix to make OSX operating system. |
| GUI | Linux is command based but some distros provide GUI based Linux. Gnome and KDE are mostly used GUI. | Initially it was command based OS, but later Common Desktop Environment was created. Most Unix distributions use Gnome. |
| Interface | The default interface is BASH (Bourne Again SHell). But some distros have developed their own interfaces. | It originally used Bourne shell. But is also compatible with other GUIs. |
| File system support | Linux supports more file system than Unix. | It also supports file system but lesser than Linux. |
| Coding | Linux is a Unix clone, behaves like Unix but doesn't contain its code. | Unix contain a completely different coding developed by AT&T Labs. |
| Operating system | Linux is just the kernel. | Unix is a complete package of Operating system. |
| Security | It provides higher security. Linux has about 60-100 viruses listed till date. | Unix is also highly secured. It has about 85-120 viruses listed till date. |
| Error detection and solution | As Linux is open-source, whenever a user post any kind of threat, developers from all over the world start working on it. And hence, it provides faster solution. | In Unix, users have to wait for some time for the problem to be resolved. |

**Unix Architecture**

Here is a basic block diagram of a Unix system −




The main concept that unites all the versions of Unix is the following four basics −

**Kernel** − The kernel is the heart of the operating system. It interacts with the hardware and most of the tasks like memory management, task scheduling and file management.

**Shell** − The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are the most famous shells which are available with most of the Unix variants.

**Commands and Utilities** − There are various commands and utilities which you can make use of in your day-to-day activities. cp, mv, cat and grep, etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various options.

**Files and Directories** − All the data of Unix is organized into files. All files are then organized into directories. These directories are further organized into a tree-like structure called the filesystem.
In Unix a **file** is just a destination for or a source of a stream of data. Thus, a printer, for example, is a file and so is the screen. A **process** is a program that is currently running. So a process may be associated with a file.

**System calls** in Unix are used for file system control, process control, inter process communication etc. Access to the Unix kernel is only available through these system calls. Generally, system calls are similar to function calls, the only difference is that they remove the control from the user process.

**Features of UNIX**

Unix is an operating system, so it has all the features that the OS must-have. UNIX also looks at a few things in a different way than other OS. Features of UNIX are listed below:

1. Multiuser System:

Unix provides multiple programs to run and compete for the attention of the CPU. This happens in 2 ways:

i) Multiple users running multiple jobs
ii) Single user running multiple jobs

In UNIX, resources are actually shared between all the users, so-called a multi-user system. For doing so, computer give a time slice (breaking unit of time into several segments) to each user. So, at any instant of time, only one user is served but the switching is so fast that it gives an illusion that all the users are served simultaneously.

2. Multitask System:

A single user may run multiple tasks concurrently. Example: Editing a file, printing another on the printer & sending email to a person, and browsing the net too at the same time. The Kernel is designed to handle user's multiple needs.

The important thing here is that only one job can be seen running in the foreground, the rest all seems to run in the background. Users can switch between them, terminate/suspend any of the jobs.

3. The Building-Block Approach:

The Unix developers thought about keeping small commands for every kind of work. So, Unix has so many commands, each of which performs one simple job only. You can use 2 commands by using pipes ('|'). Example: $ ls | wc Here, | (pipe) connects 2 commands to create a pipeline. This command counts the number of files in the directory. These types of connected commands that can filter/manipulate data in other ways are called filters.

Nowadays, many UNIX tools are designed in a way that the output of 1 can be used as an input for the others. We can create a large number of combinations by connecting a number of tools.

4. The UNIX Toolkit:

Unix has a kernel but the kernel alone can't do much that could help the user. So, we need to use the host of applications that usually come along with the UNIX systems. The applications are quite diversified. General-purpose tools, text manipulation utilities (called filters), compilers and interpreters, networked programs, and system administration tools are all

included. With every UNIX release, new tools are being added and the older ones are modified/ removed.

5. Pattern Matching:

Unix provides very sophisticated pattern matching features. The meta-char '*' is a special character used by the system to match a number of file names. There are several other metachars in UNIX. The matching is not confined to only filename. Advanced tools use a regular expression that is framed with the characters from this set.

6. Programming Facility:

Unix provides shell which is also a programming language designed for programmers, not for casual end-users. It has all the control structures, loops, and variables required for programming purposes. These features are used to design the shell scripts (programs that can invoke the UNIX commands).
Many functions of the system can be controlled and managed by these shell scripts.

7. Documentation:

It has a 'man' command that stands for the manual, which is the most important reference for any commands and their configuration files. Apart from the offline documentation, there is a vast number of resources available on the Internet.

**What is POSIX (Portable Operating System Interface)?**

POSIX (Portable Operating System Interface) is a set of standard operating system interfaces based on the Unix operating system. The most recent POSIX specifications -- IEEE Std 1003.1-2017 -- defines a standard interface and environment that can be used by an operating system (OS) to provide access to POSIX-compliant applications. The standard also defines a command interpreter (shell) and common utility programs. POSIX supports application portability at the source code level so applications can be built to run on any POSIX compliant OS.

**A brief history of the POSIX standard**

The POSIX interfaces were originally developed under the auspices of IEEE. However, the POSIX standard is now being developed and maintained by the Austin Common Standards Revision Group, commonly referred to as the Austin Group.

The Austin Group is a joint working group made up of members from IEEE, The Open Group and Joint Technical Committee 1 of the International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). IEEE owns the POSIX trademark. The Open Group, which owns the Unix trademark, is a global consortium that develops technology standards.

**Internal and external commands**

The UNIX system is command-based i.e. things happen because of the commands that you key in. All UNIX commands are seldom more than four characters long. They are grouped into two categories:

**Internal Commands:** Commands which are built into the shell. For all the shell built-in commands, execution of the same is fast in the sense that the shell doesn't have to search the given path for them in the PATH variable, and also no process needs to be spawned for executing it.
Examples: source, cd, fg, etc.

**External Commands:** Commands which aren't built into the shell. When an external command has to be executed, the shell looks for its path given in the PATH variable, and also a new process has to be spawned and the command gets executed. They are usually located in /bin or /usr/bin. For example, when you execute the "cat" command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed.
Examples: ls, cat etc.

**Password changing (password)**

Change Password

All Unix systems require passwords to help ensure that your files and data remain your own and that the system itself is secure from hackers and crackers. Following are the steps to change your password −

Step 1 − To start, type password at the command prompt as shown below.

Step 2 − Enter your old password, the one you're currently using.

Step 3 − Type in your new password. Always keep your password complex enough so that nobody can guess it. But make sure, you remember it.

Step 4 − You must verify the password by typing it again.

```
$ passwd
Changing password for amrood
(current) Unix password:******
New UNIX password:*******
Retype new UNIX password:*******
passwd: all authentication tokens updated  successfully
```

Note − We have added asterisk (*) here just to show the location where you need to enter the current and new passwords otherwise at your system. It does not show you any character when you type.

**Knowing who are logged in (who)**

who command is used to find out the following information:
1. Time of last system boot.
2. Current run level of the system 3. List of logged in users and
   more.

Description: The who command is used to get information about currently logged in user on to system.

Syntax: $who [options] [filename]

Examples:
1. The who command displays the following information for each user currently logged in to the system if no option is provided:

Login name of the users
Terminal line numbers
Login time of the users in to system Remote
host name of the user hduser@mahesh-
Inspiron-3543:~$ who
hduser   tty7        2018-03-18 19:08 (:0)
hduser@mahesh-Inspiron-3543:~$

**whoami command**

whoami command is used both in Unix Operating System and as well as in Windows Operating System.

It is basically the concatenation of the strings “who”,”am”,”i” as whoami.
It displays the username of the current user when this command is invoked.
It is similar as running the id command with the options -un.
The earliest versions were created in 2.9 BSD as a convenience form for who am i, the Berkeley Unix who command’s way of printing just the logged in user’s identity. The GNU version was written by Richard Mlynarik and is part of the GNU Core Utilities (coreutils).

Syntax:

akfrgs@HP~: whoami

**System information using uname**

The command ‘uname‘ displays the information about the system.

Syntax:

uname [OPTION]

Options and Examples

1. -a option: It prints all the system information in the following order: Kernel name, network node hostname, kernel release date, kernel version, machine hardware name, hardware platform, operating system.

Syntax:
$uname -a

2. -s option: It prints the kernel name.

Syntax:
$uname -s

3. -n option: It prints the hostname of the network node(current computer).

Syntax:
$uname -n

4. -r option: It prints the kernel release date.

Syntax:
$uname -r

5. -v option: It prints the version of the current kernel.

Syntax:
$uname -v

6. -m option: It prints the machine hardware name.

Syntax:
$uname -m

7. -p option: It prints the type of the processor.

Syntax:
$uname  -p

8. -I option: It prints the platform of the hardware.

Syntax:
$uname  -i

9. -o option: It prints the name of the operating system.

Syntax :
$uname  -o

**File name of terminal connected to the standard input (tty)**

Linux operating system represents everything in a file system, the hardware devices that we attach are also represented as a file. The terminal is also represented as a file. There a command exists called tty which displays information related to terminal. The tty command of terminal basically prints the file name of the terminal connected to standard input. tty is short of teletype, but popularly known as a terminal it allows you to interact with the system by passing on the data (you input) to the system, and displaying the output produced by the system.

Syntax:

tty [OPTION]....

Example:
[i]  tty - -version
[ii] sudo tty

**Who is Logged in?**

Sometime you might be interested to know who is logged in to the computer at the same time. There are three commands available to get you this information, based on how much you wish to know about the other users: users, who, and w.
$ users
 amrood bablu qadir

$ who amrood ttyp0 Oct 8 14:10
(limbo) bablu  ttyp2 Oct 4 09:08
(calliope)
qadir  ttyp4 Oct 8 12:09 (dent)

$

Try the w command on your system to check the output. This lists down information associated with the users logged in the system.

**Logging Out**

When you finish your session, you need to log out of the system. This is to ensure that nobody else accesses your files.

**To log out**

Just type the logout command at the command prompt, and the system will clean up everything and break the connection.

**System Shutdown**

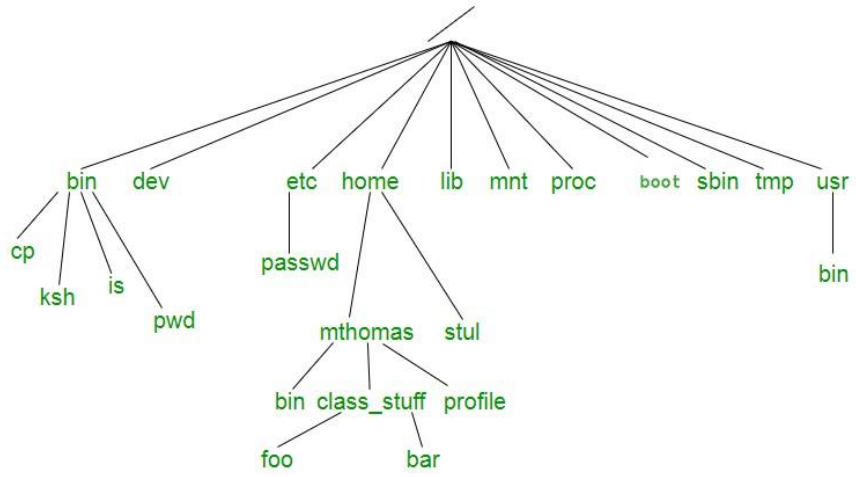
The most consistent way to shut down a Unix system properly via the command line is to use one of the following commands –

| Sr.No. | Command & Description |
|---|---|
| 1 | Halt<br>Brings the system down immediately |
| 2 | init 0<br>Powers off the system using predefined scripts to synchronize and clean up the system prior to shutting down |
| 3 | init 6<br>Reboots the system by shutting it down completely and then restarting it |
| 4 | poweroff<br>Shuts down the system by powering off |
| 5 | Reboot<br>Reboots the system |
| 6 | shutdown<br>Shuts down the system |

# Module 2

**UNIX file system**

Unix file system is a logical method of organizing and storing large amounts of information in a way that makes it easy to manage. A file is a smallest unit in which the information is stored. Unix file system has several important features. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system. Files in Unix System are organized into multi-level hierarchy structure known as a directory tree. At the very top of the file system is a directory called
"root" which is represented by a "/". All other files are "descendants" of root.




**Directories or Files and their description –**

/ : The slash / character alone denotes the root of the filesystem tree.
/bin : Stands for "binaries" and contains certain fundamental utilities, such as ls or cp, which are generally needed by all users.
/boot : Contains all the files that are required for successful booting process.
/dev : Stands for "devices". Contains file representations of peripheral devices and pseudodevices.
/etc : Contains system-wide configuration files and system databases. Originally also contained "dangerous maintenance utilities" such as init,but these have typically been moved to /sbin or elsewhere.
/home : Contains the home directories for the users.
/lib : Contains system libraries, and some critical files such as kernel modules or device drivers.
/media : Default mount point for removable devices, such as USB sticks, media players, etc.
/mnt : Stands for "mount". Contains filesystem mount points. These are used, for example, if the system uses multiple hard disks or hard disk partitions. It is also often used for remote (network) filesystems, CD-ROM/DVD drives, and so on.
/proc : procfs virtual filesystem showing information about processes as files.

/root : The home directory for the superuser “root” – that is, the system administrator. This account’s home directory is usually on the initial filesystem, and hence not in /home (which may be a mount point for another filesystem) in case specific maintenance needs to be performed, during which other filesystems are not available. Such a case could occur, for example, if a hard disk drive suffers physical failures and cannot be properly mounted.

/tmp : A place for temporary files. Many systems clear this directory upon startup; it might have tmpfs mounted atop it, in which case its contents do not survive a reboot, or it might be explicitly cleared by a startup script at boot time.

/usr : Originally the directory holding user home directories,its use has changed. It now holds executables, libraries, and shared resources that are not system critical, like the X Window System, KDE, Perl, etc. However, on some Unix systems, some user accounts may still have a home directory that is a direct subdirectory of /usr, such as the default as in Minix. (on modern systems, these user accounts are often related to server or system use, and not directly used by a person).

/usr/bin : This directory stores all binary programs distributed with the operating system not residing in /bin, /sbin or (rarely) /etc.

/usr/include : Stores the development headers used throughout the system. Header files are mostly used by the #include directive in C/C++ programming language.

/usr/lib : Stores the required libraries and data files for programs stored within /usr or elsewhere.

/var : A short for “variable.” A place for files that may change often – especially in size, for example e-mail sent to users on the system, or process-ID lock files.

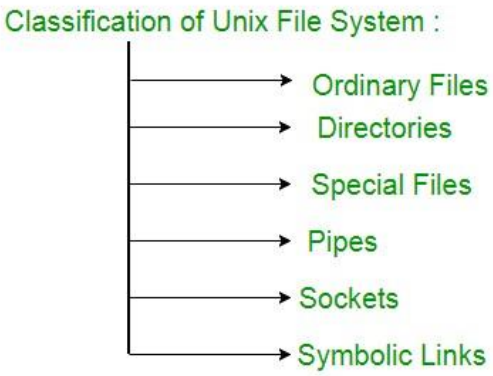
/var/log : Contains system log files.

/var/mail : The place where all the incoming mails are stored. Users (other than root) can access their own mail only. Often, this directory is a symbolic link to /var/spool/mail.

/var/spool : Spool directory. Contains print jobs, mail spools and other queued tasks.

/var/tmp : A place for temporary files which should be preserved between system reboots.

**Types of Unix files – The UNIX files system contains several different types of files:**




**1) Ordinary files** – An ordinary file is a file on the system that contains data, text, or program instructions.

Used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with.
Always located within/under a directory file. Do
not contain other files.
In long-format output of ls -l, this type of file is specified by the "-" symbol.

**2) Directories** – Directories store both special and ordinary files. For users familiar with Windows or Mac OS, UNIX directories are equivalent to folders. A directory file contains an entry for every file and subdirectory that it houses. If you have 10 files in a directory, there will be 10 entries in the directory. Each entry has two components. (1) The Filename (2) A unique identification number for the file or directory (called the inode number)

Branching points in the hierarchical tree.
Used to organize groups of files.
May contain ordinary files, special files or other directories.
Never contain "real" information which you would work with (such as text). Basically, just used for organizing files.
All files are descendants of the root directory, ( named / ) located at the top of the tree. In long-format output of ls –l , this type of file is specified by the "d" symbol.

**3) Special Files** – Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Output (I/O) operations. Device or special files are used for device Input/Output(I/O) on UNIX and Linux systems. They appear in a file system just like an ordinary file or a directory. On UNIX systems there are two flavors of special files for each device, character special files and block special files :

When a character special file is used for device Input/Output(I/O), data is transferred one character at a time. This type of access is called raw device access.

When a block special file is used for device Input/Output(I/O), data is transferred in large fixed-size blocks. This type of access is called block device access.
For terminal devices, it's one character at a time. For disk devices though, raw access means reading or writing in whole chunks of data – blocks, which are native to your disk.

In long-format output of ls -l, character special files are marked by the "c" symbol. In long-format output of ls -l, block special files are marked by the "b" symbol.

