Unit 1 : Introduction to Linux Operating System

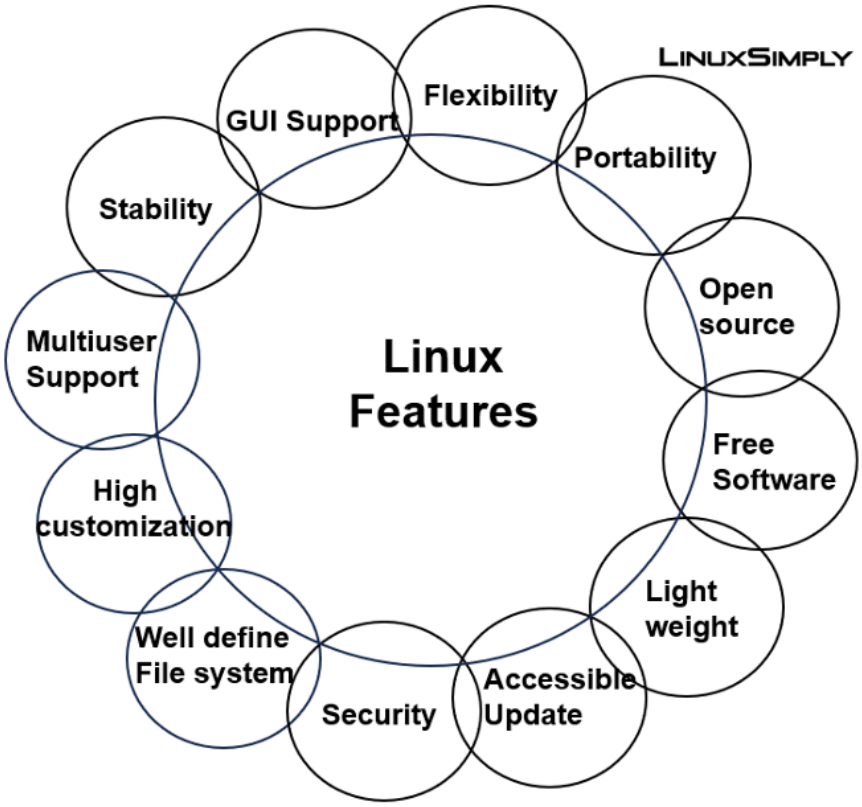
What is Linux?

Linux is based on the **UNIX** operating system. **UNIX** is a powerful, multi-user, multitasking operating system originally developed in the **1970s at AT&T Bell Labs**. It laid the foundation for many modern operating systems, including Linux.

While UNIX is a **licensed operating system** (meaning you need to purchase a license to use it), **Linux is free and open-source**, making it accessible to everyone. Anyone can inspect and modify the source code, which enables global collaboration and innovation. Its efficient performance and strong security model make it suitable for a wide variety of devices and industries.

1.1 Features of Linux OS

Linux, a popular open-source operating system, is known for its features like being free and open-source, supporting multiple users and tasks simultaneously, offering strong security, being highly customizable, and providing excellent performance and stability. It also boasts a hierarchical file system, a command-line interface through the shell, and supports various hardware architectures.



Key Features of Linux:

- **Open Source:**

  The source code is publicly available, allowing for community-driven development, modification, and distribution.

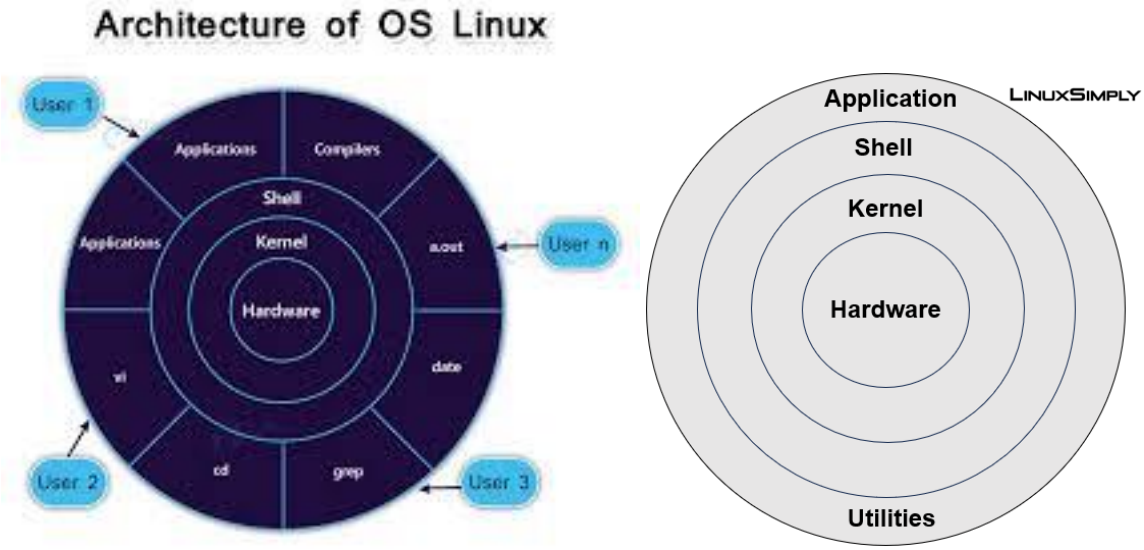
- **Multiuser:**

  Multiple users can access the system's resources concurrently, such as RAM and storage.

- **Multitasking/Multiprogramming:**
  The system can run multiple applications concurrently, improving efficiency.
- **Security:**
  Linux is known for its strong security features, including user permissions, encryption, and a proactive community focused on addressing vulnerabilities.
- **Portability:**
  The OS can be adapted to run on various hardware platforms.
- **Hierarchical File System:**
  Organizes files and directories in a structured tree-like manner for efficient access and management.
- **Shell/Command Line Interface:**
  Provides a powerful command-line interface (CLI) for executing commands and managing the system.
- **Graphical User Interface (GUI):**
  While it has a strong CLI, Linux also supports various GUI environments.
- **Stability:**
  Linux is known for its stability and reliability, rarely crashing or freezing.
- **Customization:**
  Users can customize the system to their specific needs and preferences.
- **Performance:**
  Linux can be optimized for performance, making it suitable for resource-intensive tasks.
- **Package Management:**
  Built-in package managers simplify the installation, update, and removal of software.
- **Community Support:**
  A large and active community provides support, documentation, and resources.
- **Interoperability:**
  Linux works well with other systems and platforms.
- **Live CD/USB:**
  Many Linux distributions can be run from a USB drive or CD without installation.
- **Energy Efficiency:**
  Linux can be configured to be energy efficient, making it suitable for various devices.

Architecture of Linux :

Linux architecture has the following components:



Linux Architecture

1. **Kernel:** Kernel is the core of the Linux based operating system. It virtualizes the common hardware resources of the computer to provide each process with its virtual resources. This makes the process seem as if it is the sole process running on the machine. The kernel is also responsible for preventing and mitigating conflicts between different processes. Different types of the kernel are:

- Monolithic Kernel
- Hybrid kernels
- Micro kernels

2. **System Library:** Linux uses system libraries, also known as shared libraries, to implement various functionalities of the operating system. These libraries contain pre-written code that applications can use to perform specific tasks. By using these libraries, developers can save time and effort, as they don't need to write the same code repeatedly. System libraries act as an interface between applications and the kernel, providing a standardized and efficient way for applications to interact with the underlying system.
3. **Shell:** The shell is the user interface of the Linux Operating System. It allows users to interact with the system by entering commands, which the shell interprets and executes. The shell serves as a bridge between the user and the kernel, forwarding the user's requests to the kernel for processing. It provides a convenient way for users to perform various tasks, such as running programs, managing files, and configuring the system.
4. **Hardware Layer:** The hardware layer encompasses all the physical components of the computer, such as RAM (Random Access Memory), HDD (Hard Disk Drive), CPU (Central Processing Unit), and input/output devices. This layer is responsible for interacting with the Linux Operating System and providing the necessary resources for the system and applications to function properly. The Linux kernel and system libraries enable communication and control over these hardware components, ensuring that they work harmoniously together.

5. **System Utility:** System utilities are essential tools and programs provided by the Linux Operating System to manage and configure various aspects of the system. These utilities perform tasks such as installing software, configuring network settings, monitoring system performance, managing users and permissions, and much more. System utilities simplify system administration tasks, making it easier for users to maintain their Linux systems efficiently.

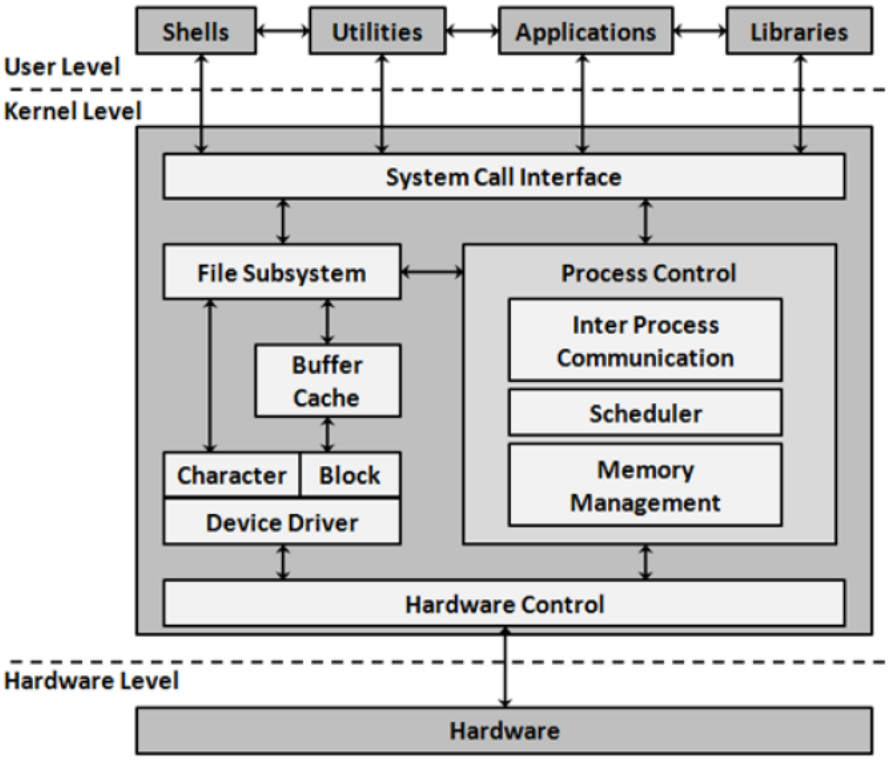
How is the Linux Operating System Used

The Linux operating system is widely used across various domains due to its flexibility, security, and open-source nature:

- **Servers and Hosting**: Powers web servers, cloud infrastructure, and database management systems.
- **Development**: Used by developers for coding, debugging, and running applications.
- **Desktop and Personal Use**: Provides secure and customizable desktop environments.
- **Cyber security**: Essential for ethical hacking, penetration testing, and security research.
- **Embedded Systems**: Runs lightweight devices like routers, IoT gadgets, and smart appliances.
- **Supercomputers**: Dominates high-performance computing for scientific research and simulations.
- **Education**: A cost-effective tool for teaching programming and system administration.

KERNEL and IT's Architecture :

The kernel is the core of the operating system. Kernel is mostly written in C. It is loaded into memory when the system is booted and communicates directly with the hardware. User programs that need to access the hardware use the services of the kernel, which performs the job on the user's behalf. The programs access kernel through set of functions called system calls. The kernel program is usually stored in a file called "Unix".



Architecture of UNIX (Kernel Architecture) :

The UNIX architecture can be divided into ***three*** levels: User level, Kernel level, Hardware level.
The system call and library Interface represent the border between user programs and the kernel as shown in the figure. System calls are ordinary function calls in C programs and libraries map these functions calls to primitive needed to enter the operating system. Programs frequently use other libraries such as standard I/O library to provide more sophisticated use of the system calls. The libraries are linked with the programs at compile time.

The **system calls** are partitioned to the system calls that interact with the file sub system and the system calls that interact with the process control subsystem.

The **file subsystem manages** the files, allocates files space, administrating the free space, controlling access to files, and retrieving data for users. Processes interact with the file system through system calls. E.g. Open, close, read, write, etc. The files subsystem accesses the data using buffering mechanism that regulates dataflow between the kernel and secondary storage devices. The buffering mechanism interacts with block I/O devices drivers to initiate data transfer to and from the kernel.

Device drivers are the kernel modules that control the operation of peripheral devices. Block I/O devices are random access storage device to the rest of the system.

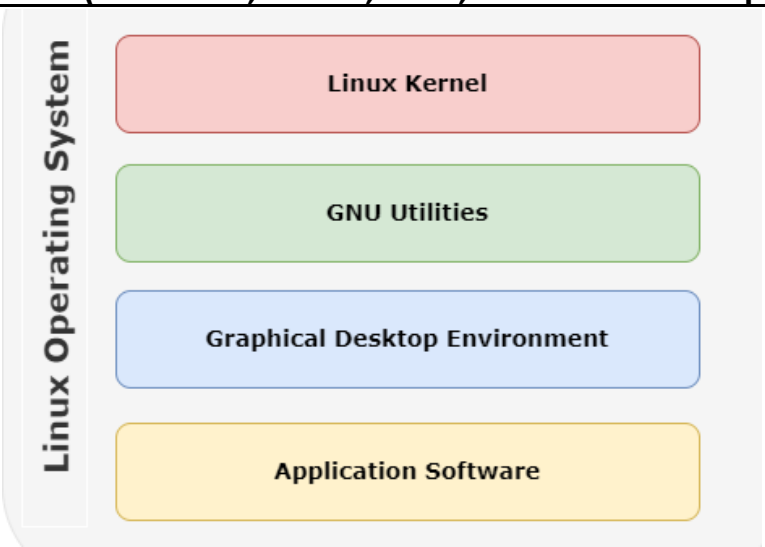
The **file subsystem also interacts** directly with the raw (Character devices) I/O device drivers without the intervention of a buffering mechanism.

The **process control subsystem** is responsible for process synchronization, inter-process communication, memory management, process scheduling. The system calls used with process control systems are fork (creating a new process), exec (overlay the image of a program onto the running process), Exit (finish executing a process), wait (Synchronize process execution with the exit of a previously forked process), and signal(control process response to extraordinary events).

The **memory management** module controls the allocation of memory. If at any time the system doesn't have enough physical memory for all processes, the kernel moves between main memory and secondary memory so that the all processes get a fair chance to execute.

The **scheduler module** allocates the CPU to processes. It schedules them to run in turn until they voluntarily relinquish the CPU while awaiting a resource or until the kernel preempts them when their recent run time exceeds a time quantum. The scheduler then chooses the highest priority eligible process to run; the original process will run when it is the highest priority eligible process available.
The **inter-process communication** provides message passing between processes. i.e.: it facilitates the communication between processes.
The **hardware control** is responsible for handling interrupts and for communicating with the machine. Devices such as disks or terminals may interrupt the CPU while a process is executing. The kernel executes the interrupt and then resumes the previously executing process. This way it provides access of hardware devices.

1.2Components of Linux OS (Hardware, Kernel, Shell, GNU Utilities & Applications)



The Linux operating system is composed of several key elements: the hardware, kernel, shell, GNU utilities, and applications. The hardware provides the physical components like CPU, RAM, and storage. The kernel acts as the core, managing resources and interacting with the hardware. The shell is the user interface, allowing commands to be executed. GNU utilities are a collection of tools for system management and user interaction. Finally, applications are the software programs that users utilize for various tasks.

- **Hardware:**

This encompasses the physical components of the computer system, including the CPU, memory (RAM), storage (hard drives, SSDs), input/output devices (keyboard, mouse, display), and network interfaces.

- **Kernel:**

The kernel is the heart of the Linux operating system. It manages the system's resources, including the CPU, memory, and input/output devices. It acts as a bridge between the hardware and the rest of the operating system, providing a layer of abstraction for applications.

- **Shell:**

The shell is a command-line interpreter that acts as an intermediary between the user and the kernel. It accepts commands from the user and translates them into actions that the kernel can understand and execute. Common shells include Bash, Zsh, and Fish.

- **GNU Utilities:**

GNU utilities are a collection of free software tools that provide a wide range of functionalities for system administration, file manipulation, text processing, and more. Examples include ls, grep, sed, awk, and bash (which is also a shell).

- **Applications:**

Applications are the software programs that users interact with to perform specific tasks. Examples include web browsers, word processors, media players, and games.

1.3 Shell in Linux (Bash, Zsh, Dash – Features and Differences)

Linux distributions utilize various shells, with Bash, Zsh, and Dash being prominent examples, each offering distinct features and use cases.

Bash (Bourne-Again Shell):

- **Features:** Default shell for most Linux distributions, widely compatible with Bourne Shell scripts, supports command history, aliases, job control, loops, conditionals, variables, arrays, and input/output redirection.

- **Role:**
  Primarily used for interactive command-line sessions and general-purpose shell scripting due to its robust features and widespread adoption.

Zsh (Z Shell):

- **Features:**

  Builds upon and extends Bash, offering enhanced features like improved command completion, built-in spelling correction, shared history, smart path expansion, and a highly customizable framework with themes and plugins (e.g., Oh My Zsh).

- **Role:**
  Favored by users seeking advanced customization, improved interactivity, and more powerful scripting capabilities than Bash.

Dash (Debian Almquist Shell):

- **Features:**

  A lightweight, POSIX-compliant shell optimized for speed and minimal resource consumption, often serving as /bin/sh in Debian-based systems (like Ubuntu) for executing system scripts. Lacks many interactive features found in Bash or Zsh.

- **Role:**
  Primarily used for system scripts and situations where performance and a small footprint are critical, rather than interactive user sessions.

Dash is significantly smaller and faster than Bash and Zsh, making it ideal for system scripts, whereas Bash and Zsh are larger and more feature-rich for interactive use.

- **Default Usage:**
  Bash is typically the default interactive shell, while Dash often serves as the default shell for executing system scripts. Zsh is an optional upgrade for users desiring more advanced features.

Table of Difference between Bash and Zsh

Bash	Zsh
Bash is the default shell for Linux and it is released in the replacement of Bourne Shell.	Z shell is built on top of the bash shell and is an extended version of the bash with plenty of new features.
Bash reads the .bashrc file in non-login interactive shell and .bash_profile in login shells.	Zsh reads .zshrc in an interactive shell and .zprofile in a login shell.
Bash uses backslash escapes.	Zsh uses percentage escapes.
Bash doesn't have an inline wildcard expansion.	Zsh has a built-in wildcard expansion.
Doesn't have customization options.	Zsh has many frameworks that provide customization.
It doesn't have many themes and plug-in support.	Has plenty of plug-in's and themes.

Bash	Zsh
Bash lacks syntax highlighting and auto-correction features.	Zsh has syntax highlighting and auto-correction features.
In bash keybinding is done using '.inputrc' and 'bind builtin'.	In zsh binding is done using 'bindkey builtin'.