**4) Pipes** – UNIX allows you to link commands together using a pipe. The pipe acts a temporary file which only exists to hold data from one command until it is read by another.A Unix pipe provides a one-way flow of data.The output or result of the first command sequence is used as the input to the second command sequence. To make a pipe, put a vertical bar (|) on the command line between two commands.For example: who | wc -l In long-format output of ls –l , named pipes are marked by the "p" symbol.

**5) Sockets** – A Unix socket (or Inter-process communication socket) is a special file which allows for advanced inter-process communication. A Unix Socket is used in a client-server application framework. In essence, it is a stream of data, very similar to network stream (and network sockets), but all the transactions are local to the filesystem. In long-format output of ls -l, Unix sockets are marked by "s" symbol.

**6) Symbolic Link** – Symbolic link is used for referencing some other file of the file system.Symbolic link is also known as Soft link. It contains a text form of the path to the file it references. To an end user, symbolic link will appear to have its own name, but when you try reading or writing data to this file, it will instead reference these operations to the file it points to. If we delete the soft link itself , the data file would still be there.If we delete the source file or move it to a different location, symbolic file will not function properly. In longformat output of ls –l , Symbolic link are marked by the "l" symbol (that's a lower case L).

File management in Unix. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

When you work with Unix, one way or another, you spend most of your time working with files. This tutorial will help you understand how to create and remove files, copy and rename them, create links to them, etc.

In Unix, there are three basic types of files −

- **Ordinary Files** − An ordinary file is a file on the system that contains data, text, or program instructions. In this tutorial, you look at working with ordinary files.

- **Directories** − Directories store both special and ordinary files. For users familiar with Windows or Mac OS, Unix directories are equivalent to folders.

- **Special Files** − Some special files provide access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters. Other special files are similar to aliases or shortcuts and enable you to access a single file using different names.

**Listing Files**

To list the files and directories stored in the current directory, use the following command −

```
$ls
```

Here is the sample output of the above command −

```
$ls

bin       hosts  lib     res.03 ch07      hw1    pub     test_results
ch07.bak  hw2    res.01  users docs       hw3    res.02  work
```

The command **ls** supports the **-l** option which would help you to get more information about the listed files −

```
$ls -l
total 1962188

drwxrwxr-x  2 amrood amrood      4096 Dec 25 09:59 uml -rw-rw-
r--   1 amrood amrood       5341 Dec 25 08:38 uml.jpg drwxr-xr-x
2 amrood amrood       4096 Feb 15   2006 univ drwxr-xr-x   2 root
root          4096 Dec   9   2007 urlspedia -rw-r--r--   1 root    root
276480 Dec  9  2007 urlspedia.tar drwxr-xr-x  8 root   root      4096
Nov 25  2007 usr drwxr-xr-x  2    200    300      4096 Nov 25  2007
webthumb-1.01 -rwxr-xr-x  1 root    root         3192 Nov 25   2007
webthumb.php
-rw-rw-r--  1 amrood amrood    20480 Nov 25  2007 webthumb.tar
-rw-rw-r--  1 amrood amrood     5654 Aug  9  2007 yourfile.mid -
rw-rw-r--  1 amrood amrood    166255 Aug  9  2007 yourfile.swf
drwxr-xr-x 11 amrood amrood      4096 May 29  2007 zlib-1.2.3 $
```

Here is the information about all the listed columns –

- **First Column** − Represents the file type and the permission given on the file. Below is the description of all type of files.
- **Second Column** − Represents the number of memory blocks taken by the file or directory.
- **Third Column** − Represents the owner of the file. This is the Unix user who created this file.
- **Fourth Column** − Represents the group of the owner. Every Unix user will have an associated group.
- **Fifth Column** − Represents the file size in bytes.
- **Sixth Column** − Represents the date and the time when this file was created or modified for the last time.
- **Seventh Column** − Represents the file or the directory name.

In the **ls -l** listing example, every file line begins with a **d**, **-**, or **l**. These characters indicate the type of the file that's listed.

| Sr.No. | Prefix & Description |
|---|---|
| 1 | **-**<br>Regular file, such as an ASCII text file, binary executable, or hard link. |
| 2 | **b**<br>Block special file. Block input/output device file such as a physical hard drive. |
| 3 | **c**<br>Character special file. Raw input/output device file such as a physical hard drive. |
| 4 | **d**<br>Directory file that contains a listing of other files and directories. |
| 5 | **l**<br>Symbolic link file. Links on any regular file. |
| 6 | **p**<br>Named pipe. A mechanism for interprocess communications. |
| 7 | **s**<br>Socket used for interprocess communication. |

**Metacharacters**

Metacharacters have a special meaning in Unix. For example, ***** and **?** are metacharacters. We use ***** to match 0 or more characters, a question mark (**?**) matches with a single character.

For Example −

```
$ls ch*.doc
```

Displays all the files, the names of which start with **ch** and end with **.doc** −

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc ch02-1.doc c
```

Here, * works as meta character which matches with any character. If you want to display all the files ending with just **.doc**, then you can use the following command −

```
$ls *.doc
```

**Hidden Files**

An invisible file is one, the first character of which is the dot or the period character (.). Unix programs (including the shell) use most of these files to store configuration information.

Some common examples of the hidden files include the files −

- **.profile** − The Bourne shell ( sh) initialization script .
- **.kshrc** − The Korn shell ( ksh) initialization script .
- **.cshrc** − The C shell ( csh) initialization script.
- **.rhosts** − The remote shell configuration file.

To list the invisible files, specify the **-a** option to **ls** −

```
$ ls -a

.profile      docs    lib    test_results.
.rhosts       hosts   pub    users.
.emacs   bin          hw1     res.01  work .
.exrc    ch07         hw2     res.02
.kshrc   ch07.bak     hw3     res.03 $
```

- **Single dot (.)** − This represents the current directory.
- **Double dot (..)** − This represents the parent directory.

**Creating Files**

You can use the **vi** editor to create ordinary files on any Unix system. You simply need to give the following command −

```
$ vi filename
```

The above command will open a file with the given filename. Now, press the key **i** to come into the edit mode. Once you are in the edit mode, you can start writing your content in the file as in the following program −

```
This is unix file....I created it for the first time..... I'm
going to save this content in this file.
```

Once you are done with the program, follow these steps −

- Press the key **esc** to come out of the edit mode.
- Press two keys **Shift + ZZ** together to come out of the file completely.

You will now have a file created with **filename** in the current directory.

```
$ vi filename $
```

**Editing Files**

You can edit an existing file using the **vi** editor. We will discuss in short how to open an existing file −

```
$ vi filename
```

Once the file is opened, you can come in the edit mode by pressing the key **i** and then you can proceed by editing the file. If you want to move here and there inside a file, then first you need to come out of the edit mode by pressing the key **Esc**. After this, you can use the following keys to move inside a file −

- **l** key to move to the right side.
- **h** key to move to the left side.
- **k** key to move upside in the file.
- **j** key to move downside in the file.

So using the above keys, you can position your cursor wherever you want to edit. Once you are positioned, then you can use the **i** key to come in the edit mode. Once you are done with the editing in your file, press **Esc** and finally two keys **Shift + ZZ** together to come out of the file completely.

**Display Content of a File**

You can use the **cat** command to see the content of a file. Following is a simple example to see the content of the above created file −

```
$ cat filename
This is unix file....I created it for the first time.....
I'm going to save this content in this file.
$
```

You can display the line numbers by using the **-b** option along with the **cat** command as follows −

```
$ cat -b filename
1   This is unix file....I created it for the first time.....
2   I'm going to save this content in this file.
$
```

Counting Words in a File

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file. Following is a simple example to see the information about the file created above −

```
$ wc filename
2  19 103 filename
$
```

Here is the detail of all the four columns –

- **First Column** − Represents the total number of lines in the file.
- **Second Column** − Represents the total number of words in the file.
- **Third Column** − Represents the total number of bytes in the file. This is the actual size of the file.
- **Fourth Column** − Represents the file name.

You can give multiple files and get information about those files at a time. Following is simple syntax −

```
$ wc filename1 filename2 filename3
```

**Copying Files**

To make a copy of a file use the **cp** command. The basic syntax of the command is −

```
$ cp source_file destination_file
```

Following is the example to create a copy of the existing file **filename**.

```
$ cp filename copyfile
$
```

You will now find one more file **copyfile** in your current directory. This file will exactly be the same as the original file **filename**.

**Renaming Files**

To change the name of a file, use the **mv** command. Following is the basic syntax −

```
$ mv old_file new_file
```

The following program will rename the existing file **filename** to **newfile**.

```
$ mv filename newfile
$
```

The **mv** command will move the existing file completely into the new file. In this case, you will find only **newfile** in your current directory.

**Deleting Files**

To delete an existing file, use the **rm** command. Following is the basic syntax −

```
$ rm filename
```

**Caution** − A file may contain useful information. It is always recommended to be careful while using this **Delete** command. It is better to use the **-i** option along with **rm** command.
Following is the example which shows how to completely remove the existing file **filename**.

```
$ rm filename
$
```

You can remove multiple files at a time with the command given below −

```
$ rm filename1 filename2 filename3 $
```

**Standard Unix Streams**

Under normal circumstances, every Unix program has three streams (files) opened for it when it starts up −

- **stdin** − This is referred to as the *standard input* and the associated file descriptor is 0. This is also represented as STDIN. The Unix program will read the default input from STDIN.
- **stdout** − This is referred to as the *standard output* and the associated file descriptor is 1. This is also represented as STDOUT. The Unix program will write the default output at STDOUT
- **stderr** − This is referred to as the *standard error* and the associated file descriptor is 2. This is also represented as STDERR. The Unix program will write all the error messages at STDERR.

A directory is a file the solo job of which is to store the file names and the related information. All the files, whether ordinary, special, or directory, are contained in directories.

Unix uses a hierarchical structure for organizing files and directories. This structure is often referred to as a directory tree. The tree has a single root node, the slash character (/), and all other directories are contained below it.

**Home Directory**

The directory in which you find yourself when you first login is called your home directory.

You will be doing much of your work in your home directory and subdirectories that you'll be creating to organize your files.

You can go in your home directory anytime using the following command −

```
$cd ~
$
```

Here ~ indicates the home directory. Suppose you have to go in any other user's home directory, use the following command −

```
$cd ~username
$
```

To go in your last directory, you can use the following command −

```
$cd -
$
```

**Absolute/Relative Pathnames**

Directories are arranged in a hierarchy with root (/) at the top. The position of any file within the hierarchy is described by its pathname.

Elements of a pathname are separated by a /. A pathname is absolute, if it is described in relation to root, thus absolute pathnames always begin with a /.

Following are some examples of absolute filenames.

```
/etc/passwd
/users/sjones/chem/notes
/dev/rdsk/Os3
```

A pathname can also be relative to your current working directory. Relative pathnames never begin with /. Relative to user amrood's home directory, some pathnames might look like this −

```
chem/notes personal/res
```

To determine where you are within the filesystem hierarchy at any time, enter the command **pwd** to print the current working directory −

```
$pwd
/user0/home/amrood

$
```

**Listing Directories**

To list the files in a directory, you can use the following syntax −

```
$ls dirname
```

Following is the example to list all the files contained in **/usr/local** directory −

```
$ls /usr/local

X11     bin     gimp    jikes    sbin ace    doc     include
lib     share atalk    etc     info    man     ami
```

**Creating Directories**

We will now understand how to create directories. Directories are created by the following command −

```
$mkdir dirname
```

Here, directory is the absolute or relative pathname of the directory you want to create. For example, the command −

```
$mkdir mydir
$
```

Creates the directory **mydir** in the current directory. Here is another example −

```
$mkdir /tmp/test-dir
$
```

This command creates the directory **test-dir** in the **/tmp** directory. The **mkdir** command produces no output if it successfully creates the requested directory.

If you give more than one directory on the command line, **mkdir** creates each of the directories. For example, −

```
$mkdir docs pub $
```

Creates the directories docs and pub under the current directory.

**Creating Parent Directories**

We will now understand how to create parent directories. Sometimes when you want to create a directory, its parent directory or directories might not exist. In this case, **mkdir** issues an error message as follows −

```
$mkdir /tmp/amrood/test
mkdir: Failed to make directory "/tmp/amrood/test";
No such file or directory
$
```

In such cases, you can specify the **-p** option to the **mkdir** command. It creates all the necessary directories for you. For example −

```
$mkdir -p /tmp/amrood/test $
```

The above command creates all the required parent directories.

**Removing Directories**

Directories can be deleted using the **rmdir** command as follows −

```
$rmdir dirname
$
```

**Note** − To remove a directory, make sure it is empty which means there should not be any file or sub-directory inside this directory.

You can remove multiple directories at a time as follows −

```
$rmdir dirname1 dirname2 dirname3 $
```

The above command removes the directories dirname1, dirname2, and dirname3, if they are empty. The **rmdir** command produces no output if it is successful.

**Changing Directories**

You can use the **cd** command to do more than just change to a home directory. You can use it to change to any directory by specifying a valid absolute or relative path. The syntax is as given below −

```
$cd dirname
$
```

Here, **dirname** is the name of the directory that you want to change to. For example, the command −

```
$cd /usr/local/bin $
```

Changes to the directory **/usr/local/bin**. From this directory, you can **cd** to the directory **/usr/home/amrood** using the following relative path −

```
$cd ../../home/amrood $
```

**Renaming Directories**

The **mv (move)** command can also be used to rename a directory. The syntax is as follows −

```
$mv olddir newdir
$
```

You can rename a directory **mydir** to **yourdir** as follows −

```
$mv mydir yourdir
$
```

The directories . (dot) and .. (dot dot)

The **filename .** (dot) represents the current working directory; and the **filename ..** (dot dot) represents the directory one level above the current working directory, often referred to as the parent directory.

If we enter the command to show a listing of the current working directories/files and use the **-a option** to list all the files and the **-l option** to provide the long listing, we will receive the following result.

```
$ls -la drwxrwxr-x    4    teacher    class    2048  Jul 16 17.56
. drwxr-xr-x    60    root              1536  Jul 13 14:18 .. --------
--    1    teacher    class    4210  May 1 08:27 .profile
-rwxr-xr-x    1    teacher    class    1948  May 12 13:42 memo
$
```

**File ownership** is an important component of Unix that provides a secure method for storing files. Every file in Unix has the following attributes −

- **Owner permissions** − The owner's permissions determine what actions the owner of the file can perform on the file.

- **Group permissions** − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file. • **Other (world) permissions** − The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators

While using **ls -l** command, it displays various information related to file permission as follows −

```
$ls -l /home/amrood
-rwxr-xr--  1 amrood   users 1024  Nov 2 00:10  myfile drwxr-
xr--- 1 amrood   users 1024  Nov 2 00:10  mydir
```

Here, the first column represents different access modes, i.e., the permission associated with a file or a directory.

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x) −

- The first three characters (2-4) represent the permissions for the file's owner. For example, **-rwxr-xr--** represents that the owner has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example, **-rwxr-xr--** represents that the group has read (r) and execute (x) permission, but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else. For example, **-rwxr-xr--** represents that there is **read (r)** only permission.

**File Access Modes**

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which have been described below −

Read

Grants the capability to read, i.e., view the contents of the file.

Write

Grants the capability to modify, or remove the content of the file.

Execute

User with execute permissions can run a file as a program.
Directory Access Modes

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned –

Read

Access to a directory means that the user can read the contents. The user can look at the **filenames** inside the directory.

Write

Access means that the user can add or delete files from the directory.

Execute

Executing a directory doesn't really make sense, so think of this as a traverse permission.

A user must have **execute** access to the **bin** directory in order to execute the **ls** or the **cd** command.

Changing Permissions

To change the file or the directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod — the symbolic mode and the absolute mode.

Using chmod in Symbolic Mode

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

| Sr.No. | Chmod operator & Description |
|---|---|
| 1 | + <br> Adds the designated permission(s) to a file or directory. |
| 2 | - <br> Removes the designated permission(s) from a file or directory. |
| 3 | = <br> Sets the designated permission(s). |

Here's an example using **testfile**. Running **ls -1** on the testfile shows that the file's permissions are as follows –

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes –

```
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod g = rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Here's how you can combine these commands on a single line −

```
$chmod o+wx,u-x,g = rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

**Using chmod with Absolute Permissions**

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

| Number | Octal Permission Representation | Ref |
|---|---|---|
| **0** | No permission | --- |
| **1** | Execute permission | --x |
| **2** | Write permission | -w- |
| **3** | Execute and write permission: 1 (execute) + 2 (write) = 3 | -wx |
| **4** | Read permission | r-- |
| **5** | Read and execute permission: 4 (read) + 1 (execute) = 5 | r-x |

| | | |
|---|---|---|
| **6** | Read and write permission: 4 (read) + 2 (write) = 6 | rw- |
| **7** | All permissions: 4 (read) + 2 (write) + 1 (execute) = 7 | rwx |

Here's an example using the testfile. Running **ls -l** on the testfile shows that the file's permissions are as follows −

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes −

```
$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 743 testfile
$ls -l testfile
-rwxr---wx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 043 testfile
$ls -l testfile
----r---wx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Changing Owners and Groups

While creating an account on Unix, it assigns a **owner ID** and a **group ID** to each user. All the permissions mentioned above are also assigned based on the Owner and the Groups.

Two commands are available to change the owner and the group of files −

- **chown** − The **chown** command stands for **"change owner"** and is used to change the owner of a file.
- **chgrp** − The **chgrp** command stands for **"change group"** and is used to change the group of a file.

**Changing Ownership**

The **chown** command changes the ownership of a file. The basic syntax is as follows −

```
$ chown user filelist
```

The value of the user can be either the **name of a user** on the system or the **user id (uid)** of a user on the system.

The following example will help you understand the concept −

```
$ chown amrood testfile $
```

Changes the owner of the given file to the user **amrood**.

**NOTE** − The super user, root, has the unrestricted capability to change the ownership of any file but normal users can change the ownership of only those files that they own.

**Changing Group Ownership**

The **chgrp** command changes the group ownership of a file. The basic syntax is as follows −

```
$ chgrp group filelist
```

The value of group can be the **name of a group** on the system or **the group ID (GID)** of a group on the system.

Following example helps you understand the concept −

```
$ chgrp special testfile
$
```

Changes the group of the given file to special group.

**SUID and SGID File Permission**

Often when a command is executed, it will have to be executed with special privileges in order to accomplish its task.

As an example, when you change your password with the **passwd** command, your new password is stored in the file **/etc/shadow**.

As a regular user, you do not have **read** or **write** access to this file for security reasons, but when you change your password, you need to have the write permission to this file. This means that the **passwd** program has to give you additional permissions so that you can write to the file **/etc/shadow**.

Additional permissions are given to programs via a mechanism known as the **Set User ID (SUID)** and **Set Group ID (SGID)** bits.

When you execute a program that has the SUID bit enabled, you inherit the permissions of that program's owner. Programs that do not have the SUID bit set are run with the permissions of the user who started the program.

This is the case with SGID as well. Normally, programs execute with your group permissions, but instead your group will be changed just for this program to the group owner of the program.

The SUID and SGID bits will appear as the letter **"s"** if the permission is available. The SUID **"s"** bit will be located in the permission bits where the owners' **execute** permission normally resides.

For example, the command −

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x  1   root   bin  19031 Feb 7 13:47  /usr/bin/passwd*
$
```

Shows that the SUID bit is set and that the command is owned by the root. A capital letter **S** in the execute position instead of a lowercase **s** indicates that the execute bit is not set.

If the sticky bit is enabled on the directory, files can only be removed if you are one of the following users −

- The owner of the sticky directory.
- The owner of the file being removed.
- The super user, root.

To set the SUID and SGID bits for any directory try the following command −

```
$ chmod ug+s dirname
$ ls -l
drwsr-sr-x 2 root root  4096 Jun 19 06:45 dirname
$
```

[File system, Types of file, File naming convention, Parent – Child relationship, HOME variable, inode number, Absolute pathname, Relative pathname, Significance of dot (.) and dotdot (..), Displaying pathname of the current directory (pwd), Changing the current directory (cd), Make directory (mkdir), Remove directories (rmdir), Listing contents of directory (ls), Very brief idea about important file systems of UNIX: /bin, /usr/bin, /sbin, /usr/sbin, /etc, /dev, /lib, /usr/lib, /usr/include, /usr/share/man, /temp, /var, /home]

**Parent and Child Processes**

Each unix process has two ID numbers assigned to it: The Process ID (pid) and the Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check the ps -f example where this command listed both the process ID and the parent process ID.

**Zombie and Orphan Processes**

Normally, when a child process is killed, the parent process is updated via a SIGCHLD signal. Then the parent can do some other task or restart a new child as needed. However, sometimes the parent process is killed before its child is killed. In this case, the "parent of all processes," the init process, becomes the new PPID (parent process ID). In some cases, these processes are called orphan processes.

When a process is killed, a ps listing may still show the process with a Z state. This is a zombie or defunct process. The process is dead and not being used. These processes are different from the orphan processes. They have completed execution but still find an entry in the process table.

**Daemon Processes**

Daemons are system-related background processes that often run with the permissions of root and services requests from other processes.

A daemon has no controlling terminal. It cannot open /dev/tty. If you do a "ps -ef" and look at the tty field, all daemons will have a ? for the tty.

To be precise, a daemon is a process that runs in the background, usually waiting for something to happen that it is capable of working with. For example, a printer daemon waiting for print commands.

If you have a program that calls for lengthy processing, then it’s worth to make it a daemon and run it in the background.

# Module 3

**Ordinary file handling**

[Displaying and creating files (cat), Copying a file (cp), Deleting a file (rm), Renaming/ moving a file (mv), Paging output (more), Printing a file (lp), Knowing file type (file), Line, word and character counting (wc), Comparing files (cmp), Finding common between two files (comm), Displaying file differences (diff), Creating archive file (tar), Compress file (gzip), Uncompress file (gunzip), Archive file (zip), Extract compress file (unzip), Brief idea about effect of cp, rm and mv command on directory.]

**Types of Files:**

Regular files (-): It contain programs, executable files and text files.
Directory files (d): It is shown in blue colour. It contains list of files.

i) Special files ii) Block file (b) iii) Character device file (c) iv) Named pipe file (p)
v) Symbolic link file (l)
vi) Socket file (s)

**Linux File Commands**

| Command | Description |
|---|---|
| file | Determines file type. |
| touch | Used to create a file. |
| rm | To remove a file. |
| cp | To copy a file. |
| mv | To rename or to move a file. |
| rename | To rename file. |

**Linux cat command:** to display file content
The 'cat' command can be used to display the content of a file.

Syntax:

cat <fileName>

**cp command in Linux with examples**

cp stands for copy. This command is used to copy files or group of files or directory. It creates an exact image of a file on a disk with different file name. cp command require at least two filenames in its arguments.

Syntax:

cp [OPTION] Source Destination cp
[OPTION] Source Directory
cp [OPTION] Source-1 Source-2 Source-3 Source-n Directory

First and second syntax is used to copy Source file to Destination file or Directory.
Third syntax is used to copy multiple Sources(files) to Directory.
cp command works on three principal modes of operation and these operations depend upon number and type of arguments passed in cp command :

Two file names : If the command contains two file names, then it copy the contents of 1st file to the 2nd file. If the 2nd file doesn't exist, then first it creates one and content is copied to it. But if it existed then it is simply overwritten without any warning. So be careful when you choose destination file name.

cp Src_file Dest_file

Suppose there is a directory named geeksforgeeks having a text file a.txt. Example:

$ ls
a.txt

$ cp a.txt b.txt

$ ls
a.txt  b.txt

One or more arguments : If the command has one or more arguments, specifying file names and following those arguments, an argument specifying directory name then this command

copies each source file to the destination directory with the same name, created if not existed but if already existed then it will be overwritten, so be careful !!.
cp Src_file1 Src_file2 Src_file3 Dest_directory

Suppose there is a directory named geeksforgeeks having a text file a.txt, b.txt and a directory name new in which we are going to copy all files.
Example:

$ ls
a.txt  b.txt  new

Initially new is empty
$ ls new

$ cp a.txt b.txt new

$ ls new
a.txt  b.txt

Note: For this case last argument must be a directory name. For the above command to work, Dest_directory must exist because cp command won't create it.

Two directory names : If the command contains two directory names, cp copies all files of the source directory to the destination directory, creating any files or directories needed. This mode of operation requires an additional option, typically R, to indicate the recursive copying of directories.

cp -R Src_directory Dest_directory
In the above command, cp behavior depend upon whether Dest_directory is exist or not. If the Dest_directory doesn't exist, cp creates it and copies content of Src_directory recursively as it is. But if Dest_directory exists then copy of Src_directory becomes sub-directory under Dest_directory.

Options:

There are many options of cp command, here we will discuss some of the useful options: Suppose a directory named geeksforgeeks contains two files having some content named as a.txt and b.txt. This scenario is useful in understanding the following options.

$ ls geeksforgeeks a.txt
b.txt

$ cat a.txt
GFG

$ cat b.txt

GksfrGks

1. -i(interactive): i stands for Interactive copying. With this option system first warns the user before overwriting the destination file. cp prompts for a response, if you press y then it overwrites the file and with any other option leave it uncopied.

```
$ cp -i a.txt b.txt
cp: overwrite 'b.txt'? y

$ cat b.txt
GFG
```

2. -b(backup): With this option cp command creates the backup of the destination file in the same folder with the different name and in different format.

```
$ ls
a.txt  b.txt

$ cp -b a.txt b.txt

$ ls
a.txt  b.txt  b.txt~
```

3. -f(force): If the system is unable to open destination file for writing operation because the user doesn’t have writing permission for this file then by using -f option with cp command, destination file is deleted first and then copying of content is done from source to destination file.

```
$ ls -l b.txt
-r-xr-xr-x+ 1 User User 3 Nov 24 08:45 b.txt
```

User, group and others doesn't have writing permission.

Without -f option, command not executed

```
$ cp a.txt b.txt
cp: cannot create regular file 'b.txt': Permission denied
```

With -f option, command executed successfully

```
$ cp -f a.txt b.txt
```

4. -r or -R: Copying directory structure. With this option cp command shows its recursive behavior by copying the entire directory structure recursively.
Suppose we want to copy gksfrgks directory containing many files, directories into gfg directory(not exist).

```
$ ls gksfrgks/
a.txt  b.txt  b.txt~  Folder1  Folder2
```

Without -r option, error $
cp gksfrgks gfg
cp: -r not specified; omitting directory 'gksfrgks'

With -r, execute successfully
$ cp -r gksfrgks gfg

$ ls gfg/
a.txt  b.txt  b.txt~  Folder1  Folder2

5. -p(preserve): With -p option cp preserves the following characteristics of each source file in the corresponding destination file: the time of the last data modification and the time of the last access, the ownership (only if it has permissions to do this), and the file permission-bits.
Note: For the preservation of characteristics you must be the root user of the system, otherwise characteristics changes.

$ ls -l a.txt
-rwxr-xr-x+ 1 User User 3 Nov 24 08:13 a.txt

$ cp -p a.txt c.txt

$ ls -l c.txt
-rwxr-xr-x+ 1 User User 3 Nov 24 08:13 c.txt
As we can see above both a.txt and c.txt(created by copying) have same characteristics.

Examples:

Copying using * wildcard: The star wildcard represents anything i.e. all files and directories. Suppose we have many text document in a directory and wants to copy it another directory, it takes lots of time if we copy files 1 by 1 or command becomes too long if specify all these file names as the argument, but by using * wildcard it becomes simple.

Initially Folder1 is empty
$ ls
a.txt  b.txt  c.txt  d.txt  e.txt  Folder1

$ cp *.txt Folder1

$ ls Folder1
a.txt  b.txt  c.txt  d.txt  e.txt

**rm command in Linux with examples**

rm stands for remove here. rm command is used to remove objects such as files, directories, symbolic links and so on from the file system like UNIX. To be more precise, rm removes

references to objects from the filesystem, where those objects might have had multiple references (for example, a file with two different names). By default, it does not remove directories. This command normally works silently and you should be very careful while running rm command because once you delete the files then you are not able to recover the contents of files and directories. Syntax:

rm [OPTION]... FILE...
Let us consider 5 files having name a.txt, b.txt and so on till e.txt.

$ ls
a.txt  b.txt  c.txt  d.txt  e.txt
Removing one file at a time
$ rm a.txt

$ ls
b.txt  c.txt  d.txt  e.txt

Removing more than one file at a time
$ rm b.txt c.txt

$ ls
d.txt  e.txt
Note: No output is produced by rm, since it typically only generates messages in the case of an error. Options: 1. -i (Interactive Deletion): Like in cp, the -i option makes the command ask the user for confirmation before removing each file, you have to press y for confirm deletion, any other key leaves the file un-deleted.

$ rm -i d.txt
rm: remove regular empty file 'd.txt'? y

$ ls
e.txt
2. -f (Force Deletion): rm prompts for confirmation removal if a file is write protected. The -f option overrides this minor protection and removes the file forcefully.

$ ls -l total
0
-r--r--r--+ 1 User User 0 Jan  2 22:56 e.txt

$ rm e.txt
rm: remove write-protected regular empty file 'e.txt'? n

$ ls
e.txt

$ rm -f e.txt

$ ls
Note: -f option of rm command will not work for write-protect directories. 3. -r (Recursive Deletion): With -r(or -R) option rm command performs a tree-walk and will delete all the files and sub-directories recursively of the parent directory. At each stage it deletes everything it finds. Normally, rm wouldn't delete the directories but when used with this option, it will delete. Below is the tree of directories and files:

$ ls
A

$ cd A

$ ls
B  C

$ ls B
a.txt  b.txt

$ ls C
c.txt  d.txt
Now, deletion from A directory(as parent directory) will be done as:

$ rm * rm: cannot remove 'B': Is a
directory rm: cannot remove 'C': Is
a directory

$ rm -r *

$ ls
Every directory and file inside A directory is deleted. 4. –version: This option is used to display the version of rm which is currently running on your system.

$ rm --version rm (GNU
coreutils) 8.26
Packaged by Cygwin (8.26-2)
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later .
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Paul Rubin, David MacKenzie, Richard M. Stallman, and
Jim Meyering.

Applications of wc Command Delete file whose name starting with a hyphen symbol (-): To remove a file whose name begins with a dash ("-"), you can specify a double dash ("–") separately before the file name. This extra dash is necessary so that rm does not misinterpret the file name as an option. Let say there is a file name -file.txt, to delete this file write command as:

```
$ ls
-file.txt

$ rm -file.txt rm:
unknown option -- l
Try 'rm ./-file.txt' to remove the file '-file.txt'.
Try 'rm --help' for more information.

$ rm -- -file.txt

$ ls
```

**mv command in Linux with examples**

mv stands for move. mv is used to move one or more files or directories from one place to another in a file system like UNIX. It has two distinct functions:
(i) It renames a file or folder.
(ii) It moves a group of files to a different directory.
No additional space is consumed on a disk during renaming. This command normally works silently means no prompt for confirmation.

Syntax:

```
mv [Option] source destination
```

Let us consider 4 files having names a.txt, b.txt, and so on till d.txt. To rename the file a.txt to geek.txt(not exist):

```
$ ls
a.txt  b.txt  c.txt  d.txt

$ mv a.txt geek.txt

$ ls
b.txt  c.txt  d.txt  geek.txt
```

If the destination file doesn't exist, it will be created. In the above command mv simply replaces the source filename in the directory with the destination filename(new name). If the destination

file exist, then it will be overwrite and the source file will be deleted. By default, mv doesn't prompt for overwriting the existing file, So be careful !!

Let's try to understand with an example, moving geeks.txt to b.txt(exist):

```
$ ls
b.txt  c.txt  d.txt  geek.txt

$ cat geek.txt
India

$ cat b.txt geeksforgeeks

$ mv geek.txt b.txt

$ ls
b.txt  c.txt  d.txt

$ cat b.txt
India
```

Options:

1. -i (Interactive): Like in cp, the -i option makes the command ask the user for confirmation before moving a file that would overwrite an existing file, you have to press y for confirm moving, any other key leaves the file as it is. This option doesn't work if the file doesn't exist, it simply renames it or move it to new location.

```
$ ls
b.txt  c.txt  d.txt  gk.txt

$ cat gk.txt
India

$ cat b.txt
gk

$ mv -i gk.txt b.txt
mv: overwrite 'b.txt'? y

$ ls
b.txt  c.txt  d.txt

$ cat b.txt
India
```

2. -f (Force): mv prompts for confirmation overwriting the destination file if a file is writeprotected. The -f option overrides this minor protection and overwrites the destination file forcefully and deletes the source file.

```
$ ls
b.txt  c.txt  d.txt  geek.txt

$ cat b.txt
gksfrgks

$ ls -l b.txt
-r--r--r--+ 1 User User 13 Jan  9 13:37 b.txt

$ mv gk.txt b.txt mv: replace 'b.txt', overriding mode
0444 (r--r--r--)? n

$ ls
b.txt  c.txt  d.txt  gk.txt

$ mv -f gk.txt b.txt

$ ls
b.txt  c.txt  d.txt

$ cat b.txt
India
```

3. -n (no-clobber): With -n option, mv prevent an existing file from being overwritten.
In the following example the effect is for nothing to happen as a file would be overwritten.

```
$ ls
b.txt  c.txt  d.txt  gk.txt

$ cat b.txt
gksfrgks

$ mv -n gk.txt b.txt

$ ls
b.txt  c.txt  d.txt  gk.txt

$ cat b.txt
Gksfrgks
```

4. -b(backup): With this option, it is easier to take a backup of an existing file that will be overwritten as a result of the mv command. This will create a backup file with the tilde character(~) appended to it.

```
$ ls
b.txt  c.txt  d.txt  gk.txt

$ mv -b gk.txt b.txt

$ ls
b.txt  b.txt~  c.txt  d.txt
```

5. –version: This option is used to display the version of mv which is currently running on your system.