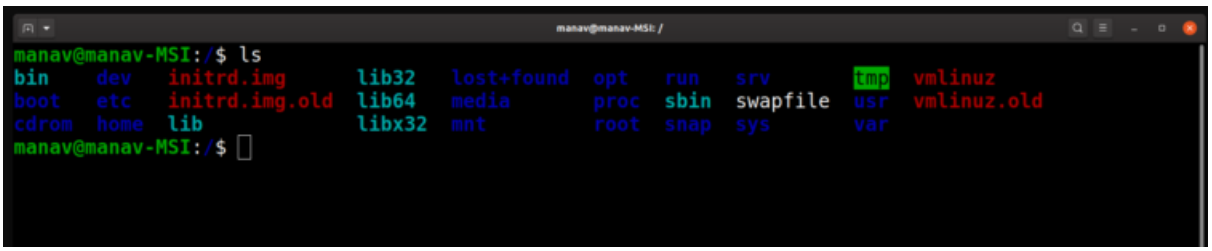
1.4 Introduction to Files and File Types in Linux (text, binary, special files)

In Linux, most of the operations are performed on files. And to handle these files Linux has directories also known as folders which are maintained in a tree-like structure. Though, these directories are also a type of file themselves. Linux has 3 types of files:

1. **Regular Files:** It is the common file type in Linux. it includes files like - text files, images, binary files, etc. Such files can be created using the touch command. They consist of the majority of files in the Linux/UNIX system. The regular file contains ASCII or Human Readable text, executable program binaries, program data and much more.
2. **Directories:** Windows call these directories as folders. These are the files that store the list of file names and the related information. The root directory(/) is the base of the system, /home/ is the default location for user's home directories, /bin for Essential User Binaries, /boot – Static Boot Files, etc. We could create new directories with mkdir command.
3. **Special Files:** Represents a real physical device such as a printer which is used for IO operations. Device or special files are used for device Input/Output(I/O) on UNIX and Linux systems. You can see them in a file system like an ordinary directory or file.
4. **Files Listing**

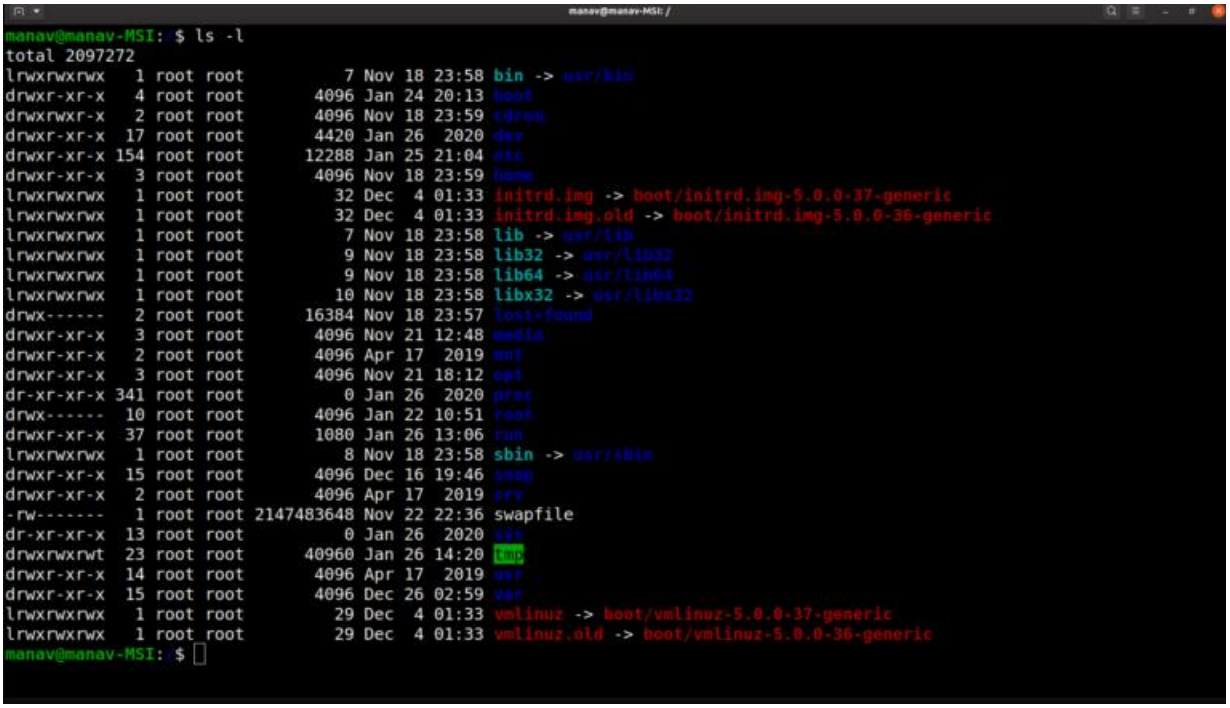
   To perform Files listings or to list files and directories ls command is used

```
$ls
```



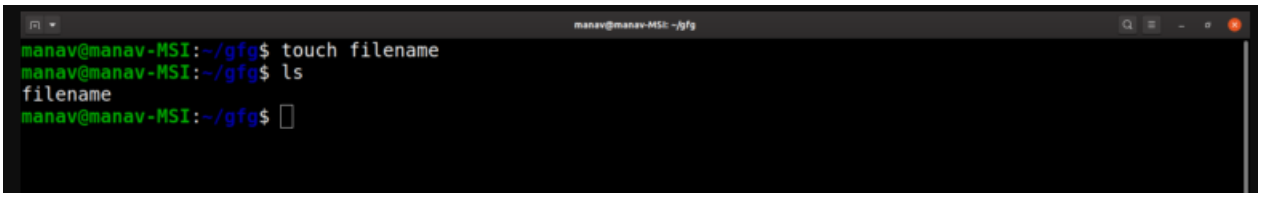
All your files and directories in the current directory would be listed and each type of file would be displayed with a different color. Like in the output directories are displayed with dark blue color.

$ls -l



Creating Files touch command can be used to create a new file. It will create and open a new blank file if the file with a filename does not exist. And in case the file already exists then the file will not be affected.

$touch filename



Displaying File Contents

cat command can be used to display the contents of a file. This command will display the contents of the 'filename' file. And if the output is very large then we could use more or less to fit the output on the terminal screen otherwise the content of the whole file is displayed at once.

$cat filename

Copying a File

cp command could be used to create the copy of a file. It will create the new file in destination with the same name and content as that of the file 'filename'.

```
$cp source/filename destination/
```

Moving a File

mv command could be used to move a file from source to destination. It will remove the file filename from the source folder and would be creating a file with the same name and content in the destination folder.

```
$mv source/filename destination/
```

Renaming a File

mv command could be used to rename a file. It will rename the filename to new_filename or in other words, it will remove the filename file and would be creating a new file with the new_filename with the same content and name as that of the filename file.

```
$mv filename new_filename
```

Deleting a File

rm command could be used to delete a file. It will remove the filename file from the directory.

```
$rm filename
```

Categories of Files in Linux/UNIX :

In Linux/UNIX, Files are mainly categorized into 3 parts:

1. **Regular Files:** Standard files like text, executable, or binary files.
2. **Directory Files:** Files that represent directories containing other files and folders.
3. **Special Files:** This category includes block device files, character device files, symbolic links, pipes, and socket files.

The easiest way to find out file type in any operating system is by looking at its extension such as .txt, .sh, .py, etc. If the file doesn't have an extension then in Linux we can use **file** utility**.**

File Type	Command to create the File	Located in	The file type using "ls -l" is denoted using	FILE command output
Regular FIle	touch	Any directory/Folder	-	PNG Image data, ASCII Text, RAR archive

File Type	Command to create the File	Located in	The file type using "ls -l" is denoted using	FILE command output
				data, etc
Directory File	mkdir	It is a directory	d	Directory
Block Files	fdisk	/dev	b	Block special
Character Files	mknod	/dev	c	Character special
Pipe Files	mkfifo	/dev	p	FIFO
Symbol Link Files	ln	/dev	l	Symbol link to <linkname>
Socket Files	socket() system call	/dev	s	Socket

Types of File and Explanation

1. Regular Files

Regular files are ordinary files on a system that contains programs, texts, or data. It is used to store information such as text, or images. These files are located in a directory/folder. Regular files contain all readable files such as text files, Docx files, programming files, etc, Binary files, image files such as JPG, PNG, SVG, etc, compressed files such as ZIP, RAR, etc.

Example:

Or we can use the "**file** *" command to find out the file type

2. Directory Files

The sole job of directory files is to store the other regular files, directory files, and special files and their related information. This type of file will be denoted in blue color with links greater than or equal to 2. A directory file contains an entry for every file and sub-directory that it houses. If we have 10 files in a directory, we will have 10 entries in the directory file. We can navigate between directories using the **cd** command

We can find out directory file by using the following command:

```
ls -l | grep ^d
```

We can also use the **file *** command

Special Files

1. Block Files:

Block files act as a direct interface to block devices hence they are also called block devices. A block device is any device that performs data Input and Output operations in units of blocks. These files are hardware files and most of them are present in '**/dev'.**

We can find out block file by using the following command:

```
ls -l | grep ^b
```

We can use the **file** command also:

2. Character device files:

A character file is a hardware file that reads/writes data in character by character in a file. These files provide a serial stream of input or output and provide direct access to hardware devices. The terminal, serial ports, etc are examples of this type of file.

We can find out character device files by:

```
ls -l | grep ^c
```

We can use the **file** command to find out the type of file:

3. Pipe Files:

The other name of pipe is a "named" pipe, which is sometimes called a FIFO. FIFO stands for "First In, First Out" and refers to the property that the order of bytes going in is the same coming out. The "name" of a named pipe is actually a file name within the file system. This file sends data from one process to another so that the receiving process reads the data first-in-first-out manner.

We can find out pipe file by using the following command:

```
ls -l | grep ^p
```

We can use the **file** command to find out file type:

4. Symbol link files:

A symbol link file is a type of file in Linux which points to another file or a folder on your device. Symbol link files are also called Symlink and are similar to shortcuts in Windows.

We can find out Symbol link file by using the following command:

```
ls -l | grep ^l
```

We can use the **file** command to find out file type:

5. Socket Files:

A socket is a special file that is used to pass information between applications and enables the communication between two processes. We can create a socket file using the socket() system call. A socket file is located in /dev of the root folder or you can use the **find / -type s** command to find socket files.

```
find / -type s
```

We can find out Symbol link file by using the following command:

```
ls -l | grep ^s
```

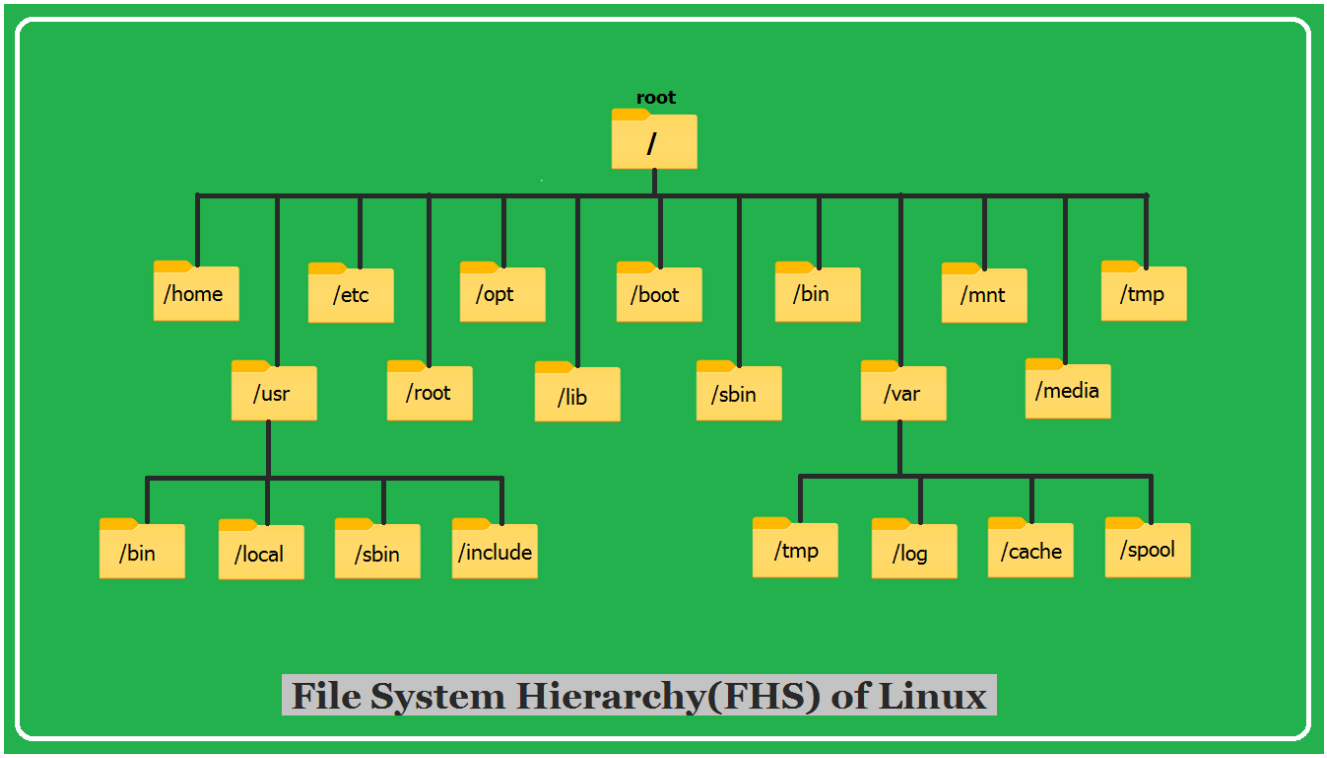
1.5 Linux Directory Structure and File System Hierarchy Standard (FHS)

The Linux Directory Structure, as defined by the File system Hierarchy Standard (FHS), is a hierarchical system where all files and directories branch off from the root directory, denoted by a forward slash "/". The FHS standardizes this structure, ensuring consistency across different Linux distributions and other Unix-like systems.

Here's a breakdown of some key directories:

- / (Root): The top-level directory, the base of the entire file system hierarchy.
- /bin: Contains essential user command binaries (executable files).
- /boot: Contains files needed for booting the system, such as the kernel and boot loader.
- /dev: Contains device files, which provide access to hardware devices.
- /etc: Contains system-wide configuration files.
- /home: Contains user home directories, where users store their personal files and settings.
- /lib: Contains shared library files needed by programs.
- /media: For mounting removable media like USB drives and CDs.
- /mnt: Temporary mount points for file systems.
- /opt: For optional, add-on software packages.
- /proc: A virtual file system that provides information about running processes.
- /root: The home directory for the root user.
- /sbin: Contains essential system binaries, often used by the root user.
- /tmp: A directory for temporary files, which may be deleted on reboot.
- /usr: Contains user-related programs, libraries, documentation, and other read-only data.
- /var: Contains variable data, such as log files, that may change frequently.

The FHS ensures a consistent and organized file system structure, making it easier for users and administrators to navigate, manage, and back up a Linux system



Difference between Windows and Linux File System

Windows and Linux differ significantly in how they organize, access, and manage files within their operating systems.

Feature	Windows	Linux
Structure	Drives (C:, D:, etc.) and folders	Single, unified tree structure starting from root (/)
Case Sensitivity	Not case-sensitive	Case-sensitive
File Permissions	Simpler (user accounts)	More granular control (user, group, others)
File System	Primarily NTFS	Ext4 (most common) FAT32, NTFS (sometimes)
Overall Remarks	User-friendly, familiar interface	Flexible, powerful for advanced users

Unit 2 : Basic Linux Commands

2.1 Directory Navigation Commands (pwd, cd, mkdir, rmdir, ls, tree)

These commands help you navigate, organize, and manage files and directories within the Linux file system.

1. pwd : print/present Working directory

Syntax:

pwd

Example:

$ pwd
/c/Users/Admin/Desktop

2. cd: change directory

Syntax:

cd [path-name/directory-name]

Example:

(a) cd : It change home directory
$ cd "C:\Users\Admin\Desktop\BCA\Fybca"

(b) cd .. : It goes up to one directory level
$ cd ..

(c) cd ../.. : It goes up to two directory level
$ cd ../..

(d) cd / : It change directory to the system's root
$ cd /

(e) cd ~ : It switches to home directory of user which is similar to cd without any argument.
$ cd ~

3. mkdir: **make a directory**. It is used to create one or more new directories.

Syntax:
mkdir [option] path-name/directory-name[path-name/directory-name]….

Example:
$ mkdir sem1

It has following option are use

(a)–p :This option create subdirectory tree under current directory.
$ mkdir -p sem1/dbms

(b)-v :If a directory created successfully then this option shows the name of the directory.
$ mkdir -v sem1/math
mkdir: created directory 'sem1/math'

(c)-m mode: This option sets the permission mode of new directory.
$ mkdir -m444 myprg
mkdir: cannot change permissions of ‘myprg’: Permission denied

4. rmdir: **Remove a directory**. It removes one or more empty directories.

Syntax:
rmdir [option] path-name/directory-name[path-name/directory-name]….

Example:
$ rmdir myprg

It has following option are use

(a)-p: It is useful for removing subdirectory trees.

```
$ rmdir -p sem1/math
```

(b)-v:

```
$ rmdir -v sem1/dbms
    rmdir: removing directory, 'sem1/dbms'
```

5. ls: It stand for **list**. It display list of file and directories in current working directory.

Syntax:

```
ls [option] [argument-list]
```

Example:

```
$ ls
```

It has following option are use

(a)-x: It displays files listing in multi-column on line by line.
(b)-C: It displays files listing in multi-column on column by
(c)-a: It display all file present in the current directory with
(d)-F: It is useful to identifying directories which is executable
(e)-l: It displays long listing or detailed information about file or
(g)-n: It list numeric user-id and group id instead of name.
(h)-R: It lists subdirectories recursively.
(i)-L: It list all symbolic files pointed by symbolic links.
(j)-d: It is used to verify the directory name exists or not.
(k)-t: It sort by last modification time .Latest modified file should be display first.
(l)-u: It sorts by last access time, latest access file display first.
(m)-i: It shows i-node number of specific file.
(n)-S: It sort by file size, the largest file display first.
(o)-r: It sort files list in reverse order.
(p)-U: It does not sort. It display files in the order in which they
(q)-1: It lists one file per line.

5. Tree: The tree command displays the contents of a directory in a hierarchical (tree-like) format, showing the structure of files and subdirectories.

Syntax:

tree [options] [directory]

Example:

1. Basic usage

tree

→ Displays the tree structure of the **current directory**.

2. Display tree of a specific directory

tree /home/user/Documents

→ Shows tree structure starting from /home/user/Documents.

3. Limit the depth of the tree

tree -L 2

→ Shows the directory structure up to **2 levels deep**.

4. Display only directories

tree -d

→ Shows **only directories**, not files.

5. Print file sizes

tree -s

→ Shows file sizes in **bytes** next to each file.

6. Include hidden files

tree -a

→ Includes files/directories starting with . (hidden files).

7. Save the output to a file

tree > tree.txt

→ Saves the directory structure to tree.txt.

2.2 File Management Commands (cat, rm, cp, mv, touch)

1. cat: cat stands for **concatenate**. It is display the contents of one or more files.

Syntax:

cat [option][file1][file2]…..

Example 1: see output of file

$ cat f1.txt

Example 2: See output of subdirectory

$ cat sem5/unix/unit1/unit1.txt

Example 3: It will take input from standards input devices and display them on standard output device.

$ cat

Example 4: User can create file using cat command.

$ cat > file1.txt

Example 5: It can display contents of one or more files.

$ cat file1.txt f1.txt

Example 6: This command appends more lines into exists file.

$ cat >> file1.txt

Example 7: This command read the standard input and a file input during the execution of command..

$ cat - file1.txt

It has following option are use

(a)-v: It display control characters and other non-printing characters.

$ cat -v f1.txt

(b)-e or -E: It print a $ to mark the end of file.

$ cat -e file1.txt

(c)-t: It prints tab character as ^I and form feed character as ^L.

$ cat -t f1.txt

(d)-n: It displays the number of line.

```
$ cat -n f1.txt
```

2. rm: rm stands for **remove**. It deletes one or more files/directories.

Syntax: rm [option]file(s)/directory-name(s)

Example 1: To remove a single file from current directory

```
$ rm file1.txt
```

Example 2: To remove more than one file from current directory

```
$ rm file1.txt file2.txt
```

It has following option are use

(a) - i (interactive): This option remove files interactively and display prompt for remove file?

```
$ rm -i f1.txt
rm: remove regular file 'f1.txt'? y
```

(b) - r (recursive) : It is used to remove non-empty directory, together with all the files and subdirectories.

```
$ rm -r sem5
$ rm -r sem5 sem6
```

(c) – f (force) : It is used to remove the file forcefully which have set permission.

```
$ rm -f myprg
```

3. cp: cp stands for copy. It create duplicate files having access and modification date-time similar to current system date-time.

Syntax:

```
cp [option] filename/directory-name filename/directory-name
```

Example 1: It create duplicate file of file1 with different name at current directory.

```
$ cp f1 f2
```

Example 2: File can be copied to and from another sub-directory with different name.

```
$ cp "sem1\math\f1" "sem2\crdbms\f2"
```

Example 3: File can be copied to and from another sub-directory with same name.

```
$ cp "sem1\math\f1" "sem2\crdbms"
```

Example 4: It also copy one or more files with the same name to another directory.

```
$ cp f1 f2 f3 myprg
```

It has following option are use

(a) - i (interactive): This option copy files interactively and display prompt for copy file?

```
$ cp -i f1 f3
cp: overwrite 'f3'? y
```

(b) - r (recursive) : It is used copy an entire directory structure to onther directory.

```
$ cp -r sem1 sem5
```

(c) – p (preserve) : It is used copy file with access and modification date-time.

```
$ ls -l f1
```

(d) – l(link) : This option create a link instead of copying a file.

```
$ cp -l f1 f1.ln
```

4. mv: mv stand for move. It is used to move an individual file, a list of file or a directory from one directory to another .It is also used to rename a file/directory.