```
$ mv --version mv (GNU
coreutils) 8.26
Packaged by Cygwin (8.26-2)
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later .
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Mike Parker, David MacKenzie, and Jim Meyering.
```

**more command in Linux with Examples**

more command is used to view the text files in the command prompt, displaying one screen at a time in case the file is large (For example log files). The more command also allows the user do scroll up and down through the page. The syntax along with options and command is as follows. Another application of more is to use it with some other command after a pipe. When the output is large, we can use more command to see output one by one.

Syntax:

more [-options] [-num] [+/pattern] [+linenum] [file_name]

[-options]: any option that you want to use in order to change the way the file is displayed. Choose any one from the followings: (-d, -l, -f, -p, -c, -s, -u)
[-num]: type the number of lines that you want to display per screen.
[+/pattern]: replace the pattern with any string that you want to find in the text file.
[+linenum]: use the line number from where you want to start displaying the text content.
[file_name]: name of the file containing the text that you want to display on the screen.

While viewing the text file use these controls:

Enter key: to scroll down line by line.
Space bar: To go to the next page. b
key: To go to back one page.

Options:

-d : Use this command in order to help the user to navigate. It displays "[Press space to continue, 'q' to quit.]" and displays "[Press 'h' for instructions.]" when wrong key is pressed. Example:

more -d sample.txt

**lp - Unix, Linux Command**

NAME
lp: submits files for printing or alters a pending job..

EXAMPLES
Example-1:

To print the /etc/motd file on printer lp0 attached to device dlp0, enter:

# lp /etc/motd

Example-2:

To queue the MyFile file and return the job number, enter:

# lp myfile

**file command in Linux with examples**

file command is used to determine the type of a file. .file type may be of human-readable(e.g. 'ASCII text') or MIME type(e.g. 'text/plain; charset=us-ascii'). This command tests each argument in an attempt to categorize it.

It has three sets of tests as follows:

filesystem test: This test is based on the result which returns from a stat system call. The program verifies that if the file is empty, or if it's some sort of special file. This test causes the file type to be printed. magic test: These tests are used to check for files with data in particular

fixed formats. language test: This test search for particular strings which can appear anywhere in the first few blocks of a file.
Syntax:

file [option] [filename]
Example: Command displays the file type

file email.py file
name.jpeg file
Invoice.pdf file
exam.ods
file videosong.mp4

**wc command in Linux with examples**

wc stands for word count. As the name implies, it is mainly used for counting purpose.

It is used to find out number of lines, word count, byte and characters count in the files specified in the file arguments.
By default it displays four-columnar output.
First column shows number of lines present in a file specified, second column shows number of words present in the file, third column shows number of characters present in file and fourth column itself is the file name which are given as argument. Syntax:

wc [OPTION]... [FILE]...
Let us consider two files having name state.txt and capital.txt containing 5 names of the Indian states and capitals respectively.

$ cat state.txt
Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chhattisgarh

$ cat capital.txt
Hyderabad
Itanagar
Dispur
Patna
Raipur
Passing only one file name in the argument.

$ wc state.txt

```
 5  7 58 state.txt
      OR
$ wc capital.txt
 5  5 39 capital.txt
```
Passing more than one file name in the argument.

```
$ wc state.txt capital.txt
  5   7  58 state.txt
  5   5  39 capital.txt
 10  12  97 total
```
Note : When more than file name is specified in argument then command will display fourcolumnar output for all individual files plus one extra row displaying total number of lines, words and characters of all the files specified in argument, followed by keyword total. Options:
1. -l: This option prints the number of lines present in a file. With this option wc command displays two-columnar output, 1st column shows number of lines present in a file and 2nd itself represent the file name.

With one file name
```
$ wc -l state.txt
5 state.txt
```

With more than one file name
```
$ wc -l state.txt capital.txt
  5 state.txt
  5 capital.txt
 10 total
```

**cmp Command in Linux with examples**

cmp command in Linux/UNIX is used to compare the two files byte by byte and helps you to find out whether the two files are identical or not.

When cmp is used for comparison between two files, it reports the location of the first mismatch to the screen if difference is found and if no difference is found i.e the files compared are identical.
cmp displays no message and simply returns the prompt if the files compared are identical.
Syntax:
```
cmp [OPTION]... FILE1 [FILE2 [SKIP1 [SKIP2]]]
```

SKIP1 ,SKIP2 & OPTION are optional and
FILE1 & FILE2 refer to the filenames .
The syntax of cmp command is quite simple to understand. If we are comparing two files then obviously we will need their names as arguments (i.e as FILE1 & FILE2 in syntax). In addition to this, the optional SKIP1 and SKIP2 specify the number of bytes to skip at the beginning of each file which is zero by default and OPTION refers to the options compatible with this

command about which we will discuss later on. cmp Example : As explained that the cmp command reports the byte and line number if a difference is found. Now let's find out the same with the help of an example. Suppose there are two files which you want to compare one is file1.txt and other is file2.txt :

$cmp file1.txt file2.txt
If the files are not identical : the output of the above command will be :
$cmp file1.txt file2.txt
file1.txt file2.txt differ: byte 9, line 2

/*indicating that the first mismatch found in two
files at byte 20 in second line*/
If the files are identical : you will see something like this on your screen: $cmp
file1.txt file2.txt
$ _
/*indicating that the files are identical*/

**comm command in Linux with examples**

comm compare two sorted files line by line and write to standard output; the lines that are common and the lines that are unique.

Suppose you have two lists of people and you are asked to find out the names available in one and not in the other, or even those common to both. comm is the command that will help you to achieve this. It requires two sorted files which it compares line by line.
Before discussing anything further first let's check out the syntax of comm command: Syntax :

$comm [OPTION]... FILE1 FILE2
As using comm, we are trying to compare two files therefore the syntax of comm command needs two filenames as arguments.
With no OPTION used, comm produces three-column output where first column contains lines unique to FILE1 ,second column contains lines unique to FILE2 and third and last column contains lines common to both the files. comm command only works right if you are comparing two files which are already sorted. Example: Let us suppose there are two sorted files file1.txt and file2.txt and now we will use comm command to compare these two.

// displaying contents of file1 //
$cat file1.txt
Apaar
Ayush Rajput
Deepak
Hemant

// displaying contents of file2 //
$cat file2.txt
Apaar

Hemant
Lucky
Pranjal Thakral
Now, run comm command as:

// using comm command for
comparing two files //
$comm file1.txt file2.txt
Apaar
Ayush Rajput
Deepak
Hemant
Lucky
Pranjal Thakral

**diff command in Linux with examples**

diff stands for difference. This command is used to display the differences in the files by comparing the files line by line. Unlike its fellow members, cmp and comm, it tells us which lines in one file have is to be changed to make the two files identical.

The important thing to remember is that diff uses certain special symbols and instructions that are required to make two files identical. It tells you the instructions on how to change the first file to make it match the second file.

Special symbols are:

a : add c :
change d
: delete
Syntax :

diff [options] File1 File2
Lets say we have two files with names a.txt and b.txt containing 5 Indian states.

$ ls
a.txt  b.txt

$ cat a.txt
Gujarat
Uttar Pradesh
Kolkata
Bihar
Jammu and Kashmir

$ cat b.txt
Tamil Nadu
Gujarat
Andhra Pradesh
Bihar
Uttar pradesh
Now, applying diff command without any option we get the following output:

$ diff a.txt b.txt
0a1
> Tamil Nadu
2,3c3
< Uttar Pradesh
 Andhra Pradesh
5c5
 Uttar pradesh
Let's take a look at what this output means. The first line of the diff output will contain:

Line numbers corresponding to the first file,
A special symbol and
Line numbers corresponding to the second file.
Like in our case, 0a1 which means after lines 0(at the very beginning of file) you have to add Tamil Nadu to match the second file line number 1. It then tells us what those lines are in each file preceded by the symbol:

Lines preceded by a < are lines from the first file.
Lines preceded by > are lines from the second file.
Next line contains 2,3c3 which means from line 2 to line 3 in the first file needs to be changed to match line number 3 in the second file. It then tells us those lines with the above symbols.
The three dashes ("—") merely separate the lines of file 1 and file 2.
As a summary to make both the files identical, first add Tamil Nadu in the first file at very beginning to match line 1 of second file after that change line 2 and 3 of first file i.e. Uttar Pradesh and Kolkata with line 3 of second file i.e. Andhra Pradesh. After that change line 5 of first file i.e. Jammu and Kashmir with line 5 of second file i.e. Uttar Pradesh.

**Tar command in Linux/Unix with Examples**

The tar command is short for tape archive in Linux. This command is used for creating Archive and extracting the archive files. In Linux, it is one of the essential commands which facilitate archiving functionality. We can use this command for creating uncompressed and compressed archive files and modify and maintain them as well.

Syntax of tar command:
tar [options] [archive-file] [directory or file to be archived]

Options in the tar command
Various options in the tar command are listed below:

-c: This option is used for creating the archive.
-f: This option is used for creating an archive along with the provided name of the file.
-x: This option is used for extracting archives.
-u: It can be used for adding an archive to the existing archive file.
-t: It is used for displaying or listing files inside the archived file.
-A: This option is used for concatenating the archive files.
-v: It can be used to show verbose information.
-j: It is used for filtering archive tar files with the help of tbzip.
-z: It is a zip file and informs the tar command that makes a tar file with the help of gzip.
-r: This option is used for updating and adding a directory or file in an existing .tar file. -W: This option is used for verifying the archive file.

**ZIP command in Linux with examples**

ZIP is a compression and file packaging utility for Unix. Each file is stored in single .zip {.zip-filename} file with the extension .zip.

zip is used to compress the files to reduce file size and also used as file package utility. zip is available in many operating systems like unix, linux, windows etc.
If you have a limited bandwidth between two servers and want to transfer the files faster, then zip the files and transfer.
The zip program puts one or more compressed files into a single zip archive, along with information about the files (name, path, date, time of last modification, protection, and check information to verify file integrity). An entire directory structure can be packed into a zip archive with a single command.
Compression ratios of 2:1 to 3:1 are common for text files. zip has one compression method (deflation) and can also store files without compression. zip automatically chooses the better of the two for each file to be compressed.
The program is useful for packaging a set of files for distribution; for archiving files; and for saving disk space by temporarily compressing unused files or directories. Syntax :

zip [options] zipfile files_list Syntax
for Creating a zip file:

$zip myfile.zip filename.txt

**unzip lists,** tests, or extracts files from archives of the zip format, which are most commonly found on MS-DOS and Windows systems. The default behavior (with no options) is to extract into the current directory (and possibly the subdirectories below it) all files from the specified zip archive. A companion program, zip, creates zip archives. Both zip and unzip are compatible with archives created by PKWARE's PKZIP and PKUNZIP programs for MSDOS.

**Gzip Command in Linux**

gzip command compresses files. Each single file is compressed into a single file. The compressed file consists of a GNU zip header and deflated data.
If given a file as an argument, gzip compresses the file, adds a ".gz" suffix, and deletes the original file. With no arguments, gzip compresses the standard input and writes the compressed file to standard output.
Difference between Gzip and zip command in Unix and when to use which command

ZIP and GZIP are two very popular methods of compressing files, in order to save space, or to reduce the amount of time needed to transmit the files across the network, or internet. In general, GZIP is much better compared to ZIP, in terms of compression, especially when compressing a huge number of files.
The common practice with GZIP, is to archive all the files into a single tarball before compression. In ZIP files, the individual files are compressed and then added to the archive. When you want to pull a single file from a ZIP, it is simply extracted, then decompressed. With GZIP, the whole file needs to be decompressed before you can extract the file you want from the archive.
When pulling a 1MB file from a 10GB archive, it is quite clear that it would take a lot longer in GZIP, than in ZIP.
GZIP's disadvantage in how it operates, is also responsible for GZIP's advantage. Since the compression algorithm in GZIP compresses one large file instead of multiple smaller ones, it can take advantage of the redundancy in the files to reduce the file size even further. If you archive and compress 10 identical files with ZIP and GZIP, the ZIP file would be over 10 times bigger than the resulting GZIP file.

Syntax :
gzip [Options] [filenames]

Example:
$ gzip mydoc.txt
This command will create a compressed file of mydoc.txt named as mydoc.txt.gz and

**gunzip command in Linux with examples**
gunzip command is used to compress or expand a file or a list of files in Linux. It accepts all the files having extension as .gz, .z, _z, -gz, -z , .Z, .taz or.tgz and replace the compressed file with the original file by default. The files after uncompression retain its actual extension.
**Syntax:**
gunzip [Option] [archive name/file name]
[File attributes, File and directory attributes listing and very brief idea about the attributes, File ownership, File permissions, Changing file permissions – relative permission & absolute permission, Changing file ownership, Changing group ownership, File system and inodes, Hard link, Soft link, Significance of file attribute for directory, Default permissions of file and directory and using umask, Listing of modification and access time, Time stamp changing (touch), File locating (find)]

# Module 4

**UNIX file attributes**

Apart from permissions and ownership, a UNIX file has several other attributes, and in this chapter, we look at most of the remaining ones. A file also has properties related to its time stamps and links. It is important to know how these attributes are interpreted when applied to a directory or a device.

This chapter also introduces the concepts of file system. It also looks at the inode, the lookup table that contained almost all file attributes. Though a detailed treatment of the file systems is taken up later, knowledge of its basics is essential to our understanding of the significance of some of the file attributes. Basic file attributes has helped us to know about - ls –l to display file attributes (properties), listing of a specific directory, ownership and group ownership and different file permissions. ls –l provides attributes like – permissions, links, owner, group owner, size, date and the file name.

**File Systems and inodes**

The hard disk is split into distinct partitions, with a separate file system in each partition. Every file system has a directory structure headed by root.

n partitions = n file systems = n separate root directories

All attributes of a file except its name and contents are available in a table – inode (index node), accessed by the inode number. The inode contains the following attributes of a file:

File type
File permissions Number
of links
The UID of the owner
The GID of the group owner
File size in bytes
Date and time of last modification
Date and time of last access
Date and time of last change of the inode
An array of pointers that keep track of all disk blocks used by the file

Please note that, neither the name of the file nor the inode number is stored in the inode. To know inode number of a file:

ls -il tulec05

9059 -rw-r--r-- 1 kumar metal 51813 Jan 31 11:15 tulec05

Where, 9059 is the inode number and no other file can have the same inode number in the same file system.

**Hard Links**

The link count is displayed in the second column of the listing. This count is normally 1, but the following files have two links,

-rwxr-xr-- 2 kumar metal 163 Jull 13 21:36 backup.sh
-rwxr-xr-- 2 kumar metal 163 Jul 13 21:36 restore.sh

All attributes seem to be identical, but the files could still be copies. It's the link count that seems to suggest that the files are linked to each other. But this can only be confirmed by using the –i option to ls.

ls -li backup.sh restore.sh

478274 -rwxr-xr-- 2 kumar metal163 jul 13 21:36 backup.sh
478274 -rwxr-xr-- 2 kumar metal163 jul 13 21:36 restore.sh

ln: Creating Hard Links

A file is linked with the ln command which takes two filenames as arguments (cp command). The command can create both a hard link and a soft link and has syntax similar to the one used by cp. The following command links emp.lst with employee:

ln emp.lst employee

The –i option to ls shows that they have the same inode number, meaning that they are actually one end the same file:

ls -li emp.lst employee

29518 -rwxr-xr-x 2 kumar metal 915 may 4 09:58 emp.lst
29518 -rwxr-xr-x 2 kumar metal 915 may 4 09:58 employee

The link count, which is normally one for unlinked files, is shown to be two. You can increase the number of links by adding the third file name emp.dat as:

ln employee emp.dat ; ls -l emp*

29518 -rwxr-xr-x 3 kumar metal 915 may 4 09:58 emp.dat
29518 -rwxr-xr-x 3 kumar metal 915 may 4 09:58 emp.lst

29518 -rwxr-xr-x 3 kumar metal 915 may 4 09:58 employee

You can link multiple files, but then the destination filename must be a directory. A file is considered to be completely removed from the file system when its link count drops to zero. ln returns an error when the destination file exists. Use the –f option to force the removal of the existing link before creation of the new one
Where to use Hard Links

ln data/ foo.txt input_files

It creates link in directory input_files. With this link available, your existing programs will continue to find foo.txt in the input_files directory. It is more convenient to do this that modifies all programs to point to the new path. Links provide some protection against accidental deletion, especially when they exist in different directories. Because of links, we don't need to maintain two programs as two separate disk files if there is very little difference between them. A file's name is available to a C program and to a shell script. A single file with two links can have its program logic make it behave in two different ways depending on the name by which it is called.

We can't have two linked filenames in two file systems and we can't link a directory even within the same file system. This can be solved by using symbolic links (soft links).

**Symbolic Links**

Unlike the hard linked, a symbolic link doesn't have the file's contents, but simply provides the pathname of the file that actually has the contents.

ln -s note note.sym

ls -li note note.sym

9948 -rw-r--r-- 1 kumar group 80 feb 16 14:52 note
9952 lrwxrwxrwx 1 kumar group 4 feb16 15:07note.sym ->note

Where, l indicate symbolic link file category. -> indicates note.sym contains the pathname for the filename note. Size of symbolic link is only 4 bytes; it is the length of the pathname of note.

It's important that this time we indeed have two files, and they are not identical. Removing note.sym won't affect us much because we can easily recreate the link. But if we remove note, we would lose the file containing the data. In that case, note.sym would point to a nonexistent file and become a dangling symbolic link.

Symbolic links can also be used with relative pathnames. Unlike hard links, they can also span multiple file systems and also link directories. If you have to link all filenames in a

directory to another directory, it makes sense to simply link the directories. Like other files, a symbolic link has a separate directory entry with its own inode number. This means that rm can remove a symbolic link even if its points to a directory.

A symbolic link has an inode number separate from the file that it points to. In most cases, the pathname is stored in the symbolic link and occupies space on disk. However, Linux uses a fast symbolic link which stores the pathname in the inode itself provided it doesn't exceed 60 characters.

**The Directory**

A directory has its own permissions, owners and links. The significance of the file attributes change a great deal when applied to a directory. For example, the size of a directory is in no way related to the size of files that exists in the directory, but rather to the number of files housed by it. The higher the number of files, the larger the directory size. Permission acquires a different meaning when the term is applied to a directory.

ls -l -d progs

drwxr-xr-x  2  kumar  metal 320 may 9 09:57 progs

The default permissions are different from those of ordinary files. The user has all permissions, and group and others have read and execute permissions only. The permissions of a directory also impact the security of its files. To understand how that can happen, we must know what permissions for a directory really mean.

**Read permission**

Read permission for a directory means that the list of filenames stored in that directory is accessible. Since ls reads the directory to display filenames, if a directory's read permission is removed, ls wont work. Consider removing the read permission first from the directory progs,

ls -ld progs

drwxr-xr-x  2  kumar  metal 128 jun 18 22:41 progs

chmod -r progs ; ls progs

progs: permission denied

**Write permission**

We can't write to a directory file. Only the kernel can do that. If that were possible, any user could destroy the integrity of the file system. Write permission for a directory implies that you are permitted to create or remove files in it. To try that out, restore the read permission and remove the write permission from the directory before you try to copy a file to it.

chmod 555 progs ; ls –ld progs

dr-xr-xr-x 2 kumar metal 128 jun 18 22:41 progs cp
emp.lst progs

cp: cannot create progs/emp.lst: permission denied

The write permission for a directory determines whether we can create or remove files in it because these actions modify the directory
Whether we can modify a file depends on whether the file itself has write permission. Changing a file doesn't modify its directory entry

**Execute permission**

If a single directory in the pathname doesn't have execute permission, then it can't be searched for the name of the next directory. That's why the execute privilege of a directory is often referred to as the search permission. A directory has to be searched for the next directory, so the cd command won't work if the search permission for the directory is turned off.

chmod 666 progs ; ls –ld progs

drw-rw-rw- 2 kumar metal 128 jun 18 22:41 progs

cd progs

permission denied to search and execute it

**umask: DEFAULT FILE AND DIRECTORY PERMISSIONS**

When we create files and directories, the permissions assigned to them depend on the system's default setting. The UNIX system has the following default permissions for all files and directories.

rw-rw-rw- (octal 666) for regular files

rwxrwxrwx (octal 777) for directories

The default is transformed by subtracting the user mask from it to remove one or more permissions. We can evaluate the current value of the mask by using umask without arguments,

$ umask
022

This becomes 644 (666-022) for ordinary files and 755 (777-022) for directories umask 000. This indicates, we are not subtracting anything and the default permissions will remain unchanged. Note that, changing system wide default permission settings is possible using chmod but not by umask

**MODIFICATION AND ACCESS TIMES**

A UNIX file has three time stamps associated with it. Among them, two are:

Time of last file modification ls -l
Time of last access ls –lu
The access time is displayed when ls -l is combined with the -u option. Knowledge of file‘s modification and access times is extremely important for the system administrator. Many of the tools used by them look at these time stamps to decide whether a particular file will participate in a backup or not.

**TOUCH COMMAND – changing the time stamps**

To set the modification and access times to predefined values, we have,

touch options expression filename(s)

touch emp.lst (without options and expression)

Then, both times are set to the current time and creates the file, if it doesn’t exist.

touch command (without options but with expression) can be used. The expression consists of MMDDhhmm (month, day, hour and minute).

touch 03161430 emp.lst ; ls -l emp.lst

-rw-r--r-- 1 kumar metal 870 mar 16 14:30 emp.lst

ls -lu emp.lst

-rw-r--r-- 1 kumar metal 870 mar 16 14:30 emp.lst

It is possible to change the two times individually. The –m and –a options change the modification and access times, respectively:

touch command (with options and expression)

-m for changing modification time
-a for changing access time

touch -m 02281030 emp.lst ; ls -l emp.lst

-rw-r--r-- 1 kumar metal 870 feb 28 10:30 emp.lst

touch -a 01261650 emp.lst ; ls -lu emp.lst

-rw-r--r-- 1 kumar metal 870 jan 26 16:50 emp.lst

find : locating files

It recursively examines a directory tree to look for files matching some criteria, and then takes some action on the selected files. It has a difficult command line, and if you have ever wondered why UNIX is hated by many, then you should look up the cryptic find documentation. How ever, find is easily tamed if you break up its arguments into three components:

find path_list selecton_criteria action where,

Recursively examines all files specified in path_list
It then matches each file for one or more selection-criteria
It takes some action on those selected files
The path_list comprises one or more subdirectories separated by white space. There can also be a host of selection_criteria that you use to match a file, and multiple actions to dispose of the file. This makes the command difficult to use initially, but it is a program that every user must master since it lets him make file selection under practically any condition.

# Module 5

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. 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.

Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time −

```
$date
Thu Jun 25 08:30:19 MST 2009
```

You can customize your command prompt using the environment variable PS1 explained in the Environment tutorial.

Shell Types

In Unix, there are two major types of shells −

- **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** − If you are using a C-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)
- POSIX shell (sh)

The different C-type shells follow −

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

The original Unix shell was written in the mid-1970s by Stephen R. Bourne while he was at the AT&T Bell Labs in New Jersey.

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell".

Bourne shell is usually installed as **/bin/sh** on most versions of Unix. For this reason, it is the shell of choice for writing scripts that can be used on different versions of Unix.

In this chapter, we are going to cover most of the Shell concepts that are based on the Borne Shell.

## Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by # sign, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

We are going to write many scripts in the next sections. It would be a simple text file in which we would put all our commands and several other required constructs that tell the shell environment what to do and when to do it.

Shell scripts and functions are both interpreted. This means they are not compiled.

## Example Script

Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the **shebang** construct. For example −

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

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang*.

To create a script containing these commands, you put the shebang line first and then add the commands −

```
#!/bin/bash
pwd
ls
```

## Shell Comments

You can put your comments in your script as follows −

```
#!/bin/bash

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com # Script follows here:
pwd
ls
```

Save the above content and make the script executable −

```
$chmod +x test.sh
```

The shell script is now ready to be executed −

```
$./test.sh
```

Upon execution, you will receive the following result −

```
/home/amrood
index.htm      unix-basic_utilities.htm      unix-directories.htm
test.sh    unix-communication.htm    unix-environment.htm
```

**Note** − To execute a program available in the current directory, use **./program_name**

Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:

echo "What is your name?"
read PERSON echo
"Hello, $PERSON"
```

Here is a sample run of the script −

```
$./test.sh
What is your name?
Zara Ali
Hello, Zara Ali
$
```

Variable Names

The name of a variable can contain only letters (a to z or A to Z), numbers ( 0 to 9) or the underscore character ( _).

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names −

```
_ALI
TOKEN_A
VAR_1
VAR_2
```

Following are the examples of invalid variable names −

```
2_VAR
-VARIABLE VAR1-VAR2 VAR_A!
```

The reason you cannot use other characters such as **!**, *****, or **-** is that these characters have a special meaning for the shell.

Defining Variables

Variables are defined as follows −

```
variable_name=variable_value
```

For example −

```
NAME="Zara Ali"
```

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

Shell enables you to store any value you want in a variable. For example −

```
VAR1="Zara Ali"
VAR2=100
```

Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (**$**) −

For example, the following script will access the value of defined variable NAME and print it on STDOUT –

```
#!/bin/sh

NAME="Zara Ali" echo $NAME
```

The above script will produce the following value −

```
Zara Ali
```

Read-only Variables

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME −

```
#!/bin/sh

NAME="Zara Ali"
readonly NAME
NAME="Qadiri"
```

The above script will generate the following result −

```
/bin/sh: NAME: This variable is read only.
```

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command −

```
unset variable_name
```

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works −

```
#!/bin/sh

NAME="Zara Ali"
unset NAME echo
$NAME
```

The above example does not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

Variable Types

When a shell is running, three main types of variables are present −

- **Local Variables** − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
- **Environment Variables** − An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
- **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

For example, the **$** character represents the process ID number, or PID, of the current shell −

```
$echo $$
```

The above command writes the PID of the current shell −

```
29949
```

The following table shows a number of special variables that you can use in your shell scripts −

| Sr.No. | Variable & Description |
|---|---|
| 1 | **$0**<br>The filename of the current script. |
| 2 | **$n**<br>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 $2, and so on). |
| 3 | **$#**<br>The number of arguments supplied to a script. |
| 4 | **$***<br>All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2. |
| 5 | **$@**<br>All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| 6 | **$?**<br>The exit status of the last command executed. |
| 7 | **$$**<br>The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |

| | |
|---|---|
| 8 | **$!**<br>The process number of the last background command. |

**Command-Line Arguments**

The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to the command line −

```
#!/bin/sh

echo "File Name: $0" echo
"First Parameter : $1" echo
"Second Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Here is a sample run for the above script −

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
```

Special Parameters $* and $@

There are special parameters that allow accessing all the command-line arguments at once. **$*** and **$@** both will act the same unless they are enclosed in double quotes, **""**.

Both the parameters specify the command-line arguments. However, the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script as shown below to process an unknown number of commandline arguments with either the $* or $@ special parameters −

```
#!/bin/sh

for TOKEN in $*
do    echo
$TOKEN done
```

Here is a sample run for the above script −

```
$./test.sh Zara Ali 10 Years Old
Zara
Ali
10
Years Old
```

**Note** − Here **do...done** is a kind of loop that will be covered in a subsequent tutorial.

Exit Status

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command −

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
$echo $?
0
$
```

**Processes in Linux/Unix**

A program/command when executed, a special instance is provided by the system to the process. This instance consists of all the services/resources that may be utilized by the process under execution.

- Whenever a command is issued in Unix/Linux, it creates/starts a new process. For example, pwd when issued which is used to list the current directory location the user is in, a process starts.
- Through a 5 digit ID number Unix/Linux keeps an account of the processes, this number is called process ID or PID. Each process in the system has a unique PID.
- Used up pid's can be used in again for a newer process since all the possible combinations are used.
- At any point of time, no two processes with the same pid exist in the system because it is the pid that Unix uses to track each process.

## Initializing a process

**A process can be run in two ways:**

**Method 1: Foreground Process :** Every process when started runs in foreground by default, receives input from the keyboard, and sends output to the screen. When issuing pwd command

**$ ls pwd Output:**

$ /home/geeksforgeeks/root

When a command/process is running in the foreground and is taking a lot of time, no other processes can be run or started because the prompt would not be available until the program finishes processing and comes out.

**Method 2: Background Process:** It runs in the background without keyboard input and waits till keyboard input is required. Thus, other processes can be done in parallel with the process running in the background since they do not have to wait for the previous process to be completed.

Adding & along with the command starts it as a background process

**$ pwd &**

Since pwd does not want any input from the keyboard, it goes to the stop state until moved to the foreground and given any data input. Thus, on pressing Enter:

**Output:**

[1] + Done pwd

$

That first line contains information about the background process – the job number and the process ID. It tells you that the ls command background process finishes successfully. The second is a prompt for another command.

## Tracking ongoing processes

ps (Process status) can be used to see/list all the running processes.

**$ ps**

```
PID      TTY      TIME       CMD
19       pts/1    00:00:00   sh
24       pts/1    00:00:00   ps
```

For more information -f (full) can be used along with ps

**$ ps –f**

```
UID      PID  PPID C STIME    TTY        TIME CMD
52471     19     1 0 07:20    pts/1  00:00:00f     sh
52471     25    19 0 08:04    pts/1  00:00:00      ps -f
```

For single-process information, ps along with process id is used

**$ ps 19**