Syntax:

```
mv [option] filename(s)/directory-name filename/directory-name
```

Example 1: File can be moved from one directory to another directory.

```
$ mv "sem2\crdbms\f1" "sem3\f2"
```

Example 2: Move more than one file in directory at the same destination directory.

```
$ mv f1 f2 sem5
```

Example 3: To rename the file in the current directory.

```
$ mv f1 f2
```

Example 4: It move one directory to another location. If new directory on same destination then it rename old directory with new name

```
$ mv myprg prog1
```

It has following option are use

(a) - i (interactive): It warm you before overwrite an existing file.

```
$ mv -i f1 f3
mv: overwrite 'f3'? y
```

5. Touch: The touch command in Linux is used to **create empty files** or **update the timestamp** (access and modification time) of existing files.

Syntax:

```
touch [options] filename
```

1. **Create a New Empty File**

```
touch file1.txt
```

This creates a new file named file1.txt in the current directory if it doesn't already exist.

2. **Create Multiple Files at Once**

```
touch file1.txt file2.txt file3.txt
```

Creates three files at once.

3. **Update the Timestamp of an Existing File**

```
touch existingfile.txt
```

If existingfile.txt already exists, touch updates its modification and access time to the current time.

4. **Create a File in Another Directory**

touch /home/user/newfile.txt

Creates newfile.txt in the /home/user/ directory.

5. **Set a Specific Timestamp**

touch -t 202507140930 file1.txt

Sets the file's time to **14th July 2025, 09:30 AM** (format: [[CC]YY]MMDDhhmm[.ss]).

2.3 File Permissions and Ownership (chmod, chgrp, chown, umask)

Each file or directory has **three types of permission** for **three types of users**:

Type	Description
	permission
	(modify) permission
	ute (run) permission

For:

- **Owner (u)** – user who owns the file
- **Group (g)** – group the file belongs to
- **Others (o)** – everyone else

Example:

-rwxr-xr-- 1 user group file.txt

- Owner: rwx (read/write/execute)

- Group: r-x (read/execute)
- Others: r-- (read only)

1. chmod – Change File Permissions

Syntax:

chmod [options] mode file

Example 1: Symbolic mode

chmod u+x script.sh

→ Adds execute (x) permission for the owner.

Example 2: Remove write permission from group

chmod g-w file.txt

Example 3: Numeric (Octal) mode

chmod 755 script.sh

→ rwxr-xr-x

2. chown – Change File Ownership

Syntax:

chown [owner][:group] file

Example 1: Change owner

sudo chown alice file.txt

Example 2: Change owner and group

sudo chown bob:developers file.txt

4. chgrp – Change Group Ownership

Syntax:

chgrp groupname file

Example:

sudo chgrp staff report.pdf

5. umask – Set Default Permissions for New Files

The umask command sets default permissions **by subtracting from full access**.

- **Default permissions:**
  - Files: 666 (rw-rw-rw-)
  - Directories: 777 (rwxrwxrwx)

Example: View current umask

umask

Example: Set umask

umask 022

This removes write (w) permission for group and others.
New files will have 644 (rw-r--r--) and directories will have 755.

2.4 Common System Commands (who, whoami, man, echo, date, clear)

1. who: Shows who is currently logged into the system.

Syntax: who

Example:

who

Output Example:

user1 tty7 2025-07-16 09:12

2. whoami: Displays the username of the current user.

Syntax:

whoami

Example:

whoami

3. man: Displays the manual (help page) for a command.

Syntax:

man <command>

Example:

man ls

4. echo: Prints text or variables to the terminal.

Syntax:

echo [text or $variable]

Example:

echo "Hello, world!"

5. date: Displays the current system date and time.

Syntax:

Date

Example:

Date

6. clear: Clears the terminal screen.

Syntax:

Clear

2.5 Text Processing Commands (head, tail, cut, sort, cmp, tr, uniq, wc, tee)

1. head: Displays the first N lines of a file.

Syntax:

head [options] filename

Example:

head -n 5 file.txt

→ Displays the first 5 lines of file.txt.

2. tail: Displays the last N lines of a file

Displays the last n lines of file.

Syntax:

tail [options] filename

Example:

tail -n 3 log.txt

→ Displays the last 3 lines of log.txt.

3. cut: Extracts columns or fields from lines.

Syntax:

cut [options] filename

Examples:

cut -c1-5 file.txt

→ Extracts characters 1 to 5 from each line.

cut -d',' -f2 data.csv

→ Extracts the second field using comma , as delimiter.

4. sort: Sorts lines alphabetically or numerically.

Syntax:

sort [options] filename

Examples:

sort names.txt

→ Sorts lines in alphabetical order.

sort -n numbers.txt

→ Sorts numbers in ascending numerical order.

5. cmp: Compares two files byte by byte.

Syntax:

cmp file1 file2

Example:

cmp file1.txt file2.txt

→ Compares files byte by byte. If they differ, it shows the first difference.

6. tr: Translate or delete characters.

Syntax:

tr [options] SET1 [SET2]

Examples:

tr 'a-z' 'A-Z' < input.txt

→ Converts lowercase to uppercase.

tr -d '0-9' < data.txt

→ Deletes all digits from the input.

7. uniq: Removes duplicate lines (requires sorted input).

Syntax:

uniq [options] filename

Examples:

sort data.txt | uniq

→ Removes duplicate lines from a sorted file.

uniq -c sorted.txt

→ Shows duplicate count for each line.

8. wc: Counts lines, words, or characters.

Syntax:

wc [options] filename

Examples:

wc file.txt

→ Displays line, word, and byte count.

wc -l file.txt

→ Displays only line count.

9. tee: Reads input and writes to file and screen.

Syntax:

command | tee filename

Example:

ls -l | tee output.txt

→ Displays the output of ls -l on the screen and writes it to output.txt.

2.6 Introduction to Process

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

$

2.7 Process Control commands: ps, fg, bg, kill, sleep

1. ps – Show Running Processes

Syntax:

ps [options]

Examples:

ps # Show your current shell's processes

ps -e # Show all system processes

ps -ef # Full-format listing of all processes

ps aux # Detailed info including CPU/memory usage

2. fg – Bring Job to Foreground

Syntax:

fg [%job_id]

Examples:

fg # Resume most recent background job

fg %1 # Resume job number 1

Use jobs to list background/suspended jobs with their job IDs.

3. bg – Resume Job in Background

Syntax:

bg [%job_id]

Examples:

bg # Resume most recently stopped job in background

bg %2 # Resume job number 2 in background

4. kill – Terminate Process

Syntax:

kill [signal] PID

Examples:

kill 1234 # Send SIGTERM (graceful termination) to process 1234

kill -9 1234 # Send SIGKILL (force kill) to process 1234

kill -l # List all available signals

5. sleep – Pause Execution

Syntax:

sleep duration

Examples:

sleep 5 # Pause for 5 seconds

sleep 2m # Pause for 2 minutes

sleep 1h # Pause for 1 hour

2.8 Job Scheduling commands : at, batch, crontab

1. at – Schedule One-Time Tasks

The at command runs a job once at a specified time.

Syntax:

at [TIME]

Examples:

at 10:00 # Run job at 10:00 AM today

at now + 1 hour # Run job 1 hour from now

 # After typing the command:

echo "echo 'Hello'" | at now + 1 minute

 # Or interactive:

$ at 17:30

at> echo "Backup started" >> backup.log

at> <Ctrl+D> # Press Ctrl+D to save and exit

2. batch – Schedule Tasks When Load is Low

The batch command schedules jobs to run when system load is low.

Syntax:

batch

Example:

echo "tar -czf backup.tar.gz /home/user" | batch

Like at, use atq to view and atrm to remove batch jobs.

3. crontab – Repeated Scheduled Tasks (Cron Jobs)

The crontab command schedules recurring tasks (daily, weekly, etc.).

Syntax:

crontab -e # Edit user's crontab

crontab -l # List current crontab entries

crontab -r # Remove current crontab

Example:

 # Open the crontab editor

crontab -e

 # Add a job (runs every day at 7 AM):

0 7 * * * /home/user/backup.sh

 # Cron Format:

 # ┌──────────────────── minute (0 - 59)

 # │ ┌──────────────────── hour (0 - 23)

 # │ │ ┌──────────────────── day of the month (1 - 31)

 # │ │ │ ┌──────────────────── month (1 - 12)

 # │ │ │ │ ┌──────────────────── day of the week (0 - 7) (Sunday = 0 or 7)

 # │ │ │ │ │

 # * * * * * command to execute

Process Management In Unix

When you execute a program on your Unix system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system.

Whenever you issue a command in Unix, it creates, or starts, a new process. When you tried out the **ls** command to list the directory contents, you started a process. A process, in simple terms, is an instance of a running program.

The operating system tracks processes through a five-digit ID number known as the **pid** or the **process ID**. Each process in the system has a unique **pid**.

Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over. 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.

Starting a Process

When you start a process (run a command), there are two ways you can run it −

- Foreground Processes
- Background Processes

Foreground Processes

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the **ls** command. If you wish to list all the files in your current directory, you can use the following command −

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

This would display 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
```

The process runs in the foreground, the output is directed to my screen, and if the **ls** command wants any input (which it does not), it waits for it from the keyboard.

While a program is running in the foreground and is time-consuming, no other commands can be run (start any other processes) because the prompt would not be available until the program finishes processing and comes out.

Background Processes

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (**&**) at the end of the command.

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

This displays all those 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
```

Here, if the **ls** command wants any input (which it does not), it goes into a stop state until we move it into the foreground and give it the data from the keyboard.

That first line contains information about the background process - the job number and the process ID. You need to know the job number to manipulate it between the background and the foreground.

Press the Enter key and you will see the following −

```
[1] + Done ls ch*.doc &
$
```

The first line tells you that the **ls** command background process finishes successfully. The second is a prompt for another command.

Listing Running Processes

It is easy to see your own processes by running the **ps** (process status) command as follows −

```
$ps
PID TTY TIME CMD
18358 ttyp3 00:00:00 sh
18361 ttyp3 00:01:31 abiword
18789 ttyp3 00:00:00 ps
```

One of the most commonly used flags for ps is the **-f** (f for full) option, which provides more information as shown in the following example −

```
$ps -f
UID PID PPID C STIME TTY TIME CMD
amrood 6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood 6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood 3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood 6892 3662 4 10:51:50 pts/6 0:00 ps -f
```

Here is the description of all the fields displayed by **ps -f** command –

Sr.No.	Column & Description
1	**UID** User ID that this process belongs to (the person running it)
2	**PID** Process ID
3	**PPID** Parent process ID (the ID of the process that started it)
4	**C** CPU utilization of process
5	**STIME** Process start time
6	**TTY** Terminal type associated with the process
7	**TIME** CPU time taken by the process
8	**CMD** The command that started this process

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

Sr.No.	Option & Description
1	**-a** Shows information about all users
2	**-x** Shows information about processes without terminals

3	**-u** Shows additional information like -f option
4	**-e** Displays extended information

Stopping Processes

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when the process is running in the foreground mode.

If a process is running in the background, you should get its Job ID using the **ps** command. After that, you can use the **kill** command to kill the process as follows –

```
$ps -f
UID PID PPID C STIME TTY TIME CMD
amrood 6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood 6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood 3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood 6892 3662 4 10:51:50 pts/6 0:00 ps -f
$kill 6738
Terminated
```

Here, the **kill** command terminates the **first_one** process. If a process ignores a regular kill command, you can use **kill -9** followed by the process ID as follows –

```
$kill -9 6738
Terminated
```

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**.

o 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.

Job ID Versus Process ID

Background and suspended processes are usually manipulated via **job number (job ID)**. This number is different from the process ID and is used because it is shorter.

In addition, a job can consist of multiple processes running in a series or at the same time, in parallel. Using the job ID is easier than tracking individual processes.

bg command in Linux with Examples

In Linux, the **bg** command is a useful tool that allows you to manage and move processes between the foreground and background. It's especially helpful when you want to multitask in the terminal by placing a process in the background, enabling you to continue using the terminal for other commands while the process runs quietly in the background.

Syntax

bg [job_spec ...]

where,

- **job_spec**: This is used to identify the job you want to move to the background. It can be specified in several formats:
    - **%n:** Refers to job number n.
    - **%str:** Refers to a job that was started by a command beginning with str.

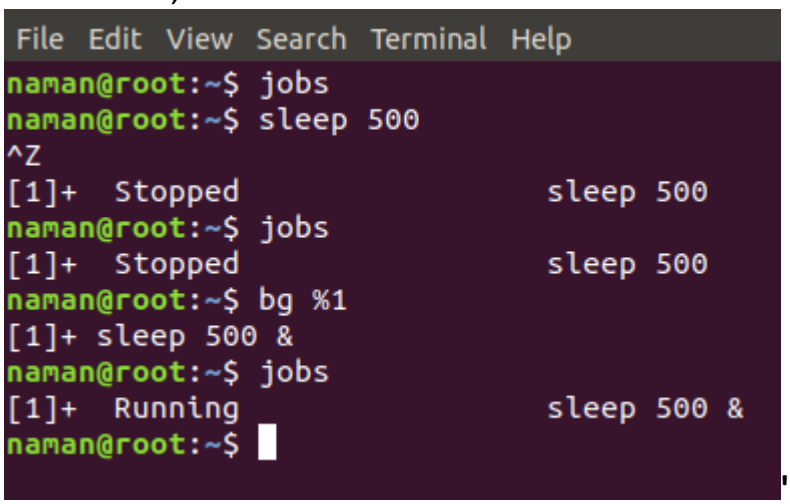
- **%?str:** Refers to a job that was started by a command containing str.
- **%% or %+:** Refers to the current job. Both fg and bg commands will operate on this job if no job_spec is provided.
- **%-:** Refers to the previous job.

If no **job_spec** is provided, the most recent job is resumed in the background.

Useful Options for bg command

1. bg [JOB_SPEC]:

This command is used to put the mentioned job in background. In the below screenshot, we do following :

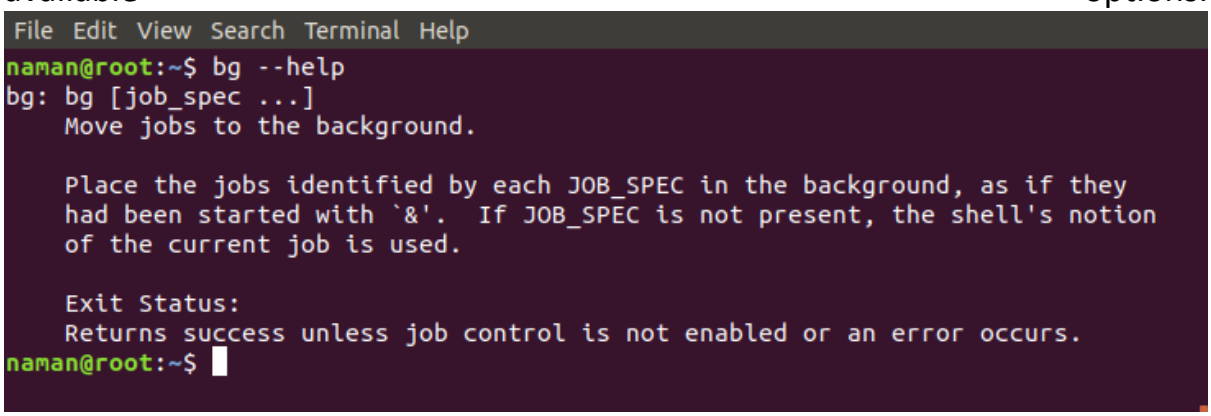


'sleep 500' is used to create dummy foreground job.

- We use jobs command to list all jobs
- We create a process using sleep command, we get its ID as 1.
- We put it in background by providing its ID to bg.

2. bg --help:

Displays the help information for the bg command. This is useful if you need more information on how to use the command or if you're unsure of the available options.



fg command in Linux with examples

The **fg command** in Linux is used to bring a background job into the foreground. It allows you to resume a suspended job or a background process directly in the terminal window, so you can interact with it.

Syntax

fg [job_spec]

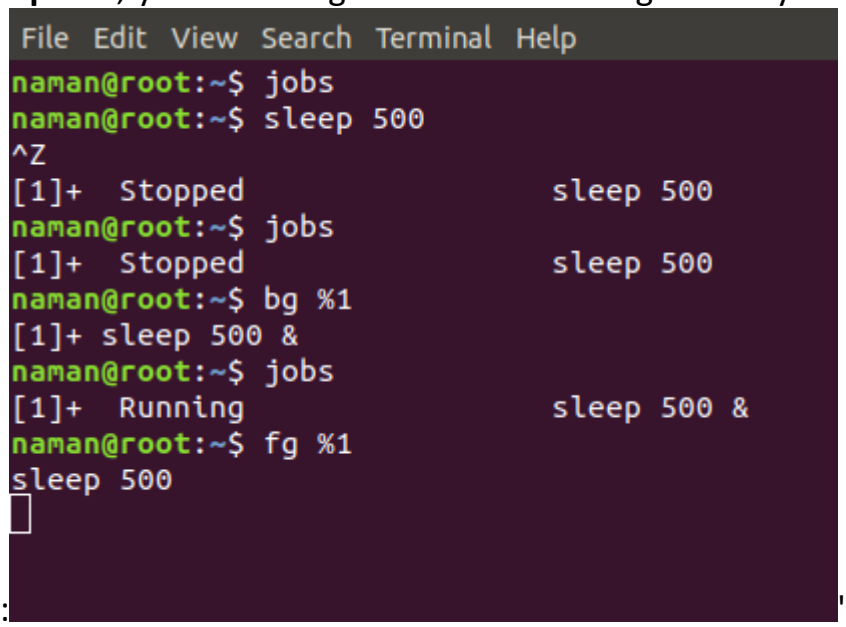
The **job_spec** is a way to refer to the background jobs that are currently running or suspended. Here are some common ways to specify a job:

- **%n:** Refers to job number n.
- **%str:** Refers to a job that was started by a command beginning with str.
- **%?str:** Refers to a job that was started by a command containing str.
- **%% or %+:** Refers to the current job (this is the default job operated on by fg if no job_spec is provided).
- **%-:** Refers to the previous job.

Key Options for the fg command

1. fg [JOB_SPEC]:

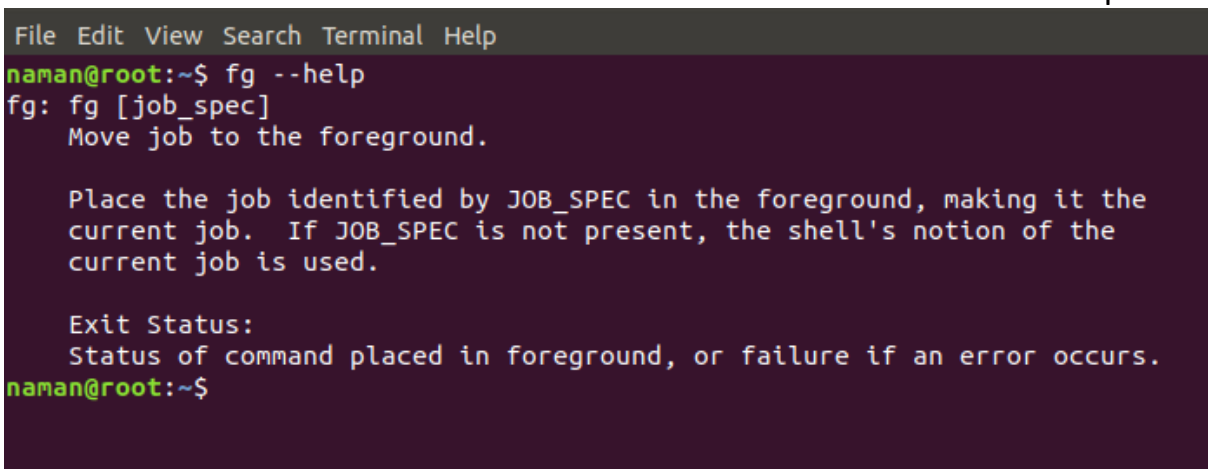
This is the primary use of the **fg** command, bringing a specified job running in the background back to the foreground. For example, if you create a dummy job using **sleep 500**, you can bring it back to the foreground by referencing its job number:



"sleep 500" is a command which is used to create a dummy job which runs for 500 seconds.