```
PID      TTY      TIME       CMD
19       pts/1    00:00:00   sh
```

For a running program (named process) **Pidof** finds the process id's (pids) **Fields described by ps are described as:**

- **UID**: User ID that this process belongs to (the person running it)
- **PID**: Process ID
- **PPID**: Parent process ID (the ID of the process that started it)
- **C**: CPU utilization of process
- **STIME**: Process start time
- **TTY**: Terminal type associated with the process
- **TIME**: CPU time is taken by the process
- **CMD**: The command that started this process

**There are other options which can be used along with ps command :**

- **-a**: Shows information about all users
- **-x**: Shows information about processes without terminals
- **-u**: Shows additional information like -f option
- **-e**: Displays extended information **Stopping a process:**

When running in foreground, hitting Ctrl + c (interrupt character) will exit the command. For processes running in background kill command can be used if it's pid is known.

**$ ps –f**

| UID | PID | PPID | C | STIME | TTY | TIME | CMD |
|---|---|---|---|---|---|---|---|
| 52471 | 19 | 1 | 0 | 07:20 | pts/1 | 00:00:00 | sh |
| 52471 | 25 | 19 | 0 | 08:04 | pts/1 | 00:00:00 | ps –f |

**$ kill 19**
Terminated
If a process ignores a regular kill command, you can use kill -9 followed by the process ID.

**$ kill -9 19**
Terminated

**Other process commands:**

**bg**: A job control command that resumes suspended jobs while keeping them running in the background Syntax: **bg [ job ]** For example:

bg %19 **fg**: It continues a stopped job by running it in the

foreground.

Syntax:
**fg [ %job_id ]** For example fg 19 **top**: This command is used to show all the running processes within the working environment of Linux.

Syntax: top **nice**: It starts a new process (job) and assigns it a priority (nice) value at the same time. Syntax: nice [-nice value] nice value ranges from -20 to 19, where -20 is of the highest priority.

**renice** : To change the priority of an already running process renice is used.

Syntax: renice [-nice value]
[process id]

**df**: It shows the amount of available disk space being used by file systems Syntax:
df Output:

Filesystem 1K-blocks Used Available Use% Mounted on

/dev/loop0 18761008 15246876 2554440 86% /

none 4 0 4 0% /sys/fs/cgroup

udev 493812 4 493808 1% /dev tmpfs

100672 1364 99308 2% /run none

5120 0 5120 0% /run/lock none

503352 1764 501588 1% /run/shm none

102400 20 102380 1% /run/user

/dev/sda3 174766076 164417964 10348112 95% /host **free**: It shows the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel Syntax:
free Output:

| | total | used | free | shared | buffers | cached |
|---|---|---|---|---|---|---|
| Mem: | 1006708 | 935872 | 70836 | 0 | 148244 | 346656 |
| -/+ buffers/cache: | | 440972 | 565736 | | | |
| Swap: | 262140 | 130084 | 132056 | | | |

## Types of Processes

1. **Parent and Child process :** The 2nd and 3rd column of the ps –f command shows process id and parent's process id number. For each user process, there's a parent process in the system, with most of the commands having shell as their parent.
2. **Zombie and Orphan process :** After completing its execution a child process is terminated or killed and SIGCHLD updates the parent process about the termination and thus can continue the task assigned to it. But at times when the parent process is killed before the termination of the child process, the child processes become orphan processes, with the parent of all processes "init" process, becomes their new pid.
   A process which is killed but still shows its entry in the process status or the process table is called a zombie process, they are dead and are not used.
3. **Daemon process :** They are system-related background processes that often run with the permissions of root and services requests from other processes, they most of the time run in the background and wait for processes it can work along with for ex print daemon. When ps –ef is executed, the process with ? in the tty field are daemon processes.

# Module 6

## Customization

### What is an environment variable?

**Environment variables** or **ENVs** basically define the behavior of the environment. They can affect the processes ongoing or the programs that are executed in the environment.

### Scope of an environment variable

Scope of any variable is the region from which it can be accessed or over which it is defined. An environment variable in Linux can have **global** or **local** scope.

**Global**

A globally scoped ENV that is defined in a terminal can be accessed from anywhere in that particular environment which exists in the terminal. That means it can be used in all kind of scripts, programs or processes running in the environment bound by that terminal.

**Local**

A locally scoped ENV that is defined in a terminal cannot be accessed by any program or process running in the terminal. It can only be accessed by the terminal( in which it was defined) itself.

### How to access ENVs?

**SYNTAX:**
$NAME

**NOTE:** Both local and global environment variables are accessed in the same way.

### How to display ENVs?

**To display any ENV**

**SYNTAX:**
$ echo $NAME

**To display all the Linux ENVs**

**SYNTAX:**
$ printenv //displays all the global ENVs or

$ set //display all the ENVs(global as well as local) or

$ env //display all the global ENVs

**How to set environment variables?**

**To set a global ENV**
$ export NAME=Value

or

$ set NAME=Value

**To set a local ENV**

**SYNTAX:**
$ NAME=Value

**To set user wide ENVs**

These variable are set and configured in ~/.bashrc, ~/.bash_profile, ~/.bash_login, ~/.profile files according to the requirement. These variables can be accessed by a particular user and persist through power offs.

Following steps can be followed to do so:
Step 1: Open the terminal. Step 2:

$ sudo vi ~/.bashrc

Step 3:Enter password.
Step 4:Add variable in the file opened. export

NAME=Value

Step 5: Save and close the file. Step 6:

$ source ~/.bashrc

**To set system wide ENVs**

These variable are set and configured in /etc/environment, /etc/profile, /etc/profile.d/, /etc/bash.bashrc files according to the requirement.These variables can be accessed by any user and persist through power offs.

Following steps can be followed to do so:
Step 1: Open the terminal.

Step 2:
$ sudo -H vi /etc/environment

Step 3:Enter password.
Step 4:Add variable in the file opened.

NAME=Value

Step 5: Save and close the file.
Step 6: Logout and Login again.

**How to unset environment variables?**

**SYNTAX:** $
unset NAME

or

$ NAME="

**NOTE**: To unset permanent ENVs, you need to re-edit the files and remove the lines that were added while defining them. **Some commonly used ENVs in Linux**

**$USER:** Gives current user's name.
**$PATH:** Gives search path for commands.
**$PWD:** Gives the path of present working directory.
**$HOME:** Gives path of home directory.
**$HOSTNAME:** Gives name of the host.
**$LANG:** Gives the default system language.
**$EDITOR:** Gives default file editor. **$UID:**
Gives user ID of current user.
**$SHELL:** Gives location of current user's shell program.

Piping in Unix or Linux

A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing. The Unix/Linux systems allow stdout of a command to be connected to stdin of another command. You can make it do so by using the pipe character '**|**'.
Pipe is used to combine two or more commands, and in this, the output of one command acts as input to another command, and this command's output may act as input to the next command and so on. It can also be visualized as a temporary connection between two or more commands/ programs/ processes. The command line programs that do the further processing are referred to as filters.

This direct connection between commands/ programs/ processes allows them to operate simultaneously and permits data to be transferred between them continuously rather than having to pass it through temporary text files or through the display screen. Pipes are unidirectional **i.e data flows from left to right through the pipeline.**

**Syntax :**

command_1 | command_2 | command_3 | .... | command_N

**Example :**

**1. Listing all files and directories and give it as input to more command.**

$ ls -l | more

The more command takes the output of $ ls -l as its input. The net effect of this command is that the output of ls -l is displayed one screen at a time. The pipe acts as a container which takes the output of ls -l and gives it to more as input. This command does not use a disk to connect standard output of ls -l to the standard input of more because pipe is implemented in the main memory. In terms of I/O redirection operators, the above command is equivalent to the following command sequence.

$ ls -l -> temp more -> temp

(or more temp)

[contents of temp] rm

temp

The novel idea of Pipes was introduced by **M.D Mcllroy** in **June 1972**– version 2, 10 UNIX installations. Piping is used to give the output of one command (written on LHS) as input to another command (written on RHS). Commands are piped together using vertical bar " | " symbol. **Syntax:** command 1|command 2

***Example:***

- **Input:** ls|more
- **Output:** more command takes input from *ls* command and appends it to the standard output. It displays as many files that fit on the screen and highlighted *more* at the bottom of the screen. To see the next screen hit enter or space bar to move one line at a time or one screen at a time respectively.

Filters in UNIX

In UNIX/Linux, filters are the set of commands that take input from standard input stream i.e. **stdin**, perform some operations and write output to standard output stream i.e. **stdout**. The stdin and stdout can be managed as per preferences using redirection and pipes. Common filter commands are: grep, more, sort.

**1.** **grep Command:**It is a pattern or expression matching command. It searches for a pattern or regular expression that matches in files or directories and then prints found matches.

**Syntax:**

$grep[options] "**pattern to be matched**" filename **Example:**
**Input :** $grep 'hello' ist_file.txt
**Output :** searches hello in the ist_file.txt and outputs/returns the lines containing 'hello'.

**2.** **sort Command:** It is a data manipulation command that sorts or merges lines in a file by specified fields. In other words it sorts lines of text alphabetically or numerically, **default sorting is alphabetical**.
**Syntax:**
$sort[options] filename

# Module 7

**Introduction to shell script**

Introduction to Linux Shell and Shell Scripting

If you are using any major operating system you are indirectly interacting to **shell**. If you are running Ubuntu, Linux Mint or any other Linux distribution, you are interacting to shell every time you use terminal. In this article I will discuss about linux shells and shell scripting so before understanding shell scripting we have to get familiar with following terminologies:

- Kernel
- Shell
- Terminal

**What is Kernel**

The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It manages following resources of the Linux system –

- File management
- Process management
- I/O management
- Memory management
- Device management etc.

**What is Shell**

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.

**Command Line Shell**

Shell can be accessed by user using a command line interface. A special program called Terminal in linux/macOS or Command Prompt in Windows OS is provided to type in the human readable commands such as "cat", "ls" etc. and then it is being execute.

**Shell Scripting**

Usually shells are interactive that mean, they accept command as input from users and execute them. However some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal.

As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called **Shell Scripts** or **Shell Programs**. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with **.sh** file extension eg. **myscript.sh**

A shell script have syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.

A shell script comprises following elements –

- Shell Keywords – if, else, break etc.
- Shell commands – cd, ls, echo, pwd, touch etc.
- • Functions Control flow – if..then..else, case and shell loops etc.

**Why do we need shell scripts**

There are many reasons to write shell scripts:

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups
- System monitoring
- Adding new functionality to the shell etc.

**Advantages of shell scripts**

- The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax
- Writing shell scripts are much quicker
- Quick start
- Interactive debugging etc.

**Disadvantages of shell scripts**

- Prone to costly errors, a single mistake can change the command which might be harmful
- Slow execution speed
- Design flaws within the language syntax or implementation
- Not well suited for large and complex task
- Provide minimal data structure unlike other scripting languages. etc

There are **5** basic operators in bash/shell scripting:

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- Bitwise Operators
- File Test Operators

1. **Arithmetic Operators**: These operators are used to perform normal arithmetics/mathematical operations. There are 7 arithmetic operators:
   - **Addition (+)**: Binary operation used to add two operands.
   - **Subtraction (-)**: Binary operation used to subtract two operands.
   - **Multiplication (*)**: Binary operation used to multiply two operands.
   - **Division (/)**: Binary operation used to divide two operands.
   - **Modulus (%)**: Binary operation used to find remainder of two operands.
   - **Increment Operator (++)**: Unary operator used to increase the value of operand by one.

- **Decrement Operator (- -)**: Unary operator used to decrease the value of a operand by one

2. **Relational Operators**: Relational operators are those operators which define the relation between two operands. They give either true or false depending upon the relation. They are of 6 types:
    - **'==' Operator**: Double equal to operator compares the two operands. Its returns true is they are equal otherwise returns false.
    - **'!=' Operator**: Not Equal to operator return true if the two operands are not equal otherwise it returns false.
    - **'<' Operator**: Less than operator returns true if first operand is less than second operand otherwise returns false.
    - **'<=' Operator**: Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false
    - **'>' Operator**: Greater than operator return true if the first operand is greater than the second operand otherwise return false.
    - **'>=' Operator**: Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

3. **Logical Operators** : They are also known as boolean operators. These are used to perform logical operations. They are of 3 types:
    - **Logical AND (&&)**: This is a binary operator, which returns true if both the operands are true otherwise returns false.
    - **Logical OR (||)**: This is a binary operator, which returns true is either of the operand is true or both the operands are true and return false if none of then is false.
    - **Not Equal to (!)**: This is a unary operator which returns true if the operand is false and returns false if the operand is true.

4. **Bitwise Operators**: A bitwise operator is an operator used to perform bitwise operations on bit patterns. They are of 6 types:
    - **Bitwise And (&)**: Bitwise & operator performs binary AND operation bit by bit on the operands.
    - **Bitwise OR (|)**: Bitwise | operator performs binary OR operation bit by bit on the operands.
    - **Bitwise XOR (^)**: Bitwise ^ operator performs binary XOR operation bit by bit on the operands.
    - **Bitwise complement (~)**: Bitwise ~ operator performs binary NOT operation bit by bit on the operand.
    - **Left Shift (<<)**: This operator shifts the bits of the left operand to left by number of times specified by right operand.
    - **Right Shift (>>)**: This operator shifts the bits of the left operand to right by number of times specified by right operand.

**5. File Test Operator**: These operators are used to test a particular property of a file.

- **-b operator**: This operator check whether a file is a block special file or not. It returns true if the file is a block special file otherwise false.
- **-c operator**: This operator checks whether a file is a character special file or not. It returns true if it is a character special file otherwise false.
- **-d operator**: This operator checks if the given directory exists or not. If it exists then operators returns true otherwise false.
- **-e operator**: This operator checks whether the given file exists or not. If it exits this operator returns true otherwise false.
- **-r operator**: This operator checks whether the given file has read access or not. If it has read access then it returns true otherwise false.
- **-w operator**: This operator check whether the given file has write access or not. If it has write then it returns true otherwise false.
- **-x operator**: This operator check whether the given file has execute access or not. If it has execute access then it returns true otherwise false.
- **-s operator**: This operator checks the size of the given file. If the size of given file is greater than 0 then it returns true otherwise it is false.

**Conditional Statements:** There are total 5 conditional statements which can be used in bash programming

1. if statement
2. if-else statement
3. if..elif..else..fi statement (Else If ladder)
4. if..then..else..if..then..fi..fi..(Nested if)
5. switch statement

Their description with syntax is as follows:

**if statement**

This block will process if specified condition is true.

***Syntax:***

```
if [ expression ] then
   statement
fi
```

**if-else statement**

If specified condition is not true in if part then else part will be execute.

***Syntax***

```
if [ expression ]
then
statement1 else
statement2
fi
```

**if..elif..else..fi statement (Else If ladder)**

To use multiple conditions in one if-else block, then elif keyword is used in shell. If expression1 is true then it executes statement 1 and 2, and this process continues. If none of the condition is true then it processes else part. ***Syntax*** if [ expression1 ] then statement1 statement2

.

.

elif [ expression2 ]
then statement3
statement4

.

.

else
statement5
fi

**if..then..else..if..then..fi..fi..(Nested if)**

Nested if-else block can be used when, one condition is satisfies then it again checks another condition. In the syntax, if expression1 is false then it processes else part, and again expression2 will be check.

***Syntax:***

if [ expression1 ]
then
statement1
statement2

.

else
if [ expression2 ]
then
statement3

.

fi fi

**switch statement**

case statement works as a switch statement if specified value match with the pattern then it will execute a block of that particular pattern

When a match is found all of the associated statements until the double semicolon (;;) is executed.

A case will be terminated when the last command is executed.

If there is no match, the exit status of the case is zero.

***Syntax:*** case in
Pattern 1) Statement 1;;
Pattern n) Statement n;; esac

**Example Programs**

**Example 1:**

Implementing if statement

```
#Initializing two variables
a=10
b=20

#Check whether they are equal
if [ $a == $b ]
then
   echo "a is equal to b"
fi

#Check whether they are not equal
if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

**Output**

```
$bash -f main.sh

a is not equal to b
```

***Example 2:***

Implementing if.else statement

```
#Initializing two variables
a=20
b=20

if [ $a == $b ] then
   #If they are equal then print this
echo "a is equal to b" else

   #else print this
   echo "a is not equal to b"
fi
```

**Output**

```
$bash -f main.sh

a is equal to b
```

**Example 3:**
Implementing switch statement

```
CARS="bmw"

#Pass the variable in string
case "$CARS" in
   #case 1
   "mercedes") echo "Headquarters - Affalterbach, Germany" ;;

   #case 2
   "audi") echo "Headquarters - Ingolstadt, Germany" ;;

   #case 3
   "bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;
esac
```

**Output**
$bash -f main.sh

Headquarters - Chennai, Tamil Nadu, India.

**Looping Statements in Shell Scripting:** There are total 3 looping statements which can be used in bash programming

1. while statement
2. for statement
3. until statement

To alter the flow of loop statements, two commands are used they are,

1. break
2. continue

As a UNIX system administrator, you will be **responsible for the installation, configuration, and maintenance of our UNIX systems**. In this role, you will troubleshoot server errors, install new system hardware, respond to user issues, and monitor the performance of the network.

**NAME**
shutdown - bring the system down

**SYNOPSIS /sbin/shutdown** [**-t** *sec*] [**-arkhncfFHP**] *time* [*warning-message*]

**DESCRIPTION**

**shutdown** brings the system down in a secure way. All logged-in users are notified that the system is going down, and **login**(1) is blocked. It is possible to shut the system down immediately or after a specified delay. All processes are first notified that the system is going down by the signal SIGTERM. This gives programs like **vi**(1) the time to save the file being edited, mail and news processing programs a chance to exit cleanly, etc. **shutdown** does its job by signalling the **init** process, asking it to change the runlevel. Runlevel **0** is used to halt the system, runlevel **6** is used to reboot the system, and runlevel **1** is used to put to system into a state where administrative tasks can be performed; this is the default if neither the *h* or *-r* flag is given to **shutdown**. To see which actions are taken on halt or reboot see the appropriate entries for these runlevels in the file */etc/inittab*.

**OPTIONS**

| Tag | Description |
|---|---|
| **-a** | Use **/etc/shutdown.allow**. |
| **-t** *sec* | Tell **init**(8) to wait *sec* seconds between sending processes the warning and the kill signal, before changing to another runlevel. |
| **-k** | Don't really shutdown; only send the warning messages to everybody. |
| **-r** | Reboot after shutdown. |
| **-h** | Halt or poweroff after shutdown. |
| **-H** | Halt action is to halt or drop into boot monitor on systems that support it. |
| **-P** | Halt action is to turn off the power. |
| **-n** | [DEPRECATED] Don't call **init**(8) to do the shutdown but do it ourself. The use of this option is discouraged, and its results are not always what you'd expect. |

| | |
|---|---|
| **-f** | Skip fsck on reboot. |
| **-F** | Force fsck on reboot. |
| **-c** | Cancel an already running shutdown. With this option it is of course not possible to give the **time** argument, but you can enter a explanatory message on the command line that will be sent to all users. |
| *Time* | When to shutdown. |
| *warning-message* | Message to send to all users. |

The *time* argument can have different formats. First, it can be an absolute time in the format *hh:mm*, in which *hh* is the hour (1 or 2 digits) and *mm* is the minute of the hour (in two digits). Second, it can be in the format **+***m*, in which *m* is the number of minutes to wait. The word **now** is an alias for **+0**.

If shutdown is called with a delay, it creates the advisory file */etc/nologin* which causes programs such as *login(1)* to not allow new user logins. Shutdown removes this file if it is stopped before it can signal init (i.e. it is cancelled or something goes wrong). It also removes it before calling init to change the runlevel.

The **-f** flag means 'reboot fast'. This only creates an advisory file */fastboot* which can be tested by the system when it comes up again. The boot rc file can test if this file is present, and decide not to run **fsck**(1) since the system has been shut down in the proper way. After that, the boot process should remove */fastboot*.

The **-F** flag means 'force fsck'. This only creates an advisory file */forcefsck* which can be tested by the system when it comes up again. The boot rc file can test if this file is present, and decide to run **fsck**(1) with a special 'force' flag so that even properly unmounted filesystems get checked. After that, the boot process should remove */forcefsck*.

The **-n** flag causes **shutdown** not to call **init**, but to kill all running processes itself. **shutdown** will then turn off quota, accounting, and swapping and unmount all filesystems.

**Creating a user account**

To create a normal user account, we use the **useradd** command, like shown in the following snippet of code:

```
sudo useradd [options] [username]
```

In the above snippet, the username is the name by which we will create the new account.
Note that the name can not be the same for two users.
In the place of options, we can pass different flags to enable or disable different settings. The following table contains those options and their brief descriptions:

| Option | Description |
|---|---|
| -b, --base-dir *BASE_DIR* | This option is used to specify the default base directory for the new user being created. *BASE_DIR* is concatenated with the account name to specify the base directory if the d flag is not used to define the home directory. |
| -c, --comment *COMMENT* | This option is used to write a short description of the new account. It is also used as the user's full name for the time being. |
| -d, --home *HOME_DIR* | This option is used to specify the home directory for the newly created user. |
| -D, --defaults | This option is used to display the current default values for the `useradd` command |
| -e, --expiredate *EXPIRE_DATE* | This option is used to specify the date in the YYYY-MMDD format on which the newly created user account will be disabled. |
| -f, --inactive | This option is used to specify the number of days that the account will stay active after its password has expired. |
| -g, --gid *GROUP* | This option is to specify the name or number of the group of newly created user. |
| -G, --groups *Group1[,Group2,...[GroupN]]]* | This option is used to specify a list of groups which will be joined by the newly created user. |
| -h, --help | This option is to display the help message |
| -k, --skel *SKEL_DIR* | This option is used to specify files and folders which will be copied into the home directory of the newly created user. |
| -K, --key *KEY=VALUE* | This option overrides the default values present in /etc/login.defs. These inclue UID_MIN, UID_MAX, UMASK etc. |
| - I, -no-log-init | This option prevents the user to be added from the lastlog and faillog databases. |
| -m, --create-home | This option is used to create the user's home directory if it does not exist. |
| -M | This option is used to prevent the creation of the home directory. |
| -N, --no-user-group | This option is used to prevent the creation of a group with the user's name. |
| -o, --non-unique | This options allows the creation of a user account with a duplicate ID. |
| -p, --password *PASSWORD* | The encrypted password, as returned by the `crypt`. |
| -r, --system | This option is used to create a system account. |

| | |
|---|---|
| -s, --shell *SHELL* | This option is used to specify the name of the user's login shell. |
| -u, --uid *UID* | This option is used to set the numerical value of the user' ID. This value must be unique. |
| -U, --user-group | This option enables the creation of a group with the same name as the user's name. |
| -Z, --selinux-user *SEUSER* | This option is used to determine the SELinux user for the user's login. |

**Modifying a user account**

To modify a user account, we use the **usermod** command as shown in the following snippet:

```
usermod [options] [username]
```

In the above snippet, the username is the name of the account that is to be modified.

In the place of options, different flags are passed for different settings.

**Deleting a user account**

To delete a user account, we use the **userdel** command as shown in the following snippet:

```
userdel [options] [username]
```

In the above snippet, the username is the account's name that is to be deleted.

In the place of options, different flags are passed for different settings.

**Groups**

In Linux or Unix-based operating systems, we can form **groups** of users' accounts. Groups are used to manage the user accounts collectively. We can manage the access permissions for the entire group.

A single user can be a part of multiple groups, and a group can have multiple users.

The terminal commands related to groups are discussed in detail below.

**Creating a new group**

We use the **groupadd** command to create a new group, as shown in the snippet below.

```
groupdadd [options] [groupname]
```

In the above snippet, groupname is the name assigned to the newly created group.

In place of options, we can pass different flags for different settings.

**Modifying a group**

We use the **groupmod** command to modify an existing group, as shown in the snippet below.

```
groupmod [options] [groupname]
```

In the above snippet, groupname is the group's name that is to be modified.

In place of options, we can pass different flags for different settings.

**Deleting a group**

To delete a group, we use the groupdel command as shown in the following snippet:

```
groupdel [groupname]
```

Root account

This is also called **superuser** and would have complete and unfettered control of the system. A superuser can run any commands without any restriction. This user should be assumed as a system administrator.

System accounts

System accounts are those needed for the operation of system-specific components for example mail accounts and the **sshd** accounts. These accounts are usually needed for some specific function on your system, and any modifications to them could adversely affect the system. User accounts

User accounts provide interactive access to the system for users and groups of users. General users are typically assigned to these accounts and usually have limited access to critical system files and directories.

Unix supports a concept of *Group Account* which logically groups a number of accounts. Every account would be a part of another group account. A Unix group plays important role in handling file permissions and process management.

Managing Users and Groups

There are four main user administration files −

- **/etc/passwd** − Keeps the user account and password information. This file holds the majority of information about accounts on the Unix system. • **/etc/shadow** − Holds the encrypted password of the corresponding account.

Not all the systems support this file.

- **/etc/group** − This file contains the group information for each account.
- **/etc/gshadow** − This file contains secure group account information.

Check all the above files using the **cat** command.

The following table lists out commands that are available on majority of Unix systems to create and manage accounts and groups −

| Sr.No. | Command & Description |
|---|---|
| 1 | **useradd**<br>Adds accounts to the system |
| 2 | **usermod**<br>Modifies account attributes |
| 3 | **userdel**<br>Deletes accounts from the system |
| 4 | **groupadd**<br>Adds groups to the system |
| 5 | **Groupmod**<br>Modifies group attributes |
| 6 | **Groupdel**<br>Removes groups from the system |

User Management in Linux

A user is an entity, in a Linux operating system, that can manipulate files and perform several other operations. Each user is assigned an ID that is unique for each user in the operating system. In this post, we will learn about users and commands which are used to get information about the users. After installation of the operating system, the **ID 0 is assigned to the root user**

and the IDs 1 to 999 (both inclusive) are assigned to the system users and hence the ids for local user begins from 1000 onwards.
In a single directory, we can create 60,000 users. Now we will discuss the important commands to manage users in Linux.

**1.** To ***list out all the users in Linux***, use the awk command with -F option. Here, we are accessing a file and printing only first column with the help of *print $1* and *awk*. awk -F':' '{ print $1}' /etc/passwd

**2.** Using id command, you can get the ***ID of any username***. Every user has an id assigned to it and the user is identified with the help of this id. By default, this id is also the group id of the user. id username

**3.** The command to add a user. *useradd command* adds a new user to the directory. The user is given the ID automatically depending on which category it falls in. The username of the user will be as provided by us in the command. sudo useradd username

**4.** Using passwd command to assign a password to a user. After using this command we have to enter the new password for the user and then the password gets updated to the new password. passwd username

**5.** ***Accessing a user configuration file***. cat /etc/passwd

This commands prints the data of the configuration file. This file contains information about the user in the format.

username : x : user id : user group id : : /home/username : /bin/bash

**6.** The command to **change the user ID for a user**. usermod -u new_id username

This command can change the user ID of a user. The user with the given username will be assigned with the new ID given in the command and the old ID will be removed.

*Example:* sudo usermod -u 1982 test

**7.** Command to **Modify the group ID of a user**. usermod -g new_group_id username

This command can change the group ID of a user and hence it can even be used to move a user to an already existing group. It will change the group ID of the user whose username is given and sets the group ID as the given new_group_id.

*Example:* sudo usermod -g 1005 test

**8.** You can **change the user login name** using *usermod* command. The below command is used to change the login name of the user. The old login name of the user is changed to the new login name provided. sudo usermod -l new_login_name old_login_name

**9.** The **command to change the home directory**. The below command change the home directory of the user whose username is given and sets the new home directory as the directory whose path is provided.
usermod -d new_home_directory_path username

**10.** You can also **delete a user name**. The below command deletes the user whose username is provided. Make sure that the user is not part of a group. If the user is part of a group then it will not be deleted directly, hence we will have to first remove him from the group and then we can delete him. userdel -r username

**Unix commands with detailed description:**

1. **who:** The '$ who' command displays all the users who have logged into the system currently. As shown above on my system I am the only user currently logged in.The thing tty2 is terminal line the user is using and the next line gives the current date and time

$ who
Output: harssh tty2 2017-07-18 09:32 (:0)

2. **pwd:** The '$pwd' command stands for 'print working directory' and as the name says,it displays the directory in which we are currently (directory is same as folder for Windows OS users).

In the output we are harssh directory(folder for Windows OS that are moving to Linux),which is present inside the home directory

$ pwd
Output: /home/harssh

3. **mkdir:** The '$ mkdir' stands for 'make directory' and it creates a new directory.We have used '$ cd' (which is discussed below) to get into the newly created directory and again on giving '$ pwd' command,we are displayed with the new 'newfolder' directory.

$ mkdir newfolder
$ cd newfolder
$ pwd
Output: /home/harssh/newfolder

4. **rmdir:** The '$ rmdir' command deletes any directory we want to delete and you can remember it by its names 'rmdir' which stands for 'remove directory'.

$ rmdir newfolder

**5. cd:** The '$ cd' command stands for 'change directory' and it changes your current directory to the 'newfolder' directory.You can understand this a double-clicking a folder and then you do some stuff in that folder.

$ cd newfolder (assuming that there is a directory named 'newfolder' on your system)

**6. ls:** The 'ls' command simply displays the contents of a directory.
$ ls
Output: Desktop Documents Downloads Music Pictures Public Scratch Templates Videos

**7. touch:** The '$ touch' command creates a file(not directory) and you can simple add an extension such as .txt after it to make it a Text File.

$ touch example
$ ls
Output: Desktop Documents Downloads Music Pictures Public Scratch Templates Videos example
Note: It is important to note that according to the Unix File structure, Unix treats all the stuff it has as a 'file', even the directories(folders) are also treated as a file.You will get to know more about this as you will further use Linux/Unix based OS.

**8. cp:** This '$ cp ' command stands for 'copy' and it simply copy/paste the file wherever you want to.In the above example, we are copying a file 'file.txt' from the directory harssh to a new directory new.

$ cp /home/harssh/file.txt /home/harssh/new/

**9. mv:** The '$ mv' command stands for 'move' and it simply move a file from a directory to another directory.In the above example a file named 'file.txt' is being moved into a new directory 'new'

$ mv /home/harssh/file.txt /home/harssh/new

**10. rm:** The '$ rm ' command for remove and the '-r' simply recursively deletes file. Try '$ rm filename.txt' at your terminal

$ rm file.txt

**11. chmod:** The '$ chmod' command stands for change mode command.As there are many modes in Unix that can be used to manipulate files in the Unix environment.Basically there are 3 modes that we can use with the 'chmod' command 1. +w (stands for write and it changes file permissions to write)
2. +r (stands for read and it changes file permissions to read)
3. +x (generally it is used to make a file executable)

$ chmod +w file.txt

$ chmod +r file.txt
$ chmod +x file.txt

**12. cal:** The '$ cal' means calendar and it simply display calendar on to your screen.

$ cal
Output : July 2017
Su Mo Tu We Th Fr Sa
1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31

**13. file:** The '$ file' command displays the type of file.As I mentioned earlier Linux treats everything as a file so on executing the command file on a directory(Downloads) it displays directory as the output

$ ls
Output: Desktop Documents Downloads Music Pictures Public Scratch Templates Videos
$ file Downloads
Output: Downloads: directory

**14. sort:** As the name suggests the '$ sort' sorts the contents of the file according to the ASCII rules.

$ sort file

**15. grep:** grep is an acronym for 'globally search a regular expression and print it'.The '$ grep' command searches the specified input fully(globally) for a match with the supplied pattern and displays it.

In the example, this would search for the word 'picture' in the file newsfile and if found,the lines containing it would be displayed on the screen.

$ grep picture newsfile

**16. man:** The '$ man' command stands for 'manual' and it can display the in-built manual for most of the commands that we ever need.In the above example, we can read about the '$ pwd' command.

$ man pwd

**17. lpr:** The '$ lpr' command send a file to the printer for printing.

$ lpr new.txt

**18. passwd:** The '$ passwd' command simply changes the password of the user.In above case 'harssh' is the user.

$ passwd
Output: Changing password for harssh.
(current) UNIX password:

**19. clear:** The '$ clear' command is used to clean up the terminal so that you can type with more accuracy

$ clear

**20. history:** The '$ history' command is used to get list of previous commands may be obtained by executing the following command. you can also use parameters like !n to reexecute the nth command, !! to executes the most recent command, and !cp this will execute the most recent command that starts with cp.

$ history

**21. cat --- for creating and displaying short files**

Cat(concatenate) command is very frequently used in Linux. It reads data from the file and gives their content as output. It helps us to create, view, concatenate files. So let us see some frequently used cat commands.

To view a single file Command:

$cat filename

**22. date --- display date** date command is used to display the system date and time. date command is also used to set date and time of the system. By default the date command displays the date in the time zone on which unix/linux operating system is configured.You must be the super-user (root) to change the date and time.

Syntax:

date [OPTION]... [+FORMAT]
date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]

**23. echo --- echo argument** echo command in linux is used to display line of text/string that are passed as an argument . This is a built in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.

Syntax :

echo [option] [string]

Displaying a text/string : Syntax :

echo [string]

**24. ftp --- connect to a remote machine to download or upload files** ftp command examples
Both ftp and secure ftp (sftp) has similar commands. To connect to a remote server and download multiple files, do the following.

$ ftp IP/hostname
ftp> mget *.html

**25. head --- display first part of file**
It is the complementary of Tail command. The head command, as the name implies, print the top N number of data of the given input. By default, it prints the first 10 lines of the specified files. If more than one file name is provided then data from each file is preceded by its file name.

Syntax:

head [OPTION]... [FILE]...

**26. more --- use to read files** more command is used to view the text files in the command prompt, displaying one screen at a time in case the file is large (For example log files). The more command also allows the user do scroll up and down through the page. The syntax along with options and command is as follows. Another application of more is to use it with some other command after a pipe. When the output is large, we can use more command to see output one by one.

Syntax:

more [-options] [-num] [+/pattern] [+linenum] [file_name]

[-options]: any option that you want to use in order to change the way the file is displayed. Choose any one from the followings: (-d, -l, -f, -p, -c, -s, -u)
[-num]: type the number of lines that you want to display per screen.
[+/pattern]: replace the pattern with any string that you want to find in the text file.
[+linenum]: use the line number from where you want to start displaying the text content.
[file_name]: name of the file containing the text that you want to display on the screen.

**27. tail --- display last part of file**
It is the complementary of head command.The tail command, as the name implies, print the last N number of data of the given input. By default it prints the last 10 lines of the specified files. If more than one file name is provided then data from each file is precedes by its file name.

Syntax:

tail [OPTION]... [FILE]...

**28. tar --- create an archive, add or extract files**
The Linux 'tar' stands for tape archive, is used to create Archive and extract the Archive files. tar command in Linux is one of the important command which provides archiving functionality in Linux. We can use Linux tar command to create compressed or uncompressed Archive files and also maintain and modify them.

Syntax:

tar [options] [archive-file] [file or directory to be archived]
Options:
-c : Creates Archive
-x : Extract the archive
-f : creates archive with given filename
-t : displays or lists files in archived file
-u : archives and adds to an existing archive file
-v : Displays Verbose Information
-A : Concatenates the archive files
-z : zip, tells tar command that creates tar file using gzip
-j : filter archive tar file using tbzip
-W : Verify a archive file
-r : update or add file or directory in already existed .tar file

$ tar cvf archive_name.tar dirname/ Extract
from an existing tar archive.

$ tar xvf archive_name.tar View
an existing tar archive.

$ tar tvf archive_name.tar

**29. wc --- count characters, words, lines** wc stands for word count. As the name implies, it is mainly used for counting purpose.

It is used to find out number of lines, word count, byte and characters count in the files specified in the file arguments.
By default it displays four-columnar output.
First column shows number of lines present in a file specified, second column shows number of words present in the file, third column shows number of characters present in file and fourth column itself is the file name which are given as argument. Syntax:

wc [OPTION]... [FILE]...

**30. find command**
Find files using file-name ( case in-sensitve find)

# find -iname "MyCProgram.c"
Execute commands on files found by the find command

$ find -iname "MyCProgram.c" -exec md5sum {} \;
Find all empty files in home directory

# find ~ -empty

**31. sed command**
When you copy a DOS file to Unix, you could find \r\n in the end of each line. This example converts the DOS file format to Unix file format using sed command.

$sed 's/.$//' filename
Print file content in reverse order

$ sed -n '1!G;h;$p' thekstuff.txt
Add line number for all non-empty-lines in a file

$ sed '/./=' thekstuff.txt | sed 'N; s/\n/ /'

**32. awk command**
Remove duplicate lines using awk

$ awk '!($0 in array) { array[$0]; print }' temp
Print all lines from /etc/passwd that has the same uid and gid

$awk -F ':' '$3==$4' passwd.txt Print
only specific field from a file.

$ awk '{print $2,$5;}' employee.txt

**33. vim command**
Go to the 123rd line of file

$ vim +123 filename.txt
Go to the first match of the specified

$ vim +/search-term filename.txt Open
the file in read only mode.

$ vim -R /etc/passwd

**34. gzip command**
To create a *.gz compressed file:

$ gzip test.txt
To uncompress a *.gz file:

$ gzip -d test.txt.gz

**35. unzip command**
To extract a *.zip compressed file:

$ unzip test.zip
View the contents of *.zip file (Without unzipping it):

$ unzip -l jasper.zip

**36. shutdown command**
Shutdown the system and turn the power off immediately.

# shutdown -h now
Shutdown the system after 10 minutes.

# shutdown -h +10
Reboot the system using shutdown command.

# shutdown -r now
Force the filesystem check during reboot.

# shutdown -Fr now

**37. kill command**
Use kill command to terminate a process. First get the process id using ps -ef command, then use kill -9 to kill the running Linux process as shown below. You can also use killall, pkill, xkill to terminate a unix process.

$ ps -ef | grep vim
ram    7243  7222  9 22:43 pts/2    00:00:00 vim

$ kill -9 7243

**38. ifconfig command** ifconfig(interface configuration) command is used to configure the kernel-resident network interfaces. It is used at the boot time to set up the interfaces as necessary. After that, it is usually used when needed during debugging or when you need system tuning. Also, this command is used to assign the IP address and netmask to an interface or to enable or disable a given interface.

Syntax:

ifconfig [...OPTIONS] [INTERFACE]

**39. mount command**
To mount a file system, you should first create a directory and mount it as shown below.

# mkdir /u01

# mount /dev/sdb1 /u01
You can also add this to the fstab for automatic mounting. i.e Anytime system is restarted, the filesystem will be mounted.

/dev/sdb1 /u01 ext2 defaults 0 2

**40. host command** host command in Linux system is used for DNS (Domain Name System) lookup operations. In simple words, this command is used to find the IP address of a particular domain name or if you want to find out the domain name of a particular IP address the host command becomes handy. You can also find more specific details of a domain by specifying the corresponding option along with the domain name.