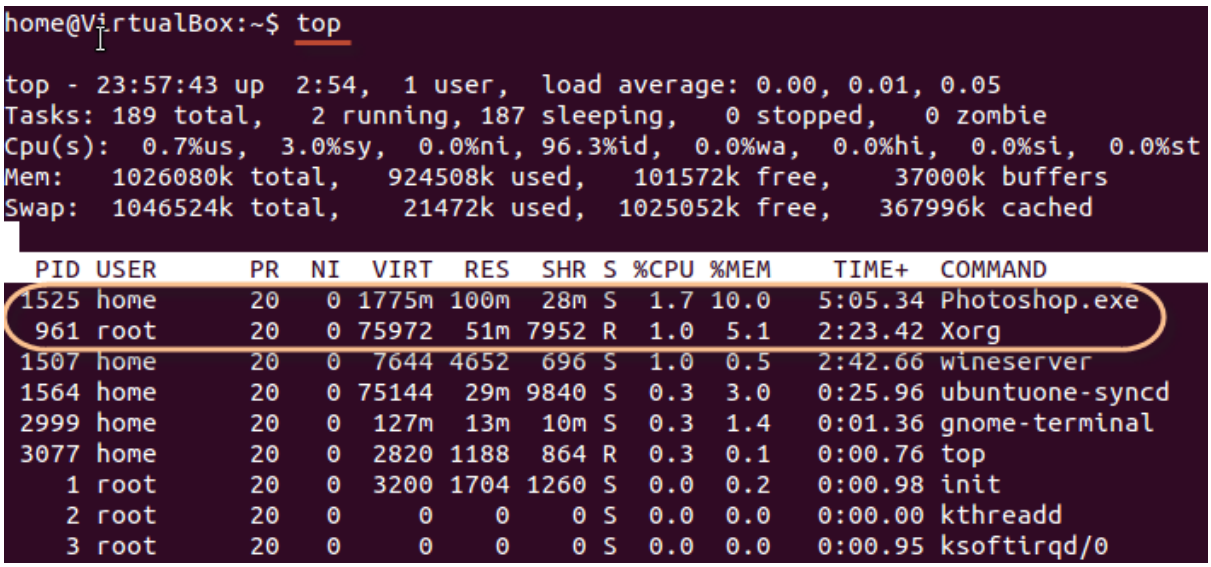
2. fg --help:

This option displays help information for the **fg** command, explaining usage and available options.



Top

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



Press ‘q’ on the keyboard to move out of the process display.

The terminology follows:

Field	Description	Example 1
PID	The process ID of each task	1525
User	The username of task owner	Home
PR	Priority Can be 20(highest) or -20(lowest)	20
NI	The nice value of a task	0
VIRT	Virtual memory used (kb)	1775
RES	Physical memory used (kb)	100
SHR	Shared memory used (kb)	28
S	Status There are five types:	S

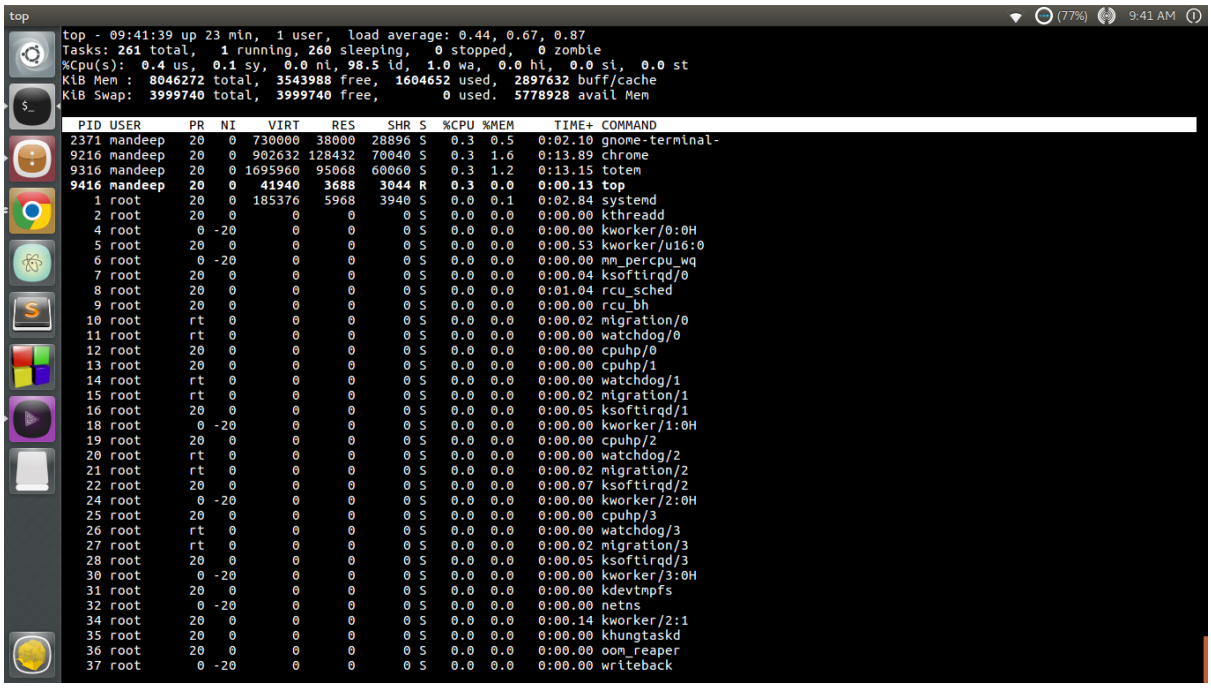
Field	Description	Example 1
	‘D’ = uninterruptible sleep ‘R’ = running ‘S’ = sleeping ‘T’ = traced or stopped ‘Z’ = zombie	
%CPU	% of CPU time	1.7
%MEM	Physical memory used	10
TIME+	Total CPU time	5:05.34
Command	Command name	Photoshop.exe

Priority of process in Linux | nice value

The running instance of program is process, and each process needs space in RAM and CPU time to be executed, each process has its priority in which it is executed.

Now observe the below image and see column **NI**
top

Output:



top command output

The column NI represents nice value of a process. It's value ranges from -20 to 20(on most unix like operating systems).

-20 20

most priority least priority

process process

One important thing to note is nice value only controls CPU time assigned to process and not utilisation of memory and I/O devices.

nice and renice command

nice command is used to start a process with specified nice value, which renice command is used to alter priority of running process.

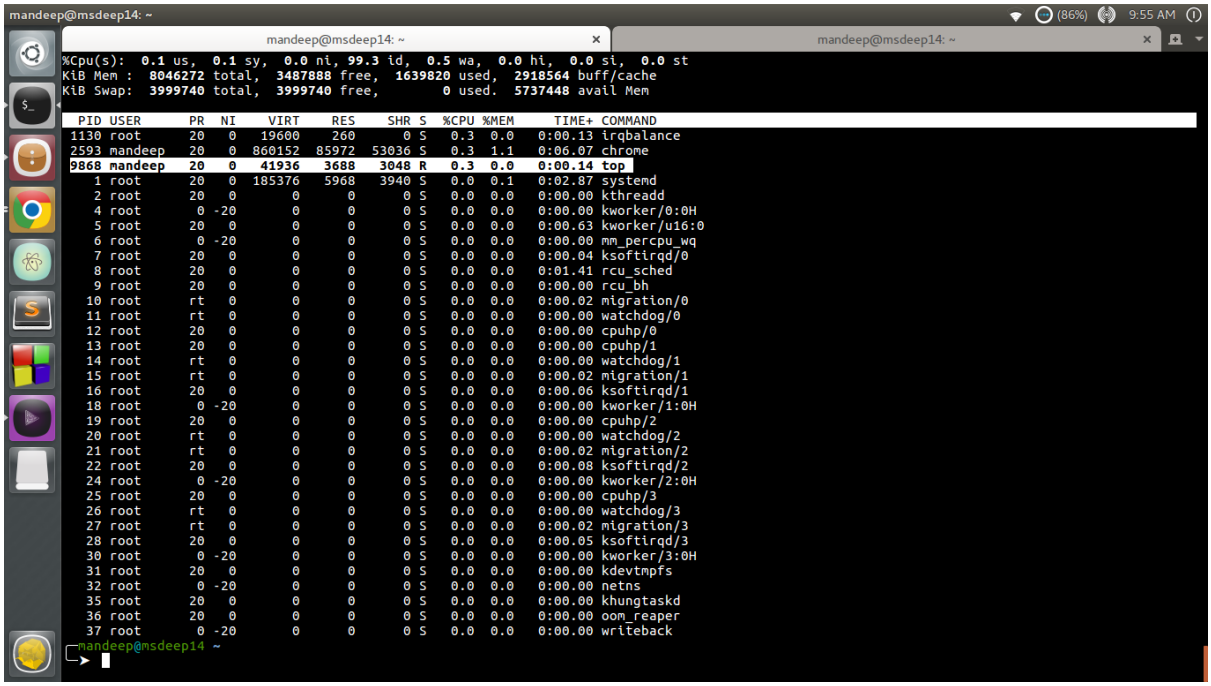
Usage of nice command :

Now let's assume the case that system has only 1GB of RAM and it's working really slow, i.e. programs running on it(processes) are not responding quickly, in that case if you want to kill some of the processes, you need to start a terminal, if you start your bash shell normally, it will also produce lag but you can avoid this by starting the bash shell with high priority.

For example:

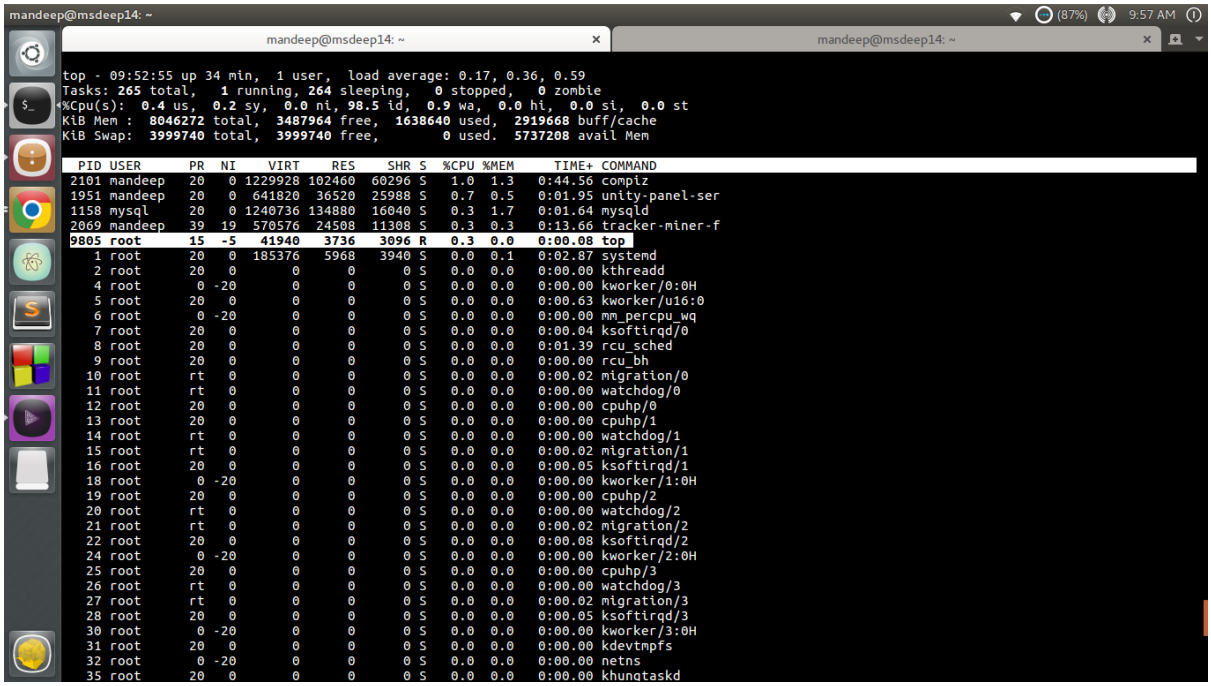
nice -n -5 bash

First observe output of top without setting nice value of any process in below image



nice value of top is 0

Now start a bash shell with nice value -5, if you see the highlighted line, the top command which is running on bash shell has nice value set to -5



nice value of bash shell is -5

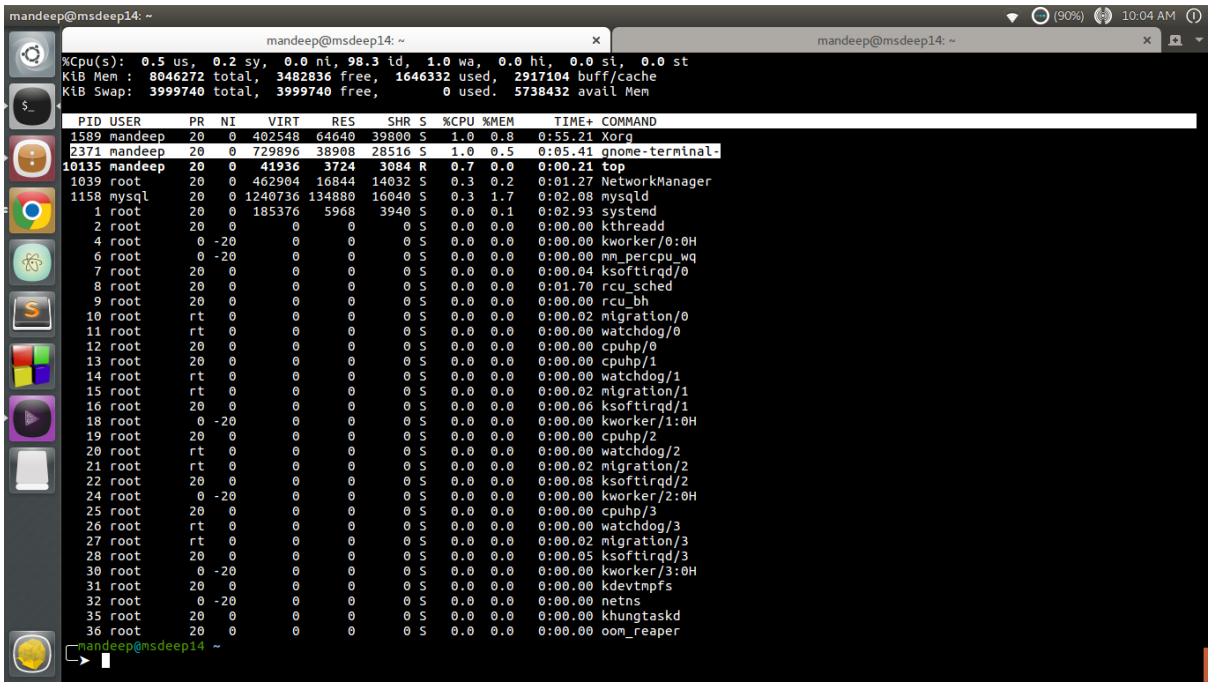
Usage of renice command :

To alter priority of running process, we use renice command.
renice value PID

value is new priority to be assigned
PID is PID of process whose priority is to be changed

One thing to note is you can't set high priority to any process without having root permissions though any normal user can set high priority to low priority of a process.

We will see one example of how you alter priority of process.



nice value of gnome terminal is 0

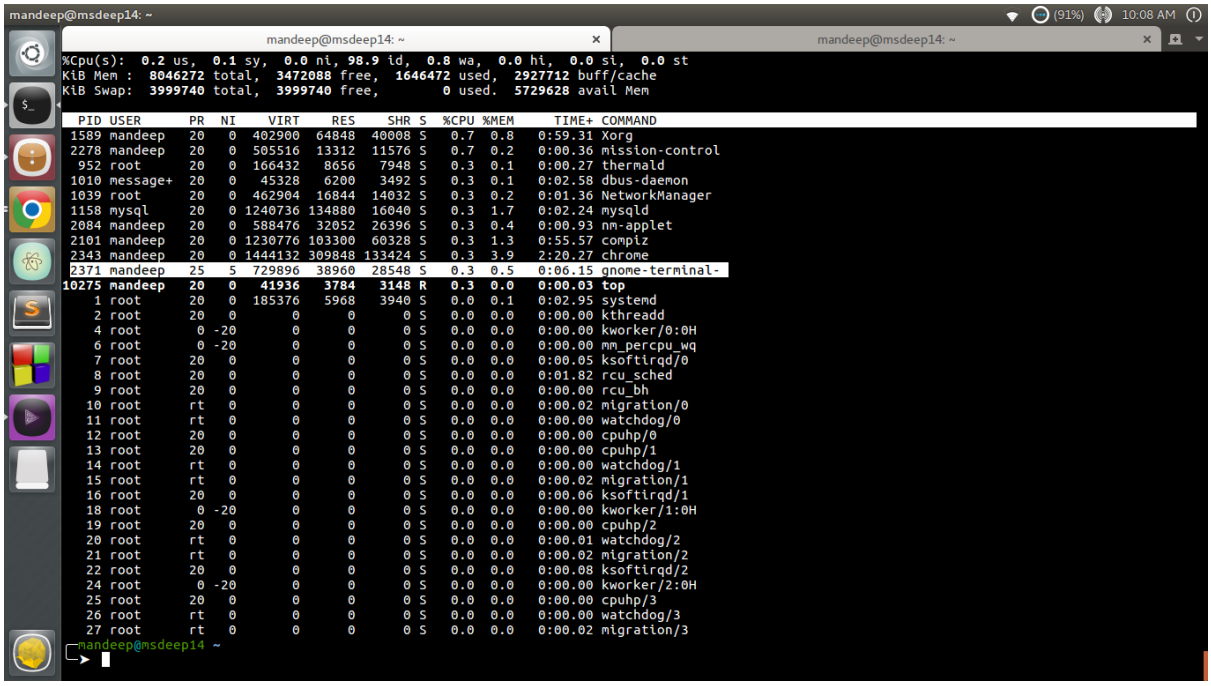
You can observe that nice value of process(PID = 2371) is 0, now let's try to set the new priority of 5 to this process.

renice 5 2371

Output:

2371 (process ID) old priority 0, new priority 5

You can also see this priority using top command(see highlighted line in image).



process 2371 has nice value 5

Command	Description
bg	To send a process to the background
fg	To run a stopped process in the foreground
top	Details on all Active Processes
ps	Give the status of processes running for a user
ps PID	Gives the status of a particular process
pidof	Gives the Process ID (PID) of a process
kill PID	Kills a process
nice	Starts a process with a given priority
renice	Changes priority of an already running process
df	Gives free hard disk space on your system
free	Gives free RAM on your system

nohup Command in Linux with Examples

Every command in Linux starts a process at the time of its execution, which automatically gets terminated upon exiting the terminal. Suppose, you are executing programs over SSH and if the connection drops, the session will be terminated, all the executed processes will stop, and you may face a huge accidental crisis. In such cases, running commands in the background can be very helpful to the user and this is where **nohup command** comes into the picture. **nohup (No Hang Up)** is a command in Linux systems that runs the process even after logging out from the shell/terminal.

Nohup Command

Usually, every process in Linux systems is sent a **SIGHUP (Signal Hang UP)** which is responsible for terminating the process after closing/exiting the terminal. Nohup command prevents the process from receiving this signal upon closing or exiting the terminal/shell. Once a job is started or executed using the nohup command, **stdin** will not be available to the user and **nohup.out** file is used as the default file for **stdout** and **stderr**. If the output of the nohup command is redirected to some other file, **nohup.out** file is not generated.

Nohup Command Syntax

The syntax for using the Nohup command is straightforward:

nohup command [options] &

- **`command`**: Specifies the command or script that you want to execute.
- **`[options]`**: Optional arguments or flags that modify the behavior of the command.
- **`&`**: Placing an ampersand (&) at the end of the command instructs the shell to run the command in the background.

Vi Editor

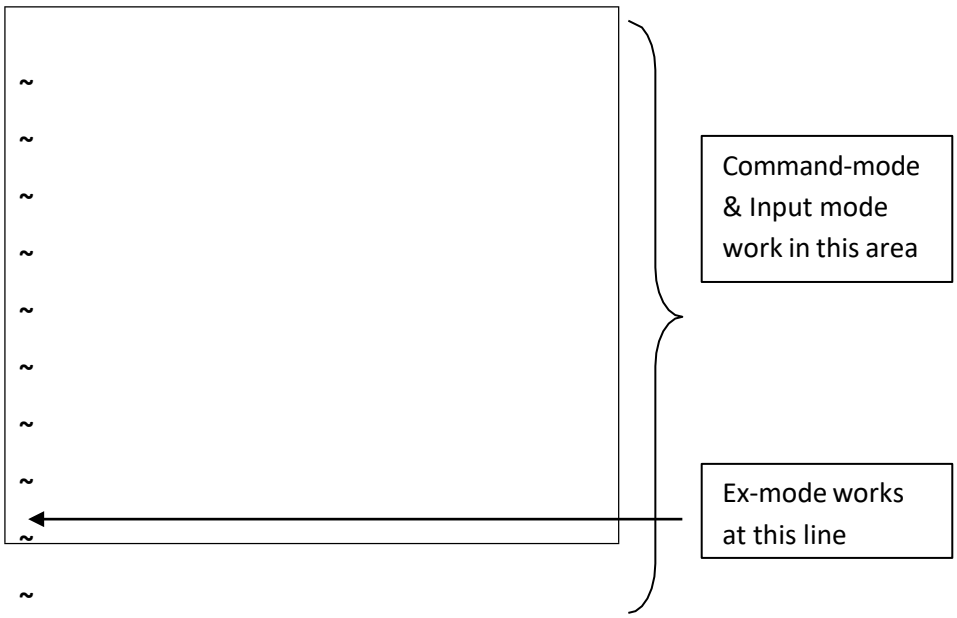
The vi Editor is a text based editor used in Linux and Unix for editing configuration files and creating text documents.

Vi- editor is one of the most versatile editors of linux. The vi-editor was created by Bill Joy for BSD versions for unix.Bram Moolenaar improved the editor and called it "vim" (vi-improved) editor. Vi uses number of internal commands to navigate to any point in a text file and edit the text there. It allows to copy and move text within a file and also from one file to another. Vi offers cryptic and sometimes mnemonic, internal commands for editing work.

Syntax (for invoking vi editor)

$ vi (enter) [will open vi editor with a temporary file name]

$ vi filename (enter) [will open vi editor with the given file name in the current directory]



e.g. vi test (enter)

After applying above command the vi editor gets open as shown in the figure above.

Vi editor operates in two mode.

1. **Command mode:** This mode enables you to perform administrative tasks such as saving files, executing commands, moving the cursor, cutting (yanking) and pasting lines or words, and finding and replacing. In this mode, whatever you type is interpreted as a command.
2. **Insert mode:** This mode enables you to insert text into the file. Everything that's typed in this mode is interpreted as input and finally it is put in the file .

Hint: If you are not sure which mode you are in, press the Esc key twice, and then you'll be in command mode.

Creating a script using the vi editor involves the following steps:

Open or Create the Script File.

Step-1:

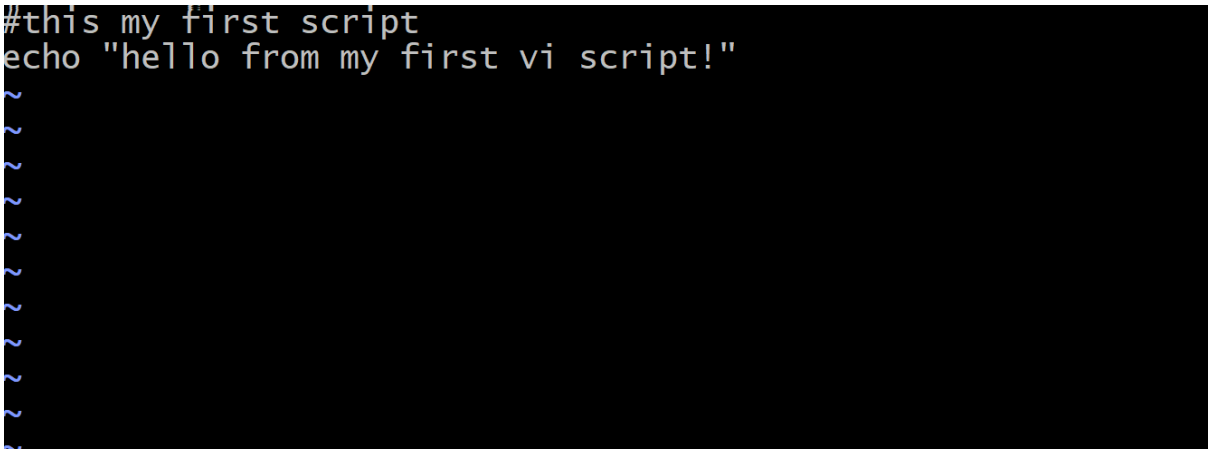
Open your terminal and type vi script_name.sh, replacing script_name.sh with your desired script filename. The .sh extension is conventional for shell scripts. If the file doesn't exist, vi will create a new one; otherwise, it will open the existing file.

Step-2: Enter insert mode.

Upon opening vi, you are in command mode. To begin typing your script, you must switch to insert mode. Press the i key to enter insert mode. Write Your Script.

In insert mode, you can type the commands and logic for your script.

Code



Step-2: Exit Insert Mode.

Once you have finished writing your script, press the Esc key to return to command mode. Save and Exit.

In command mode, type :wq and press Enter. : initiates a command line within vi, w saves the changes to the file, and q quits the vi editor.

If you want to exit without saving changes, use :q!.

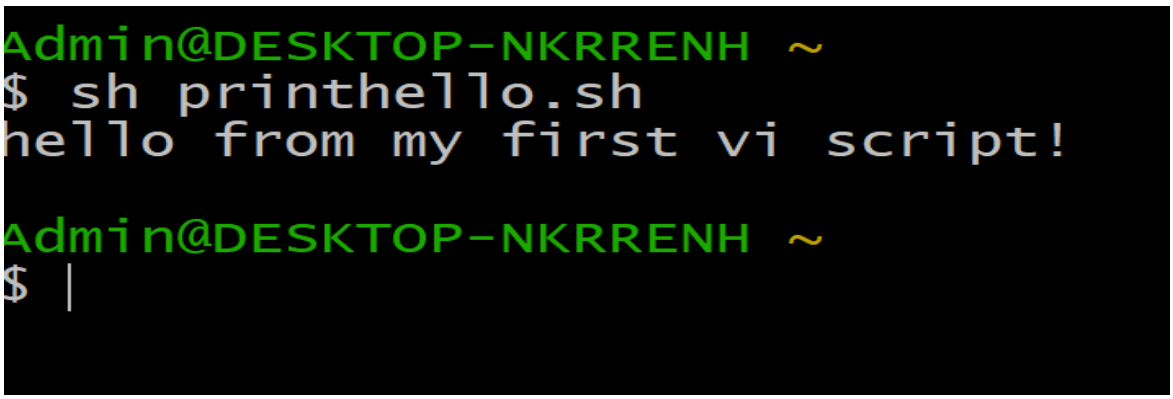
Step-3: Execute script from the terminal:

You can now execute your script from the terminal:

Code

```
sh script_name.sh
```

Ex: sh printhello.sh



Insert Command:

you can enter Insert mode using various keys, each with a slightly different effect:

i: Inserts text before the current cursor position.
I: Inserts text at the beginning of the current line.
a: Appends text after the current cursor position.
A: Appends text at the end of the current line.
o: Insert a new line below the current line.
O: Insert a new line above the current line.

Navigation Commands:

h, j, k, l: Move cursor left, down, up, right respectively.
w, b, e: Move to beginning of next word, beginning of previous word, end of current word.
0, $: Move to beginning of line, end of line.
gg, G: Move to first line, last line of file.
nG: Move to line number 'n'.

Editing Commands:

x: Delete character under cursor.
dd: Delete the current line.
ndd: Delete 'n' lines.
yy: Yank (copy) the current line.
nyy: Yank (copy) 'n' lines.
p: Paste yanked or deleted text below the current line.
P: Paste yanked or deleted text above the current line.
u: Undo the last change.

Saving and Exiting:

:w: Save the file.
:wq or ZZ: Save and quit.
:q!: Quit without saving changes.
:x: Save and quit (similar to :wq).

Shell Metacharacters and operator.

Metacharacters in Linux are special symbols that have specific meanings to the shell, allowing for pattern matching, redirection, and other powerful command-line operations. They are used to manipulate files, directories, and command output.

Filename Expansion(wildcard:*,?,[])

There are mainly three metacharacters for filename substitution:

1. **Asterisk (*)**
2. **Question mark (?)**
3. **Character class [].**

They are used for matching filenames in a directory.

1. asterisk (*)

It matches zero or one more occurrences of any characters in a filename. When the **Asterisk(*)** is appended to the sring file, the pattern file* matches all filenames in the directory with the string file and also include the file fileitself. You can also use * as an argument to ls as follow:

```
$ls  *     #display all files of current directory
```

When shell encounters this command line, it immediately identifies the * as a metacharacter. It then generates a list of files from the current directory that match this pattern and passes it on to the kernel for execution. So, this command display name of all files in the directory.

2. Question-mark (?)

It is another metacharacter used for filename substitution. It **matches any single character**. If we use it with file string then it matches filename of current directory that start

with string file followed by any single character. For example, you can write a command as follow:

$ls file? #display filename begins with string file followed by any character

Then it display all files begins with 'file' and the last character is any.

Character class ([])

The character class uses two metacharacter represented by a pair of square brackets i.e. [].You can write as many characters inside the square brackets, but matching takes place for **only a single character** in the class i.e. it matches any single character from character set. For example, a single character expression taking the value either 1 or 2 or 3 or 4 can be represented by character class as [1234]. This can be combined with any string or another wild-card expression.

Character class uses two another metacharacter : **!(bang or exclamation) and (hyphen).** Hyphen (i.e. -) is used to specify range inside the class eg. [1-4]. A valid range specification requires that the character on the left have a lower ASCII value than the one on the right.

Exclamation (i.e. !) reverses the matching criteria i.e., it matches all other characters except the ones in the class. It is placed at the beginning of character or character set within the class. For example, consider command as follow:

$ls [!0-9]*

It displays all the files whose filename begins with other than digit.

Eliminate the meaning of wild-card character:

The metacharacters asterisk (i.e. *) and question mark (i.e. ?) lose their meaning when used inside the character class, and are matched literally. Similarly, '-' and '!' also lose their significance when placed outside the class. Additionally '!' loses its meaning when placed anywhere but at the beginning within the class. The '-' loses its meaning if it is not bounded properly on either side by a single character. e.g. [a-] or [-a].

Matching a Dot

The * does not match all files beginning with a . (dot) or the /of a pathname. If you want to list all the hidden files in your directory, then the dot must be matched explicitly as follow:

$ls .*
It displays all files begins with **.** and also display**.** and **..** contents.

$ls .[!.]*
It displays only those files in your directory begins with. (dot).

Linux - Shell Input/Output Redirections

In linux, commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from the standard input, which happens to be your terminal by default. Similarly, a command normally writes its output to standard output, which is again your terminal by default.

Redirection allows users to change the default behaviour of standared streams. Redirection is the process through which is a user can use file instead of the standard I/O devices. There are three types of redirection:

1. **Output Redirection**
2. **Input Redirection**
3. **Error Redirection**

1. Output Redirection

In output redirection, the output of command sends to file instead of standard output devices (i.e Monitor) this capability is known as output redirection. When shell encountered > in the command line then it understand that standard output is to be sent to the file instead of monitor. The shell first opens the file, writes output of command into it and then close the file.

Check the following who command which redirects the complete output of the command in the users file.

$ who > users

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. You can check the users file for the complete content –

$ cat users

```
User1        tty01   Sep 12 07:30
User2        tty15   Sep 12 13:32
User3       tty21   Sep 12 10:10
User4        tty24   Sep 12 13:07
User5       tty25   Sep 12 13:03
$
```

The above command is also written using file discripter as follow:

$ who 1 > users

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider the following example –

$ echo line 1 > users
$ cat users
line 1
$

You can use >> operator to append the output in an existing file as follows –

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

2. Input Redirection

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command. The commands that normally take their input from the standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file users generated above, you can execute the command as follows –

```
$ wc -l users
2 users
$
```

Upon execution, you will receive the following output. You can count the number of lines in the file by redirecting the standard input of the wc command from the file users −

```
$ wc -l < users
2
$
```

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not. In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

The above command is also written using file discripter as follow:

```
$ wc -l  0< users
```

3. Error Redirection

In Linux, it is possible to redirect the error messages of an invalid command to a file oth

er than the standard error file so that error messages do not appear on the terminal. This is done using a file descriptor for the standard error file. To redirect the error message of a command to a file other than the terminal, we use error redirection as 2>

Assume that the file file1 does not exist in the current working directory and you apply a command like this:

$cat file1 2 > err_msg <enter>

So, linux will generate an error message because the execution is unsuccessful. This error message will be written into the file err_msg.

Piping mechanism

A user can combine more than commands using a metacharacter known as pipe. It is denoted by vertical bar (i.e. |)) symbol. This allows for sequential processing of data, where the output of the first command becomes the input for the second, and so on. The general form of piping mechanism is as follow:

command1\command2\....\commandN

There is no restriction on number of commands used in pipeline. Here, output of commandl send as standard input to command2, the output of command2 sends as standard input to command3 and so forth. In this way, the standard output of one filter command can be sent as standard input to another filter command.

Let us consider a command as follow:

```
$ cat fl | wc-c
123
$
```

Here, the output of cat command can be use as standard input for wc command. So, the result shows number of characters in file fl.

Consider the command sequences as follow:

```
$sort fl > fl.sort
Suniq –u fl.sort >fl.uniq
$wc -1 <fl.uniq
5
$
```

The purpose of these command sequences is that to count number of unique lines in file f1. Through the pipe feature, these three steps can be done in one command without creating a temporary files as shown below:

```
$sort fl | uniq-u | wc-l
5
$
```

The vertical bar (i.e.|) indicates to the Linux to send the output of the command before | (i.e. sort fl) as input to the command after | (i.e. uniq-u).Again sends the output of the command before | (i.e. uniq-u) as input to the command after | (i.e.wc –l).Thus, the commands sort, uniq and wc create a pipeline through which data to flow without creating temporary files.

Therefore, the advantages of the pipe feature is that

- There is no need to create intermediate temporary files to perform complex tasks.
- As compare to redirection, it is faster.

Command Substitution

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

Command substitution in Linux allows the output of a command to be used as an argument or value within another command or statement. This mechanism enables dynamic scripting and the creation of more complex and flexible shell commands.

Syntax

The command substitution is performed when a command is given as −
`command`

When performing the command substitution make sure that you use the back quote, not the single quote character.

Example

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrates the command substitution −

```
DATE=`date`
echo "Date is $DATE"

USERS=`who | wc -l`
echo "Logged in user are $USERS"
```

Upon execution, you will receive the following result −

```
Date is Thu Jul  2 03:59:57 MST 2009
Logged in user are 1
```

Unit 3 : Shell Scripting in Linux

3.3 Control Flow Structures (if-else, case, for, while, until)
3.4 Logical Operators (&&, ||, !)
3.5 test and [ ] command for Condition Testing (file, numeric, string)
3.6 Arithmetic Operations (expr, $(( )))

Read statement

- The read statement is used to make the shell interactive.
- It is the input taking tool of the shell script.
- Syntax: read var1 var2 var3 ….
- We can read values for multiple variables using a single read statement.
- If the number of input values given is more than the number of variables then the last values are assigned to the last variable.
- If the number of input values given is less than the number of variables then the last variable is left unassigned.

Example: read n1 n2 n3

i/p 1 2 3 (n1=1,n2=2,n3=3)

1 2 (n1=1,n2=2 & n3 is left unassigned)

1 2 3 4 5 (n1=1,n2=2,n3=3 4 5)

3.3 Control Flow Structures (if-else, case, for, while, until)

Control flow structures allow you to control the execution sequence of commands based on conditions or repetitive patterns. Here are the commonly used control structures in Linux shell scripting:

1. if-else Statement

Used to execute commands conditionally.

Syntax:

```
if [ condition ]; then
   commands
elif [ another_condition ]; then
   other_commands
else
   fallback_commands
fi
```

Example:

```
num=10
if [ $num -gt 0 ]; then
    echo "Positive number"
else
    echo "Non-positive number"
fi
```

2. case Statement

Used for multiple choice decision making, like a switch-case in other languages.

Syntax:

```
case $variable in
    pattern1)
        commands ;;
    pattern2)
        commands ;;
    *)
        default_commands ;;
esac
```

Example:

```
day="Mon"
case $day in
    "Mon") echo "Start of the week" ;;
    "Fri") echo "End of the week" ;;
    *) echo "Midweek day" ;;
esac
```

3. for Loop

Used to iterate over a list or range.

Syntax:

```
for var in list; do
    commands
done
```

Example:

```
for i in 1 2 3 4 5; do
    echo "Number: $i"
done
```

4. while Loop

Executes commands as long as a condition is true.

Syntax:

```
while [ condition ]; do
    commands
done
```

Example:

```
count=1
while [ $count -le 5 ]; do
    echo "Count is $count"
    count=$((count + 1))
done
```

5. until Loop

Opposite of while: Executes commands until a condition becomes true.

Syntax:

```
until [ condition ]; do
    commands
done
```

Example:

```
count=1
until [ $count -gt 5 ]; do
    echo "Count is $count"
    count=$((count + 1))
done
```

Summary Table

Structure	Purpose	Common Use Case
if-else	Conditional branching	Check file existence, number test
case	Multi-branch decision	Menu selection, option handling
for	Iteration over list or range	Loop through numbers, files
while	Loop while condition is true	Read file line-by-line
until	Loop until condition becomes true	Retry logic until success

Break and continue in Linux.

for i in `seq 1 5` do if(($i== 2)) then break fi echo "value of i is $i" done	for i in `seq 1 5` do if(($i== 2)) then continue fi echo "value of i is $i" done
o/p : value of i is 1	o/p : value of i is 1 value of i is 3 value of i is 4 value of i is 5

Select loop in linux

```
clear
select DRINK in tea cofee water juice appey all none
do
case $DRINK in
tea|cofee|water|all)
        echo "Go to canteen";;
juice|appey)
        echo "Available at home";;
none)
        break ;;
*)
         echo "ERROR: Invalid selection";;
esac
done
```

output :

```
1) tea
2) cofee
3) water
4) juice
5) appey
6) all
7) none
#? 3
Go to canteen
#? 8
ERROR: Invalid selection
#? 2
Go to canteen
#?
```

3.4 Logical Operators (&&, ||, !) in Linux

In Linux shell scripting (especially bash), logical operators are used to control the flow of execution based on the success or failure of commands. These are particularly useful in conditional statements and command chaining.

1. && – Logical AND

- Executes the second command only if the first command succeeds (i.e., returns exit status 0).
- Commonly used for chaining commands.

Syntax:

command1 && command2

- Example:

mkdir newdir && cd newdir

✅ Here, cd newdir will execute only if mkdir newdir succeeds.

2. || – Logical OR

- Executes the second command only if the first command fails (i.e., returns a non-zero exit status).

- Syntax:

command1 || command2

- Example:

cd unknown_dir || echo "Directory not found!"

❌ If cd unknown_dir fails, then echo will be executed.

3. ! – Logical NOT

- Negates the exit status of a command.
- If the command succeeds, ! makes it fail and vice versa.

- Syntax:

! command

- Example:

! ls somefile && echo "File does not exist"

✅ If ls somefile fails (file doesn't exist), the ! makes it "succeed", and the message is printed.

✅ Combined Example:

```
file="myfile.txt"
[ -f "$file" ] && echo "$file exists" || echo "$file does not exist"
```

- This checks if a file exists.
- If it exists, it prints the first message.
- If not, the second message is printed.

Example

```
# Logical Operators with Mathematical Operations
a=10
b=5
c=0
echo "a = $a, b = $b, c = $c"
echo
# 1. && (Logical AND) - Perform multiplication only if a > b
[ $a -gt $b ] && echo "$a is greater than $b, so a * b = $((a * b))"

# 2. || (Logical OR) - If b is not greater than a, then show subtraction
[ $b -gt $a ] || echo "$b is not greater than $a, so a - b = $((a - b))"

# 3. ! (Logical NOT) - Check if c is NOT greater than 0, then do addition
if ! [ $c -gt 0 ]; then
  echo "$c is not greater than 0, so a + b = $((a + b))"
fi
```

The exit status of a command

Each Linux command returns a status when it terminates normally or abnormally. You can use value of exit status in the shell script to display an error message or take some sort of action. For example, if tar command is unsuccessful, it returns a code which tells the shell script to send an e-mail to sysadmin.

Exit Status

- Every Linux command executed by the shell script or user, has an exit status.
- The exit status is an integer number.
- The Linux man pages stats the exit statuses of each command.
- 0 exit status means the command was successful without any errors.
- A non-zero (1-255 values) exit status means command was failure.
- You can use special shell variable called $? to get the exit status of the previously executed command. To print $? variable use the echo command:

```
echo $?
date  # run date command
echo $? # print exit status
foobar123 # not a valid command
echo $? # print exit status
```

How Do I See Exit Status Of The Command?