Syntax:

host [-aCdlriTWV] [-c class] [-N ndots] [-t type] [-W time]
[-R number] [-m flag] hostname [server]

**41. wait Command**
wait is a built-in command of Linux that waits for completing any running process. wait command is used with a particular process id or job id. If no process id or job id is given with wait command then it will wait for all current child processes to complete and returns exit status. Create a file named ‘wait_example.sh’ and add the following script.

#!/bin/bash echo "Wait
command" & process_id=$!
wait $process_id echo "Exited

with status $?" Run the file with
bash command.

$ bash wait_example.sh

**42. sleep Command**

When you want to pause the execution of any command for specific period of time then you can use sleep command. You can set the delay amount by seconds (s), minutes (m), hours (h) and days (d). Create a file named 'sleep_example.sh' and add the following script. This script will wait for 5 seconds after running.

#!/bin/bash

echo "Wait for 5 seconds"
sleep 5 echo
"Completed"
Run the file with bash command.

$ bash sleep_example.sh

**Unix Basic Commands**

| Command | Description |
|---|---|
| ls | Lists all files and directories in the present working directory |
| ls – r | Lists files in sub-directories as well |
| ls – a | Lists hidden files as well |
| ls – al | Lists files and directories with detailed information like permissions, size, owner, etc. |
| cat > filename | Creates a new file |
| cat filename | Displays the file content |
| cat file1 file2 > file3 | Joins two files (file1, file2) and stores the output in a new file (file3) |
| mv file "new file path" | Moves the files to the new location |
| mv filename new_file_name | Renames the file to a new filename |
| sudo | Allows regular users to run programs with the security privileges of the superuser or root |
| rm filename | Deletes a file |
| Man | Gives help information on a command |
| History | Gives a list of all past basic Linux commands list typed in the current terminal session |
| clear | Clears the terminal |
| mkdir directoryname | Creates a new directory in the present working directory or a at the specified path |

| | |
|---|---|
| Rmdir | Deletes a directory |
| mv | Renames a directory |
| pr -x | Divides the file into x columns |
| pr -h | Assigns a header to the file |
| pr -n | Denotes the file with Line Numbers |
| lp -nc<br>lpr c | Prints “c” copies of the File |
| lp -d lpr<br>-P | Specifies name of the printer |
| apt-get | Command used to install and update packages |
| mail -s ‘subject’ -c ‘ccaddress’<br>-b ‘bcc-address’ ‘toaddress’ | Command to send email |
| mail -s “Subject” to-address < filename | Command to send email with attachment |

**Shell Programming**

**Create and Execute First BASH Program:**

You can run bash script from the terminal or by executing any bash file. Run the following command from the terminal to execute a very simple bash statement. The output of the command will be ‘Hello World’

$ echo “Hello World”

Open any editor to create a bash file. Here, nano editor is used to create the file and filename is set as ‘First.sh’

$ nano First.sh
Add the following bash script to the file and save the file.

#!/bin/bash
echo “Hello World”

You can run bash file by two ways. One way is by using bash command and another is by setting execute permission to bash file and run the file. Both ways are shown here.

$ bash First.sh
Or,

$ chmod a+x First.sh
$ ./First.sh

**Use of echo command:**

You can use echo command with various options. Some useful options are mentioned in the following example. When you use 'echo' command without any option then a newline is added by default. '-n' option is used to print any text without new line and '-e' option is used to remove backslash characters from the output. Create a new bash file with a name, 'echo_example.sh' and add the following script.

```
echo "Printing text with newline" echo -n
"Printing text without newline" echo -e
"\nRemoving \t backslash \t characters\n" Run
```

the file with bash command.

```
$ bash echo_example.sh
```

**Use of comment:**

'#' symbol is used to add single line comment in bash script. Create a new file named 'comment_example.sh' and add the following script with single line comment.

```
#!/bin/bash

# Add two numeric value
((sum=25+35))

#Print the result echo
$sum
```

Run the file with bash command.

```
$ bash comment_example.sh
```

**Use of Multi-line comment:**

You can use multi line comment in bash in various ways. A simple way is shown in the following example. Create a new bash named, 'multiline-comment.sh' and add the following script. Here, ':' and " ' " symbols are used to add multiline comment in bash script. This following script will calculate the square of 5.

```
#!/bin/bash
: '
The following script calculates the
square value of the number, 5.
'
((area=5*5)) echo
$area
```

Run the file with bash command.

$ bash multiline-comment.sh

**Using While Loop:**

Create a bash file with the name, 'while_example.sh', to know the use of while loop. In the example, while loop will iterate for 5 times. The value of count variable will increment by 1 in each step. When the value of count variable will 5 then the while loop will terminate.

```
#!/bin/bash
valid=true
count=1 while [
$valid ] do echo
$count if [ $count
-eq 5 ];
then
break
fi
((count++)) done
```

Run the file with bash command.

$ bash while_example.sh

**Using For Loop:**

The basic for loop declaration is shown in the following example. Create a file named 'for_example.sh' and add the following script using for loop. Here, for loop will iterate for 10 times and print all values of the variable, counter in single line.

```
#!/bin/bash
for (( counter=10; counter>0; counter-- )) do
echo -n "$counter "
done printf
"\n"
```

Run the file with bash command.

$ bash for_example.sh

**Get User Input:**

'read' command is used to take input from user in bash. Create a file named 'user_input.sh' and add the following script for taking input from the user. Here, one string value will be taken from the user and display the value by combining other string value.

```
#!/bin/bash echo "Enter Your Name"
read name echo "Welcome $name to
LinuxHint" Run the file with bash
command.

$ bash user_input.sh
```

**Using if statement:**

You can use if condition with single or multiple conditions. Starting and ending block of this statement is define by 'if' and 'fi'. Create a file named 'simple_if.sh' with the following script to know the use if statement in bash. Here, 10 is assigned to the variable, n. if the value of $n is less than 10 then the output will be "It is a one digit number", otherwise the output will be "It is a two digit number". For comparison, '-lt' is used here. For comparison, you can also use '-eq' for equality, '-ne' for not equality and '-gt' for greater than in bash script.

```
#!/bin/bash n=10
if [ $n -lt 10 ]; then
echo "It is a one digit number" else
echo "It is a two digit number" fi
Run the file with bash command.

$ bash simple_if.sh
```

**Using if statement with AND logic:**

Different types of logical conditions can be used in if statement with two or more conditions. How you can define multiple conditions in if statement using AND logic is shown in the following example. '&&' is used to apply AND logic of if statement. Create a file named 'if_with_AND.sh' to check the following code. Here, the value of username and password variables will be taken from the user and compared with 'admin' and 'secret'. If both values match then the output will be "valid user", otherwise the output will be "invalid user".

```
#!/bin/bash

echo "Enter
username" read
username echo "Enter
password"
read password

if [[ ( $username == "admin" && $password == "secret" ) ]];
then echo "valid user" else
echo "invalid user"
fi
```

Run the file with bash command.

$ bash if_with_AND.sh

**Using if statement with OR logic:**

'||' is used to define OR logic in if condition. Create a file named 'if_with_OR.sh' with the following code to check the use of OR logic of if statement. Here, the value of n will be taken from the user. If the value is equal to 15 or 45 then the output will be "You won the game", otherwise the output will be "You lost the game".

```
echo "Enter any number"
read n

if [[ ( $n -eq 15 || $n  -eq 45 )
]] then echo "You won the
game" else
echo "You lost the game"
fi
```

Run the file with bash command.

$ bash if_with_OR.sh

**Using else if statement:**

The use of else if condition is little different in bash than other programming language. 'elif' is used to define else if condition in bash. Create a file named, 'elseif_example.sh' and add the following script to check how else if is defined in bash script.

```
Echo "Enter your lucky number"
read n

if [ $n -eq 101 ];
then
echo "You got 1st prize" elif
[ $n -eq 510 ];
then
echo "You got 2nd prize" elif
[ $n -eq 999 ];
then
echo "You got 3rd prize"
else
echo "Sorry, try for the next time" fi
```

Run the file with bash command. $ bash elseif_example.sh

**Using Case Statement:**

Case statement is used as the alternative of if-elseif-else statement. The starting and ending block of this statement is defined by 'case' and 'esac'. Create a new file named, 'case_example.sh' and add the following script. The output of the following script will be same to the previous else if example.

```
#!/bin/bash

echo "Enter your lucky number"
read n case $n in 101) echo
echo "You got 1st prize" ;;
510)
echo "You got 2nd prize" ;;
999)
echo "You got 3rd prize" ;;
*)
echo "Sorry, try for the next time" ;; esac
```

Run the file with bash command.

```
$ bash case_example.sh
```

**Get Arguments from Command Line:**

Bash script can read input from command line argument like other programming language. For example, $1 and $2 variable are used to read first and second command line arguments.
Create a file named "command_line.sh" and add the following script. Two argument values read by the following script and prints the total number of arguments and the argument values as output.

```
#!/bin/bash echo "Total
arguments : $#" echo "1st
Argument = $1" echo "2nd
argument = $2" Run the file
with bash command.

$ bash command_line.sh Linux Hint
```

**Get arguments from command line with names:**

How you can read command line arguments with names is shown in the following script. Create a file named, 'command_line_names.sh' and add the following code. Here, two arguments, X and Y are read by this script and print the sum of X and Y.

```
#!/bin/bash for arg in "$@" do
index=$(echo $arg | cut -f1 -
d=) val=$(echo $arg | cut -f2 -
d=)
case $index in
X) x=$val;;

Y) y=$val;;

*) esac done
((result=x+y)) echo
"X+Y=$result"
```

Run the file with bash command and with two command line arguments.

```
$ bash command_line_names X=45 Y=30
```

**Combine String variables:**

You can easily combine string variables in bash. Create a file named "string_combine.sh" and add the following script to check how you can combine string variables in bash by placing variables together or using '+' operator.

```
#!/bin/bash
string1="Linux" string2="Hint" echo
"$string1$string2"
string3=$string1+$string2 string3+="
is a good tutorial blog site" echo
$string3
```

Run the file with bash command.

```
$ bash string_combine.sh
```

**Get substring of String:**

Like other programming language, bash has no built-in function to cut value from any string data. But you can do the task of substring in another way in bash that is shown in the following script. To test the script, create a file named 'substring_example.sh' with the following code. Here, the value, 6 indicates the starting point from where the substring will start and 5 indicates the length of the substring.

```
#!/bin/bash
Str="Learn Linux from LinuxHint"
subStr=${Str:6:5} echo $subStr
```
Run the file with bash command.

```
$ bash substring_example.sh
```

**Add Two Numbers:**

You can do the arithmetical operations in bash in different ways. How you can add two integer numbers in bash using double brackets is shown in the following script. Create a file named 'add_numbers.sh' with the following code. Two integer values will be taken from the user and printed the result of addition.

```
#!/bin/bash
echo "Enter first number" read
x
echo "Enter second number"
read  y  ((
sum=x+y ))
echo "The result of addition=$sum" Run
```
the file with bash command.

```
$ bash add_numbers.sh
```

**Create Function:**

How you can create a simple function and call the function is shown in the following script. Create a file named 'function_example.sh' and add the following code. You can call any function by name only without using any bracket in bash script.

```
#!/bin/bash function
F1()
{
echo 'I like bash programming'
}

F1
```

Run the file with bash command.

```
$ bash function_example.sh
```

**Create function with Parameters:**

Bash can't declare function parameter or arguments at the time of function declaration. But you can use parameters in function by using other variable. If two values are passed at the time of function calling then $1 and $2 variable are used for reading the values. Create a file named 'function|_parameter.sh' and add the following code. Here, the function, 'Rectangle_Area' will calculate the area of a rectangle based on the parameter values.

```
#!/bin/bash

Rectangle_Area() {
area=$(($1 * $2))
echo "Area is : $area"
}

Rectangle_Area 10 20
```

Run the file with bash command.

```
$ bash function_parameter.sh
```

**Pass Return Value from Function:**

Bash function can pass both numeric and string values. How you can pass a string value from the function is shown in the following example. Create a file named, 'function_return.sh' and add the following code. The function, greeting() returns a string value into the variable, val which prints later by combining with other string.

```
#!/bin/bash  function
greeting() {

str="Hello, $name"
```

```
echo $str

}
echo "Enter your name" read
name

val=$(greeting) echo "Return value of the
function is $val" Run the file with bash
command.

$ bash function_return.sh
```

**Make Directory:**

Bash uses ‘mkdir’ command to create a new directory. Create a file named ‘make_directory.sh’ and add the following code to take a new directory name from the user. If the directory name is not exist in the current location then it will create the directory, otherwise the program will display error.

```
#!/bin/bash
echo “Enter directory name” read
newdir
`mkdir $newdir`
```

Run the file with bash command.

```
$ bash make_directory.sh
```

**Make directory by checking existence:**

If you want to check the existence of directory in the current location before executing the ‘mkdir’ command then you can use the following code. ‘-d’ option is used to test a particular directory is exist or not. Create a file named, ‘directory_exist.sh’ and add the following code to create a directory by checking existence.

```
#!/bin/bash     echo     "Enter
directory name"
read ndir if [ -d
"$ndir" ]
then
echo "Directory exist"
else `mkdir $ndir` echo
"Directory created"
fi
```

Run the file with bash command. $ bash directory_exist.sh

**Read a File:**

You can read any file line by line in bash by using loop. Create a file named, 'read_file.sh' and add the following code to read an existing file named, 'book.txt'.

```
#!/bin/bash
file='book.txt'
while read line;
do echo $line
done < $file
```

Run the file with bash command.

```
$ bash read_file.sh
```

Run the following command to check the original content of 'book.txt' file.

```
$ cat book.txt
```

**Delete a File:**

'rm' command is used in bash to remove any file. Create a file named 'delete_file.sh' with the following code to take the filename from the user and remove. Here, '-i' option is used to get permission from the user before removing the file.

```
#!/bin/bash
echo "Enter filename to remove"
read fn rm
-i $fn
```

Run the file with bash command.

```
$ ls
$ bash delete_file.sh
$ ls
```

**Append to File:**

New data can be added into any existing file by using '>>' operator in bash. Create a file named 'append_file.sh' and add the following code to add new content at the end of the file. Here, 'Learning Laravel 5' will be added at the of 'book.txt' file after executing the script.

```
#!/bin/bash

echo "Before appending the file"
cat book.txt
```

```
echo “Learning Laravel 5”>>
book.txt echo “After appending the
file” cat book.txt
```
Run the file with bash command.
```
$ bash append_file.sh
```

**Test if File Exist:**

You can check the existence of file in bash by using ‘-e’ or ‘-f’ option. ‘-f’ option is used in the following script to test the file existence. Create a file named, ‘file_exist.sh’ and add the following code. Here, the filename will pass from the command line.

```
#!/bin/bash
filename=$1 if [ -f
"$filename" ]; then echo
"File exists"
else
echo "File does not exist" fi
```
Run the following commands to check the existence of the file. Here, book.txt file exists and book2.txt is not exist in the current location.

```
$ ls
$ bash file_exist.sh book.txt
$ bash file_exist.sh book2.txt
```

**Send Email:**

You can send email by using ‘mail’ or ‘sendmail’ command. Before using these commands, you have to install all necessary packages. Create a file named, ‘mail_example.sh’ and add the following code to send the email.

```
#!/bin/bash
Recipient=”admin@example.com”
Subject=”Greeting”
Message=”Welcome to our site”
`mail -s $Subject $Recipient <<< $Message` Run
```
the file with bash command.

```
$ bash mail_example.sh
```

**Get Parse Current Date:**

You can get the current system date and time value using `date` command. Every part of date and time value can be parsed using 'Y', 'm', 'd', 'H', 'M' and 'S'. Create a new file named 'date_parse.sh' and add the following code to separate day, month, year, hour, minute and second values.

```
#!/bin/bash
Year=`date +%Y`
Month=`date +%m`
Day=`date +%d`
Hour=`date +%H`
Minute=`date +%M` Second=`date +%S`
echo `date` echo "Current Date is: $Day-
$Month-$Year" echo "Current Time is:
$Hour:$Minute:$Second" Run the file
with bash command.
```

```
$ bash date_parse.sh
```

**Wait Command:**

wait is a built-in command of Linux that waits for completing any running process. wait command is used with a particular process id or job id. If no process id or job id is given with wait command then it will wait for all current child processes to complete and returns exit status. Create a file named 'wait_example.sh' and add the following script.

```
#!/bin/bash echo "Wait
command" & process_id=$!
wait $process_id echo
"Exited with status $?" Run
the file with bash
command.
```

```
$ bash wait_example.sh
```

**Sleep Command:**

When you want to pause the execution of any command for specific period of time then you can use sleep command. You can set the delay amount by seconds (s), minutes (m), hours (h) and days (d). Create a file named ‘sleep_example.sh’ and add the following script. This script will wait for 5 seconds after running.

```
#!/bin/bash

echo "Wait for 5 seconds"
sleep  5  echo
"Completed"
```

Run the file with bash command.

```
$ bash sleep_example.sh
```