Type the following command:

```
date
```

To view exist status of date command, enter:

```
echo $?
```

Sample Output:

```
0
```

Try non-existence command

```
date1
echo $?
ls /eeteec
echo $?
```

Sample Output:

```
2
```

According to ls man page - *exit status is 0 if OK, 1 if minor problems, 2 if serious trouble*

3.5 test and [] Commands for Condition Testing

In Linux shell scripting, condition testing is essential for decision-making. The test command and the [] (square brackets) are used to evaluate conditions such as comparisons on files, numbers, and strings. Both are functionally equivalent, but the square brackets are more commonly used in scripts for readability.

Test uses certain operators to evaluate the condition on its right and returns either a true or false exit status, which is then used by if statement for decision making

Test works in three ways:

Compare two numbers

Compare two strings

Checks file's attributes

General Syntax

- Using test:

  test condition

  test -f "$file"

- Using [ ]:

  [condition]

  [-f /etc/passwd]

Note: In [ ], spaces are required after the opening [and before the closing].

1. File Condition Tests

These check the attributes of files or directories.

Expression	Meaning
-e file	File exists
-f file	File exists and is a regular file
-d file	File exists and is a directory
-r file	File is readable
-w file	File is writable
-x file	File is executable
-file	File exits and is a directory

Expression	Meaning
-u file	File exits and has SUID bit set
-k file	File exits and has sticky bit set
-L file	File exits and has symbolic link (Korn and Bash only)
F1 –nt f2	F1 is newer than f2(Korn and Bash only)
F1 –ot f2	F1 is older than f2(Korn and Bash only)
F1 –ef f2	F1 is linked to f2(Korn and Bash only)

Example:

```
if [ -f /etc/passwd ]; then
  echo "File exists"
fi
```

File Condition Testing with test

```
file="/etc/passwd"
if test -f "$file"; then
  echo "The file exists and is a regular file."
else
  echo "File does not exist or is not a regular file."
fi
```

2. Numeric Condition Tests

Used to compare integer values.

Expression	Meaning
n1 -eq n2	n1 is equal to n2
n1 -ne n2	n1 is not equal to n2
n1 -gt n2	n1 is greater than n2
n1 -lt n2	n1 is less than n2
n1 -ge n2	n1 is greater than or equal to n2

Expression	Meaning
n1 -le n2	n1 is less than or equal to n2

Example:

```
a=10
b=20
if [ "$a" -lt "$b" ]; then
  echo "a is less than b"
fi
```

Numeric Condition Testing with test

```
a=10
b=20
if test "$a" -lt "$b"; then
  echo "a is less than b"
else
  echo "a is not less than b"
fi
```

3. String Condition Tests

Used to compare strings or check their properties.

Expression	Meaning
str1 = str2	Strings are equal
str1 != str2	Strings are not equal
-z str	String is empty
-n str	String is not empty

Example:

```
name="admin"
if [ "$name" = "admin" ]; then
  echo "Welcome, admin"
fi
```

String Condition Testing with test

```
username="admin"
if test "$username" = "admin"; then
  echo "Welcome, admin"
else
  echo "Access denied"
fi
```

4. Using [[...]] (Advanced Bash)

[[...]] is a more modern and safer alternative to [...], with features like:

- Pattern matching using == with wildcards
- No need to quote variables (avoids word splitting issues)
- Supports logical operators like && and ||

Example:

```
if [[ $user == a* ]]; then
  echo "Username starts with 'a'"
fi
```

Summary

- test, [ ], and [[ ]] are tools for conditional checks in shell scripts.
- Use -f, -d, etc., for file tests.
- Use -eq, -lt, etc., for numeric comparisons.
- Use =, !=, -z, -n for string evaluations.
- Prefer [[...]] for safer and more advanced scripting when using Bash.

3.6 Arithmetic Operations : expr and $(( ))

In Linux shell scripting, arithmetic operations are commonly performed using the expr command or the $(()) syntax. These tools allow you to perform basic integer arithmetic like addition, subtraction, multiplication, etc.

The expr statement

The expr is an external command which is used for computations in Linux. It performs following features.

- Perform arithmetic operations on integers
- Manipulates strings.

Arithmetic Computations

expr can perform 4 basic arithmetic operations and the modulus function.

i.e for two variables x any y the arithmetic operations can be:

Indicates the shell prompt.

$expr $x + $y

$expr $x - $y

Is required as * is considered as meta-character and \ will hide its special meaning

$expr $x * $y

$expr $x / $y

$expr $x % $y

The above example shows the working of expr command on the command line. If the user wants to use the expr in shell scripts the user has to use it with command substitution.

e.g. to store the value of the summation of two values in a variable.

c=`expr $x + $y`

the value of sum is stored in variable c.

1. Using expr

expr is a command-line utility used to evaluate expressions.

Syntax:

expr <operand1> <operator> <operand2>

- Example:

a=10

b=5

result=$(expr $a + $b)

echo "Sum is: $result"

⚠️ Notes:

- Spaces between operands and operator are mandatory.
- You need to escape special characters like *:

result=$(expr $a * $b)

2. Using Arithmetic Expansion $(( ))

This is the preferred and modern method for arithmetic operations in bash. It's more concise and readable.

Syntax:

$((expression))

Example:

a=10

b=5

sum=$((a + b))

echo "Sum is: $sum"

product=$((a * b))

echo "Product is: $product"

⚡ Advantages:

- No need to escape operators.
- Works directly within scripts and command substitution.
- Faster and more readable than expr.

Supported Operators

Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo	%

String Handling

The expr command is also used for handling strings. Using expr we can perform basic string functions like,

1. Finding length of string

```
echo "enter string"
read s
len=`expr length $s`
echo $len
```

2. Substring of string

```
st="good-morning"
t=`expr substr $st 4 4`
echo $t
```

3. Concat string

```
echo "enter string"
read s1
echo "enter string"
read s2
echo "concate is"  $s1 $s2
```

Logical operator -a(and) -o(or) and !(not)

What they mean:

- -a = AND (both conditions must be true)
- -o = OR (either condition can be true)
- ! = NOT (negates the condition)

```
# Example files for testing
file1="/etc/passwd"
file2="/tmp/nonexistentfile"

# Using -a (AND)
if [ -e "$file1" -a ! -e "$file2" ]; then
  echo "File1 exists AND File2 does NOT exist."
else
  echo "Either File1 does not exist OR File2 exists."
fi
```

```
# Using -o (OR)
if [ -e "$file1" -o -e "$file2" ]; then
  echo "Either File1 exists OR File2 exists (or both)."
else
  echo "Neither File1 nor File2 exists."
fi
```

Explanation:

- [-e "$file1" -a ! -e "$file2"]
  Checks if **file1 exists AND file2 does NOT exist**
- [-e "$file1" -o -e "$file2"]
  Checks if **either file1 exists OR file2 exists**

Important notes:

- The -a and -o operators are sometimes considered **deprecated** because they can cause ambiguous parsing in complex expressions.
- It's better to use [[...]] with && (AND) and || (OR) operators instead:

```
if [[ -e "$file1" && ! -e "$file2" ]]; then
  echo "File1 exists AND File2 does NOT exist."
fi

if [[ -e "$file1" || -e "$file2" ]]; then
  echo "Either File1 exists OR File2 exists."
fi
```

Unit 4 : Advanced Text Processing Tools

4.1 Introduction to Regular Expressions (Basic and Extended)
4.2 Pattern Matching using grep, egrep, and fgrep
4.3 Stream Editing with sed (search, replace, line deletion, insertion)

4.1 Introduction to Regular Expressions (Basic and Extended)

Regular expressions (regex) are powerful tools for pattern matching and text processing. In Linux, they are widely used with command-line tools such as grep, sed and awk. There are **two main types** of regular expressions in Linux:

- **Basic Regular Expressions (BRE)**
- **Extended Regular Expressions (ERE)**

1. Basic Regular Expressions (BRE)

BRE is the default mode used by many commands such as grep.

Key Features and Symbols:

Symbol	Description
.	Matches any single character
^	Matches the beginning of a line
$	Matches the end of a line
*	Matches zero or more of the preceding char
[]	Matches any one character in the set
[^]	Matches any one character **not** in the set
\{n,m\}	Matches between n and m occurrences
\?	Matches zero or one occurrence
\+	Matches one or more occurrences
|	Alternation (OR)
\(and \)	Group expressions

In BRE, special characters like {}, ?, +, |, (), must be **escaped with a backslash** (\).

Example Commands (BRE):

```
grep '^a' file.txt      # Lines starting with 'a'
grep 'ing$' file.txt    # Lines ending with 'ing'
grep '[0-9]' file.txt   # Lines containing a digit
grep '^\(ab\)*$' file.txt # Lines with zero or more repetitions of 'ab'
```

2. Extended Regular Expressions (ERE)

Used by tools like egrep or grep -E, ERE supports more features without needing to escape characters.

Key Features and Symbols (additional from BRE):

Symbol	Description
?	Matches zero or one of the preceding element
+	Matches one or more of the preceding element
`	`
()	Group expressions

I
n ERE, you do **not** need to escape +, ?, |, ().

Example Commands (ERE):

```
grep -E 'a|b' file.txt       # Lines containing 'a' or 'b'
grep -E 'ab+' file.txt       # 'a' followed by one or more 'b'
grep -E '(foo|bar)' file.txt  # Lines with 'foo' or 'bar'
grep -E '^a(bc)*$' file.txt   # Lines starting with 'a' followed by zero or more 'bc'
```

Common Tools Using Regex in Linux

Tool	Regex Type Used	Notes
grep	BRE by default	Use -E for ERE
sed	BRE by default	Can use ERE with -r
awk	Uses ERE	Powerful for structured text

4.2 Pattern Matching using grep, egrep, and fgrep

grep [OPTIONS] PATTERN [FILE...]

1. grep (Basic Regular Expressions - BRE)

- Uses **Basic Regular Expressions** by default.
- You must **escape special characters** like ?, +, |, ().

```
grep '^hello' file.txt       # Lines starting with 'hello'
grep 'world$' file.txt       # Lines ending with 'world'
grep '[0-9]' file.txt        # Lines containing digits
grep 'ab\+c' file.txt        # 'a' followed by one or more 'b' then 'c' (BRE requires \+)
```

🔹 2. egrep (Extended Regular Expressions - ERE)

- Equivalent to grep -E
- **No need to escape** special characters like +, ?, |, ().

```
egrep 'ab+c' file.txt        # 'a' followed by one or more 'b' then 'c'
egrep '(foo|bar)' file.txt   # Match 'foo' or 'bar'
egrep 'a[0-9]+z?' file.txt   # 'a' followed by one or more digits and optional 'z'
```

✅ Use egrep when you want more complex regex without escaping.

3. fgrep (Fixed grep / Literal Match)

- Searches for **exact strings**, **no regex** processing.
- Much faster when regex is not needed.
- Equivalent to grep -F

fgrep 'a.b' file.txt # Matches literal string 'a.b', not any character

fgrep 'hello?' file.txt # Matches literal string 'hello?'

fgrep '[a-z]' file.txt # Matches literal string '[a-z]', not a character class

✅ Use fgrep when searching for **plain text**, especially with special characters.

Comparison Table

Feature	**grep (BRE)**	**egrep (ERE)**	**fgrep (Fixed)**
Regex Type	Basic	Extended	None (literal)
Special Chars	Escaped	Direct use	Treated as plain
Grouping ()	\(\)	()	Literal
Alternation `	`	|	`
Use Case	Simple patterns	Complex patterns	Exact text search

4.3 Stream Editing with sed (search, replace, line deletion, insertion)

sed stands for **Stream Editor** — a powerful tool to **search**, **replace**, **delete**, and **insert** text in a stream or file.

Basic Syntax

sed [options] 'command' file

You can also use sed with pipelines:

cat file | sed 'command'

1. **Search and Replace (s)**

sed 's/pattern/replacement/' file

Common Variants:

Command	Description
s/foo/bar/	Replace **first** occurrence of foo with bar on each line
s/foo/bar/g	Replace **all** occurrences on each line
s/foo/bar/2	Replace **2nd occurrence** only
s/foo/bar/gI	Replace all (case-insensitive)

Example:

sed 's/hello/hi/' file.txt

sed 's/ERROR/OK/g' log.txt

sed 's/[0-9]\+/#/' numbers.txt

2. Delete Lines (d)

sed 'Nd' file.txt # Delete line N

Examples:

sed '3d' file.txt # Delete line 3

sed '1,5d' file.txt # Delete lines 1 to 5

sed '/^$/d' file.txt # Delete all blank lines

sed '/error/d' file.txt # Delete lines containing 'error'

3. Insert and Append Lines

Action	Syntax	Example
Insert	sed 'Nd i\text'	sed '2i\New line before 2'
Append	sed 'Nd a\text'	sed '2a\New line after 2'

- Use \ to continue text on a new line in shell.

sed '1i\Start of file' file.txt # Insert at beginning

sed '$a\End of file' file.txt # Append at end

4. Change a Line (c)

sed '3c\This is new content' file.txt

Replaces line 3 completely.

5. Multiple Commands

You can run **multiple sed commands** using:

- **Semicolon (;)**
- **Multiple -e options**

sed -e '1d' -e 's/foo/bar/' file.txt

OR

sed '1d; s/foo/bar/' file.txt

- **In-place Editing (-i)**

To **edit a file directly** (no output to stdout):

sed -i 's/foo/bar/g' file.txt

Optionally backup original:

sed -i.bak 's/foo/bar/g' file.txt # Backs up to file.txt.bak

Summary Table

Task	**Command Example**
Replace	sed 's/old/new/' file.txt
Replace globally	sed 's/old/new/g' file.txt
Delete line	sed '5d' file.txt
Delete range	sed '10,20d' file.txt
Delete blank	sed '/^$/d' file.txt

Task	Command Example
Insert before	sed '3i\Inserted line' file.txt
Append after	sed '3a\Appended line' file.txt
Change line	sed '2c\New content' file.txt

grep command in Unix/Linux

The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression (grep stands for global search for regular expression and print out).

Syntax:

grep [options] pattern [files]

Options Description

-c : This prints only a count of the lines that match a pattern
-h : Display the matched lines, but do not display the filenames.
-i : Ignores, case for matching
-l : Displays list of a filenames only.
-n : Display the matched lines and their line numbers.
-v : This prints out all the lines that do not matches the pattern
-e exp : Specifies expression with this option. Can use multiple times.
-f file : Takes patterns from file, one per line.
-E : Treats pattern as an extended regular expression (ERE)
-w : Match whole word
-o : Print only the matched parts of a matching line,
with each such part on a separate output line.

-A n : Prints searched line and nlines after the result.
-B n : Prints searched line and n line before the result.
-C n : Prints searched line and n lines after before the result.

Sample Commands
Consider the below file as an input.

$cat > geekfile.txt

unix is great os. unix is opensource. unix is free os.

learn operating system.

Unix linux which one you choose.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

1. Case insensitive search : The -i option enables to search for a string case insensitively in the given file. It matches the words like “UNIX”, “Unix”, “unix”.

$grep -i "UNix" geekfile.txt

Output:

unix is great os. unix is opensource. unix is free os.

Unix linux which one you choose.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

2. Displaying the count of number of matches : We can find the number of lines that matches the given string/pattern

$grep -c "unix" geekfile.txt
Output:

2

3. Display the file names that matches the pattern : We can just display the files that contains the given string/pattern.

$grep -l "unix" *

or

$grep -l "unix" f1.txt f2.txt f3.xt f4.txt
Output:

geekfile.txt

4. Checking for the whole words in a file : By default, grep matches the given string/pattern even if it is found as a substring in a file. The -w option to grep makes it match only the whole words.

$ grep -w "unix" geekfile.txt
Output:

unix is great os. unix is opensource. unix is free os.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

5. Displaying only the matched pattern : By default, grep displays the entire line which has the matched string. We can make the grep to display only the matched string by using the -o option.

$ grep -o "unix" geekfile.txt
Output:

unix

unix

unix

unix

unix

unix

6. Show line number while displaying the output using grep -n : To show the line number of file with the line matched.

$ grep -n "unix" geekfile.txt
Output:

1:unix is great os. unix is opensource. unix is free os.

4:uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

7. Inverting the pattern match : You can display the lines that are not matched with the specified search string pattern using the -v option.

$ grep -v "unix" geekfile.txt
-vn
Output:

learn operating system.

Unix linux which one you choose.

8. Matching the lines that start with a string : The ^ regular expression pattern specifies the start of a line. This can be used in grep to match the lines which start with the given string or pattern.

$ grep "^unix" geekfile.txt
Output:

unix is great os. unix is opensource. unix is free os.

9. Matching the lines that end with a string : The $ regular expression pattern specifies the end of a line. This can be used in grep to match the lines which end with the given string or pattern.

$ grep "os$" geekfile.txt

10.Specifies expression with -e option. Can use multiple times :

$grep –e "Agarwal" –e "Aggarwal" –e "Agrawal" geekfile.txt

11. -f file option Takes patterns from file, one per line.

$cat pattern.txt

Agarwal
Aggarwal
Agrawal

$grep –f pattern.txt geekfile.txt

12. Print n specific lines from a file: -A prints the searched line and n lines after the result, -B prints the searched line and n lines before the result, and -C prints the searched line and n lines after and before the result.

Syntax:
$grep -A[NumberOfLines(n)] [search] [file]

$grep -B[NumberOfLines(n)] [search] [file]

$grep -C[NumberOfLines(n)] [search] [file]

Example:
$grep -A1 learn geekfile.txt

Output:
learn operating system.
Unix linux which one you choose.
--
uNix is easy to **learn**.unix is a multiuser os.Learn unix .unix is a powerful.

(Prints the searched line along with the next n lines (here n = 1 (A1).)
(Will print each and every occurrence of the found line, separating each output by --)
(Output pattern remains the same for -B and -C respectively)
Unix linux which one you choose.
--
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a
powerful.
Unix linux which one you choose.
--
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

13. Search recursively for a pattern in the directory: -R prints the searched pattern in the given directory recursively in all the files.
Syntax

$grep -R [Search] [directory]

Example :
$grep -iR geeks /home/geeks

Output:

./geeks2.txt:Well Hello Geeks

./geeks1.txt:I am a big time geek

-i to search for a string case insensitively

-R to recursively check all the files in the directory.

Linux grep

The 'grep' command stands for **"global regular expression print"**. grep command filters the content of a file which makes our search easy.

grep with pipe

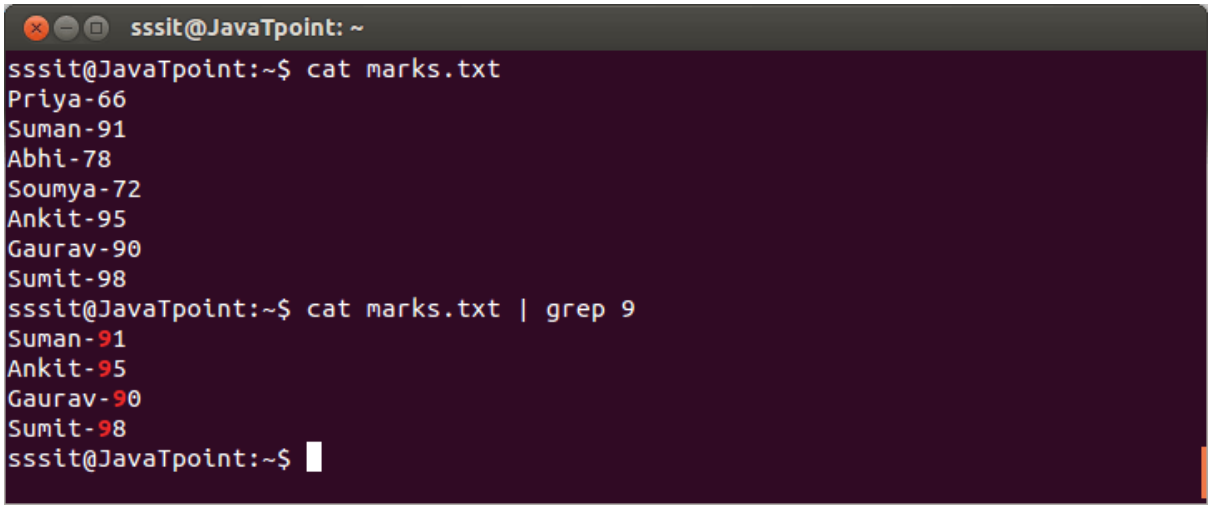
The 'grep' command is generally used with pipe **(|)**.

Syntax:

command | grep **<searchWord>**

Example:

```
cat marks.txt | grep 9
```



Look at the above snapshot, grep command filters all the data containing '9'.

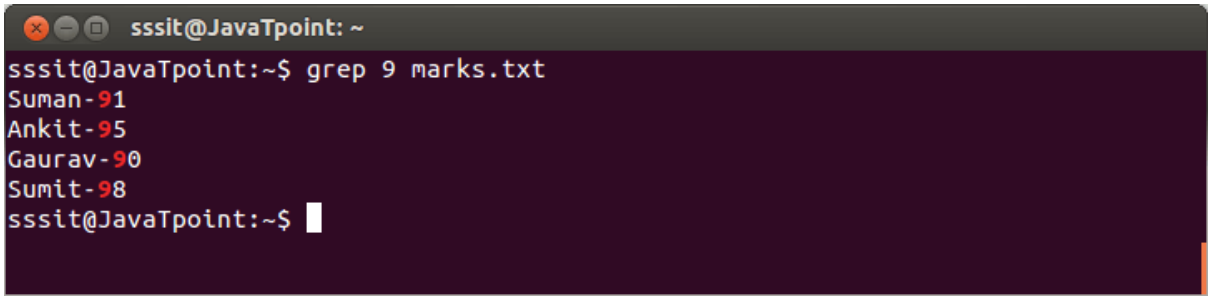
grep without pipe

It can be used without pipe also.

Syntax:

grep **<searchWord>** **<file** name**>**

Example: grep 9 marks.txt



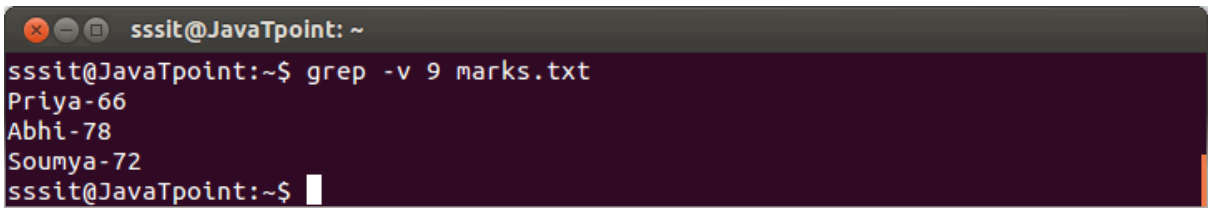
Look at the above snapshot, grep command do the same work as earlier example but without pipe.

grep options

- **grep -vM**: The 'grep -v' command displays lines not matching to the specified word.

  Syntax: grep -v **<searchWord> <fileName>**

  Example: grep -v 9 marks.txt

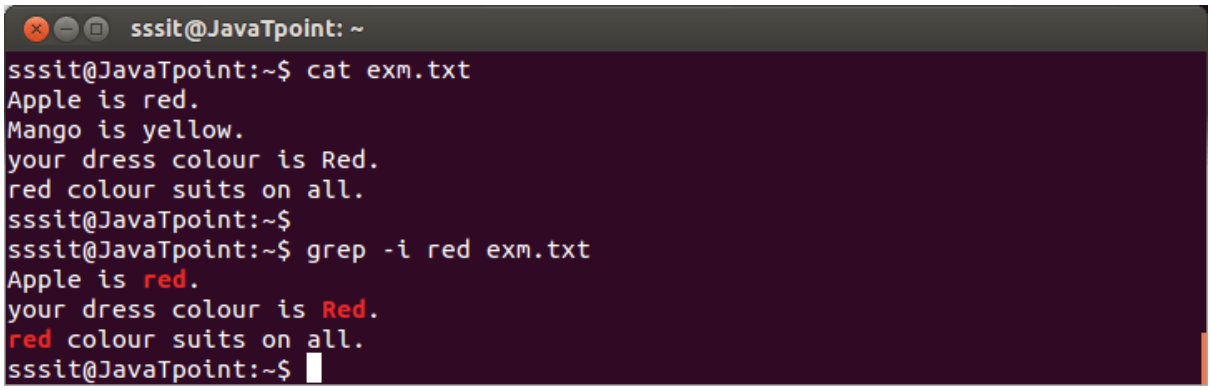
  

  Look at the above snapshot, command **"grep -v 9 marks.txt"** displays lines hwich don't contain our search word '9'.

- **grep -i**: The 'grep -i' command filters output in a case-insensitive way.

  Syntax: grep -i **<searchWord> <fileName>**

  Example: grep -i red exm.txt

  Look at the above snapshot, command **"grep -i red exm.txt"** displays all lines containing 'red' whether in upper case or lower case.

- **grep -A/ grep -B/ grep -C**

  grep -A command is used to display the **line after the result**.

  grep -B command is used to display the **line before the result**.

  grep -C command is used to display the **line after and line before** the result.

  You can use (A1, A2, A3.....)(B1, B2, B3....)(C1, C2, C3....) to display any number of lines.

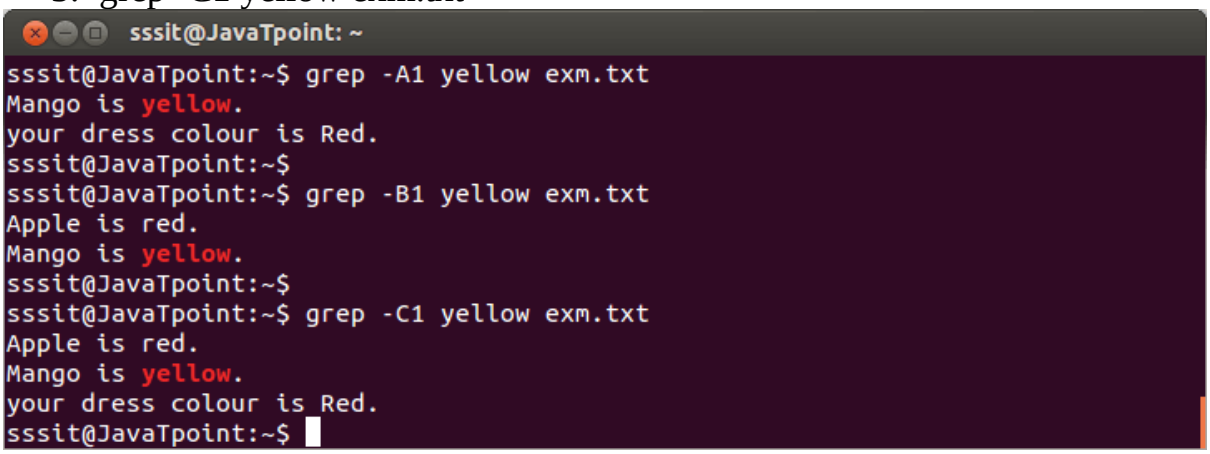
  Syntax: grep -A**<lineNumber> <searchWord> <fileName>**

  grep -B**<lineNumber> <searchWord> <fileName>**

  grep -C**<lineNumber> <searchWord> <fileName>**

  Example:

  1. grep -A1 yellow exm.txt
  2. grep -B1 yellow exm.txt
  3. grep -C1 yellow exm.txt



  Look at the above snapshot, command **"grep -A1 yellow exm.txt"** displays searched line with next succeeding line, command **"grep -B1 yellow exm.txt"** displays searched line with one preceding line and command **"grep -C1 yellow exm.txt"** displays searched line with one preceding and succeeding line.

BRE:(BASIC REGULAR EXPRESSION)

Admin@DESKTOP-TRR2ACF ~/program/grep

$ cat bre.txt
khyati
rajesh
solanki

```
khushi
pinky
kinjal

Admin@DESKTOP-TRR2ACF ~/program/grep

$ grep ^k bre.txt
khyati
khushi
kinjal

Admin@DESKTOP-TRR2ACF ~/program/grep

$ cat bre.txt
1khyati
2rajesh
3solanki
4khushi
5pinky
6kinjal

Admin@DESKTOP-TRR2ACF ~/program/grep
$ grep ^[3-4] bre.txt
3solanki
4khushi


Admin@DESKTOP-TRR2ACF ~/program/grep

$ grep ^[3-5] bre.txt
3solanki
4khushi
5pinky

Admin@DESKTOP-TRR2ACF ~/program/grep

$ grep ^[^3-5] bre.txt
1khyati
2rajesh
6kinjal

Admin@DESKTOP-TRR2ACF ~/program/grep
$ grep -i "agg*[ar][ar][vw]al" p1.txt
Agarwal
Aggarwal
Agrawal

Admin@DESKTOP-TRR2ACF ~/program/grep
$ grep -i "sa[kx]s*ena" bre.txt
```

```
saksena
saxena

Admin@DESKTOP-TRR2ACF ~/program/grep
$ grep -i "qui[te][te]" bre.txt
quite
quiet

Admin@DESKTOP-TRR2ACF ~/program/grep
$ grep k. bre.txt
1khyati
3solanki
4khushi
5pinky
6kinjal
saksena
```

Basic Regular Expression

Regular Expression provides an ability to match a “string of text” in a very flexible and concise manner. A “string of text” can be further defined as a single character, word, sentence or particular pattern of characters.

Like the shell’s wild–cards which match similar filenames with a single expression, grep uses an expression of a different sort to match a group of similar patterns.

- **[ ]**: Matches any one of a set characters
- **[ ] with hyphen**: Matches any one of a range characters
- ^: The pattern following it must occur at the beginning of each line
- ^ **with [ ] :** The pattern must not contain any character in the set specified
- **$**: The pattern preceding it must occur at the end of each line
- **. (dot)**: Matches any one character
- \ **(backslash)**: Ignores the special meaning of the character following it
- ***:** zero or more occurrences of the previous character
- **(dot).***: Nothing or any numbers of characters.

Examples

(a) [ ] : Matches any one of a set characters

1. ```
   $grep  “New[abc]”  filename
   ```

   It specifies the search pattern as :

   ```
   Newa , Newb or Newc
   ```

2. ```
   $grep  “[aA]g[ar][ar]wal”  filename
   ```

   It specifies the search pattern as

   ```
   Agarwal , Agaawal , Agrawal , Agrrwal

   agarwal , agaawal , agrawal , agrrwal
   ```

(b) Use [ ] with hyphen: Matches any one of a range characters

1. ```
   $grep  “New[a-e]” filename
   ```

   It specifies the search pattern as

   ```
   Newa , Newb or Newc , Newd, Newe
   ```

2. ```
   $grep  “New[0-9][a-z]”  filename
   ```

   It specifies the search pattern as: New followed by a number and then an alphabet.

   ```
   New0d, New4f etc
   ```

(c) Use ^: The pattern following it must occur at the beginning of each line

1. ```
   $grep  “^san”  filename
   ```

   Search lines beginning with san. It specifies the search pattern as

   ```
   sanjeev ,sanjay, sanrit , sanchit , sandeep etc.
   ```

2. ```
   $ls -l |grep  “^d”
   ```

   Display list of directories only

3. ```
   $ls -l |grep  “^-”
   ```

   Display list of regular files only

(d) Use ^ with [ ]: The pattern must not contain any character in the set specified

1. ```
   $grep  “New[^a-c]”  filename
   ```

   It specifies the pattern containing the word “New” followed by any character other than an ‘a’,’b’, or ‘c’

2. ```
   $grep  “^[^a-z A-Z]”  filename
   ```

   Search lines beginning with an non-alphabetic character

(e) Use $: The pattern preceding it must occur at the end of each line

```
$ grep "vedik$" file.txt
```

(f) Use . (dot): Matches any one character

```
$ grep "..vik" file.txt
$ grep "7..9$" file.txt
```
```

(g) Use \ (backslash): Ignores the special meaning of the character following it

1. ```
   $ grep "New\.\[abc\]" file.txt
   ```

   It specifies the search pattern as New.[abc]

2. ```
   $ grep "S\.K\.Kumar" file.txt
   ```

   It specifies the search pattern as

   ```
   S.K.Kumar
   ```

(h) Use *: zero or more occurrences of the previous character

```
$ grep "[aA]gg*[ar][ar]wal" file.txt
```

(i) Use (dot).*: Nothing or any numbers of characters.

```
$ grep "S.*Kumar" file.txt
```

ERE:(EXTENDED REGULAR EXPRESSION)

agarwal
aggarwal
agrawal
agarval
agraval
Admin@DESKTOP-TRR2ACF ~/program/grep
$ grep -i "ag+[ar][ar][vw]al" bre.txt

Admin@DESKTOP-TRR2ACF ~/program/grep
$ grep -i "agg?[ar][ar][vw]al" bre.txt

saksena
Saxena

Admin@DESKTOP-TRR2ACF ~/program/grep
$ grep -i "sa[kx]s*ena" bre.txt
saksena
saxena

Admin@DESKTOP-TRR2ACF ~/program/grep
$ grep -i "sa(ks/x)ena" bre.txt

sengupta
dasgupta
Admin@DESKTOP-TRR2ACF ~/program/grep
$ grep -i "(sen/das)gupta" bre.txt

Character Classes

Grep offers standard character classes as predefined functions to simplify bracket expressions. Below is a table that outlines some classes and the bracket expression equivalent.

Syntax	Description	Equivalent
`[[:alnum:]]`	All letters and numbers.	`"[0-9a-zA-Z]"`
`[[:alpha:]]`	All letters.	`"[a-zA-Z]"`
`[[:blank:]]`	Spaces and tabs.	`[CTRL+V<TAB> ]`
`[[:digit:]]`	Digits 0 to 9.	`[0-9]`
`[[:lower:]]`	Lowercase letters.	`[a-z]`
`[[:punct:]]`	Punctuation and other characters.	`"[^a-zA-Z0-9]"`
`[[:upper:]]`	Uppercase letters.	`[A-Z]`
`[[:xdigit:]]`	Hexadecimal digits.	`"[0-9a-fA-F]"`

Quantifiers

Quantifiers are metacharacters that specify the number of appearances. The following table shows each grep quantifier syntax with a short description.

Syntax	Description
`*`	Zero or more matches.
`?`	Zero or one match.
`+`	One or more matches.
`{n}`	`n` matches.
`{n,}`	`n` or more matches.
`{,m}`	Up to `m` matches.
`{n,m}`	From `n` up to `m` matches.

Sed Command in Linux/Unix with examples

SED command in UNIX stands for stream editor and it can perform lots of functions on file like searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace. By using SED you can edit files even without opening them, which is much quicker way to find and replace something in file, than first opening that file in VI Editor and then changing it.

- SED is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
- SED command in unix supports regular expression which allows it perform complex pattern matching.

Syntax:

sed OPTIONS... [SCRIPT] [INPUTFILE...]

Example:
Consider the below text file as an input.

$cat > geekfile.txt

unix is great os. unix is opensource. unix is free os.

learn operating system.

unix linux which one you choose.

unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

Sample Commands

1. **Replacing or substituting string :** Sed command is mostly used to replace the text in a file. The below simple sed command replaces the word “unix” with “linux” in the file.

   $sed 's/unix/linux/' geekfile.txt

   Output :
   linux is great os. unix is opensource. unix is free os.

   learn operating system.

   linux linux which one you choose.

   linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

   Here the “s” specifies the substitution operation. The “/” are delimiters. The “unix” is the search pattern and the “linux” is the replacement string.

By default, the sed command replaces the first occurrence of the pattern in each line and it won't replace the second, third…occurrence in the line.

2. **Replacing the nth occurrence of a pattern in a line :** Use the /1, /2 etc flags to replace the first, second occurrence of a pattern in a line. The below command replaces the second occurrence of the word "unix" with "linux" in a line.

   $sed 's/unix/linux/2' geekfile.txt

   Output:
   unix is great os. linux is opensource. unix is free os.

   learn operating system.

   unix linux which one you choose.

   unix is easy to learn.linux is a multiuser os.Learn unix .unix is a powerful.

3. **Replacing all the occurrence of the pattern in a line :** The substitute flag /g (global replacement) specifies the sed command to replace all the occurrences of the string in the line.

   $sed 's/unix/linux/g' geekfile.txt

   Output :
   linux is great os. linux is opensource. linux is free os.

   learn operating system.

   linux linux which one you choose.

   linux is easy to learn.linux is a multiuser os.Learn linux .linux is a powerful.

4. **Replacing from nth occurrence to all occurrences in a line :** Use the combination of /1, /2 etc and /g to replace all the patterns from the nth occurrence of a pattern in a line. The following sed command replaces the third, fourth, fifth… "unix" word with "linux" word in a line.

   $sed 's/unix/linux/3g' geekfile.txt

   Output:
   unix is great os. unix is opensource. linux is free os.

   learn operating system.

   unix linux which one you choose.

unix is easy to learn.unix is a multiuser os.Learn linux .linux is a powerful.

5. **Parenthesize first character of each word :** This sed example prints the first character of every word in parenthesis.

   $ echo "Welcome To The Geek Stuff" | sed 's/\(\b[A-Z]\)/\(\1\)/g'

   Output:

   (W)elcome (T)o (T)he (G)eek (S)tuff

6. **Replacing string on a specific line number :** You can restrict the sed command to replace the string on a specific line number. An example is

   $sed '3 s/unix/123/' geekfile.txt

   Output:
   unix is great os. unix is opensource. unix is free os.

   learn operating system.

   linux linux which one you choose.

   unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

   The above sed command replaces the string only on the third line.

7. **Duplicating the replaced line with /p flag :** The /p print flag prints the replaced line twice on the terminal. If a line does not have the search pattern and is not replaced, then the /p prints that line only once.

   $sed 's/unix/linux/p' geekfile.txt

   Output:
   linux is great os. unix is opensource. unix is free os.

   linux is great os. unix is opensource. unix is free os.

   learn operating system.

   linux linux which one you choose.

   linux linux which one you choose.

   linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

8. **Printing only the replaced lines :** Use the -n option along with the /p print flag to display only the replaced lines. Here the -n option suppresses the duplicate rows generated by the /p flag and prints the replaced lines only one time.

   $sed -n 's/unix/linux/p' geekfile.txt

   Output:
   linux is great os. unix is opensource. unix is free os.

   linux linux which one you choose.

   linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

   If you use -n alone without /p, then the sed does not print anything.

9. **Replacing string on a range of lines :** You can specify a range of line numbers to the sed command for replacing a string.

   $sed '1,3 s/unix/linux/' geekfile.txt

   Output:
   linux is great os. unix is opensource. unix is free os.

   learn operating system.

   linux linux which one you choose.

   unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

   Here the sed command replaces the lines with range from 1 to 3. Another example is

   $sed '2,$ s/unix/linux/' geekfile.txt

   Output:
   unix is great os. unix is opensource. unix is free os.

   learn operating system.

   linux linux which one you choose.

   linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful

   Here $ indicates the last line in the file. So the sed command replaces the text from second line to last line in the file.

10. **Deleting lines from a particular file :** SED command can also be used for deleting lines from a particular file. SED command is used for performing deletion operation without even opening the file

    Examples:

1. To Delete a particular line say n in this example

Syntax:

$ sed 'nd' filename.txt

Example:

$ sed '5d' filename.txt

2. To Delete a last line

Syntax:

$ sed '$d' filename.txt

3. To Delete line from range x to y

Syntax:

$ sed 'x,yd' filename.txt

Example:

$ sed '3,6d' filename.txt

4. To Delete from nth to last line

Syntax:

$ sed 'nth,$d' filename.txt

Example:

$ sed '12,$d' filename.txt

5. To Delete pattern matching line

Syntax:

$ sed '/pattern/d' filename.txt

Example:

$ sed '/abc/d' filename.txt

SED is used for finding, filtering, text substitution, replacement and text manipulations like insertion, deletion search, etc. It's a one of the powerful utilities offered by Linux/Unix systems. We can use sed with regular expressions. I hope atleast you have the basic knowledge about Linux regular expressions.

It provides Non-interactive editing of text files that's why it's used to automate editing and has two buffers – **pattern buffer** and **hold buffer**. *Sed* use *Pattern buffer* when it read files, line by line and that currently read line is inserted into pattern buffer whereas *hold buffer* is a long-term storage, it catch the information, store it and reuse it when it is needed. Initially, both are empty. SED command is used for performing different operations without even opening the file.
sed general syntax –

sed OPTIONS… [SCRIPT] [INPUTFILE…]

 # Let's start with File Spacing

1 – Insert one blank line after each line –

Admin@DESKTOP-TRR2ACF ~/program/sed
$ sed G f1.txt
unix is great os. unix is opensource. unix is free os.

learn operating system.

Unix linux which one you choose unix.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

2 – To insert two blank lines –

$ sed 'G;G' f1.txt
unix is great os. unix is opensource. unix is free os.

learn operating system.

Unix linux which one you choose unix.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

3 – Delete blank lines and insert one blank line after each line –

```
Admin@DESKTOP-TRR2ACF ~/program/sed
$ cat f1.txt
unix is great os. unix is opensource. unix is free os.
learn operating system.

Unix linux which one you choose unix.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a
powerful.

Admin@DESKTOP-TRR2ACF ~/program/sed
$ sed '/^$/d;G' f1.txt
unix is great os. unix is opensource. unix is free os.

learn operating system.

Unix linux which one you choose unix.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a

powerful.
```

4 – Insert a black line above every line which matches “Unix” –

```
Admin@DESKTOP-TRR2ACF ~/program/sed
$ cat f1.txt
unix is great os. unix is opensource. unix is free os.
learn operating system.
Unix linux which one you choose unix.
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a
powerful.

Admin@DESKTOP-TRR2ACF ~/program/sed
$ sed '/Unix/{x;p;x;}' f1.txt
unix is great os. unix is opensource. unix is free os.
learn operating system.
```

Unix linux which one you choose unix.
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a
powerful.

5 – Insert a blank line below every line which matches “Unix” –

```
Admin@DESKTOP-TRR2ACF ~/program/sed
$ sed '/Unix/G' f1.txt
unix is great os. unix is opensource. unix is free os.
learn operating system.
Unix linux which one you choose unix.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a
powerful.
```

6 – Insert 5 spaces to the left of every lines –

```
Admin@DESKTOP-TRR2ACF ~/program/sed
$ sed 's/^/     /' f1.txt
     unix is great os. unix is opensource. unix is free os.
     learn operating system.
     Unix linux which one you choose unix.
     uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a
     powerful.
```

 # Numbering lines

1 – Number each line of a file (left alignment). **=** is used to number the line. \t is used for tab between number and sentence –

```
Admin@DESKTOP-TRR2ACF ~/program/sed
$ sed =  f1.txt | sed 'N;s/\n/\t/'
1       unix is great os. unix is opensource. unix is free os.
2       learn operating system.
3       Unix linux which one you choose unix.
4       uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a
5       powerful.
```

2 – Number each line of a file (number on left, right-aligned). This command is similar to `cat -n filename`.
cl

```
Admin@DESKTOP-TRR2ACF ~/program/sed
$ sed = f1.txt | sed 'N; s/^/     /; s/ *\(.\{4,\}\)\n/\1  /'
  1  unix is great os. unix is opensource. unix is free os.
  2  learn operating system.
  3  Unix linux which one you choose unix.
  4  uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a
```

5 powerful.

3 – Number each line of file, only if line is not blank –

```
Admin@DESKTOP-TRR2ACF ~/program/sed
$ sed '/./=' f1.txt | sed '/./N; s/\n/ /'
1 unix is great os. unix is opensource. unix is free os.
2 learn operating system.
3 Unix linux which one you choose unix.
4 uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a
5 powerful.
```

```
Admin@DESKTOP-TRR2ACF ~/program/sed
$ sed = f1.txt
1
unix is great os. unix is opensource. unix is free os.
2
learn operating system.
3
Unix linux which one you choose unix.
4
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a
5
powerful.
```

 # Deleting lines

1 – Delete a particular line –
Syntax: sed 'nd' filename
Example :
sed '5d' a.txt

2 – Delete the last line
Syntax: sed '$d' filename

3 – Delete line from range x to y
Syntax: sed 'x,yd' filename

Example :
sed '3,5d' a.txt

4 – Delete from nth to last line
Syntax: sed 'nth,$d' filename
Example :
sed '2,$d' a.txt

5 – Delete the pattern matching line –
Syntax: sed '/pattern/d' filename
Example :
sed '/life/d' a.txt

6 – Delete lines starting from nth line and every 2nd line from there –
Syntax: sed 'n~2d' filename
Example :
sed '3~2d' a.txt

7 – Delete the lines which matches the pattern and 2 lines after to that –
Syntax: sed '/pattern/,+2d' filename
Example :
sed '/easy/,+2d' a.txt

8 – Delete blank Lines
sed '/^$/d' a.txt

9 – Delete empty lines or those begins with "#" –
sed -i '/^#/d;/^$/d' a.txt

 # View/Print the files

If we want to view content of file, then we use **cat** command and if we want to view the bottom and the top content of any file, we use tools such as **head** and **tail.** But what if we need to view a particular section in the middle of any file? Here we'll discuss, how to use SED command to view a section of any file.

1 – Viewing a file from x to y range –
Syntax: sed -n 'x,yp' filename
Example :
sed -n '2,5p' a.txt

2 – View the entire file except the given range –
Syntax: sed 'x,yd' filename
Example :
sed '2,4d' a.txt

3 – Print nth line of the file –
Syntax: sed -n 'address'p filename
Example :
sed -n '4'p a.txt

4 – Print lines from xth line to yth line.
Syntax: sed -n 'x,y'p filename
Example :
sed -n '4,6'p a.txt

5 – Print only the last line –
Syntax: sed -n '$'p filename

6 – Print from nth line to end of file –
Syntax: sed -n 'n,$p' filename
Example :
sed -n '3,$'p a.txt

Pattern Printing

7 – Print the line only which matches the pattern –
Syntax: sed -n /pattern/p filename
Example :
sed -n /every/p a.txt

8 – Print lines which matches the pattern i.e from input to xth line.
Syntax: sed -n '/pattern/,xp' filename
Example :
sed -n '/everyone/,5p' a.txt

Following prints lines which matches the pattern, 3rd line matches the pattern "everyone", so it prints from 3rd line to 5th line. Use **$** in place of 5, if want to print the file till end.

9 – Prints lines from the xth line of the input, up-to the line which matches the pattern. If the pattern doesn't found then it prints up-to end of the file.
Syntax: sed -n 'x,/pattern/p' filename
Example :
sed -n '1,/everyone/p' a.txt

10 – Print the lines which matches the pattern up-to the next xth lines –
Syntax: sed -n '/pattern/,+xp' filename
Example :
sed -n '/learn/,+2p' a.txt

 # Replacement with the sed command

1 – Change the first occurrence of the pattern –
sed 's/life/leaves/' a.txt

2 – Replacing the nth occurrence of a pattern in a line –
Syntax: sed 's/old_pattern/new_pattern/n' filename
Example :
sed 's/to/two/2' a.txt
We wrote "2" because we replaces the second occurrence. Likewise you can use 3, 4 etc according to need.

3 – Replacing all the occurrence of the pattern in a line.
sed 's/life/learn/g' a.txt

4 – Replace pattern from nth occurrence to all occurrences in a line.
Syntax: sed 's/old_pattern/new_pattern/ng' filename
Example :
sed 's/to/TWO/2g' a.txt

Note – This sed command replaces the second, third, etc occurrences of pattern "to" with "TWO" in a line.
If you wish to print only the replaced lines, then use "-n" option along with "/p" print flag to display only the replaced lines –

sed -n 's/to/TWO/p' a.txt

And if you wish to print the replaced lines twice, then only use "/p" print flag without "-n" option-

sed 's/to/TWO/p' a.txt

5 – Replacing pattern on a specific line number. Here, "m" is the line number.
Syntax: sed 'm s/old_pattern/new_pattern/' filename
Example :
sed '3 s/every/each/' a.txt

If you wish to print only the replaced lines –

sed -n '3 s/every/each/p' a.txt

6 – Replace string on a defined range of lines –
Syntax: sed 'x,y s/old_pattern/new_pattern/' filename
where,
x = starting line number
and y = ending line number
Example :
sed '2,5 s/to/TWO/' a.txt

Note – *$* can be used in place of "y" if we wish to change the pattern up-to last line in the file.
Example :
sed '2,$ s/to/TWO/' a.txt

7 – If you wish to replace pattern in order to ignore character case (beginning with uppercase or lowercase), then there are two ways to replace such patterns

–
First, By using “/i” print flag –
Syntax: sed ‘s/old_pattern/new_pattern/i’ filename
Example :
sed 's/life/Love/i' a.txt

Second, By using regular expressions –

sed 's/[Ll]ife/Love/g' a.txt

8 – To replace multiple spaces with a single space –
sed 's/ */ /g' filename

9 – Replace one pattern followed by the another pattern –
Syntax: sed ‘/followed_pattern/ s/old_pattern/new_pattern/’ filename
Example :
sed '/is/ s/live/love/' a.txt

10 – Replace a pattern with other except in the nth line.
Syntax: sed ‘n!s/old_pattern/new_pattern/’ filename
Example :
sed -i '5!s/life/love/' a.txt

Unix Sed Tutorial: Printing File Lines using Address and Patterns

Let us review how to print file lines using address and patterns in this first part of sed tutorial.

We’ll be posting several awesome sed tutorials with examples in the upcoming weeks.

Unix Sed Introduction

- sed is a “non-interactive” stream-oriented editor. Since its an “non-interactive” it can be used to automate editing if desired.
- The name sed is an abbreviation for stream editor, and the utility derives many of its commands from the ed line-editor (ed was the first UNIX text editor).
- This allows you to edit multiple files, or to perform common editing operations without ever having to open vi or emacs.
- sed reads from a file or from its standard input, and outputs to its standard output.
- sed has two buffers which are called pattern buffer and hold buffer. Both are initially empty.

Unix Sed Working methodology

This is called as one execution cycle. Cycle continues till end of file/input is reached.

1. Read a entire line from stdin/file.
2. Removes any trailing newline.
3. Places the line, in its pattern buffer.
4. Modify the pattern buffer according to the supplied commands.
5. Print the pattern buffer to stdout.

Printing Operation in Sed

Linux Sed command allows you to print only specific lines based on the line number or pattern matches. **“p”** is a command for printing the data from the pattern buffer. To suppress automatic printing of pattern space use -n command with sed. sed -n option will not print anything, unless an explicit request to print is found.

```
Syntax:

# sed -n 'ADDRESS'p filename



# sed -n '/PATTERN/p' filename
```

Let us first create thegeekstuff.txt file that will be used in all the examples mentioned below.

```
# cat thegeekstuff.txt

1. Linux - Sysadmin, Scripting etc.

2. Databases - Oracle, mySQL etc.

3. Hardware

4. Security (Firewall, Network, Online Security etc)

5. Storage

6. Cool gadgets and websites

7. Productivity (Too many technologies to explore, not much time available)

8. Website Design

9. Software Development
```

```
10.Windows- Sysadmin, reboot etc.
```

5 Sed ADDRESS Format Examples

Sed Address Format 1: NUMBER

This will match only Nth line in the input.

 # sed -n 'N'p filename
For example, 3p prints third line of input file thegeekstuff.txt as shown below.

```
# sed -n '3'p thegeekstuff.txt

3. Hardware
```

Sed Address Format 2: NUMBER1~NUMBER2

M~N with "p" command prints every Nth line starting from line M.

 # sed -n 'M~N'p filename
For example, 3~2p prints every 2nd line starting from 3rd line as shown below.

```
# sed -n '3~2'p thegeekstuff.txt

3. Hardware

5. Storage

7. Productivity (Too many technologies to explore, not much time available)

9. Software Development
```

Sed Address Format 3: START,END

M,N with "p" command prints Mth line to Nth line.

 # sed -n 'M,N'p filename
For example, 4,8p prints from 4th line to 8th line from input file thegeekstuff.txt

```
# sed -n '4,8'p thegeekstuff.txt

4. Security (Firewall, Network, Online Security etc)

5. Storage

6. Cool gadgets and websites
```

```
7. Productivity (Too many technologies to explore, not much time available)

8. Website Design
```

Sed Address Format 4: '$' Last Line

$ with "p" command matches only the last line from the input.

 # sed -n '$'p filename
For example, $p prints only the last line as shown below.

```
# sed -n '$'p thegeekstuff.txt

10.Windows- Sysadmin, reboot etc.
```

Sed Address Format 5: NUMBER,$

N,$ with "p" command prints from Nth line to end of file.

 # sed -n 'N,$p' filename
For example 4,$p prints from 4th line to end of file.

```
# sed -n '4,$p' thegeekstuff.txt

4. Security (Firewall, Network, Online Security etc)

5. Storage

6. Cool gadgets and websites

7. Productivity (Too many technologies to explore, not much time available)

8. Website Design

9. Software Development

10.Windows- Sysadmin, reboot etc.
```

6 Sed PATTERN Format Examples

Sed Pattern Format 1: PATTERN

PATTERN could be unix regular expression. The below command prints only the line which matches the given pattern.

 # sed -n /PATTERN/p filename
For example, following prints the line only which matches the pattern "Sysadmin".

```
# sed -n /Sysadmin/p thegeekstuff.txt

1. Linux - Sysadmin, Scripting etc.

10.Windows- Sysadmin, reboot etc.
```

Sed Pattern Format 2: /PATTERN/,ADDRESS

 # sed -n '/PATTERN/,Np' filename
For example, following prints lines which matches the pattern to Nth line, from input. 3rd line matches the pattern "Hardware", so it prints from 3rd line to 6th line.

```
# sed -n '/Hardware/,6p' thegeekstuff.txt

3. Hardware

4. Security (Firewall, Network, Online Security etc)

5. Storage

6. Cool gadgets and websites
```

Sed Pattern Format 3: ADDRESS,/PATTERN/

It prints from the Nth line of the input, to the line which matches the pattern. If the pattern doesnt match, it prints upto end of the input.

 # sed -n 'N,/PATTERN/p' filename
For example, 4th line matches the pattern "Security", so it prints from 3rd line to 4th line.

```
# sed -n '3,/Security/p' thegeekstuff.txt

3. Hardware

4. Security (Firewall, Network, Online Security etc)
```

Sed Pattern Format 4: /PATTERN/,$

It prints from the line matches the given pattern to end of file.

 # sed -n '/PATTERN/,$p' filename

```
# sed -n '/Website/,$p' thegeekstuff.txt

8. Website Design

9. Software Development
```

```
10.Windows- Sysadmin, reboot etc.
```

Sed Pattern Format 5: /PATTERN/,+N

It prints the lines which matches the pattern and next N lines following the matched line.

 # sed -n '/PATTERN/,+Np' filename
For example, following prints the 5th line which matches the pattern /Storage/ and next two lines following /Storage/.

```
# sed -n '/Storage/,+2p' thegeekstuff.txt

5. Storage

6. Cool gadgets and websites

7. Productivity (Too many technologies to explore, not much time available)
```

Sed Pattern Format 6: /PATTERN/,/PATTERN/

Prints the section of file between two regular expression (including the matched line).

 # sed -n '/P1/,/P2/p' filename
For example, 5th line matches "Storage" and 8th line matches "Design", so it prints 5th to 8th.

```
# sed -n '/Storage/,/Design/p' thegeekstuff.txt

5. Storage

6. Cool gadgets and websites

7. Productivity (Too many technologies to explore, not much time available)

8. Website Design
```

Unix Sed Tutorial : 7 Examples for Sed Hold and Pattern Buffer Operations

As its name implies, **sed hold buffer** is used to save all or part of the **sed pattern space** for subsequent retrieval. The contents of the pattern space can be copied to the hold space, then back again. No operations are performed directly on the hold space. sed provides a set of hold and get functions to handle these movements.
Sed h function
The h (hold) function copies the contents of the pattern space into a holding area (also called as **sed hold space**), destroying any previous contents of the holding area.

Sed H function
The H function appends the contents of the pattern space to the contents of the holding area. The former and new contents are separated by a newline.

Sed g function
The g function copies the contents of the holding area into the pattern space, destroying the previous contents of the pattern space.

Sed G function
The G function appends the contents of the holding area to the contents of the pattern space. The former and new contents are separated by a newline. The maximum number of addresses is two.

Sed x function
The exchange function interchanges the contents of the pattern space and the holding area. The maximum number of addresses is two.

Now let us see some examples to learn about the above commands.

Let us first create thegeekstuff.txt file that will be used in the examples mentioned below.

```
$ cat thegeekstuff.txt

#Linux

        Administration

        Scripting

        Tips and Tricks



#Windows

        Administration



#Database

        Mysql

        Oracle

        Queries

        Procedures
```

1. Double Space a File Content Using Sed Command

```
$sed 'G' thegeekstuff.txt

#Linux



        Administration



        Scripting



        Tips and Tricks



#Windows



        Administration



#Database



        Mysql



        Oracle



        Queries



        Procedures



$
```

In this example,

1. Sed reads a line and places it in the pattern buffer.
2. G command appends the hold buffer to the pattern buffer separated by \n. so one newline will be appended with the pattern space content.
3. Similarly, If you want to triple space a file, append hold buffer content to the pattern buffer twice. (G;G)

2. Print File Content in Reverse Order Using Sed Command

Print the lines of a file in reverse order (similar to tac command that we discussed earlier).

```
$sed -n '1!G;h;$p' thegeekstuff.txt

        Procedures

        Queries

        Oracle

        Mysql

#Database



        Administration

#Windows



        Tips and Tricks

        Scripting

        Administration

#Linux
```

In this example,

1. First line will be placed into the hold space as it is.
2. From the 2nd line onwards, just append the hold space content with the pattern space. (Remember 2nd line is in pattern space, and 1st line is in hold space).
3. Now 1st and 2nd line got reversed and move this to the hold space.
4. Repeat the above steps till last line.
5. Once the last line is reached, just append the hold space content with the pattern space and print the pattern space.

3. Print a Paragraph (Only if it contains given pattern) Using Sed Command

In thegeekstuff.txt print paragraph only if it contains the pattern “Administration”.

```
$ sed -e '/./{H;$!d;}' -e 'x;/Administration/!d' thegeekstuff.txt



Linux

        Administration

        Scripting

        Tips and Tricks



Windows

        Administration
```

In this example,

1. Till the empty line comes, keep appending the non empty lines into the hold space
2. When empty line comes i.e paragraph ends, exchange the data between pattern and hold space. So that whole paragraph will be available in pattern space. Check if pattern “Administration” is available, if yes don’t delete it i.e print the pattern space

4. Print the line immediately before a pattern match using Sed Command

Print only the line immediately before,the pattern “Mysql”.

```
$ sed -n '/Mysql/{g;1!p;};h' thegeekstuff.txt


#Database
```

In this example,

1. For each cycle, place the line into hold buffer, if it doesn’t match with the pattern “Mysql”.
2. If the line matches with the pattern, get the data from the hold space(previous line) using g command and print it.
3. In case, if the first line matches with the pattern “Mysql”,anyway hold space will be empty.(There is no previous line to the first line).So first line should not get printed(1!p)

5. Delete the last line of each paragraph using Sed Command

```
$ sed -n -e '/^$/{x;d}' -e '/./x;p' thegeekstuff.txt



#Linux

        Administration

        Scripting



#Windows



#Database

        Mysql

        Oracle

        Queries
```

In this example,

1. If the line is not empty,then exchange the line between pattern and hold space. So first line will be placed in the hold space.
2. When next non empty line comes, exchange the pattern space and hold space, and print the pattern space. i.e first non empty line will be printed and 2nd line goes to hold. And in next cycle, 2nd non empty line is printed when 3rd line goes to hold and goes on like this.
3. When empty line comes (previous line to the empty line will be available in hold buffer) just exchange pattern and hold space, and delete the line (last line of the paragraph) and start the next cycle.

6. For each line, append the previous line to the end of it using Sed Command

```
$ sed 'H;x;s/^\(.*\)\n\(.*\)/\2\1/' thegeekstuff.txt

#Linux

        Administration#Linux

        Scripting        Administration

        Tips and Tricks        Scripting
```

```
        Tips and Tricks

#Windows

        Administration#Windows

        Administration

#Database

        Mysql#Database

        Oracle        Mysql

        Queries Oracle

        Procedures                      Queries
```

In this example,

1. Place the first line in Hold buffer.
2. When the second line comes, append to Hold space (first line)
3. Then exchange pattern and hold buffer. So now pattern space will have first and second line separated by \n, Hold space will have only second line.
4. So interchange the lines in the pattern space.
5. The above steps happens till the end of the file

7. Prepend tag of every block to every line of that block

```
$ sed '

/^#/{

h

d

}

G

s/^\(.*\)\n#\(.*\)/\2 \1/' thegeekstuff.txt

Linux         Administration

Linux         Scripting

Linux         Tips and Tricks

Linux
```

```
Windows         Administration

Windows

Database         Mysql

Database        Oracle

Database        Queries

Database        Procedures
```

In this example,

1. When the first line of a block is met (beginning with #)
    - keep that line to the Hold Space via command `h'
    - Then delete using 'd' to start another cycle.
2. For the rest lines of a block, Command `G' appends the tag line from the Hold Space and substitute command interchanges tag and lines properly.